

Модуль 1. Основы программирования

Тема 1.10. Указатели на примере языка Си

2 часа

Оглавление


1.10. Указатели на примере языка Си.....	2
1.10.1. Понятие указателя и его объявление	3
1.10.2. Инициализация указателя	5
1.10.3. Разыменование указателя.....	5
1.10.4. Указатели и массивы. Адресная арифметика	6
1.10.5. Зачем нужны указатели?	7
1.10.6. Передача данных в функцию	8
1.10.7. Массивы в качестве параметров функции	10
1.10.8. Динамическое выделение памяти	10
1.10.9. Указатели в Си и ссылки в Java	12
Задание 1.10.1.....	14
Задание 1.10.2.....	15

1.10. Указатели на примере языка Си

Для тех, кто имеет серьезный опыт программирования на Си и Паскале возможно будет сюрпризом то, что в языке Java нет такого понятия, как указатель. В тоже время в Java есть такое понятие, как ссылки на объекты классов. В каком-то смысле их можно рассматривать как указатели, над которыми нельзя выполнять арифметические операции (о них здесь будет рассказано). Работа с массивами в Java также на самом деле связана с понятием “указатель”.

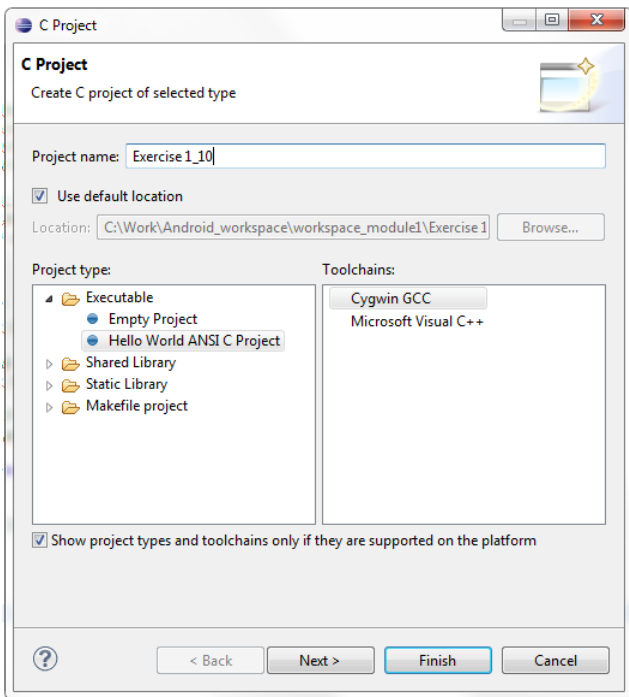
Указатели позволяют работать на уровне адресов, так называемом «низком уровне». Они дают языку Си огромные возможности, но в тоже время усложняют и являются источником трудно обнаруживаемых ошибок. Не зря создатель языка Си Брайан Керниган говорил: «Си — инструмент, острый, как бритва: с его помощью можно создать и элегантную программу, и кровавое месиво». Как результат создатели языка Java отказались от явного использования указателей. Однако, если программист не владеет понятием указателя, то он будет совершать при программировании на Java грубейшие ошибки при работе с памятью. Поэтому авторы курса сочли необходимым включить тему «Указатели» в курс ИТ ШКОЛЫ SAMSUNG, и мы разберем ее на примере языка Си.

