



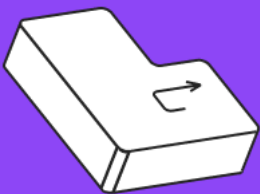
# Урок 7

## Указатели

## Массивы



Программирование на языке Си  
(базовый уровень)



# Оглавление

<b>Введение</b>	<b>2</b>
<b>Термины, используемые в лекции</b>	<b>3</b>
<b>Указатель</b>	<b>4</b>
Определение указателя	5
Получение адреса данных	6
Получение значения по адресу	8
Основные операции над указателями	9
Указатели на указатели	11
Указатели как параметры функции	12
Адресная арифметика	14
Особенности арифметических операций с указателями	17
<b>Массивы</b>	<b>19</b>
Что неправильно?	20
<b>Массивы и указатели</b>	<b>21</b>
Передача массива в функцию	21
Генератор случайных чисел массива	23
Манипуляции с массивами	25
Циклически сдвинуть массив влево на 1 элемент.	25
Сделать реверс массива	26
Отсортировать массив по возрастанию пузырьком	26
Отсортировать массив по возрастанию выбором	26
<b>Операции над указателями массива</b>	<b>27</b>
Что будет напечатано?	28
<b>Вычисление корней квадратного уравнения (массивы)</b>	<b>28</b>
<b>Подведение итогов</b>	<b>32</b>
<b>Дополнительные материалы</b>	<b>33</b>
<b>Используемые источники</b>	<b>34</b>

## Введение

На предыдущей лекции вы узнали:

- Что такое рекурсия, её достоинства и недостатки
- Какие бывают рекурсии

- Как происходит замена цикла на рекурсию и наоборот
- Практические примеры рекурсий
- Как размещаются в памяти вещественные числа
- Разобрали примеры

На этой лекции вы найдете ответы на такие вопросы как / узнаете:

- Что такое указатель
- Как получить адрес переменной и значение по адресу
- Как передать аргумент по указателю в функцию
- Что такое адресная арифметика
- Что такое массив и как с ним работать
- Разберем примеры

## Термины, используемые в лекции

**Указатель** — это переменная значением которой является адрес.

**Рекурсивная функция** — вызов функции из нее же самой.

**Итерация** — организация обработки данных, при которой действия повторяются многократно, не приводя при этом к вызовам самих себя (в отличие от рекурсии).

**Разыменование** — Получение значения по указателю минуя идентификатор переменной

**Массив** — это последовательность однотипных элементов имеющих общее имя.

**Адрес элемента** — адрес начальной ячейки памяти, в которой расположен этот элемент.

**Индекс элемента** — порядковый номер элемента в массиве.

**Значение элемента массива** — значение, хранящееся в определенной ячейке памяти, расположенной в пределах массива.

**Размер массива** — количество элементов массива.

**Размер элемента** — количество байт, занимаемых одним элементом массива.

# Указатель

Для управления памятью в языке Си используют указатели. Их знание, понимание и умелое использование помогает составлению хороших программ.

**Указатель** — это переменная значением которой является адрес другой переменной.

Для чего нужны указатели?

1. Мы уже говорили о том, что данные в языке Си передаются только по значению. Изменять же значение передаваемых в функцию параметров помогают указатели.
2. Также при помощи указателей происходит динамическое выделение и перераспределение памяти.
3. Позволяют повысить эффективность многих процедур.
4. Поддерживают динамические структуры данных (например, связанные списки, двоичные деревья)

Однако, при работе с указателями стоит быть очень внимательными. Достаточно легко ошибиться при использовании указателей, защита же от таких ошибок в Си невелика. Неправильные ссылки или неверные данные в свою очередь приведут к ошибкам в программе или ее краху. Ошибки, в значении указателей довольно сложно обнаружить.

Таким образом, указатели являются мощным инструментом языка СИ. Они позволяют напрямую обращаться к адресам в которых хранятся переменные. Что дает возможность обращаться к этим адресам и оперировать исходными данными.

Правильное понимание и использование указателей особенно необходимо для составления хороших программ на языке С.

И вот почему:

1. Указатели являются средством, с помощью которого функция может изменять значения передаваемых в нее аргументов.
2. С помощью указателей выполняется динамическое распределение памяти.
3. Указатели позволяют повысить эффективность многих процедур.
4. Обеспечивают поддержку динамических структур данных, таких, например, как двоичные деревья и связанные списки.

Таким образом, указатели являются весьма мощным средством языка C. Но и весьма опасным. Например, если указатель содержит неправильное значение, программа может потерпеть крах. Указатели весьма опасны еще и потому, что легко ошибиться при их использовании. К тому же ошибки, связанные с неправильными значениями указателей, найти очень трудно.

Все определенные в программе данные, например, переменные, хранятся в памяти по определенному адресу. И указатели позволяют напрямую обращаться к этим адресам и благодаря этому манипулировать данными.

## Определение указателя

Чтобы задать указатель необходимо описать тип данных объекта, далее звездочка “\*” и имя переменной-указателя.

```
тип_данных*имя_указателя;
```

Например, так будет выглядеть указатель на объект с типом int.

```
int*pointer;
```

Для работы с указателем необходимо его проинициализировать, т.е. присвоить ему адрес переменной. Указатель не ссылается на объект пока он не проинициализирован.



**Внимание!** Указатель должен быть проинициализирован.

При работе с указателями применяется косвенная адресация, т.е. в переменной-указателе лежит адрес другой переменной, на которую ссылается указатель.

Для определения указателя надо указать тип объекта, на который указывает указатель, и символ звездочки \*.

```
тип_данных* название_указателя;
```

Сначала идет тип данных, на который указывает указатель, и символ звездочки \*. Затем имя указателя.

Например, указатель на объект типа int записывается следующим образом:

```
int *pointer;
```

Пока указатель не инициализирован, он не ссылается ни на какой объект. Для работы с ним нужно присвоить ему адрес переменной:

```
int main(void)
{
    int x = 10;           // определяем переменную
    int *p_x;             // определяем указатель
    p_x = &x;             // указатель получает адрес переменной
    return 0;
}
```

## Получение адреса данных

При упрощенном рассмотрении устройства памяти

Если рассматривать схему организации памяти упрощенно, то память компьютера представляет собой массив последовательно пронумерованных ячеек, каждая из которых имеет свой адрес.

