Лекция IV

27 октября 2017

Работа с текстом и символами в C++
Часть 2

Работа с текстом

Далее ставится задача ввода строки с текстового терминала. Для начала, можно воспользоваться функцией **get_value** из вспомогательной библиотеки **"ffhelpers.h"**

```
1 string my_str;
2 get_value(my_str, "Введите строку текста: ");
3
4 print("Вы ввели: <<", my_str, ">>\n");
```

Результат может быть не совсем ожидаемым:

```
Введите строку текста: первая строка, ура-ура!
Вы ввели: <<первая>>
```

Функция **get_value** позволяет из всего ввода записать в переменную строки **исключительно** первое слово. Под «словом» понимается непрерывная последовательность символов до первого пробела либо переноса строки.

Работа с текстом

Для того, чтобы получить всю строку, все символы до переноса строк, следует воспользоваться функцией **getline** из библиотеки <**string**>

```
getline(cin, string& str_to_fill,
        char separator = '\n');
```

- Функция считывает все введённые символы в строку, которую указали в аргументе **str_to_fill**
- Первым параметром идёт переменная, которая отвечает за ввод информации из какого-нибудь источника. Для консольного ввода это всегда **std::cin** (определён в <**iostream**>)
- Возможно задать разделитель **separator** - символ, **после которого** текст считан не будет. Сам разделитель в строку **str_to_fill** не помещается и не учавствует в дальнейших операциях ввода.

Как **getline**, так и **cin** также доступны напрямую при использовании **"ffhelpers.h"**

Пример ввода двух строк с терминала

```
1 string s1, s2;
2
3 print("Введите первую строку: ");
4 getline(cin, s1);
5 print("Введите вторую строку: ");
6 getline(cin, s2, '*');
7
8 print("Первая строка: [[", s1, "]]\n");
9 print("Вторая строка: [[", s2, "]]\n");
```

В строку **s1** попадут все символы до первого переноса строки, в **s2** - все, предшествующие символу «*».
Более того, во втором случае ввод не закончится, пока срезди символов не появится «звёздочка».

Работа с текстом

С текстовым вводом числовых значений и строк могут возникать некоторые проблемы. Пример:

```
1  string s1;
2  double real_num;
3
4  print("Введите число: ");
5  get_value(real_num);
6
7  print("Введите строку: ");
8  getline(cin, s1);
9
10 print("Введённое число: [[", real_num, "]]\n");
11 print("Введённая строка: [[", s1, "]]\n");
```

Если на первый призыв ввести некоторое число и нажать "Enter то в строку **s1** попадёт **только** символ переноса строки! Едва ли так хотелось.

Для избежания проблемы, можно воспользоваться функцией **clear_stdin** из **"ffhelpers.h"**

```
1  string s1;
2  double real_num;
3
4  print("Введите число: ");
5  get_value(real_num);
6
7  clear_stdin();
8  print("Введите строку: ");
9  getline(cin, s1);
10
11 print("Введённое число: [[", real_num, "]]\n");
12 print("Введённая строка: [[", s1, "]]\n");
```

По крайней мере, здесь программа подождёт, пока строке не будет введена и проигнорирует символ переноса строки.

Работа с текстом

Пример абстрактной «угадайки»: ввод текста по словам

```
1 string key = "зелёный", answer;
2
3 do {
4   get_value(answer, "\nУгадайте цвет: ");
5 } while ( key != answer ) ;
6
7 print("Правильный ответ!\n");
```

Возможный вывод программы:

```
Угадайте цвет: красный
Угадайте цвет: коричневый
Угадайте цвет: зелёный
Правильный ответ!
```

Как удалить всё содержимое строки - метод **clear**:

```
void str.clear()
```

```
1 std::string s1 = "All right";
2 // Покажет длину в 9 байт
3 print("Длина s1: ", s1.size());
4
5 s1.clear();
6 // Покажет длину равную 0
7 print("Длина s1: ", s1.size());
```

Работа с текстом

Далее будут показаны действия со строкой, затрагивающие позиции конкретных текстовых символов в строке. Каждая позиция символа выражается некоторым числом, начинающимся с нуля (аналогично индексу статических массивов). Эти числа становятся важными, например, при операциях поиска подстрок. И возникает единственный вопрос - как понять, что были просмотрены все символы данной строки?

Сам тип **string** является библиотечным и ему доступны более широкие возможности, по сравнению с базовыми типами. Так, для окончания строки в нём заведена специальная константа с именем **npos**. Её полное название:

```
1 string::npos
```

Значение константы можно узнать:

```
2 print("Число в npos: ", string::npos);
```

но само по себе оно мало чем интересно.

Работа с текстом

Частичное сравнение строк - метод **compare**

```
(1) int str.compare(other_str)
(2) int str.compare(size_t pos, size_t len,
                    other_str)
(3) int str.compare(size_t pos, size_t len,
      other_str, size_t o_pos, size_t o_len)
```

, где **str** - переменная типа **string**, которую сравниваем. А **other_str** - с которой сравниваем.

