

## Лекция 9

Указатели. Ссылки.

Функции, объявление и определение.

Параметры функции.

Аргументы в командной строке.

# Указатели и адреса

**Указатель** – это группа ячеек, в которых может храниться адрес.

Различают **три вида указателей**

- указатели на объект,
- указатели на функцию
- указатели на **void**.

Указатель не является самостоятельным типом, он всегда связан с каким-либо другим конкретным типом.



**Указатель на объект** содержит адрес области памяти, в которой хранятся данные определенного типа. Объявление указателя на объект (далее просто указателя) имеет вид:

**тип \*имя;**

Например, в операторе

**int \*x, y, \*z;**

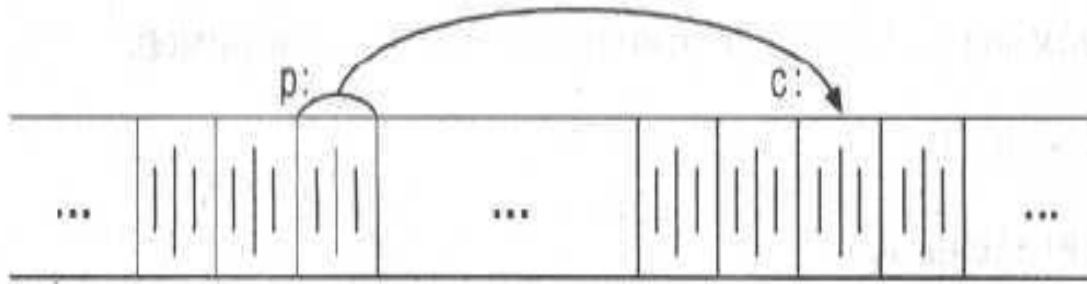
описываются два указателя на целое – **x**, **z**, а также целочисленная переменная **y**.

# Указатели и адреса

Пусть имеются описания: `char c;`      `char *p;`

**Унарный оператор &** определяет адрес объекта, так что инструкция  
`p = &c;`

присваивает переменной `p` адрес ячейки `c` (говорят, что `p` указывает на `c`).



**Оператор &** применяется только к объектам, расположенным в памяти: к переменным и элементам массивов. Его операндом не может быть ни выражение, ни константа, ни регистровая переменная.

# Указатели и адреса

**Унарный оператор `*`** – оператор *разадресации*.

Примененный к указателю, он выдает объект, на который данный указатель указывает.

Пусть `x` и `y` имеют тип `int`, а `ip` – указатель на `int`. Тогда

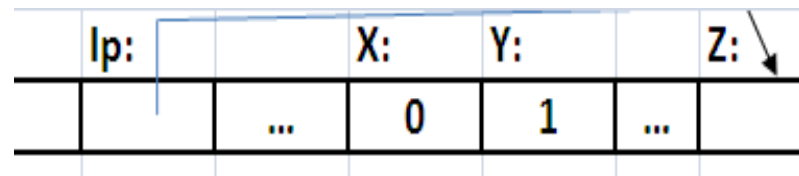
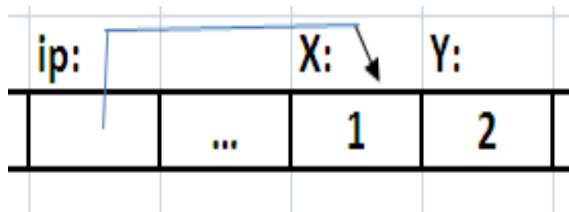
```
int x = 1, y = 2, z[10], *ip;
```

```
ip = &x;           /* теперь ip указывает на x */
```

```
y = *ip;           /* y теперь равен 1 */
```

```
*ip = 0;            /* x теперь равен 0 */
```

```
ip = &z[0];         /* ip теперь указывает на z[0] */
```



Объявление указателя `int *ip` гласит: "выражение `*ip` имеет тип `int`".

Указанный принцип применим и в объявлениях функций.

Например, запись `double *dp, atof (char *)`;

означает, что выражения `*dp` и `atof(s)` имеют тип `double`, а аргумент функции `atof` есть указатель на `char`.