С ячейками памяти можно работать по отдельности или связными кусками: один байт может хранить значение типа `char`, двухбайтовые ячейки могут рассматриваться как целое типа `short`, а четырехбайтные — как целые типа `long` и т.д.

И для получения адреса к переменной применяется операция `&`. Эта операция применяется только к таким объектам, которые хранятся в памяти компьютера, то есть к переменным и элементам массива, например к макросам она не приемлема.

Что важно, переменная `x` имеет тип `int`, и указатель, который указывает на ее адрес тоже имеет тип `int`.

```
char c = 'N';
char *pc = &c;
int *pd = (int *)pc;
```



**Внимание!** Указатель и переменная должны иметь тот же тип

Допустимо явное преобразование типов.

Для вывода значения указателя можно использовать специальный спецификатор %p:

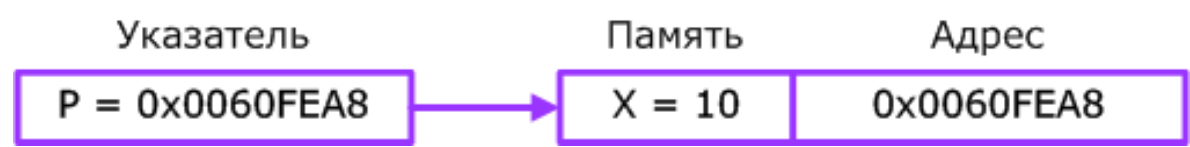
Для хранения адресов в памяти применяется шестнадцатеричная система. Адрес представляет целочисленное значение которое в зависимости от разрядности операционной системы занимает 32 или 64 бита. В микроконтроллерах и IoT может быть 8 и 16 бит.

Рассмотрим пример, где адрес переменной x равен 0x0060FEA8.

```
#include <stdio.h>
int main(void)
{
    int x = 10;
    int *p_x;
    p_x = &x;
    printf("%p \n", p_x);    // 0060FEA8
    return 0;
}
```

Допустим адрес переменной x будет равен 0x0060FEA8. (Для адресов в памяти применяется шестнадцатеричная система.)

Фактически адрес представляет целочисленное значение, выраженное в шестнадцатеричном формате 32 или 64 бита в зависимости от разрядности операционной системы. В микроконтроллерах и IoT может быть 8 и 16 бит.



Так как переменная x представляет тип int, то на большинстве архитектур она будет занимать следующие 4 байта (в IoT может быть и меньше, но не менее 2-х байт). Таким образом, переменная типа int последовательно займет ячейки памяти с адресами 0x0060FEA8, 0x0060FEA9, 0x0060FEAA, 0x0060FEAB.

Объект типа int занимает 4 байта

60FEA8	60FEA9	60FEAA	60FEAB
1 байт	1 байт	1 байт	1 байт
10			

переменная x

объект типа int занимает 4 байта

60FEA8	60FEA9	60FEAA	60FEAB
1 байт	1 байт	1 байт	1 байт
10			
переменная x			

И указатель p\_x будет ссылаться на адрес, по которому располагается переменная x, то есть на адрес 0x0060FEA8.

## Получение значения по адресу

Так как указатель хранит адрес переменной x, то мы можем по этому адресу считать хранящееся там значение, то есть значение переменной x. Для этого применяется операция \* или операция разыменования (dereference operator). Результатом этой операции всегда является сам объект, на который указывает указатель. Применим данную операцию и получим значение переменной x:

```
#include <stdio.h>

int main(void)
{
    int x = 10;
    int *p_x;
    p = &x;
    // При выводе адреса указателя функция printf() ожидает,
    // что указатель будет представлять void*
    // поэтому переданный указатель нужно преобразовать
    // в указатель типа void *:
    printf("Address = %p \n", (void*) p);
    printf("x = %d \n", *p);
    return 0;
}
```

Используя полученное значение в результате операции разыменования мы можем присвоить его другой переменной.



```
int x = 10;
int *p = &x;
int y = *p;      // присваиваем переменной y значение по адресу из
указателя p
printf("x = %d \n", y); // 10
```

Здесь присваиваем переменной y значение по адресу из указателя p, то есть значение переменной x.

И также используя указатель, мы можем менять значение по адресу, который хранится в указателе:

```
int x = 10;
int *p = &x;
*p = 45;
printf("x = %d \n", x); // 45
```

Так как по адресу, на который указывает указатель, располагается переменная x, то соответственно ее значение изменится.

По адресам можно увидеть, что переменные часто расположены в памяти рядом, но не обязательно в том порядке, в котором они определены в тексте программы:

60FE9A	60FE9B	60FE9C	60FE9D	60FE9E	60FE9F	60FEA0	60FEA1	60FEA2	60FEA3
1 байт	1 байт	1 байт	1 байт	1 байт	1 байт	1 байт	1 байт	1 байт	1 байт
2	10								N
переменная s		переменная d							переменная c

## Основные операции над указателями

Указатели в языке Си определены операции: присваивание, получение адреса указателя, получение значения по указателю, некоторые арифметические операции и операции сравнения.

= — присвоение

& — взятие адреса объекта

\* — разыменование. Получение значения по указанному адресу.

Указателю можно присвоить либо адрес объекта того же типа, либо значение другого указателя или константу NULL.

```
int *p, n; //объявление переменной p - указатель на
целочисленный объект
p = &n; //присвоение адреса n в p
*p = 10; //n = 10 или положить значение по адресу в
переменной p
printf("n = %d\n", n); // n = 10
```

Указатель p может содержать адрес любого объекта целочисленного типа.