1. полное сравнение строк **str** и **other_str**
2. сравнение фрагмента внутри **str**, начиная с символа на позиции **pos** и длиной **len**, со строкой **other_str**
3. сравнение фрагментов из **str** и **other_str**. Позиция и длина для второго задаются аргументами **o_pos** и **o_len**

Метод **compare** возвращает **нуль**, если строки илиих фрагменты равны; **число больше нуля**, если str $>$ other_str; **число меньше нуля**, если str $<$ other_str (сравнение - лексикографическое).

Работа с текстом

Для манипуляции с позициями символов используется тип **size_t**

```
1  string s1 = "два отличия найдите",
2         s2 = "найдите кота";
3  size_t nlen = strlen("найдите");
4
5  if (s1.compare(s2) < 0) {
6    print("Вторая строка больше первой\n");
7  }
8
9  size_t end = string::npos,
10        pos1 = s1.length() - nlen,
11        pos2 = 0;
12
13 if (s1.compare(pos1, end, s2, pos2, nlen) == 0) {
14   print("слово 'найдите' есть в обеих строках");
15 }
```

Работа с текстом

Частичное сравнение с «базовыми» строками (символы в двойных кавычках или соответствующий массив типа **char**)

```
(4) int str.compare(base_str)
(5) int str.compare(size_t pos, size_t len,
                    base_str)
(6) int str.compare(size_t pos, size_t len,
                    base_str, size_t base_len)
```

, где **str** - переменная типа **string**, которую сравниваем. А **base_str** - «базовая» строка.

4. полное сравнение строк **str** и **base_str**
5. сравнение фрагмента внутри **str**, начиная с символа на позиции **pos** и длиной **len**, со строкой **base_str**
6. сравнение фрагментов из **str** и **base_str**. Для базовой строки можно указать только длину фрагмента для сравнения через аргумент **base_len**

Частичное сравнение со строками языка C

```
1  string s1 = "два отличия найдите",
2  size_t nlen = strlen("найдите");
3
4  size_t end = string::npos,
5         pos1 = s1.size() - nlen;
6
7  char base_str[] = "найдите что-нибудь";
8
9  if (s1.compare(pos1, end, base_str, nlen) == 0) {
10   print("Строки равны только по "
11         "слову 'найдите' на "
12         "соответствующих позициях\n");
13 }
```

Работа с текстом

Вставка на указанную позицию - метод **insert**

```
(1) str.insert(size_t pos, other_str)
(2) str.insert(size_t pos, other_str,
               size_t o_pos, size_t o_len)
(3) str.insert(size_t pos, base_str)
(4) str.insert(size_t pos, base_str, size_t base_len)
(5) str.insert(size_t pos, size_t count, char sym)
```

1. Вставляет строку **other_str** в **str** сразу **перед** номером символа, заданного аргументом **pos**
2. Вставляет фрагмент из **other_str**, длиной **o_len** и начиная с символа **o_pos** в **str**
3. Вставляет строку **base_str** в **str** перед символом с номером **pos**.
4. Вставляет фрагмент строки **base_str**, длиною **base_len**, в **str** перед символом с номером **pos**.
5. Вставляет в строку **str** символ **sym** в количестве **count** штук перед символом с номером **pos**.

Вставка на указанную позицию - метод **insert**

```
1 string s1 = "Что дела?";
2 size_t w_len = strlen("Что");
3
4 s1.insert(w_len + 1, "за ");
5
6 // Напечатает "Что за дела?"
7 print(s1, "\n");
```

Работа с текстом

Преобразование в базовую строку (символы в двойных кавычках или соответствующий массив типа **char**) - метод **c_str**

```
char* str.c_str()
```

Метод вернёт массив (точнее - *указатель на массив*), который является корректной базовой строкой.

```
1 #include <cstring> // *strcmp*
2
3 string s1 = "Странное сообщение";
4 char c_str[] = "и не говори";
5
6 if ( strcmp(c_str, s1.c_str()) == 0 ) {
7   print("Такого не может быть\n");
8 }
```

Работа с текстом

Выделение фрагмента строки - метод **substr**

```
string str.substr(size_t start,
                  size_t len = string::npos)
```

start - переменная типа **size_t**, указывающая позицию первого символа подстроки. **len** - количество символов для извлечения.

```
1  string s1 = "Phase transitions are "
2               "great part of physics";
3  size_t len1 = strlen("Phase transitions are "),
4         len2 = strlen("great");
5
6  string s2 = s1.substr(len1, len2);
7  // Печатаем: "great"
8  print(s2, "\n");
9
10 string s3 = s1.substr(len1 + len2 + 1);
11 // Печатаем: "part of physics"
12 print(s3, "\n");
```

Работа с текстом

Поиск в строке - методы **find** и **rfind** (*reverse find*)

```
size_t str.find(frag, size_t pos = 0)
size_t str.rfind(frag, size_t rpos = npos)
```