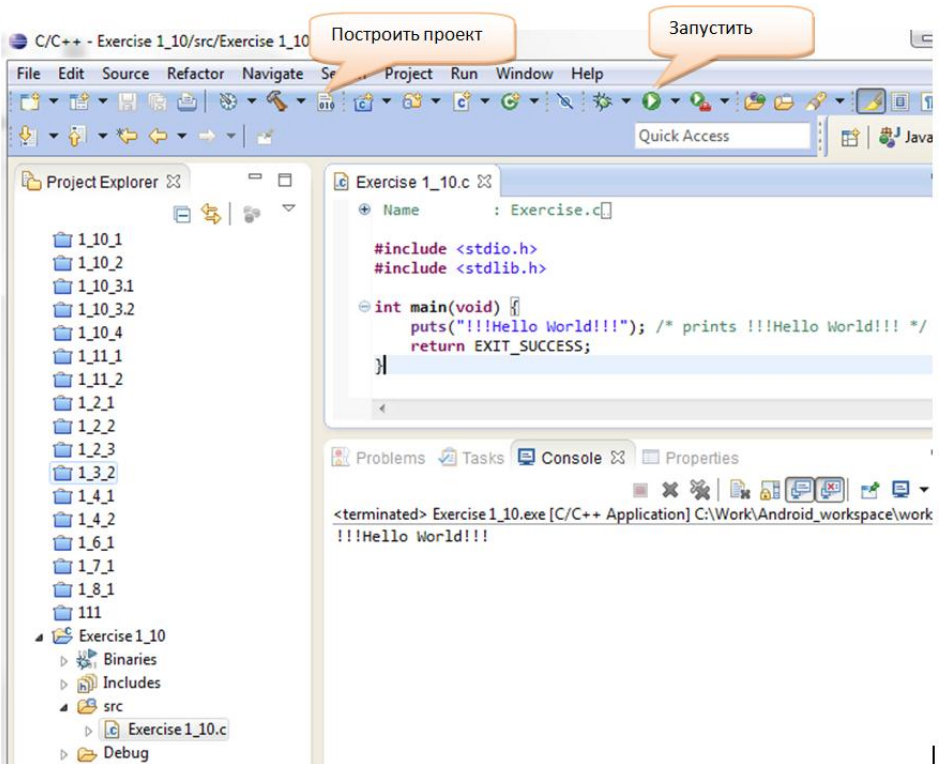
В ходе изучения темы мы будем использовать готовые консольные программы на языке Си, демонстрирующие работу с указателями. Их очень просто запускать в уже используемой нами среде Eclipse. Язык Си имеет очень схожий с Java синтаксис, поэтому разбираемые программы будут понятны.



Для создания проекта на языке Си в среде Eclipse выберите в меню: **File⇒New⇒Project** и в открывшемся окне выберите тип проекта **C Project**, затем кнопку **Next**. В следующем окне введите имя проекта, шаблон **Hello World ANSI C Project** и библиотеку **Cygwin GCC** (она входит в инсталляционный пакет ИТ ШКОЛЫ SAMSUNG, но если Eclipse настраивался самостоятельно, то как установить эту библиотеку можно посмотреть [здесь](https://cygwin.com/install.html) <https://cygwin.com/install.html>)

В данном разделе мы будем рассматривать простые программы, для запуска которых текст программы необходимо просто вставить вместо начального кода в файле с расширением **.c**, который откроется в основном окне. Затем последовательно построить и запустить проект на выполнение:





В окне (снизу справа) - консоль, в которой видим результат работы программы.

1.10.1. Понятие указателя и его объявление

Итак, что же такое указатель? **Указатель** - это переменная, значением которой является адрес ячейки памяти либо нулевой адрес. Последний используется для указания того, что в данный момент указатель не содержит адреса.



Адрес переменной

Переменная — это область памяти для хранения данных. Таким образом для облегчения программирования придумали давать области памяти имя - имя переменной.

Память состоит из ячеек, каждая из которых имеет свой адрес, выраженный числом в системе, понятной процессору компьютера.

Ячейка - минимальная единица памяти, к которой процессор может обратиться, ее размер 1 байт (8 бит). Байты группируются в машинные слова. Сейчас более широко распространены компьютеры с 32-разрядными (4 байта) и 64-разрядными словами (8 байт). Такая единица как машинное слово, необходима, поскольку большинство команд производят операции над целыми словами.

Подведем итог. Получается, что адрес переменной - это адрес ячейки, с которой начинается память, выделенная под хранение данных этой переменной.