Особенности указателей:

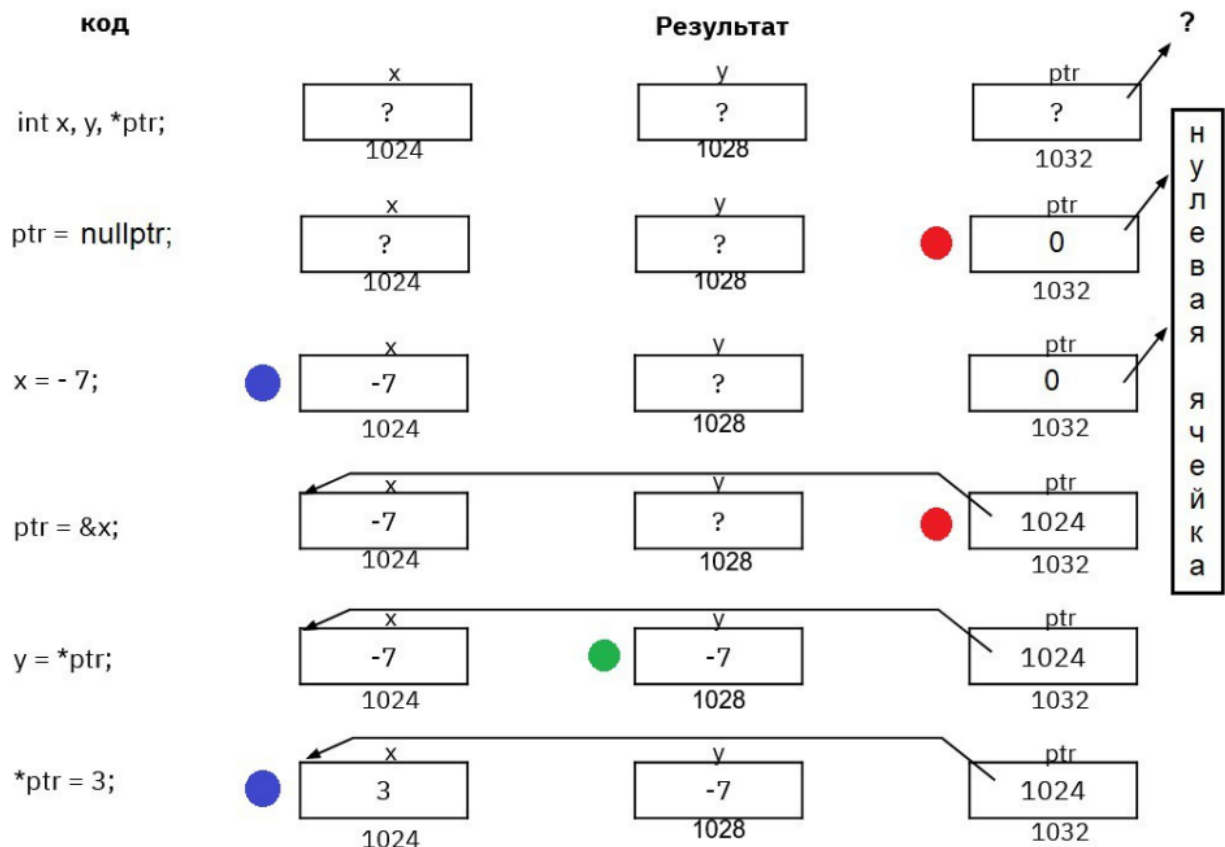
- Указатель это переменная которая хранит адрес другой переменной.
- Для работы с указателями используются операции \* и & Это не логическая "И"
- Инициализатор nullptr для указателей появился в стандарте C++11, до него использовался макрос NULL.

К указателям могут применяться операции сравнения >, >=, <, <=, ==, !=. Операции сравнения применяются только к указателям одного типа и константе NULL.

Рассмотрим практический пример изменение значения переменной через указатель.

```
#include <stdio.h>
int main() {
    int x, y, * ptr; // объявляем 3 переменные
    ptr = NULL; // инициализируем указатель нулевым значением
    x = -7;
    ptr = &x; // адрес переменной x записываем в переменную ptr
    y = *ptr; // Записываем в y значение на которое указывает указатель ptr
    *ptr = 3; // Записываем в ячейку (x) на которую ссылается указатель ptr число
    3
    //вывод на экран x = 3 y = -7
    printf("x = %d y = %d", x, y);
    return 0;
}
```

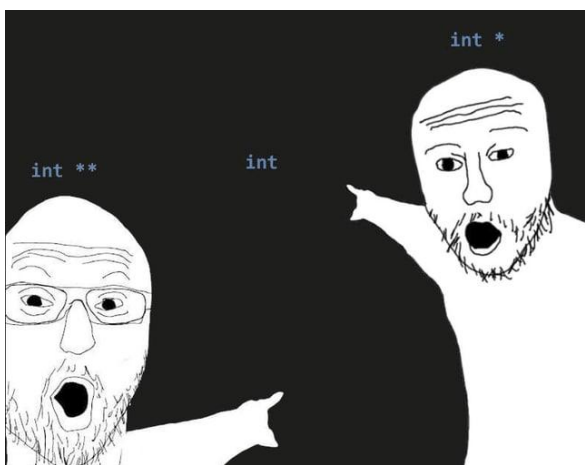
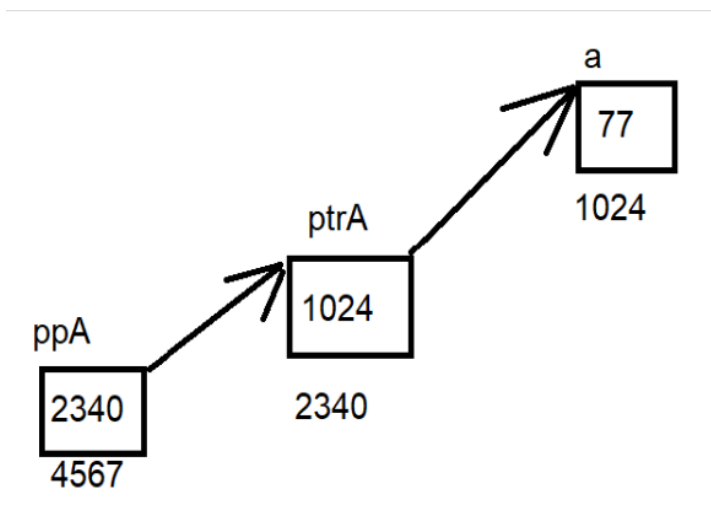
Графическое пояснение операции взятие адреса & и разименование \*



