

Ravesli [Ravesli](#)


- [Уроки по C++](#)
- [OpenGL](#)
- [SFML](#)
- [Qt5](#)
- [RegExr](#)
- [Ассемблер](#)
- [Купить .PDF](#)


Урок №83. Адресная арифметика и индексация массивов

 [Юрий](#) |

- [Уроки C++](#)

|

 Обновл. 15 Сен 2020 |

 33505

[1](#)  [11](#)

Язык C++ позволяет выполнять целочисленные операции сложения/вычитания с [указателями](#). Если `ptr` указывает на целое число, то `ptr + 1` является адресом следующего целочисленного значения в памяти после `ptr`. `ptr - 1` — это адрес предыдущего целочисленного значения (перед `ptr`).

Оглавление:

1. [Адресная арифметика](#)
2. [Расположение элементов массива в памяти](#)
3. [Индексация массивов](#)
4. [Использование указателя для итерации по массиву](#)

Адресная арифметика

Обратите внимание, `ptr + 1` не возвращает следующий *любой адрес памяти*, который находится сразу после `ptr`, но он возвращает *адрес памяти следующего объекта, тип которого совпадает* с типом значения, на которое указывает `ptr`. Если `ptr` указывает на адрес памяти целочисленного значения (размер которого 4 байта), то `ptr + 3` будет возвращать адрес памяти третьего целочисленного значения после `ptr`. Если `ptr` указывает на адрес памяти значения [типа `char`](#), то `ptr + 3` будет возвращать адрес памяти третьего значения типа `char` после `ptr`.

При вычислении результата выражения **адресной арифметики** (или «**арифметики с указателями**») компилятор всегда умножает целочисленный операнд на размер объекта, на который указывает указатель. Например:

```
1 | #include <iostream>
```

```
2
3 int main()
4 {
5     int value = 8;
6     int *ptr = &value;
7
8     std::cout << ptr << '\n';
9     std::cout << ptr+1 << '\n';
10    std::cout << ptr+2 << '\n';
11    std::cout << ptr+3 << '\n';
12
13    return 0;
14 }
```

Результат на моем компьютере:

```
002CF9A4
002CF9A8
002CF9AC
002CF9B0
```

Как вы можете видеть, каждый последующий адрес увеличивается на 4. Это связано с тем, что размер типа `int` на моем компьютере составляет 4 байта.

Та же программа, но с использованием типа `short` вместо типа `int`:

```
1 #include <iostream>
2
3 int main()
4 {
5     short value = 8;
6     short *ptr = &value;
7
8     std::cout << ptr << '\n';
9     std::cout << ptr+1 << '\n';
10    std::cout << ptr+2 << '\n';
11    std::cout << ptr+3 << '\n';
12
13    return 0;
14 }
```

Результат:

```
002BFA20
002BFA22
002BFA24
002BFA26
```

Поскольку тип `short` занимает 2 байта, то каждый следующий адрес больше предыдущего на 2.

Расположение элементов массива в памяти

Используя оператор адреса `&`, мы можем легко определить, что элементы массива расположены в памяти последовательно. То есть, элементы 0, 1, 2 и т.д. размещены рядом (друг за другом):

```
1 #include <iostream>
2
3 int main()
4 {
5     int array[] = { 7, 8, 2, 4, 5 };
6
7     std::cout << "Element 0 is at address: " << &array[0] << '\n';
8     std::cout << "Element 1 is at address: " << &array[1] << '\n';
9     std::cout << "Element 2 is at address: " << &array[2] << '\n';
10    std::cout << "Element 3 is at address: " << &array[3] << '\n';
11
12    return 0;
13 }
```

Результат на моем компьютере:

```
Element 0 is at address: 002CF6F4
Element 1 is at address: 002CF6F8
Element 2 is at address: 002CF6FC
Element 3 is at address: 002CF700
```

Обратите внимание, каждый из этих адресов по отдельности занимает 4 байта, как и размер типа `int` на моем компьютере.

Индексация массивов

Мы уже знаем, что элементы массива расположены в памяти последовательно. Из [урока №82](#) мы знаем, что **фиксированный массив** может распасться на указатель, который указывает на первый элемент (элемент под индексом 0) массива.

Также мы уже знаем, что добавление единицы к указателю возвращает адрес памяти следующего объекта этого же типа данных.