Адрес ячейки	Ячейка - 1 байт	
0	0 1 0 1 1 0 1 1	} Переменная <i>a</i>
1	0 0 0 1 1 1 1 1	
2	0 1 0 0 1 0 1 0	
3	1 1 0 1 0 0 1 0	
...	0 0 1 1 0 0 1 1	

Запустим на выполнение приведенную ниже программу на Си и увидим, что на экран будут выведены шестнадцатиричные числа (их используют, чтобы в краткой форме представить двоичные числа) - это адреса переменных *a* и *b*:

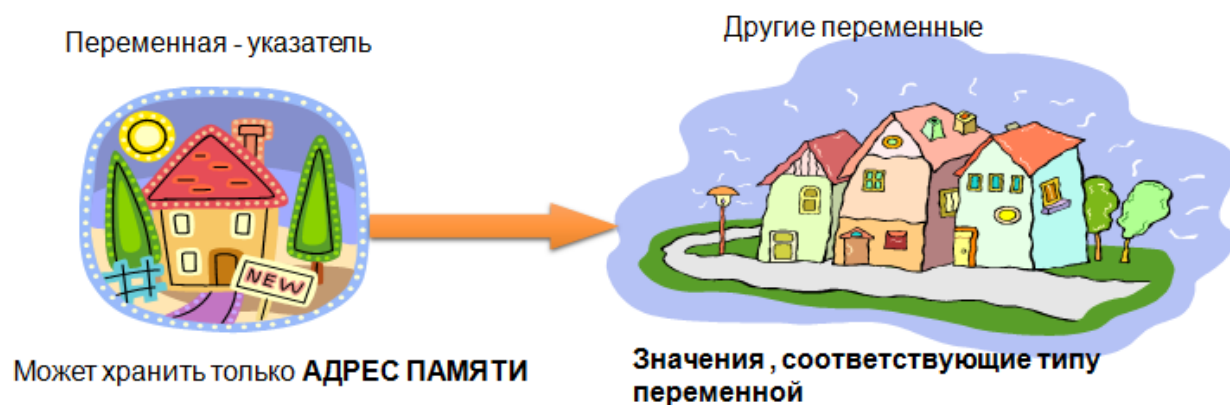
```
#include <stdio.h>

void main(void) {
    int a, b;
    printf (" a - %p\n b - %p", &a,&b);
}
```

Здесь мы остановимся и не будем углубляться в рассмотрение типов памяти и способов адресации к ней, это слишком обширный вопрос, который изучается в рамках программы подготовки ИТ специальностей в вузах.

В переменную-указатель можно сохранить только адрес. Впрочем это аналогично тому, как в переменную целого типа можно сохранить только целое число, а в символьного только символ.

А так как указатели позволяют сохранить адрес другой переменной, то мы можем прочесть его, перейти по этому адресу и считать, что находится в ячейках памяти, начиная с адреса, который хранился в указателе. И тут возникает вопрос “Как компилятору правильно интерпретировать содержимое памяти?” Например, в Си, если это символ (*char*), компилятору нужно прочесть ровно 1 байт памяти, а если это целое *int*, то, как правило, 4 байта (в языке Си в отличие от Java размеры базовых типов жестко не определены и зависят от разрядности машинного слова) . Следовательно указателю необходимо задавать тип, который определяется типом переменной, на которую он указывает (ссылается).



Ниже приведены примеры объявления переменных-указателей различных типов:

```
int a,*pa;      //pa - указатель на int
double b, *pb; //pb - указатель на double
char *pc;      //pc - указатель на char
```

Символ «*» в объявлении переменной показывает, что объявляется переменная-указатель.