## Указатели на указатели

Кроме обычных указателей в языке Си мы можем создавать указатели на другие указатели. Если указатель хранит адрес переменной, то указатель на указатель хранит адрес указателя, на который он указывает. Такие ситуации еще называются многоуровневой адресацией. Две звездочки в определении указателя говорят о том, что мы имеем дело с двухуровневой адресацией.

```
//Можно объявлять указатели на указатели:
#include <stdio.h>
int main() {
    int a = 77;
    int *ptrA = &a;
    int** ppA = &ptrA;
    *ptrA = 88;
    printf("%d\n", a); // << std::endl; // 88
    **ppA = 99;
    printf("%d\n", a); // << std::endl; // 99
    return 0;
}
```



## Указатели как параметры функции

При рассмотрении передачи параметров в функцию указывалось, что параметры передаются в функцию по значению. То есть функция не изменяет значения передаваемых аргументов.

```

#include <stdio.h>
void swap(int a, int b) {
    int tmp;
    tmp = a;
    a = b;
    b = tmp;
}
int main(void)
{
    int n=7, m=5;
    swap(n, m); // Передаются
значения
    printf("n = %d m =
%d\n", n, m);
  
```

`n = 7 m = 5`

}	
---	--

Как видно из вывода значение переменных `n` и `m` не изменилось. Параметры функции `swap` передаются по значению и, следовательно, поменялись местами только копии параметров `n` и `m` внутри функции, при этом никакого влияния на сами фактические параметры `n` и `m` не произошло. Для достижения нужного эффекта необходимо в качестве параметров функции использовать указатели на соответствующие переменные, т.е. передавать в функцию адреса переменных, значения которых требуется изменить:

<pre>#include &lt;stdio.h&gt; void swap(int *pa, int *pb) {     int tmp;     tmp = *pa;     *pa = *pb;     *pb = tmp; } int main(void) {     int n=7, m=5;     swap(&amp;n, &amp;m); // Передается адрес     printf("n = %d m = %d\n", n, m); }</pre>	<pre>n = 5 m = 7</pre>
---	------------------------

Для обеспечения доступа к объектам в вызывающей функции - параметры должны передаваться в виде указателей на эти объекты.

Что будет напечатано?

<pre> #include &lt;stdio.h&gt; int y=1; int f(int *pa) {     static int x=3;     x += y;     return x + *pa + y; } int main(void) {     int n = 10;     y++;     printf("Answer = %d\n", f(&amp;n));\     printf("Answer = %d\n", f(&amp;n)); } </pre>	<pre> Answer = 17 Answer = 19 </pre>
--	--------------------------------------

## Адресная арифметика

Над указателями определена следующая адресная арифметика или арифметика указателей

- Операции сравнения: ==, !=, >=, <=, <, >
- Вычитание и добавление целочисленной константы
- Вычитание одного указателя из другого
- Унарные операции: ++, --
- Операции: -=, +=

Однако сами операции производятся по-другому, чем с числами. И тут многое зависит от типа указателя. К указателю можно прибавлять или вычитать целое число, можно вычитать из одного указателя другой указатель.

Рассмотрим вначале операции инкремента и декремента и для этого возьмем указатель на объект типа int:

<pre> #include &lt;stdio.h&gt;  int main(void) {     long int n = 10; </pre>
--

```

long int *ptr = &n;
printf("address=%p \t value=%d \n", (void*)ptr, *ptr);

ptr++;
printf("address=%p \t value=%d \n", (void*)ptr, *ptr);

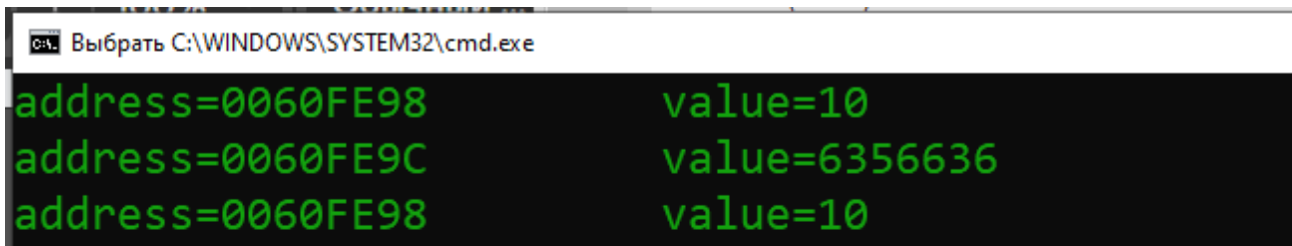
ptr--;
printf("address=%p \t value=%d \n", (void*)ptr, *ptr);

return 0;
}

```

Операция инкремента ++ увеличивает значение на единицу, но в случае увеличения на единицу будет означать увеличение адреса, на размер типа указателя `long int`.

То есть в данном случае указатель на тип `long int`, а размер объектов `long int` равен 4 байтам. Поэтому увеличение указателя типа `long int` на единицу означает увеличение значение указателя на 4. Консольный вывод выглядит следующим образом:



```

address=0060FE98      value=10
address=0060FE9C      value=6356636
address=0060FE98      value=10

```

Здесь видно, что после инкремента значение указателя увеличилось на 4: с 0x0060FE98 до 0x0060FE9C и по этому адресу лежит число 6356636 — это “мусор”, так как значение не определено.

А после декремента, то есть уменьшения на единицу, указатель получил предыдущий адрес в памяти.

Увеличение адреса на единицу означает, что мы хотим перейти к следующему объекту в памяти. А уменьшение на единицу означает переход назад к предыдущему объекту в памяти.