Следовательно, можно предположить, что добавление единицы к идентификатору массива приведет к возврату адреса памяти второго элемента (элемента под индексом 1) массива. Проверим на практике:

```
1 #include <iostream>
2
3 int main()
4 {
5     int array [5] = { 7, 8, 2, 4, 5 };
6 }
```

```
7 | std::cout << &array[1] << '\n'; // выведется адрес памяти элемента под индексом 1
8 | std::cout << array+1 << '\n'; // выведется адрес памяти указателя на массив + 1
9 |
10 | std::cout << array[1] << '\n'; // выведется 8
11 | std::cout << *(array+1) << '\n'; // выведется 8 (обратите внимание на скобки, они
12 |
13 | return 0;
14 | }
```

При разыменовании результата выражения адресной арифметики скобки необходимы для соблюдения **приоритета операций**, поскольку оператор `*` имеет более высокий приоритет, чем оператор `+`.

Результат выполнения программы на моем компьютере:

001AFE74

001AFE74

8

8

Оказывается, когда компилятор видит оператор индекса `[]`, он, на самом деле, конвертирует его в указатель с операцией сложения и разыменования! То есть, `array[n]` — это то же самое, что и `*(array + n)`, где `n` является целочисленным значением. Оператор индекса `[]` используется в целях удобства, чтобы не нужно было всегда помнить о скобках.

Использование указателя для итерации по массиву

Мы можем использовать указатели и адресную арифметику для выполнения итераций по массиву. Хотя обычно это не делается (использование оператора индекса, как правило, читабельнее и менее подвержено ошибкам), следующий пример показывает, что это возможно:

```
1 | #include <iostream>
2 |
3 | int main()
4 | {
5 |     const int arrayLength = 9;
6 |     char name[arrayLength] = "Jonathan";
7 |     int numVowels(0);
8 |     for (char *ptr = name; ptr < name + arrayLength; ++ptr)
9 |     {
10 |         switch (*ptr)
11 |         {
12 |             case 'A':
13 |             case 'a':
14 |             case 'E':
15 |             case 'e':
16 |             case 'I':
17 |             case 'i':
```

```
18     case '0':
19     case 'o':
20     case 'U':
21     case 'u':
22         ++numVowels;
23     }
24 }
25
26 std::cout << name << " has " << numVowels << " vowels.\n";
27
28 return 0;
29 }
```

Как это работает? Программа использует указатель для *прогона* каждого элемента массива поочередно. Помните, что массив распадается в указатель на первый элемент массива? Поэтому, присвоив `name` для `ptr`, сам `ptr` стал указывать на первый элемент массива. Каждый элемент разыменовывается с помощью выражения `switch`, и, если текущий элемент массива является гласной буквой, то `numVowels` увеличивается. Для перемещения указателя на следующий символ (элемент) массива в цикле `for` используется оператор `++`. Работа цикла завершится, когда все символы будут проверены.

Результат выполнения программы:

Jonathan has 3 vowels.

Оценить статью:

★★★★★ (243 оценок, среднее: 4,87 из 5)



[← Урок №82. Указатели и массивы](#)



[Урок №84. Символьные константы строк C-style](#) →

Комментариев: 11



1. [zashiki:](#)

[22 октября 2019 в 15:25](#)

разыменованный указатель, по-видимому, можно представлять с индексом, как элемент массива? т.е. можно так?:

```
1 int x=2,y=4;
2 int *ptr=&x;
3 std::cout<<ptr[0]<<"\n"<<ptr[1];
```

и это будет равнозначно?

```
1 | std::cout<<*ptr<<"\n"<<*(ptr+1);
```

[Ответить](#)



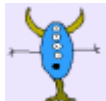
1. Евгений:

[13 февраля 2020 в 21:17](#)

Нет, не будет. Проверяем:

```
1 | int x = 2, y = 4;
2 |     int *ptr = &x;
3 |     std::cout << ptr[0] << '\n' << ptr[1] << '\n';
4 |     // check
5 |     int *px = &x;           // 0x7ffee3e2b9bc
6 |     int *py = &y;           // 0x7ffee3e2b9b8
7 |     int *px2 = &px[0];      // 0x7ffee3e2b9bc
8 |     int *py2 = &px[1];      // 0x7ffee3e2b9c0
9 |     int *px3 = &*(ptr);     // 0x7ffee3e2b9bc
10 |    int *py3 = &*(ptr+1);    // 0x7ffee3e2b9c0
11 |    std::cout << "px..." << px << "\npy..." << py << "\npx2.." << px2 << "\npy2.." << py2 << "\npx3.." << px3 << "\npy3.." << py3 << '\n';
12 |
```

[Ответить](#)



2. zashiki:

[9 сентября 2019 в 12:42](#)

Объясните:

1. почему

`int *pointer=&y` выводит адрес y(при `cout<<pointer;`)

а `char *pointer=&y` выводит сам символ??(при `cout<<pointer;`). Почему не адрес?