# Указатели и адреса

Указателю разрешено указывать только на объекты определенного типа. Существует одно исключение: "**указатель на void**" может указывать на объекты любого типа, но **к такому указателю нельзя применять оператор разадресации.**

Если **ip** указывает на **x** целочисленного типа, то **\*ip** можно использовать в любом месте, где допустимо применение **x**;  
например,

```
*ip = *ip + 10;
```

увеличивает **\*ip** на 10.

Оператор **\*ip += 1;** увеличивает на единицу то, на что указывает **ip**; те же действия выполняют

```
++*ip;
```

и **(\*ip)++;**

# Указатели и аргументы функций

Поскольку в Си функции в качестве своих аргументов получают значения параметров, нет прямой возможности, находясь в вызванной функции, изменить переменную вызывающей функции.

```
void swap(int x, int y) {  
    int temp;  
    temp = x;  
    x = y;  
    y = temp;  
}
```

Поскольку `swap(a,b)` получает лишь *копии* переменных **a** и **b**, она не может повлиять на переменные **a** и **b** той программы, которая к ней обратилась.

# Указатели и аргументы функций

Чтобы получить желаемый эффект, вызывающей программе надо передать *указатели* на те значения, которые должны быть изменены:

```
swap(&a, &b);
```

В самой же функции **swap** параметры должны быть объявлены как указатели:

```
void swap(int *px, int *py) { // перестановка *px и *py
    int temp;
    temp = *px;
    *px = *py;
    *py = temp;
}
```

# Ссылки

**Ссылки** – это особый тип данных, являющийся скрытой формой указателя, который при использовании автоматически разыменовывается. Формат объявления ссылки:

***тип & имя\_ссылки = имя\_переменной;***

где тип – это тип величины, на которую указывает ссылка,

**&** - оператор ссылки, означающий, что следующее за ним имя является именем переменной ссылочного типа, например:

```
int x;
```

```
int & p = x; // ссылка p - альтернативное имя для x
```

```
const char & CR = '\n'; // ссылка на константу.
```

Любое изменение значения ссылки повлечет за собой изменение того объекта, на который данная ссылка указывает:

```
int i=0;  int &r=i;  r+=10; // то же, что i+=10;
```

После выполнения приведенного фрагмента значения переменных **i** и **r** будет равно 10.



# Ссылки

Использование ссылок не связано с дополнительными затратами памяти.

Ссылки нельзя переназначать – инициализировав ссылку однажды адресом одной переменной, любое действие над ссылкой сказывается на самом объекте. Попытка переназначить имеющуюся ссылку какой-либо другой переменной приведет к присвоению оригиналу объекта значения другой переменной:

```
char letA='A';  
char &refA=letA;  
char letB='B';  
refA=letB; // то есть letA=letB
```

Ссылки не могут указывать на нулевой объект (принимающий значение **NULL**). То есть, если есть вероятность того, что объект в результате работы станет нулевым, то от ссылки следует отказаться в сторону применения указателя.

## Правила при работе со ссылками:

- Переменная-ссылка должна явно инициализироваться при ее описании, кроме случаев, когда она является параметром функции или описана как **extern**, или ссылается на поле данных класса.
- После инициализации ссылке **не может** быть присвоена другая переменная.
- Тип ссылки должен совпадать с типом величины, на которую она ссылается.
- **Не разрешается** определять указатели на ссылки, создавать массивы ссылок и ссылки на ссылки.

Ссылки применяются чаще всего в качестве параметров функций и типов возвращаемых функциями значений. Ссылки позволяют использовать в функциях переменные, передаваемые по адресу, без операции разадресации, что улучшает читаемость программы.

Ссылка, в отличие от указателя, не занимает дополнительного пространства в памяти и является просто другим именем величины. Операция над ссылкой приводит к изменению величины, на которую она ссылается.

# Функции, объявление и определение

**Функция** — это именованная последовательность описаний и операторов, выполняющая какое-либо законченное действие.

Функция может принимать параметры и возвращать значение.

Любая программа на C++ состоит из функций, одна из которых должна иметь имя **main** (с нее начинается выполнение программы). Функция начинает выполняться в момент вызова.