- **frag** - текстовый фрагмент, который ищется в строке **str**
- Возращает позицию первого символа из **frag** в строке **str**, если **frag** присутствует в **str**
- **frag** может быть переменной типа **string**, базовой строкой, и переменной типа **char**
- Поиск начинается с позиции, определяемой вторым аргументом
- **rfind** - поиск с конца строки (просмотр символов идёт справа налево)
- если фрагмента не было найдено, методы вернут значение **string::npos**

Поиск в строке: базовый пример

```
1  string s1 = "Сопротивление обратно ↩
       пропорционально силе тока";
2  size_t found_pos = s1.find("обр");
3
4  if ( found_pos != std::string::npos ) {
5    print("\"обр\" в строке находится на ",
6          found_pos, " позиции\n");
7  }
8
9  found_pos = s1.find("ТЧК", 6);
10 if ( found_pos == string::npos ) {
11   print("\"ТЧК\" в исходной строке "
12         "не обнаружена\n");
13 }
```

Работа с текстом

Пример: поиск всех вхождений символа в строку

```
1  string text = "Да, были люди в наше время,\n"
2                "Не то, что нынешнее племя:\n"
3                "Богатыри — не вы!\n"
4                "Плохая им досталась доля:\n"
5                "Немногие вернулись с поля...\n"
6                "Не будь на то господня воля,\n"
7                "Не отдали б Москвы!\n";
8  print("Ищем все запятые в тексте\n", text);
9  print("\n--------------------------------\n");
10
11 size_t comma_pos = text.find(',');
12 while ( comma_pos != string::npos ) {
13   size_t place = comma_pos + 1;
14   print("',' найдена на ", place, " месте\n");
15   // Поиск продолжается с первого символа,
16   // идущего после ','
17   comma_pos = text.find(',', place);
18 }
```

Работа с текстом

Замена части текста в строке - метод **replace**

```
(1) str.replace(size_t pos, size_t len, other_str)
(2) str.replace(size_t pos, size_t len, other_str,
                size_t o_pos, size_t o_len)
(3) str.replace(size_t pos, size_t len, base_str)
(4) str.replace(size_t pos, size_t len, base_str,
                size_t base_len)
(5) str.replace(size_t pos, size_t len, size_t count,
                char sym)
```

1. В строке **str** символы в количестве **len** штук (сколько символов из исходной строки удаляем), начиная с номера **pos**, заменяются на строку **other_str**
2. Аналогично, но замена происходит на фрагмент из **other_str**, длиной **o_len** и начиная с символа **o_pos**
3. Замена нужного количества символов на строку **base_str**
4. Замена нужного количества символов на **base_len** символов из строки **base_str**
5. Замена происходит на символ **sym** в количестве **count**

Замена части текста в строке

```
1 string str = "сегодня будет абракадабра!";
2 size_t w_len = strlen("абракадабра!");
3
4 str.replace(str.size() - w_len,
5               w_len, "всё хорошо");
6 print("И у нас ", str, "\n");
```

Преобразование строк в числа возможно с помощью функций из библиотеки <**string**>

```
(1) int    std::stoi(  str )
(2) size_t std::stoul( str )
(3) double std::stod(  str )
```

Данные три функции пытаются преобразовать переданную им строку в соответствующее числовое значение (целое со знаком, целое без знака, действительное число). Если преобразование не удалось - строка содержала некорректное число - то происходит ошибка времени выполнения.

Преобразование строк в числа, пример

```
1 string  s_double = "456.7788",
2         s_int    = "-7485",
3         s_size_t = "313377317135";
4
5 double d_num  = stod(  s_double );
6 int    i_num  = stoi(  s_int );
7 size_t sz_num = stoul( s_size_t );
8
9 print(d_num, " ", i_num, " ", sz_num, "\n");
```

Работа с текстом

Обратное преобразование числа в строку можно сделать с помощью функции **to_string** (<**string**>)

```
string std::to_string( number_value );
```

Данная функция принимает число (целое, действительное) и возвращает его представление в виде строки. Для действительных чисел число знаков после запятой **ограничено шестью**.

```
1  string s1;
2
3  double real_num = 456.3247899;
4  int integer     = -899;
5  size_t natural  = 13340089;
6
7  s1 += to_string(real_num) + "##";
8  s1 += to_string(integer) + "##";
9  s1 += to_string(natural);
10
11 print("Итоговая строка: ", s1, "\n");
```

Работа с текстом

```
1  void process_str1(string, string );
2  void process_str2(string&, string&);
3  void process_str3(const string&, const string& );
4
5  string s_one = "строка", s_two = "опять строка";
6  // Здесь всегда происходит копирование
7  process_str1(s_one, s_two);
8  // Передача по ссылке => нет копирования
9  // Но строки могут внутри функции изменяться
10 process_str2(s_one, s_two);
11 // Передача по неизменяемой ссылке
12 process_str3(s_one, s_two);
```

Передача переменных типа string в функции

Передача таких переменных должны происходить по **ссылке**. В случае передачи по значению, каждый аргумент типа **string** будет вызывать копирование всего текста, который содержит передаваемая переменная.