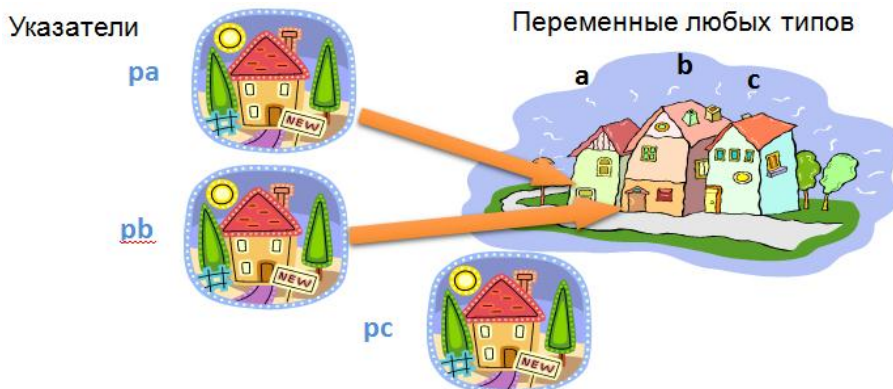
1.10.2. Инициализация указателя

Разберем пример присваивания указателю адреса переменной:

```
int a = 10;
int *pa;
pa = &a;      //pa присвоен адрес переменной a
double b, *pb;
pb = &b;      //pb присвоен адрес переменной b
pc = NULL;    //pc присвоен нулевой адрес
```

При инициализации указателей:

- Может применяться **унарная операция &** - операция взятия адреса переменной. Например, результатом выражения `&a` будет адрес переменной `a`.
- Можно использовать **ключевое слово NULL** - это нулевой указатель или нулевой адрес, который показывает, что данная переменная-указатель не указывает ни на какой объект.
- Соблюдается **соответствие типов**. Например, в примере нельзя написать: `pb=&a;` Потому что `a` - это переменная типа `int`, а `pb` - указатель типа `double` (не `int`).



На рисунке стрелками показано, что после выполнения приведенного фрагмента программы в указателях `pa` и `pb` хранятся адреса переменных `a` и `b` соответственно, т.е. теперь **`pa` и `pb` «указывают»** или еще говорят **«ссылаются» на `a` и `b`**.

1.10.3. Разыменование указателя

Как сохранить адрес переменной в указателе мы разобрались, а теперь рассмотрим, как через указатель работать с ее значением. Для этого в Си введена специальная операция «*» - **операция разыменования**, иначе ее еще называют операцией косвенного обращения к переменной или косвенной адресацией.

Запустите программу, посмотрите, что будет выведено на консоль.

```
#include <stdio.h>

void main(void) {
    int a=10;
    int *pa;
    pa=&a;
    printf ("%d %d", a, *pa); //10 10
    double b, *pb;
    pb=&b;
    *pb=3.5;
    printf ("%lf %lf", b, *pb); //3.5 3.5
}
```

В примере мы видим, что доступ к значению переменной можно получить как уже известным нам способом непосредственно через переменную, так и через указатель, который на нее ссылается.

Обратите внимание, что использование «*» в языке Си имеет различное значение:

1. при объявлении переменной показывает, что объявляется указатель;
2. в качестве унарной операции – операция разыменования;
3. в арифметических выражениях - операция умножения.

Можно ли дополнить нашу программу такими командами? В чем ошибка?

```
char c, *pc;
*pc = 'A';
```

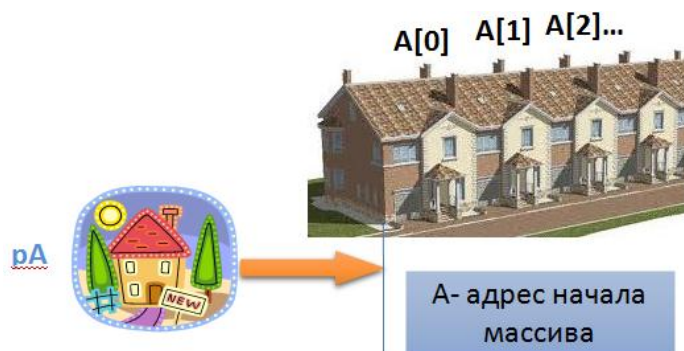
Так обращаться с указателем категорически нельзя! Операция разыменования не может быть корректно применена к указателю *pc*, потому что он не инициализирован: указатель не хранит адреса переменной (в нем хранится случайный мусор, который хранился в области памяти, выделенной под указатель, в момент его создания), а это значит, что перейти по адресу и уж тем более положить туда значение невозможно.