Инкремент указателя `address++`; эквивалентно `address + sizeof(тип);`

Декремент указателя `address--`; эквивалентно `address - sizeof(тип);`

Аналогично, для указателя типа short эти операции изменяли бы адрес на 2, а для указателя типа char на 1.

```
#include <stdio.h>

int main(void)
{
    double d = 10.6;
    double *pd = &d;
    printf("Pointer pd: address=%p \n", (void*)pd);
    pd++;
    printf("Pointer pd: address=%p \n", (void*)pd);

    char c = 'N';
    char *pc = &c;
    printf("Pointer pc: address=%p \n", (void*)pc);
    pc++;
    printf("Pointer pc: address=%p \n", (void*)pc);

    return 0;
}
```

Как видно из консольного вывода, увеличение на единицу указателя типа double дало увеличения хранимого в нем адреса на 8 единиц (размер объекта double - 8 байт), а увеличение на единицу указателя типа char дало увеличение хранимого в нем адреса на 1 (размер типа char - 1 байт).

А добавление и вычитание константы

```
address += shift; эквивалентно address + shift*sizeof(тип);
address -= shift; эквивалентно address - shift*sizeof(тип);
```

В отличие от сложения операция вычитание может применяться не только к указателю и целому числу, но и к двум указателям одного типа:

```
#include <stdio.h>
int main(void)
{
    int a = 10;
```



```

int b = 23;
int *pa = &a;
int *pb = &b;
int c = pa - pb;    // разница между адресами

printf("pa=%p \n", (void*)pa);
printf("pb=%p \n", (void*)pb);
printf("c=%d \n", c);

return 0;
}

```

## Особенности арифметических операций с указателями

При работе с указателями надо отличать операции с самим указателем и операции со значением по адресу, на который указывает указатель.

```

int a = 10;
int *pa = &a;
int b = *pa + 20;    // операция со значением, на который
// указывает указатель
pa++;                // операция с самим указателем
printf("b=%d \n", b);    // 30

```

То есть в данном случае через операцию разыменования `*pa` получаем значение, на которое указывает указатель `pa`, то есть число 10, и выполняем операцию сложения. То есть в данном случае обычная операция сложения между двумя числами, так как выражение `*pa` представляет число.

Но в то же время есть особенности, в частности, с операциями инкремента и декремента. Дело в том, что операции `*`, `++` и `--` имеют одинаковый приоритет и при размещении рядом выполняются справа налево.

Например, выполним постфиксный инкремент:

```

int a = 10;

```

```
int *pa = &a;
printf("pa: address=%p \t value=%d \n", (void*)pa, *pa);
int b = *pa++;          // инкремент адреса указателя

printf("b: value=%d \n", b);
printf("pa: address=%p \t value=%d \n", (void*)pa, *pa);
```

В выражении `b = *pa++`; сначала к указателю присваивается единица (то есть к адресу добавляется 4, так как указатель типа `int`). Затем так как инкремент постфиксный, с помощью операции разыменования возвращается значение, которое было до инкремента - то есть число 10. И это число 10 присваивается переменной `b`.

Изменим выражение:

```
b = (*pa)++;
```

Скобки изменяют порядок операций. Здесь сначала выполняется операция разыменования и получение значения, затем это значение увеличивается на 1. Теперь по адресу в указателе находится число 11. И затем так как инкремент постфиксный, переменная `b` получает значение, которое было до инкремента, то есть опять число 10. Таким образом, в отличие от предыдущего случая все операции производятся над значением по адресу, который хранит указатель, но не над самим указателем.

Аналогично будет с префиксным инкрементом:

```
b = ++*pa;
```

Теперь сначала изменяет адрес в указателе, затем мы получаем по этому адресу значение и присваиваем его переменной `b`. Полученное значение в этом случае может быть неопределенным.

Рассмотрим порядок выполнения строки `*pa++ = n+3`; в данном случае необходимо правильно расставить приоритет операций: `*(pa++) = n+3`; Операция инкремента меняет адрес (прибавляет 4 байта).

```
*pa = n+3;
pa++;
```

# Массивы

При решении задач с большим количеством данных одинакового типа использование переменных с различными именами, не упорядоченных по адресам памяти, затрудняет программирование. В подобных случаях в языке Си используют объекты, называемые массивами.

**Массив** — это последовательность однотипных элементов имеющих общее имя.

Массив представляет набор однотипных элементов и характеризуется следующими основными понятиями:

**Адресом элемента** — адресом начальной ячейки памяти, в которой расположен этот элемент;

**Индексом элемента** — порядковым номером элемента в массиве. Индексы указываются в квадратных скобках после названия массива и начинаются с нуля, поэтому для обращения к первому элементу необходимо использовать выражение `numbers[0]`;

**Значение элемента массива** — значение, хранящееся в определенной ячейке памяти, расположенной в пределах массива.

**Размер массива** — количество элементов массива

**Размер элемента** — количество байт, занимаемых одним элементом массива.

Все элементы массива располагаются в памяти подряд. Доступ к элементу массива осуществляется по его индексу, в качестве индекса используются целые неотрицательные числа, индексация элементов массива начинается с 0. Таким образом, в массиве из  $n$  элементов индексация элементов изменяется в диапазоне от 0 до  $n - 1$ .

Объявление массива выглядит следующим образом:

тип\_переменной название\_массива [длина\_массива]

**Примеры:**

- Квартиры в доме
- Сотрудники
- Данные о температуре

```
//Примеры объявления массивов
int a[3]; //массив из трех элементов a[0], a[1], a[2]
```

```
float f[5]; //массив из пяти элементов
char c[3] = {'A', 'z', '0'}; //массив из трех элементов типа char
с инициализацией
int x[5] = {7, 5}; //массив из 5 элементов с инициализацией
x[0]=7, x[1]=5
```



**Внимание!** Если начальные значения не заданы, то в памяти лежит мусор

## Что неправильно?

```
int buffer[100];
```

```
enum {BUFFER_SIZE=100};
int buffer[BUFFER_SIZE];
```

```
int x[5.5];
```

индекс не целое число

```
float f[5];
int n=1;
f[n-2]=3.14159;
f[n+5]=100;
```

n-2 = -1 начинается с 0  
n+5 = 6 выход за границы

```
int a[2] = {1, 3.5};
float b[2] = {1., 4.5, 3.14};
```

дробное число в целый массив  
третий элемент выйдет за  
границы памяти

Стандарт Си-99 допускает также использование массивов переменной длины, объявленных локально. Например:

```
int n;
scanf ("%d", &n); // ввод количества элементов массива
int arr[n];
for (int i =0; i < n; i++) { // ввод массива
    scanf ("%d", &arr[i]);
}
```

## Массивы и указатели

В языке Си массивы и указатели тесно связаны. С помощью указателей мы также легко можем манипулировать элементами массива, как и с помощью индексов.