Любая функция должна быть объявлена и определена. Как и для других величин, объявлений может быть несколько, а определение только одно.

**Объявление функции** должно находиться в тексте раньше ее вызова для того, чтобы компилятор мог осуществить проверку правильности вызова. **Объявление функции (прототип, заголовок, сигнатура)** задает ее имя, тип возвращаемого значения и список передаваемых параметров.

**Определение функции:**

```
[класс] тип имя ([список_параметров]) [throw (исключения)]  
{  
    тело функции  
}
```

# Функции, объявление и определение

*Определение функции* содержит, кроме объявления, тело функции, представляющее собой последовательность операторов и описаний в фигурных скобках.

С помощью необязательного модификатора класс можно явно задать область видимости функции, используя ключевые слова **extern** и **static**:

**extern** — глобальная видимость во всех модулях программы (по умолчанию);

**static** — видимость только в пределах модуля, в котором определена функция.

Тип возвращаемого функцией значения может быть любым, кроме массива и функции (но может быть указателем на массив или функцию). Если функция не должна возвращать значение, указывается тип **void**.

Список параметров определяет величины, которые требуется передать в функцию при ее вызове. Элементы списка параметров разделяются запятыми. Для каждого параметра, передаваемого в функцию, указывается его тип и имя (в объявлении имена можно опускать). Об исключениях, обрабатываемых функцией, рассказывается в ООП (2 курс).

# Функции, объявление и определение

Пример 1

```
#include<stdio.h>
int f1(int a, int b) {//определение функции f1
    int i=a+b;
    return i*i;
}

void f2(int a,float b) {//определение функции f2
    printf("\na=%d b=%f",a,b) ;
    return;
}

int main() {
    int x=2,y;
    y=f1(x,3); //вызов функции f1
    printf("\ny=%d",y) ;
    f2(3,4.5f); //вызов функции f2
    return 0;
}
```

# Функции, объявление и определение

Пример 2

```
#include<stdio.h>
int f1(int a, int b);           //объявление функции f1
void f2(int ,float );          //объявление функции f2

int main() {
    int x=2,y;
    y=f1(x,3);                  //вызов функции f1
    printf("\ny=%d",y);
    f2(3,4.5f);                 //вызов функции f2
    return 0;
}

void f2(int ,float );           //объявление функции f2
int f1(int a, int b) {          //определение функции f1
    int i=a+b;
    return i*i;
}

void f2(int a,float b) {        //определение функции f2
    printf("\na=%d b=%f",a,b);
    return;
}
```

# Функции, объявление и определение

В определении, в объявлении и при вызове одной и той же функции типы и порядок следования параметров должны совпадать. На имена параметров ограничений по соответствию не накладывается, поскольку функцию можно вызывать с различными аргументами, а в прототипах имена компилятором игнорируются (они служат только для улучшения читаемости программы).

Тип возвращаемого значения и типы параметров совместно определяют тип функции. Для вызова функции в простейшем случае нужно указать ее имя, за которым в круглых скобках через запятую перечисляются имена передаваемых аргументов. Вызов функции может находиться в любом месте программы, где по синтаксису допустимо выражение того типа, который формирует функция. Если тип возвращаемого функцией значения не **void**, она может входить в состав выражений или, в частном случае, располагаться в правой части оператора присваивания.

# Функции, объявление и определение

**Формальные параметры** описываются в определении функции.

**Фактические параметры** – при вызове функции.

Типы формальных и фактических параметров должны совпадать, а имена могут различаться.

В приведенном выше примере:

```
#include<stdio.h>
int f1(int a, int b) { //определение функции f1
    int i=a+b;
    return i*i;
}
void f2(int a, float b) { //определение функции f2
    printf("\na=%d b=%f", a, b);
    return;
}
int main() {
    int x=2, y;
    y=f1(x, 3); //вызов функции f1
    printf("\ny=%d", y);
    f2(3, 4.5f); //вызов функции f2
    return 0;
}
```

**формальные параметры**

**фактические параметры**

**фактические параметры**



# Функции, объявление и определение

Все величины, описанные внутри функции, а также ее параметры, являются локальными. Областью их действия является функция. При вызове функции, как и при входе в любой блок, в стеке выделяется память под локальные автоматические переменные. Кроме того, в стеке сохраняется содержимое регистров процессора на момент, предшествующий вызову функции, и адрес возврата из функции для того, чтобы при выходе из нее можно было продолжить выполнение вызывающей функции. При выходе из функции соответствующий участок стека освобождается, поэтому значения локальных переменных между вызовами одной и той же функции не сохраняются.