1.10.4. Указатели и массивы. Адресная арифметика

Указатели широко используются при работе с массивами. Здесь используется тот факт, что массивы - это значения одинакового типа, которые хранятся в памяти друг за другом. Поэтому вовсе не обязательно знать, где хранится каждый элемент массива, достаточно знать адрес начала массива и его тип. Тогда, зная размер занимаемой под каждый элемент массива памяти, можно легко посчитать адрес необходимого элемента.

```
int A[10];
int *pA = A; //эквивалентно pA = &A[0];
```

В языке Си имя массива (в примере - A) – это адрес его начала, т.е. указатель на первый элемент



массива.

Над указателями можно производить операции, которые называют операциями **адресной арифметики**:

- **++ (инкремент)** - сдвиг указателя на одну переменную вправо
- **-- (декремент)** - сдвиг указателя на одну переменную влево
- к указателю можно **прибавить или отнять целое число** - он будет сдвинут вправо или влево ровно столько раз. При этом размер сдвига в байтах соответствует типу указателя. Например, `pA++` означает 1 сдвиг вправо на один `int`, т.е 4 байта.
- указатели можно **сравнивать**, например, верно неравенство `&A[0]<&A[1]`, потому что `A[0]` лежит в памяти левее (раньше) чем `A[1]` (см. рисунок).

```
pA++; //эквивалентно pA=&A[1];  
*(pA+1)=3; //эквивалентно A[2]=3;
```

Однако, необходимо быть осторожным и отслеживать, чтобы не зайти за пределы массива:

```
*(pA+10)=7; //???
```

можем получить непредсказуемый результат.

1.10.5. Зачем нужны указатели?

В предыдущих разделах мы рассмотрели, что можно делать с помощью указателей, но не ответили на вопрос: «Для чего нужны указатели?»

Передача данных и возврат результатов из функции. Указатели применяются, если:

- необходимо получить в качестве результата работы функции больше, чем одну переменную,
- передать в функцию или получить из функции большой связный объем данных (массив, структуру данных).

Динамическое выделение памяти. Указатели позволяют работать с динамическими переменными, т.е. переменными, которые создаются по ходу выполнения программы. Это полезно, когда не известно заранее, сколько переменных нам будет необходимо для хранения данных

1.10.6. Передача данных в функцию

Из предыдущей темы мы знаем, что практически всегда в функцию необходимо передавать значения и получать из нее результаты. Передача значений в функцию осуществляется через параметры. При этом в языках программирования существует два способа передачи параметров: по значению и по ссылке. Разберемся в чем суть каждого из них и каким образом указатели здесь применяются.

Передача по значению

При этом способе значения фактических параметров копируются в формальные параметры функции.

Параметры функции могут рассматриваться как локальные переменные, для которых выделяется память при вызове функции и в начале ее выполнения параметрам функции присваиваются значения фактических параметров. При выходе из функции значения этих переменных теряются.

Рассмотрим пример программы и иллюстрацию к ней:

```
#include <stdio.h>

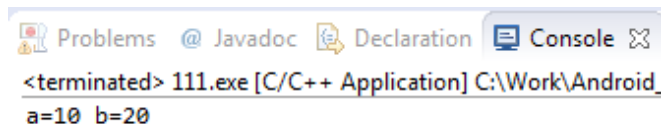
/*хотим поменять местами
значения a и b */
void sort(int a, int b)
{
    int c;
    if(a < b)
    { c = a;
      a = b;
      b = c;}
}
int main(void)
{
    int a, b;
    a = 10;
    b = 20;
    sort(a, b);
    printf("a=%d b=%d\n", a, b);
}
```



Программа начинает работу с запуска функции `main()`, в которой созданным переменным `a` и `b` присваиваются значения 10 и 20. Затем они передаются в вызываемую функцию `sort()`.