Работа с текстом. Резюме

- Работа с текстом становится гораздо удобнее с библиотекой <**string**>
- Базовый строки - просто продвинутые массивы типа **char**
- По сути, в любом случае строка хранится как массив байт
- Тип **string** позволяет свести многие операции со строкой к манипуляции позициями фрагментов строк
- В функции строки лучше всего передавать в виде ссылки (константной/неконстантной) во избежаниия ненужного копирования
- Больше полезных методов для типа **string** можно найти тут: http://www.cplusplus.com/reference/string/

Адреса, переменные и память
Углублённое расследование

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 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**

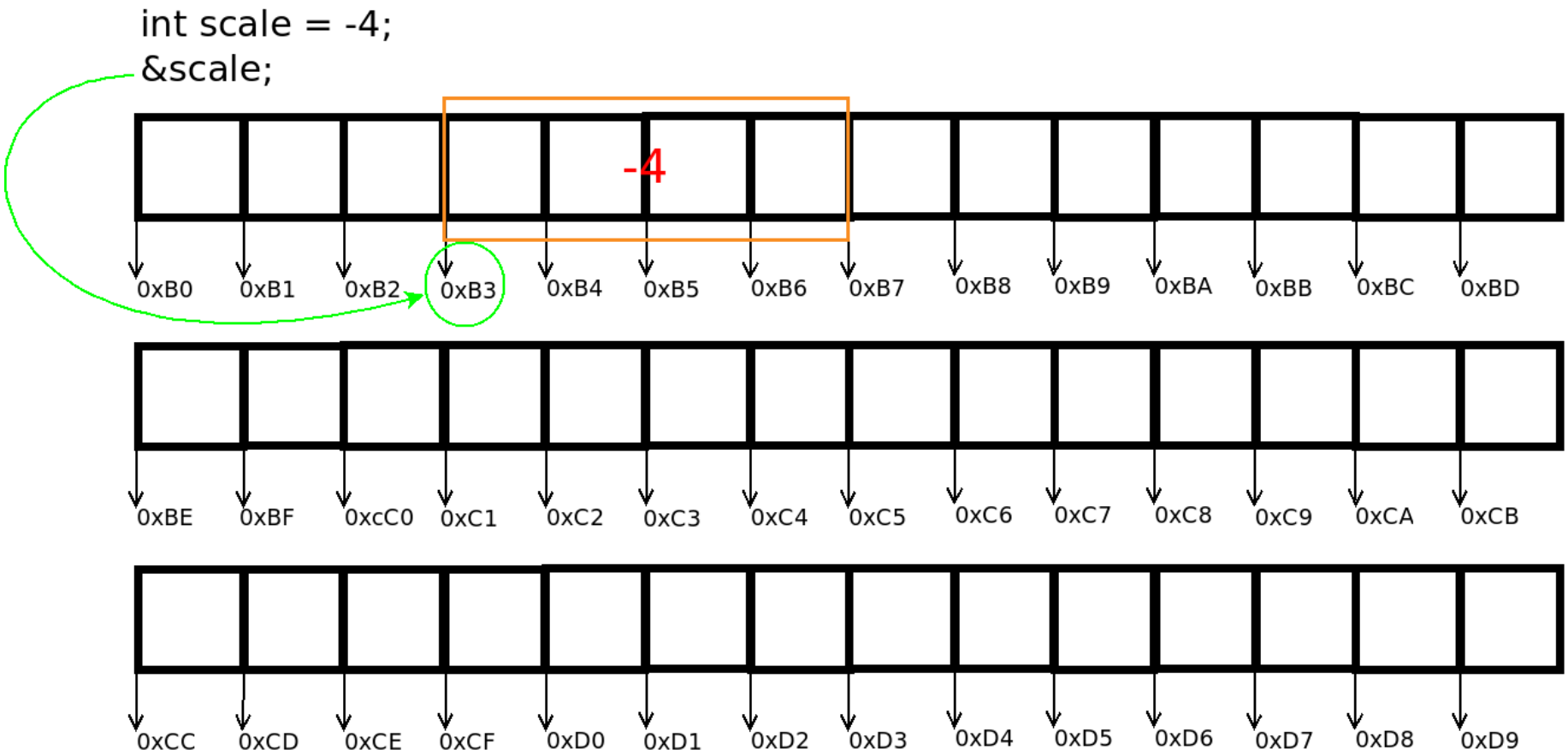
```
1 int scale = -4;
2 double rate = 2.64;
3
4 print("Размер int    : ", sizeof(int), "\n");
5 print("Размер double: ", sizeof(rate), "\n");
6
7 string str = "Тестируем и строку";
8 print("Размер string: ", sizeof(str), "(!?)\n");
```

