



# Функции и переменные

Основы языка C, лекция 2

# Повторяем: переменная

- Переменная — именованная область памяти

- `int x;`

`int y;`

`x = 2;`

`y = 5;`



- `int x, y;`

`// через запятую`

`x = 2;`

`y = 5;`

- `int x = 2, y = 5;`

`// явная инициализация`

# Повторяем: сложение чисел

- `#include <stdio.h>`

```
int main ( ) {
```

```
    int x, y, res;           // декларация переменных
```

```
    scanf("%d", &x);        // ввод данных
```

```
    scanf("%d", &y);
```

```
                                // обработка данных
```

```
    res = x + y;
```

```
                                // вывод результатов
```

```
    printf("%d plus %d is %d\n", x, y, res);
```

```
    return 0;
```

```
}
```

# Повторяем: типы данных

- Целочисленные типы данных
  - char            %hhd
  - short           %hd
  - int             %d
  - long            %ld
  - long long    %lld
- unsigned, *signed*    %d → %u
- данные с плавающей точкой
  - float        %f %g %e
  - double    %lf

# Что такое функция

- Математика: отображение множества входных аргументов на множество результатов.

$$y = \sin(x) \quad y = x^k$$

- Программирование: именованный кусок кода. Фрагмент кода, к которому можно *обратиться (call)* из другого места программы.

`scanf`, `printf` — уже вызывали стандартные функции

- У каждой функции есть идентификатор — имя функции.

# Зачем нужны функции

- Модульность кода — разбиение 1 задачи на много маленьких задач — наша цель
  - повторное использование кода
    - уже написано
    - меньше писать
    - легче отлаживать
  - скрывает несущественные для других частей детали реализации (printf, scanf, qsort)
    - можно переписать 1 функцию
  - взять из библиотеки / написать самому
    - уже сделана, быстрые алгоритмы

# Математические функции

- `#include <stdio.h>` `// ex_sin.c`  
`#include <math.h>`  
`int main( ) {`  
    `double x, y;`  
    `x = M_PI;`  
    `y = sin(x) + 2*cos(x);`  
    `printf("%ld\n", y);`  
    `return 0;`  
`}`
- `gcc -Wall -Wextra -lm -o ex_sin ex_sin.c`  
Указать, что нужно линковать библиотеку стандартных математических функций

# man 3 sin

```
tatyderb@nearbird: /mnt/c/work/conline/1_int/text
SIN(3) Linux Programmer's Manual SIN(3)

NAME
    sin, sinf, sinl - sine function

SYNOPSIS
    #include <math.h>

    double sin(double x);
    float sinf(float x);
    long double sinl(long double x);

    Link with -lm.

    Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

    sinf(), sinl():
        _ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L
        || /* Since glibc 2.19: */ _DEFAULT_SOURCE
        || /* Glibc versions <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE

DESCRIPTION
    These functions return the sine of x, where x is given in radians.

RETURN VALUE
    On success, these functions return the sine of x.

    If x is a NaN, a NaN is returned.

Manual page sin(3) line 1/65 33% (press h for help or q to quit)
```



# Создадим свою функцию foo

- `#include <stdio.h>`

`// axx + bx + c — реализовали функцию foo:`

```
float foo (float a, float b, float c, float x) {  
    float res;  
    res = a*x*x + b*x + c;  
    return res;  
}
```

```
int main ( ) {  
    float y1, y2, y3;  
    y1 = foo (1, 2, 3, 0.5);           // call foo  
    y2 = foo (-3.1, 5.16, -0.01, 10); // call foo  
    printf("%f %f\n", y1, y2);  
}
```

# `y = foo(1, 2, 31, 4.5);`

- Создаются переменные `a`, `b`, `c`, `x` и в них записываются значения `1`, `2`, `31`, `4.5`
  - неявное преобразование типа
- управление передается в функцию  
выполняются одна за другой инструкции  
`float res;`  
`res = a*x*x + b*x + c;`
- когда выполнение доходит до **`return res;`**  
выход из функции в точку вызова
- Значение выражения `foo(1, 2, 31, 4.5)` — что вернул оператор `return`

# Создание своей функции

- тип имя\_функции (список аргументов) {  
    тело функции  
}
- **имя\_функции** — идентификатор (как придумать?)
- **тип** — тип возвращаемого значения **void**
- **список аргументов**: через запятую или пустой
  - для каждого аргумента — свой тип и имя (придумать самим)
- **return** — возвратиться из функции
  - может стоять в любом месте функции

# void — нечего возвращать

- #include <stdio.h>

```
void hi ( ) {  
    printf("Здравствуй, уважаемый \n");  
    return ;           // это можно не писать  
}  
  
int main ( ) {        // main — тоже функция  
    hi ( );  
    hi ( );  
    hi ( );  
    hi ( );  
    hi ( );  
    return 0;         // кто вызвал main и кому вернули 0?  
}
```

# Безграмотность

- **void** main ( ) { ... }
  - по такой книжке нельзя учиться, она плохая
  - это не язык C
-