При запуске функции `sort()` в области данных этой функции создаются **другие переменные-параметры `a` и `b`**, в которые копируются переданные из функции `main()` значения. Все переменные функции `sort()` уничтожаются в момент завершения ее работы.

Таким образом действия внутри функции `sort()` никак не отразились на переменных функции `main()`. В результате работы программы значения переменных `a` и `b` функции `main()` остались прежними:



```
<terminated> 111.exe [C/C++ Application] C:\Work\Android_  
a=10 b=20
```



В языке Java переменные примитивных типов передаются в функцию всегда по значению

Передача по ссылке

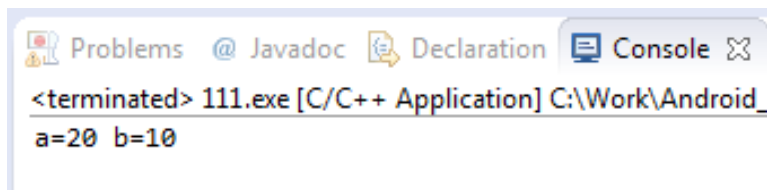
Этот способ передачи параметров в функцию позволяет получить задуманные в предыдущем примере результаты – поменять местами значения переменных. И здесь нам помогут указатели!

Если в качестве параметров передать адреса переменных, то путем использования операции разыменования можно изменить исходные переменные:

```
#include <stdio.h>  
void sort(int* pa, int* pb)  
{  
    int c;  
    if(*pa < *pb)  
    {  
        c = *pa;  
        *pa = *pb;  
        *pb = c;  
    }  
}  
int main(void)  
{  
    int a, b;  
    a = 10;  
    b = 20;  
    sort(&a, &b);  
    printf("a=%d b=%d\n", a, b);  
    return 1;  
}
```



В результате работы программы мы получим желаемый результат - значения переменных `a` и `b` поменяются местами:



```
<terminated> 111.exe [C/C++ Application] C:\Work\Android_  
a=20 b=10
```

1.10.7. Массивы в качестве параметров функции

Для того, чтобы передать в функцию массив или переменную сложного типа (структуру) в Си также используется передача по ссылке:

```
#include <stdio.h>
```

```
/* функция переворачивает массив заданного размера */
```

```
void reverse(int* pA, int n){
```

```
    int c;
```

```
    /*установить на посл.элемент массива*/
```

```
    int *p = pA + n - 1;
```

```
    while (pA<p){
```

```
        c=*pA; *pA = *p;*p = c;
```

```
        pA++;
```

```
        p--;
```

```
    }
```

```
}
```

```
int main(void){
```

```
    const int n=5;
```

```
    int i, A[n]={1, 2, 3, 4, 5};
```

```
    reverse(A,n);
```

```
    for(i=0;i<n;i++)
```

```
        printf("A[%d]=%d\n", i, A[i]);
```

```
    return 1;
```

```
}
```



Область данных функции reverse()



A- адрес начала массива

Область данных функции main()

Работа через указатели позволяет изменить значения исходного массива A:

Problems

<terminated> 1:

A[0]=5

A[1]=4

A[2]=3

A[3]=2

A[4]=1

1.10.8. Динамическое выделение памяти

В тех ситуациях, когда при написании программы неизвестно, какого размера память нам необходима для работы программы, но предполагается, что это станет известно в процессе выполнения программы, в языке Си предусмотрен механизм динамического выделения памяти.

Используем функцию *reverse()* из предыдущего примера. Пусть в функции *main()* создадим динамический массив целых чисел того размера, который вводит пользователь в переменную *n*. Т.е. мы создаем массив необходимого размера по ходу работы программы, а не заранее.