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 (то есть, обыкновенных переменных простых, 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
```

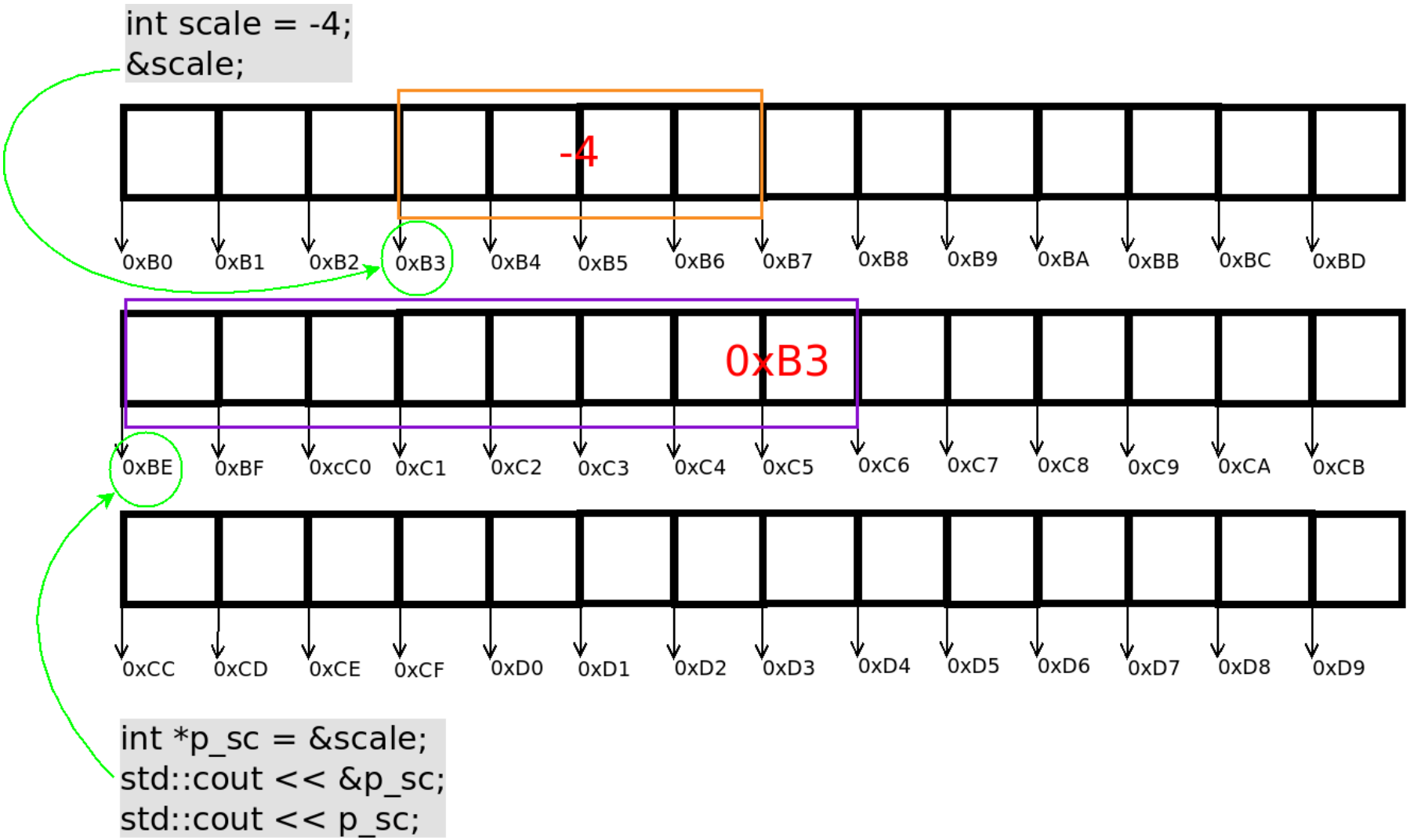
Операции с указателями: **присвоение значения**
Указателю может быть присвоено значение (адрес в памяти) либо с помощью операции взятия адреса у переменной, либо копированием значения из другого указателя.

```
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.