В языке Си имя массива - это указатель на самый первый элемент массива (элемент с индексом 0). Изменять значение такого указателя нельзя.

```
int a[5];
int b[5];
a = b; // так делать нельзя
```

Мы можем создать обычный указатель и установить его на нулевой элемент массива и изменять уже его.

```
int a[5];
int *pa;
pa = a; // так можно
```

**sizeof** - операция для вычисления размера объекта в памяти.

<pre>int a[5]; int *pa; pa = a; printf("sizeof(a) = %lu\n", sizeof(a)); printf("sizeof(pa) = %lu\n", sizeof(pa));</pre>	<pre>sizeof(a) = 20 sizeof(pa) = 8</pre>
---	--

Размер массива  $a = \text{<количество элементов>} * \text{<размер типа>} = 5 * \text{sizeof(int)} = 20$ .

Размер указателя  $pa = 8$  для 64-х разрядной системы или 4 для 32-х.

## Передача массива в функцию

Имя массива является неизменяемым указателем на его нулевой элемент: при указании в качестве параметра имени массива в функцию передается копия адреса начала той области памяти, в которой находятся элементы массива. Также известно, сколько места занимает каждый элемент массива, что позволяет обратиться к любому элементу массива. Однако информация о количестве элементов массива теряется, поэтому размер массива следует передавать в функцию через дополнительный параметр. Например, рассмотрим функцию вычисления суммы элементов массива из  $n$  элементов.

<pre>int sum(int a[], int size) {     int i, s = 0;     for(i = 0; i &lt; size; i++)         s += a[i];     return s; }</pre>	<pre>int sum(int *a, int size) {     int i, s = 0;     for(i = 0; i &lt; size; i++)         s += a[i];     return s; }</pre>
---	--

Вызов функции даст следующие результаты

<pre>int a[5] = {10,20,30,40,50}; printf("%d\n",sum(a,5)); printf("%d\n",sum(a,2)); printf("%d\n",sum(a+2,3));</pre>	<pre>150 30 120</pre>
--	-----------------------

При этом записи `int sum(int a[], int size)` и `int sum(int *a, int size)` эквивалентны

## Задачи

1. Написать функцию вычисления скалярного произведения двух вещественных массивов.

<pre>#include &lt;stdio.h&gt; float scalar(float*arrA,float*arrB,int len) {     float result = 0;     for(int i=0;i&lt;len;i++)     {         // result += arrA[i]*arrB[i];         // result += (float*)(arrA+i) * (float*)(arrB+i);         result += *(arrA+i) * *(arrB+i);         // printf("%f,%f,%f\n",result,*(arrA+i),*(arrB+i));     }     return result; } #define SIZE 3 int main(int argc, char **argv) {     float arrA[SIZE]={1,2,3};     float arrB[SIZE]={1,2,3};     //float arrC[SIZE]={1,2,3};     printf("%f",scalar(arrA,arrB,SIZE));     //Print(arrC,SIZE);     return 0; }</pre>
---

2. Написать функцию сдвига массива на N элементов вправо.
3. Составить функцию которая определяет в массиве, состоящем из положительных и отрицательных чисел, сколько элементов превосходят по модулю максимальный элемент. Показать пример ее работы на массиве из 10 элементов.

## Задачи

1. Ввести с клавиатуры массив из 5 элементов и умножить все его элементы на число 3. Распечатать полученный массив.
2. Укажите, какому элементу массива будет присвоено значение 12, а также значение переменной `i` после выполнения следующего фрагмента программы:

```
int i = 8;  
a[i++] = 12;
```

```
int i = 8;  
a[++i] = 12;
```

3. Найти минимальный элемент в массиве.
4. Найти максимальный элемент в массиве.
5. Поменять местами максимальный и минимальный элемент в массиве.
6. На стандартном потоке ввода задан текст, состоящий из латинских букв и цифр и оканчивающийся точкой. На стандартный поток вывода вывести цифру, наиболее часто встречающуюся в тексте (если таких цифр несколько, вывести любую из них).

## Генератор случайных чисел массива

Для организации тестирований программ с использованием массивов удобно заполнить массив случайными числами. Используем библиотеку `stdlib.h`

```
#include <stdio.h>  
#include <stdlib.h>  
#include <time.h>  
  
int main() {
```

```

    int x;
    printf("MAX RANDOM = %d\n", RAND_MAX); //2147483647
    // Задать начальное значение последовательности
    srand(123); // одни и те же числа, seed задан константно
    //Случайное целое число в интервале [0, RAND_MAX]
    x = rand();
    printf("x = %d\n", x); // первое случайное число
    srand( time(NULL) ); // числа меняются при каждом запуске
    программы
    x = rand();
    printf("x = %d\n", x); // другое случайное число
    return 0;
}

```

Обособим генератор случайных чисел в отдельную функцию `random_number()`

```

#include <stdio.h>
#include <stdlib.h>
enum {SIZE = 10, SEED = 123};

int random_number(int n) {
    return rand() % n;
}

int main(void)
{
    int a[SIZES], i;
    srand(SEED);
    printf("Array:\n");
    for (i = 0; i < SIZE; i++) {
        a[i] = random_number(100) + 50;
        printf("%4d", a[i]);
    }
    return 0;
}