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```

void main(void){
    int n;
    int i, *p, *pA;
    //ввод с консоли размера массива
    scanf("%d",&n);
    pA= (int*)malloc(sizeof(int) * n); // создание динамического массива
    p = pA;
    //заполнение массива случ. числами от 0 до 9
    for(i=0;i<n;i++){
        *p=(int)rand() % 10;
        p++;
    }
    for(i=0;i<n;i++)
        printf("%d ",pA[i]);
    reverse(pA,n);
    printf("\n");
    for(i=0;i<n;i++)
        printf("%d ",pA[i]);
    free(pA); // освобождение памяти
    pA = NULL;
}

```

Результат работы программы:

```

<terminated> 111.exe [C/C++ Application] C:\Work\Android_work
10
3 3 2 9 0 8 2 6 6 9
9 6 6 2 8 0 9 2 3 3

```

Обратите внимание на выделенные строки программы:

- Функция malloc() выделяет память в динамической памяти (куче) заданного размера (в байтах);
- Функция free() освобождает память;
- указатель pA обнуляется.

Последнее необходимо, потому что после вызова free() переменная pA хранит указатель на уже несуществующую переменную (получается «висячая ссылка»). Считается правилом хорошего тона такой указатель обнулять.

Каждому созданию динамической переменной с помощью malloc() должен соответствовать свой вызов free(). Если этого не соблюсти, то возникнет **утечка памяти** - ситуация, когда выделенная память уже не нужна, но и не освобождена. В языке Си необходимо отслеживать соответствие каждому malloc() своего free(). Иначе программа в ходе своей работы может занять неоправданно много ресурсов памяти.



О синтаксисе функций Си, используемых при работе с динамической памятью

Этот материал предназначен для тех, кто уже знаком с языком Си и хочет разобраться более глубоко в приведенной выше программе

void *malloc(size_t size)

функция возвращает значение void*. Для того, чтобы присваивание было

произведено корректно необходимо преобразование типа указателя. В примере: (int)*
size_t size – параметр, задающий объем необходимой памяти в байтах.

***sizeof(int)** – оператор определения размера памяти, отводимого под одну переменную заданного типа данных*

void free(void *ptr)

функция для освобождения памяти, на которую указывает параметр ptr

1.10.9. Указатели в Си и ссылки в Java

С ссылками в Java мы уже неоднократно сталкивались, когда работали с массивами. С учетом полученных в этой теме знаний, рассмотрим этот вопрос подробнее.

В предыдущих темах мы не раз использовали new:

```
Scanner in = new Scanner(System.in);
in.nextInt();
```

```
int[] vas_ar = new int[10];
```

Каждый раз оператор **new** выделял в памяти место под объект/массив и затем возвращал ссылку на эту область памяти. Это значит, что в выше приведенных примерах после вызова new переменные **in** и **vas_ar** содержат адрес области памяти для хранения объекта/массива.

В языке Java, несмотря на то, что нет понятия “указатель”, на этом механизме строится вся работа с массивами и объектами.

Например, в Java можно написать так:

```
int[] arr = { 1, 2, 3, 4, 5 };
System.out.println(arr[0]);
arr= new int[30];
System.out.println(arr[0]);
```

В результате будет выведено :

```
1
0
```

В этом примере переменная-массив **arr** сначала хранила указатель на массив чисел от 1 до 5, а затем вызовом new ей был присвоен указатель на новый массив из 30 int, которые были автоматически проинициализированы значением 0. Т.е. переменные-массивы (а также объекты) по сути своей являются указателями.

При передаче параметров в методы в Java часто говорят, что примитивные переменные передаются по значению, а массивы и объекты - по ссылке.

Строго говоря, в Java все параметры в методы передаются по значению. А то, что называют **передачей по ссылке** - это передача адреса массива или объекта по значению: в метод передается указатель на массив или объект в памяти, который затем копируется в параметры

функции, и тем самым мы получаем возможность работать с исходным массивом либо через методы объекта менять его (если это предусмотрено интерфейсом соответствующего класса).

Пример:

```
public class Example1 {

    static void changeArray(int[] a) {
        a[0] = 100;
    }

    public static void main(String[] argv) {
        int[] arr = { 1, 2, 3, 4, 5 };
        System.out.println("Befor changing : " + arr[0]);
        changeArray(arr);
        System.out.println("After changing : " + arr[0]);
    }