2. как образуются 16-ричные адреса, почему они зависят от размера переменной? т.е. почему если друг за другом идут две 4-байтные переменные, то адреса их различаются на 4, а не просто по порядку +1?

сам механизм хочется понять, гугл не помогает.

[Ответить](#)



1. zashiki:

[9 сентября 2019 в 15:12](#)

короче, выяснилось

1й вопрос: ответ в следующем уроке

2й вопрос: адрес по сути — это адрес 1 байта. 4-х байтная переменная занимает 4 адреса друг за другом. А когда называем ее адрес — это адрес первого ее байта. Поправьте, если не так.

[Ответить](#)



1. *Наталья:*

[3 октября 2020 в 19:25](#)

Именно так. В оперативной памяти нумеруется каждый байт. Если переменная занимает 4 байта, адрес следующей будет больше на 4.

[Ответить](#)



3. *Валерий:*

[9 апреля 2019 в 14:24](#)

Вы уверены, что последний пример будет работать одозначно? Здесь недостаточно "Хотя обычно это не делается". Мало того, этот код может сносно работать при текущей архитектуре. Но переход к другой архитектуре вызовет очень много проблем. Хотел бы я посмотреть на производительность этого цикла например с адресами типа `far` из времен древнего ДОСа.

Но даже это еще не все. Если мне не изменяет память, конструкция `"name + arrayLength"` уже выходит за границы объекта. Что нам говорит по этому поводу стандарт? Результат не определен.

Сравнение указателей гарантировано работает только внутри одного объекта. А дальше — как бог на душу положит.

[Ответить](#)



1. *Алексей:*

[31 мая 2019 в 16:52](#)

все верно, выходит. Только вот в цикле стоит `<` значит сравнение указателей корректно. А в остальном вы, пожалуй, правы

[Ответить](#)

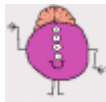


2. *Наталья:*

[27 сентября 2020 в 17:06](#)

Сравнение указателя с адресом, следующим сразу за последним элементом массива, гарантировано стандартом. Но только сравнение, а не обращение по такому адресу.

[Ответить](#)



4. *Алексей:*

[28 февраля 2019 в 15:39](#)

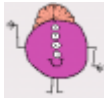
Спасибо за статью!

[Ответить](#)

1. Юрий:

[28 февраля 2019 в 15:46](#)

Пожалуйста.

[Ответить](#)

1. Алексей:

[28 февраля 2019 в 16:13](#)

А вот интересный момент, допустим есть указатель: `int32_t* pntr`, указывающий на определенный адрес и мы к нему прибавим, например, `+next`, который `int8_t next = 1`. Таким образом `pntr` должен всё также сдвинуться на 4 байта.

[Ответить](#)

Добавить комментарий






Ваш E-mail не будет опубликован. Обязательные поля помечены *

Имя * Email *

Комментарий

☐ Сохранить моё Имя и E-mail. Видеть комментарии, отправленные на модерацию☐ Получать уведомления о новых комментариях по электронной почте. Вы можете [подписаться](#) без комментирования.[TELEGRAM](#)  [КАНАЛ](#)[ПАБЛИК](#) 

ТОП СТАТЬИ

-  [Словарь программиста. Сленг, который должен знать каждый кодер](#)
-  [Урок №1. Введение в программирование](#)
-  [70+ бесплатных ресурсов для изучения программирования](#)
-  [Урок №1: Введение в создание игры «SameGame» на C++/MFC](#)
-  [Урок №4. Установка IDE \(Интегрированной Среды Разработки\)](#)

- [Ravesli](#)
- - [О проекте/Контакты](#) -
- - [Пользовательское Соглашение](#) -
- - [Все статьи](#) -
- Copyright © 2015 - 2020