```

Сделаем функции `init_array` и `print_array`. Далее будем придерживаться парадигмы: разделять ввод, вывод и вычисления. Это удобно, т.к. теперь для вывода в файл или из файла достаточно заменить соответствующие функцию ввода и вывода, не меняя остальной, уже отлаженный код.

```

#include <stdio.h>
#include <stdlib.h>

```



```

enum {SIZE = 10, SEED = 123};

int random_number(int n) {
    return rand() % n;
}

void init_array(int size, int a[], int max_random) {
    for (size_t i = 0; i < size; i++) {
        a[i] = random_number(max_random);
    }
    return;
}

void print_array(int size, int a[]) {
    for (size_t i = 0; i < size; i++) {
        printf("%d ", a[i]);
    }
    printf("\n");
    return;
}

int main(void)
{
    int a[SIZE], i;
    srand(SEED);
    init_array(SIZE, a, 100);
    print_array(SIZE, a);
    return 0;
}

```

## Манипуляции с массивами

### Циклически сдвинуть массив влево на 1 элемент

```

// циклический сдвиг массива влево на 1 элемент
void shift_array_left(int size, int a[]) {
    int tmp=a[0];
    for (size_t i=0; i < size-1; i++) {
        a[i] = a[i+1];
    }
    a[size-1] = tmp;
}

```

### Сделать реверс массива

```

/*
Реверс массива
*/

```

```

void revers_array(int size, int a[]) {
    for(size_t i=0; i<size/2; i++) {
        swap(&a[i], &a[size-1-i]);
    }
    return;
}

```

## Отсортировать массив по возрастанию пузырьком

```

/*
Сортировка пузырьком
*/
void bubble_sort_array(int size, int a[]) {
    int i = 0;
    _Bool flag;
    do {
        flag = 0; // сбросить флаг
        for (int j = size-2; j >= i ; j-- )
            if ( a[j] > a[j+1] ) {
                swap(&a[j], &a[j+1]);
                flag = 1; // поднять флаг если есть обмен
            }
        i++;
    }
    while ( flag ); // выход при flag = 0
}

```

## Отсортировать массив по возрастанию выбором

```

/*
Сортировка выбором
*/
void choose_sort_array(int size, int a[]) {
    int nMin;
    for(int i = 0; i < size-1 ; i ++ ) {
        for (int j = i+1; j < size; j ++ )
            if( a[j] < a[nMin] ) {
                nMin = j;
            }
        if( nMin != i ) {
            swap(&a[i], &a[nMin]);
        }
    }
}

```

## Операции над указателями массива

Над указателями определена следующая адресная арифметика

- Операции сравнения: ==, !=, >=, <=, <, >
- Вычитание и добавление целочисленной константы
- Вычитание одного указателя из другого
- Унарные операции: ++, --
- Операции: -=, +=

Рассмотрим пример. Обратите внимание чему равны адреса соседних элементов.

<pre>int a[5]; printf("&amp;a[0] = %p\n", &amp;a[0]); printf("&amp;a[1] = %p\n", &amp;a[1]);</pre>	<pre>&amp;a[0] = 0x7ffee1a20ab0 &amp;a[1] = 0x7ffee1a20ab4</pre>
--	--

Рассмотрим пример.

<pre>int a[5] = {10, 20, 30, 40, 50}; int *pa, n; pa = a; n = *(pa+2); // a[2] printf("n = %d\n", n);</pre>	<pre>n = 30</pre>
---	-------------------

При вычислении операции `pa+2` к адресу хранящемуся в переменной `pa` присваивается значение `2 * sizeof(int)` это как раз соответствует элементу с индексом 2, массива `a`.

<pre>int a[5] = {10, 20, 30, 40, 50}; int *pa, n; pa = a; n = *pa+2; // a[0] + 2 printf("n = %d\n", n);</pre>	<pre>n = 12</pre>
---	-------------------

При вычитании одного указателя из другого, вычисляется разность адресов, которая затем делится на размер (в байтах) элемента массива.

Результатом является количество элементов массива, расположенных между данными указателями.

<pre>int a[5] = {10,20,30,40,50}; int *pa, *qa; pa = &amp;a[1]; qa = &amp;a[3]; printf("%ld\n", qa-pa);</pre>	2
---	---

Рассмотрим другой пример. Тут операция инкремента меняет значение, а не адрес.

<pre>int a[5] = {10,20,30,40,50}; int *pa, n=10; pa = a+2; n = ++*pa; // ++(*pa) printf("%d\n", n);</pre>	31
---	----

### Что будет напечатано?

<pre>int a[5] = {10,20,30,40,50}; int *pa, n=10; pa = a+2; *pa++ = n+3; printf("%d\n", *pa); printf("%d %d %d %d %d\n", a[0], a[1], a[2], a[3], a[4]);</pre>	40 10 20 13 40 50
--	----------------------

## Вычисление корней квадратного уравнения (массивы)

Перепишем программу в соответствии с парадигмой: разделять ввод, вывод и вычисления.

<pre>#include &lt;stdio.h&gt; #include &lt;conio.h&gt; #include &lt;locale.h&gt; #include &lt;math.h&gt;  #define COEF_N 3 #define ROOT_N 2 enum {NO_ROOTS=-1, COMPLEX_ROOTS=0};</pre>
--

```

float InputFloat(char* message)
{
float number;
static int counter = 0;
    counter++;
//    printf("%d,%s",counter,message);
    printf("%s",message);
    scanf("%f",&number);
    return number;
}

void Input(int size, float coef[])
{
    printf("Вычисление корней квадратного уравнения \\ \\ \"a*x*x+b*x+c=0\\\"\\n");
    coef[2] = InputFloat("Введите a:\\n");//1
    coef[1] = InputFloat("Введите b:\\n");//18
    coef[0] = InputFloat("Введите c:\\n");//32
}

void CalcRealRoots(float sqrD,float B,float a,float roots[])
{
    float d = sqrtf(sqrD);
    roots[0] = (-B + d)/a; //2
    roots[1] = (-B - d)/a; //16
}

void Print(int size, float roots[])
{
    if(size==2)
    {
        printf("Корни квадратного уравнения \\n");
        printf("X1 = %f \\n",roots[0]);
        printf("X2 = %f \\n",roots[1]);
    }
    else if(size==1)
        printf("Корень линейного уравнения %f\\n",roots[0]);
    else if(size==COMPLEX_ROOTS)
        printf("Корни квадратного уравнения комплексные \\n");
    else if(size==NO_ROOTS)
        printf("Корней НЕТ!\\n");
    else
        printf("Ошибка количества корней%d!\\n",size);
}

int CalcRoots(float coef[],float roots[])
{
int rootsNumber;
float B = coef[1]/2;
if(coef[2]!=0)
{
    float d = B*B-coef[2]*coef[0];
    if(d<0)
        rootsNumber=COMPLEX_ROOTS;
    else
    {
        rootsNumber=2;
        CalcRealRoots(d,B,coef[2],roots);
        return 2;
    }
}
else
{
    if(coef[1]!=0)