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 int *p2 = nullptr;
2
3 // Операции с p2 ...
4 if (p2 != nullptr) {
5   // что-нибудь делаем
6 }
```

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

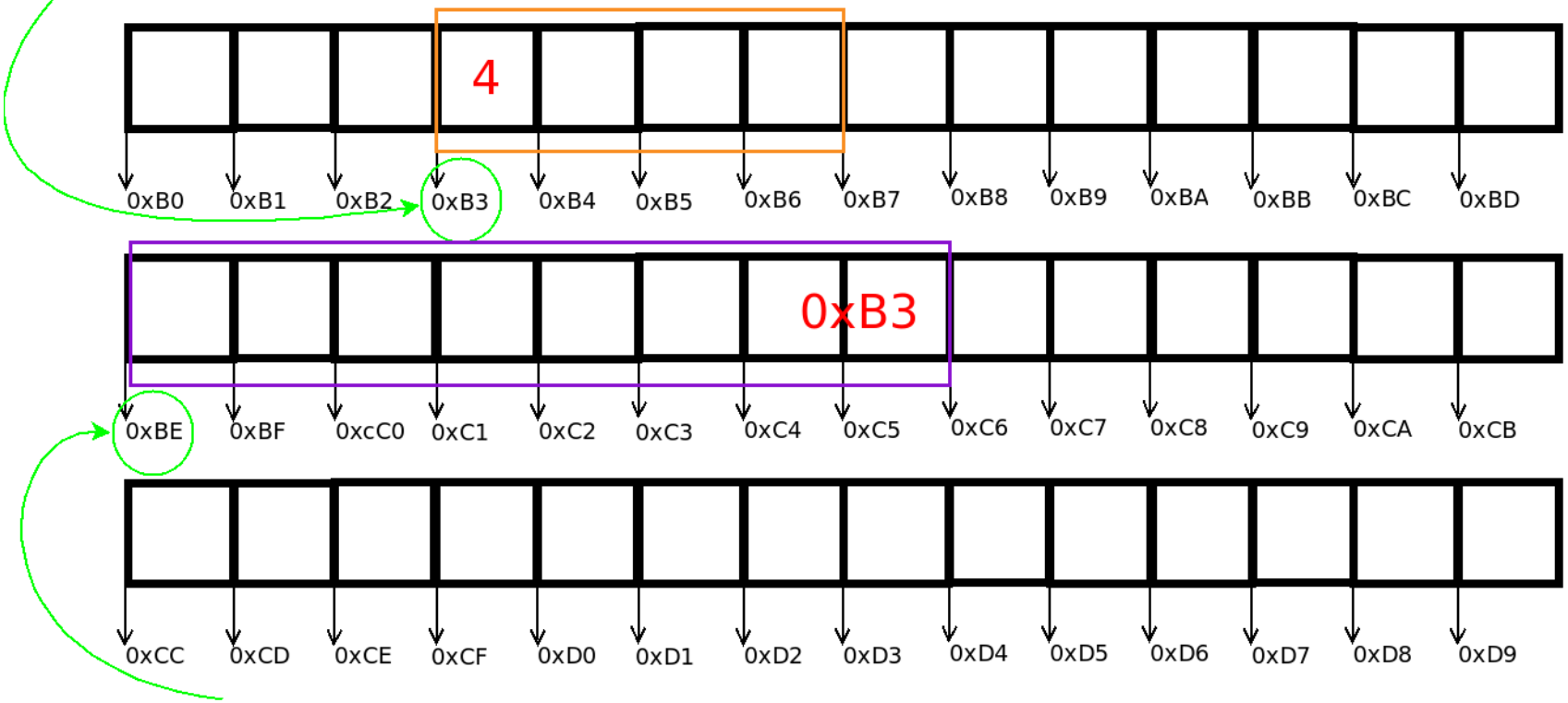
Операции с указателями: **разыменование** - получение значения переменной, адрес которой сохранён в указателе.

`*<имя_переменной_указателя>;`

```
1  int scale = 5;
2  int *p_sc = &scale;
3
4  // Вывод 5 на экран
5  print("Значение, на которое ссылается ",
6        "p_sc: ", *p_sc, "\n");
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;
14 // Печатаем 15
15 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 **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");
```

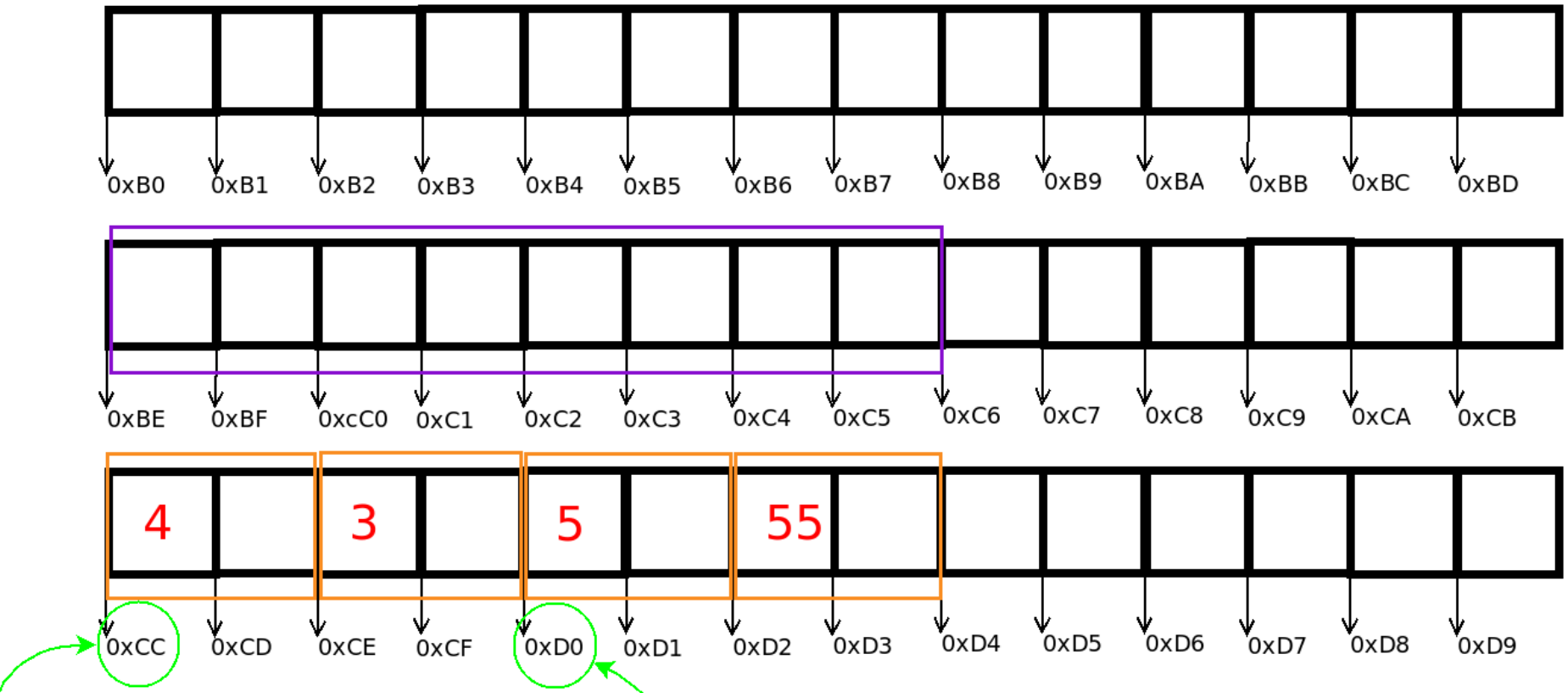
Операции с указателями: **разыменование**