Все идентификаторы, объявленные в блоках функций, являются **локальными**, т.е. они не видимы во внешних блоках. Кроме локальных идентификаторов, внутри функций могут быть видны некоторые идентификаторы, объявленные во внешних блоках. Такие идентификаторы называют **глобальными**.

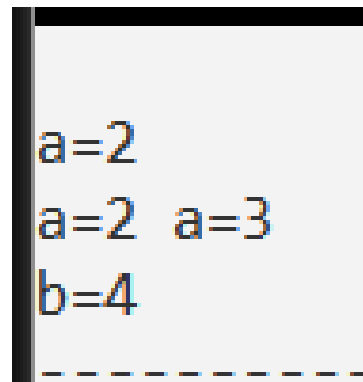
Имя, локализованное в подпрограмме, может совпадать с глобальным именем, т.к. им назначаются разные области памяти. В этом случае локальное имя **перекрывает** глобальное и делает его недоступным. Область видимости совпадает с областью действия за исключением ситуации, когда во вложенном блоке описана переменная с таким же именем. Доступ к глобальному имени можно получить с помощью операции ::.

# Функции, объявление и определение

```
#include<stdio.h>
int a;
int f1() { //...
}

int f2() { //...
}

int main() {
    int b=2;
    a = 1;
    int a;
    a = 2;
    printf("\na=%d",a);
    ::a = 3;
    printf("\na=%d",a);
    printf(" a=%d", ::a);
    b+=a;
    printf("\nb=%d",b);
}
```



```
a=2
a=2 a=3
b=4
-----
```

# Функции, объявление и определение

Пример использования глобальной переменной:

```
#include<stdio.h>
int a;
void f1() { // Определение функции f1()
    a+=1; printf("\na=%d",a) ; //b+=2;
}
void f2() { // Определение функции f2()
    a+=1; printf("\na=%d",a) ; //b+=2;
}
int main() {
    int b=2;
    a = 1;
    f1() ; // Вызов функции f1()
    f2() ; // Вызов функции f2()
    printf("\na=%d",a) ;
}
```

a=2

a=3

a=3

---

Область видимости переменной **a** – вся программа.

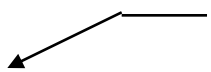
**Мораль: все переменные нужно локализовать!**

# Механизм передачи параметров и возврата значения

```
#include<stdio.h>
void inc(int b) {
    b+=1; printf("\nb=%d",b); return;
}

int dec(int a) {
    a-=1; printf("\na=%d",a);
    return a;
}
```

Список формальных  
параметров



```
int main() {
    int b=2;
    inc(b);
    printf("\nb=%d",b);
    b=dec(b);
    printf("\nb=%d",b); return 0;
}
```

Список фактических  
параметров



b=3

b=2

a=1

b=1

-----

Функция **inc** ничего не возвращает, поэтому тип **void**.

Функция **dec** возвращает целое значение, поэтому тип **int**.

# Виды параметров функций

При совместной работе функции должны обмениваться информацией. Это можно осуществить

- с помощью глобальных переменных;
- через возвращаемое функцией значение (**return**);
- через параметры.

Параметры бывают четырех видов:

- 1) параметры-значения;
- 2) параметры-переменные;
- 3) параметры константы.

Отличие между параметрами заключается в способе обмена данными между вызывающей программой и подпрограммой.