```

```

        {
            roots[0] = -coef[0]/coef[1];
            rootsNumber=1;
        }
        else
            rootsNumber=NO_ROOTS;
    }
    return rootsNumber;
}

//Сделаем вычисление корней квадратного уравнения отдельной функцией.
void SquareEquation(void)
{
    float coef[COEF_N];
    float roots[ROOT_N];
    Input(COEF_N,coef);
    int rootsNumber = CalcRoots(coef,roots);
    Print(rootsNumber,roots);
}

int main(int argc, char **argv)
{
    char Choice;
    setlocale(LC_ALL, "Rus");
    while(1)
    {
        printf("1. Вычисление корней квадратного уравнения\n");
        printf("0. Выход\n");
        printf("Для выход нажмите Q\n");
        Choice = getch();
        switch(Choice)
        {
            case '1':
                SquareEquation();
                break;
            case '0':
            case 'q':
            case 'Q':
                return 0;
                break;
            default:
                printf("Непонятный выбор %x\n",Choice);
                break;
        }
    }
    return 0;
}

```

## Подведение итогов

Итак, на этой лекции мы начали с изучили [рекурсию в функциях](#), как работает [стек вызовов](#) при рекурсии.

Рассмотрели [префиксная и постфиксная форма записи](#) и [сложную рекурсию](#).

Сравнили [рекурсия и итерирование](#) и [что лучше?](#)

Также отметили [проблемы рекурсии](#), что такое [хвостовая рекурсия](#) и [замена цикла на рекурсию](#).

Порешали [задачи на рекурсию](#), среди которых хочется особо отметить [рекурсивное вычисления Sin через ряд Тейлора](#).

Затронули вопросы [представления чисел в памяти](#) и в частности [представление вещественных чисел по стандарту IEEE 754](#), а также [арифметические операции](#) с ними.

Узнали про [погрешность вещественных чисел](#), и каким последствиям она может привести.

[Указатель](#)

[Определение указателя](#)

[Получение адреса данных](#)

[Получение значения по адресу](#)

[Основные операции над указателями](#)

[Указатели на указатели](#)

[Указатели как параметры функции](#)

[Адресная арифметика](#)

[Особенности арифметических операций с указателями](#)

[Массивы](#)

[Что неправильно?](#)

[Массивы и указатели](#)

[Передача массива в функцию](#)

[Задачи](#)

[Задачи](#)

[Генератор случайных чисел массива](#)

[Манипуляции с массивами](#)

[Циклически сдвинуть массив влево на 1 элемент](#)

[Сделать реверс массива](#)

[Отсортировать массив по возрастанию пузырьком](#)

[Отсортировать массив по возрастанию выбором](#)

[Операции над указателями массива](#)

[Что будет напечатано?](#)

[Вычисление корней квадратного уравнения \(массивы\)](#)

# Дополнительные материалы

1. Домашнее задание задачи B1 - B21
2. Оформление программного кода  
<http://www.stolyarov.info/books/pdf/codestyle2.pdf>
3. Стандарт разработки программного обеспечения MISRA на языке C  
[http://easyelectronics.ru/files/Book/misra\\_c\\_rus.pdf](http://easyelectronics.ru/files/Book/misra_c_rus.pdf)
4. <https://metanit.com/c/tutorial/5.3.php> Арифметика указателей
5. <https://metanit.com/c/tutorial/5.1.php> Указатели
6. [https://cpp.com.ru/shildt\\_spr\\_po\\_c/05/05.html](https://cpp.com.ru/shildt_spr_po_c/05/05.html) Указатели
- 7.

# Используемые источники

1. [cprogramming.com](http://cprogramming.com) - учебники по C и C++
2. [free-programming-books](http://free-programming-books) - ресурс содержащий множество книг и статей по программированию, в том числе по C и C++ (там же можно найти ссылку на распространяемую бесплатно автором html-версию книги Eckel B. «Thinking in C++»)
3. [tutorialspoint.com](http://tutorialspoint.com) - ещё один ресурс с множеством руководств по изучению различных языков и технологий, в том числе содержит учебники по C
4. Юричев Д. [«Заметки о языке программирования Си/Си++»](#) - «для тех, кто хочет освежить свои знания по Си/Си++»
5. [Онлайн версия «Си для встраиваемых систем»](#)
6. <https://metanit.com/sharp/tutorial/2.4.php> Массивы
7. <https://prog-cpp.ru/c-massiv/> Массивы в языке Си
8. <https://metanit.com/c/tutorial/2.13.php> Введение в массивы и строки
9. [https://robotclass.ru/articles/c\\_book\\_language/7/](https://robotclass.ru/articles/c_book_language/7/) Указатели и массивы
10. <https://metanit.com/c/tutorial/5.5.php> Указатели, массивы и строки



11. <https://habr.com/ru/articles/545674/> Кратко об указателях в Си: присваивание, разыменованное и перемещение по массивам
12. [https://cpp.com.ru/kr\\_cbook/ch5kr.html](https://cpp.com.ru/kr_cbook/ch5kr.html) Глава 5. Указатели и массивы
13. <https://studfile.net/preview/2426175/> Указатели
14. <https://sohabr.net/habr/post/257485/> Указатели, массивы и структуры С. Приподнимаем занесу