```
1  void old_swap(int *i1, int *i2)
2  { // Как делать в стиле языка C, не C++
3    if (i1 != nullptr and i2 != nullptr) {
4      int tmp = *i1; *i1 = *i2; *i2 = tmp;
5    }
6  }
7
8  int n1 = 15, n2 = 103;
9  old_swap(&n1, &n2);
10
11 print("n1 = ", n1, "; n2 = ", n2, "\n");
```

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

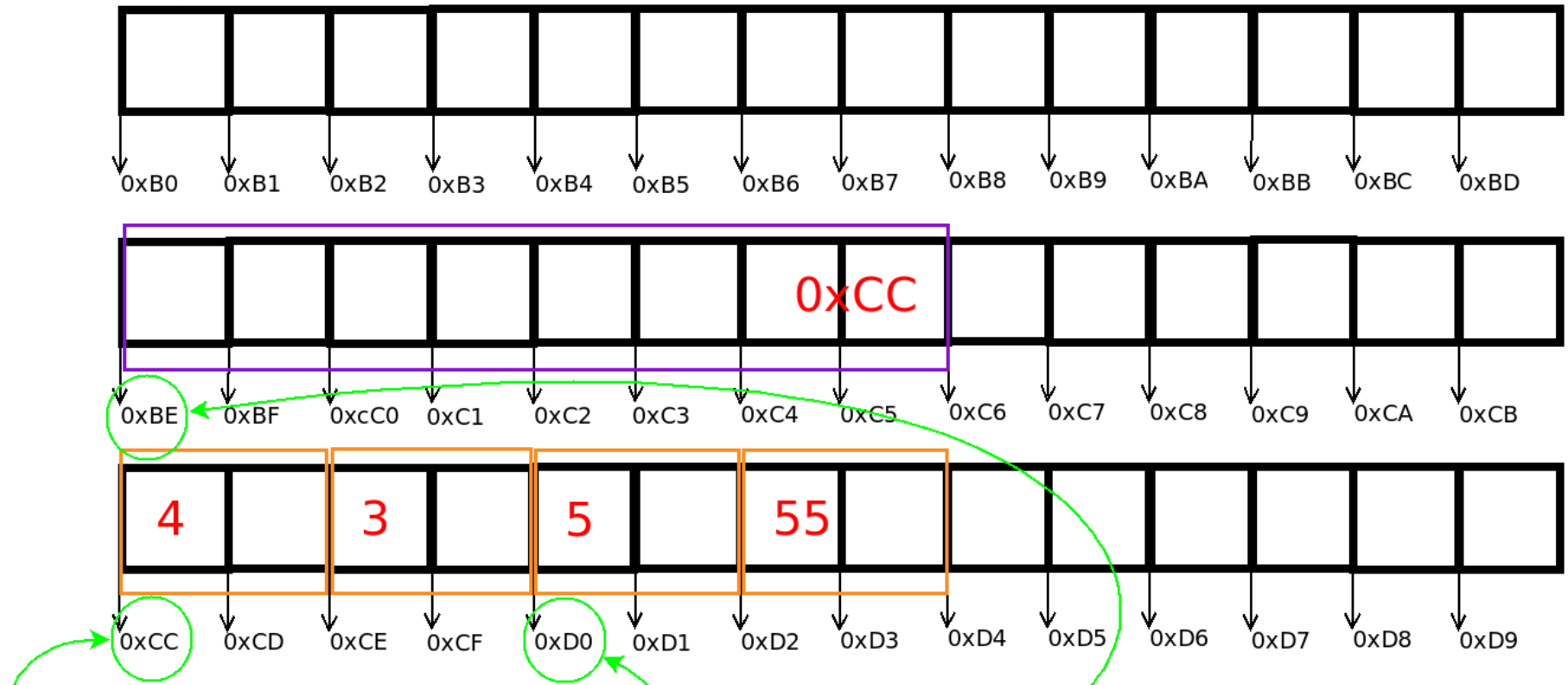
Связь указателей и массивов

- Имя переменной-массива (выше - **vec**) является указателем на его первый элемент
- Массивы, 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 предыдущей лекции
2 void print_array(int* arr, size_t count);
3 ...
4 int vec[4] = {4, 3, 5, 55};
5
6 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];
```