# Параметры значения

Рассмотрим на примере:

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

```
void my_fun(int a, int b, int c) {  
    a++;  
    b++;  
    c=c+1;  
    printf("\n a=%d b=%d c=%d",a,b,c) ;  
}
```

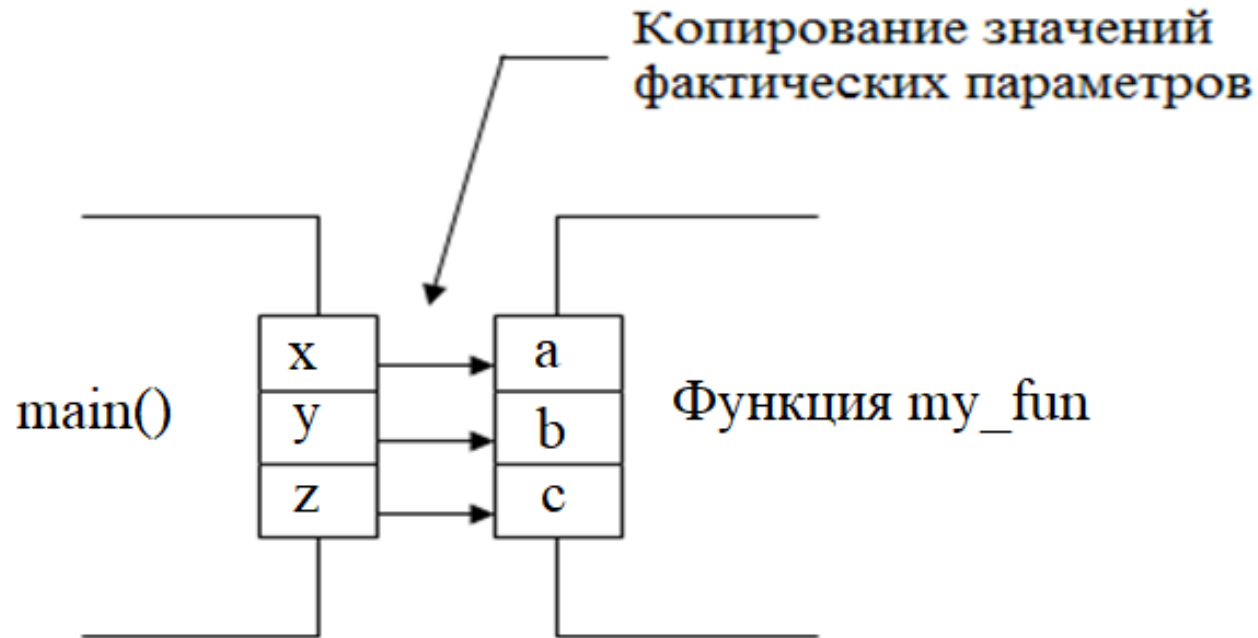
```
int main() {  
    int x=1,y=1,z=1;  
    my_fun(x,y,z) ;  
    printf("\nx=%d y=%d z=%d",x,y,z) ;  
}
```

```
a=2 b=2 c=2  
x=1 y=1 z=1
```

# Виды параметров функций

## 1. Параметры – значения

Механизм передачи параметров-значений можно пояснить с помощью рисунка:



# Виды параметров функций

Передача параметров-значений в функцию выполняется по следующим правилам:

1) основная функция **main()** отводит место в памяти компьютера для хранения значений фактических параметров: **x, y, z**;

2) в момент вызова функция **my\_fun** отводит место в памяти для хранения формальных параметров **a, b, c** в соответствии с их описанием;

3) в этот же момент происходит передача фактических значений в функцию (**копирование** значений фактических параметров в область памяти, где размещаются формальные параметры);

4) после этого связь между **main()** и функций **my\_fun** разрывается.

В функции можно изменять формальные параметры-значения, однако соответствующие им фактические параметры останутся без изменений.

**При вызове функции вместо параметров-значений можно записывать выражения. Например:**

```
fun3 (2 + x, 0.5, 3) ;
```

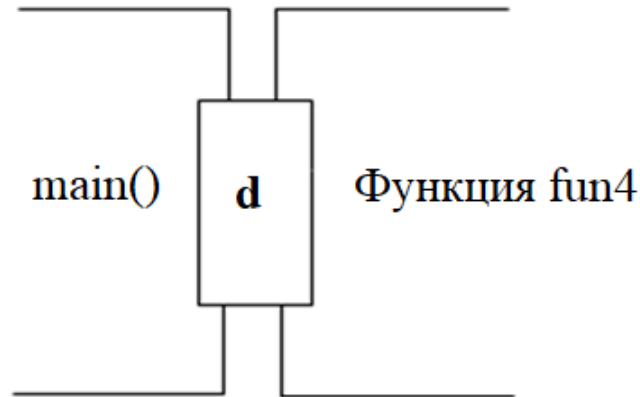


# Виды параметров функций

**2. Параметры-переменные.** Перед такими параметрами в списке формальных параметров записывают &.

Например: `void fun4(int &d) ;`

Механизм передачи параметра-переменной можно пояснить с помощью рисунка:



В этом случае `main()` и функция `fun4` для хранения параметра-переменной использует одну и ту же ячейку памяти. Для этого функции передается ссылка (адрес) на область памяти, где располагается фактический параметр.

**Фактический параметр для этого должен быть переменной!** Все действия над формальным параметром-переменной представляют собой, по сути, действия над фактическим параметром.

# Виды параметров функций

## 3 Параметры-константы

Перед параметром-константой записывается **const**.

В этом случае в подпрограмму передается ссылка (адрес) на область памяти, где располагается фактический параметр. Однако, компилятор блокирует любые присваивания параметру-константе нового значения в теле подпрограммы.

.

# Виды параметров функций (C++)

```
#include<stdio.h>
void my_fun(int a, const int b, int &c) {
    a++;
    //b++;
    c=c+1;
    printf("\n a=%d b=%d c=%d",a,b,c);
}

int main() {
    int x=1,y=1,z=1;
    my_fun(x,y,z);
    printf("\nx=%d y=%d z=%d",x,y,z);
}
```

```
a=2 b=1 c=2
x=1 y=1 z=2
```

Первый параметр-значение, второй параметр-константа, третий параметр-переменная.

# Виды параметров функций (C)

```
#include<stdio.h>

void my_fun(int a, const int b, int *c) {
    a++;
    //b++;
    *c=*c+1;
    printf("\n a=%d b=%d c=%d",a,b,*c);
}

int main() {
    int x=1,y=1,z=1;
    my_fun(x,y,&z);
    printf("\nx=%d y=%d z=%d",x,y,z);
}
```

```
a=2 b=1 c=2
x=1 y=1 z=2
```

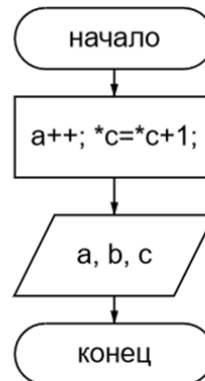


Рисунок 1 – Структурная схема функции `myfun(x,y,&z)`

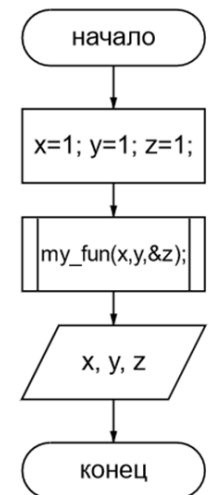


Рисунок 2 – Структурная схема основной функции

# Параметры со значениями по умолчанию

Чтобы упростить вызов функции, в ее заголовке можно указать значения параметров по умолчанию. Эти параметры должны быть последними в списке и могут опускаться при вызове функции. Если при вызове параметр опущен, должны быть опущены и все параметры, стоящие за ним. В качестве значений параметров по умолчанию могут использоваться константы, глобальные переменные и выражения:

```
#include<stdio.h>
int f1(int a, int b = 0);
void f2(int, int = 100, char = 'S');

int main() {
    int a=5;
    f1(100); f1(a, 1); // варианты вызова функции f1
    f2(a); f2(a, 10); f2(a, 10, 'x'); // варианты вызова f2
    // f2(a, 'x'); // неверно!
}
```

# Функции с переменным числом параметров

Если список формальных параметров функции заканчивается многоточием, это означает, что при ее вызове на этом месте можно указать еще несколько параметров.

Проверка соответствия типов для этих параметров **не выполняется**, **char** и **short** передаются как **int**, а **float** – как **double**.