}
```

Получаем результат:

Befor changing : 1

After changing : 100

Передача объектов в методы более подробно будет рассмотрена в Модуле 2.

В заключение приведем краткую таблицу с сравнением реализаций указателей в Си и ссылок на объекты в Java

	Указатели в Си	Ссылки на объекты и массивы в Java
Адресная арифметика	Возможны операции адресной арифметики	Невозможны
Передача параметров в функцию по ссылке	Можно передавать указатели на любые типы данных	Нельзя передать ссылки на примитивные типы, только на объекты классов и массивы. Функция не может изменить полученную ей ссылку. Однако она может вызвать метод объекта, изменяющий поля этого объекта
Выделение памяти	Используется функция <i>malloc()</i>	Используется оператор <i>new</i>
Освобождение памяти	Используется функция <i>free()</i> Необходимо отслеживать утечки памяти	Память освобождается автоматически специальным процессом, называемым <i>сборщик мусора</i> (англ. <i>garbage collector</i>)

Задание 1.10.1

Рассмотрим пример программы на Java, которая использует функции, похожие на те, что были рассмотрены в предыдущей теме:

```
// Это на Java!
public class Example1 {

    // Создаем и заполняем массив заданного размера и диапазона
    static void randArray(int[] ar, int range) {

        for (int i = 0; i < ar.length; i++) {
            ar[i] = (int) (Math.random() * (range + 1));
        }

    }

    // Вывод массива на консоль
    static void printArray(int[] a) {

        for (int i = 0; i < a.length; i++) {
            System.out.print(a[i] + " ");
        }

    }

    public static void main(String[] args) {

        // Создаем массив vas_ar из 10 чисел 0..100
        int[] vas_ar = new int[10];
        int range = 20;
        randArray(vas_ar, range);
        // Выводим полученный массив на экран
        printArray(vas_ar);

    }

}
```

Метод `randArray` определен так, что в качестве параметров он ждет адрес массива и целое число, задающее диапазон генерации элементов массива. При вызове `randArray` в качестве фактических параметров мы подставили результат работы оператора `new int[10]` и переменную `range`. После этого, работает метод `randArray`, и заполняет массив случайными значениями от 0 до `range`. При этом метод имеет тип `void`, т.е. явно массив не возвращается в качестве результата работы метода.

Ответьте на следующие вопросы и проверьте ответ путем запуска программ:

1. Изменится ли массив `vas_ar` после вызова `randArray()`? Будет ли методом `printArray()` выведен массив с сгенерированными значениями?
2. Изменим метод `randArray` приведенным ниже образом. Изменится ли переменная `range` в методе `main()` после вызова `randArray()`? Почему?

```
static void randArray(int[] ar, int range) {  
  
    for (int i = 0; i < ar.length; i++) {  
        ar[i] = (int) (Math.random() * (range + 1));  
    }  
    range = 0; // Изменим переменную  
}
```

3. Каким способом (по значению или по ссылке) были переданы параметры в метод *randArray*?

Задание 1.10.2

В приведенных фрагментах программ на Си определить изменится ли значение переменной после выполнения функции.

*Проверить ответ: дополнить программы выводом результата на консоль, отладить и запустить.

1)

```
void inc(int a){  
    a++;  
}  
void main(){  
    int a=1;  
    inc(a); //Изменится ли значение a?  
}
```

2)

```
void swap(int a, int b){  
    int t;  t=b; b=a; a=t;  
}  
  
void main(){  
    int p=3,q=5;  
    swap(p,q); //Изменяются ли значения переменных p,q?  
}
```

3)

```
void dec(int* a){  
    (*a)--;  
}  
  
void main(){  
    int a=1;  
    dec(&a); //Изменится ли значение a?  
}
```

4)

```
void sum(int a[]){  
    a[0]=a[1]+a[2];  
}  
void main(){  
    int m[3]={0, 20, 30};  
    sum(m); //Изменится ли массив m?  
}
```