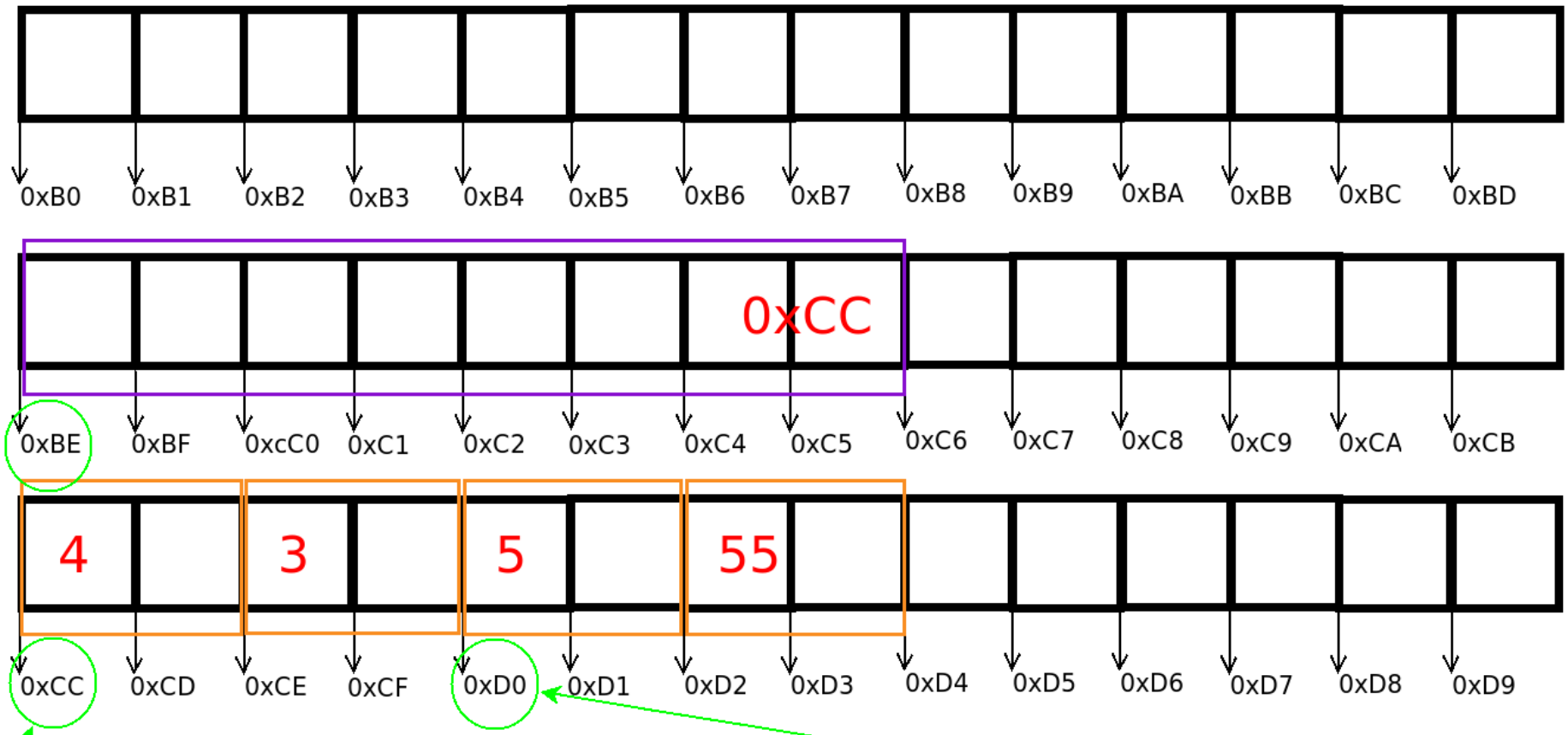
```
short *p_vec = vec;
std::cout << *p_vec;
```

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).

Сложение указателя с целым числом графически




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

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

Указатели и массивы

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;
```

Операции с указателями: **индексация**

Индексация указателя выполняет два действия:

1. Сместиться на количество блоков, равных индексу, от текущего адреса
2. Получить значение из блока 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 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  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");
```