В качестве примера можно привести функцию **printf**, прототип которой имеет вид:

```
int printf (const char*, ...);
```

Это означает, что вызов функции должен содержать по крайней мере один параметр типа **char\*** и может либо содержать, либо не содержать другие параметры:

```
printf("Введите исходные данные"); // один параметр  
printf("Сумма: %5.2f рублей", sum); // два параметра  
printf("%d %d %d %d", a, b, c, d); // пять параметров
```

# Аргументы в командной строке

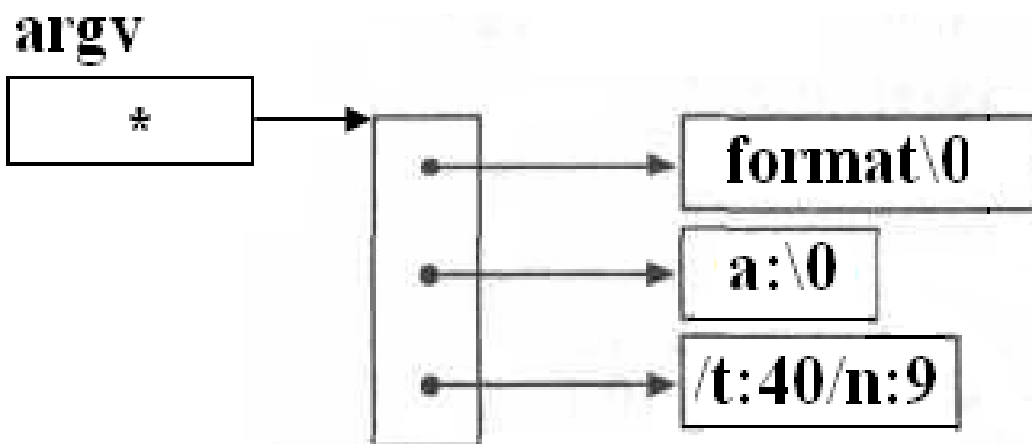
В момент вызова **main** получает два аргумента.

В первом, обычно называемом **argc**, содержится количество аргументов, задаваемых в командной строке.

Второй **argv** является указателем на массив литерных указателей.

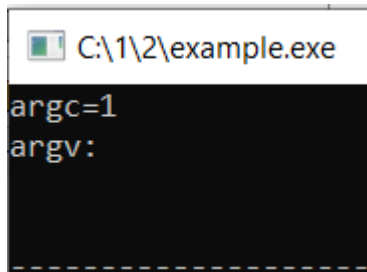
**argv[0]** – имя программы; **argc**  $\geq 1$

Команда: **format a: /t:40/n:9**

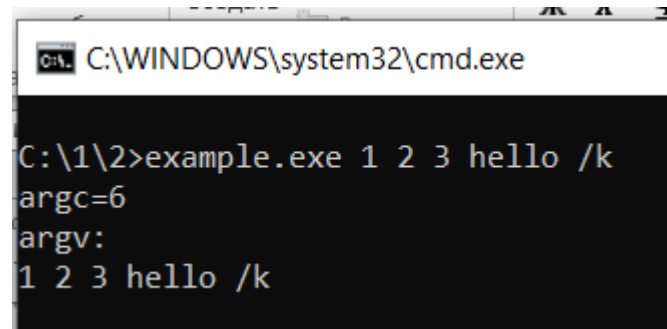


# Аргументы в командной строке

```
#include <stdio.h>
/* печать аргументов командной строки: */
main(int argc, char *argv[]) {
    int i;
    printf("argc=%d \n", argc);
    printf("argv: \n");
    for (i = 1; i < argc; i++)
        printf("%s%s", argv[i], (i < argc-1) ? " " : "");
    printf("\n");
    return 0;
}
```



```
C:\1\2\example.exe
argc=1
argv:
-----
```



```
C:\WINDOWS\system32\cmd.exe
C:\1\2>example.exe 1 2 3 hello /k
argc=6
argv:
1 2 3 hello /k
```

Аргумент **argv** – указатель на начало массива строк аргументов.

Поэтому инструкцию **printf** можно было бы написать и так:

```
printf((argc > 1) ? "%s_" : "%s", *++argv);
```

Как видим, формат в **printf** тоже может быть выражением.