# аргумент group

- #include <stdio.h>

```
void hi ( int group ) {  
    printf("Здравствуйте, группа %d!\n", group);
```

```
}
```

```
int main ( ) {  
    hi ( 951 ); // Здравствуйте, группа 951!  
    hi ( 954 ); // Здравствуйте, группа 954!  
    hi ( 978 ); // Здравствуйте, группа 978!  
    hi ( 916 ); // Здравствуйте, группа 916!  
    hi ( );      // ошибка, мало аргументов  
    return 0;
```

```
}
```

# Запрещено

- Функции с одинаковыми именами
- Значения по умолчанию
- *Переменное число аргументов — позже printf*
- Реализация функции внутри другой функции (pascal, python, ...)

```
int main ( ) {  
    void hi ( ) {  
        printf("Здравствуй");  
    }  
    // дальше идет функция main  
    hi ( );  
}
```

# ИМЕНА И ТИПЫ

- Имя функции должно быть значимым
- 1 функция — 1 действие
  - get\_font, set\_size, draw\_line, to\_upper
  - is\_empty, ~~is\_not\_empty~~
- Если не можете назвать функцию, разбейте ее на несколько мелких функций
- тип — у каждого аргумента  
long power(**int** x, **int** n)      // ok  
long power(**int** x, n)            // error



# Нет дара предвидения

- ДО вызова функции нужна информация компилятору о:
  - количестве аргументов и типе каждого аргумента
  - типе возвращаемого значения
- Сначала вас учат,  
потом вы используете знания

# Какую функцию раньше?



```
void foo ( ) {
```

```
    ...
```

```
    bzz( );           // что такое bzz ???
```

```
    ...
```

```
}
```

```
void bzz ( ) {
```

```
    ...
```

```
    foo ( );
```

```
    ...
```

```
}
```

# Прототип функции

- Информация компилятору об:
  - имени функции,
  - возвращаемом типе,
  - количестве аргументов и их типах
- Вместо **{ тело функции }** пишем **;**
- ```
void bzz();    // прототип функции bzz (prototype)
void foo( ) {
    bzz( );    // вызов bzz (call)
}
void bzz( ) {  // реализация bzz (implementation)
    foo( );
}
```


# Прототип и реализация

- // Прототип  
тип имя\_функции (список *типов* аргументов);
- // Реализация  
тип имя\_функции (список аргументов) {  
    тело функции  
}
- зачем писать имена аргументов? Понятность  
  
    long power (int x, int n);     //  $x^{**n}$   
  
    long power (int, int);         // где степень?
  - понятно компилятору
  - но не человеку



# Перерыв

# Характеристики переменной

- **Имя**
- **Тип**
- **Значение**  


A diagram illustrating a variable. A blue box with a purple border contains the number '2'. Below the box, the letter 'x' is written in blue, with a thin blue line connecting it to the bottom of the box.
- **Область видимости**  
в каком месте программы можно обратиться к данной переменной
- **Время жизни**  
Когда переменная создается и когда уничтожается
- Чему равно значение при создании, если переменной явно ничего не присвоили?  
`int x;`

# Локальные переменные

- область видимости — от места декларации до конца соответствующего блока { ... } (if, циклы)
- время жизни — до конца выполнения этого блока
- начальное значение — "мусор" (что было записано в этом месте памяти, то и осталось)

```
• int main ( ) {  
    int a = 2,  
        b;  
    printf ("%d %d\n", a, b );    // 2 ???  
    return 0;  
}
```

a 

b 

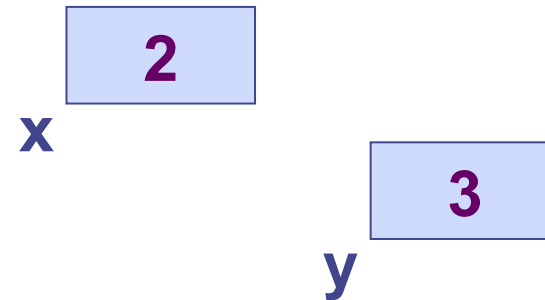
# Аргументы функций

- Область видимости — эта функция
- Время жизни — время 1 (одного) выполнения этой функции
  - создаются при вызове функции
  - уничтожаются после ее выполнения
  - и так для каждого вызова — новые наборы аргументов
- Начальное значение — значение аргумента при вызове

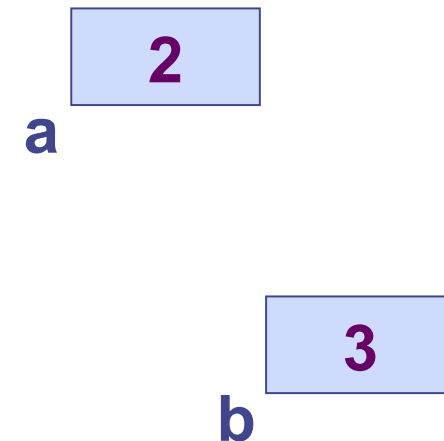


# Передача аргументов

- ```
int sum (int x, int y) {  
    return x + y;  
}
```



```
int main ( ) {  
    int a = 2, b = 3;  
    printf("%d %d %d\n", a, b, sum(a, b) );    // 2 3 5  
    a = sum(a, b);  
    printf("%d %d\n", a, b);                  // ??  
    b = sum(a, b);  
    printf("%d %d\n", a, b);                  // ??  
  
    return 0;  
}
```

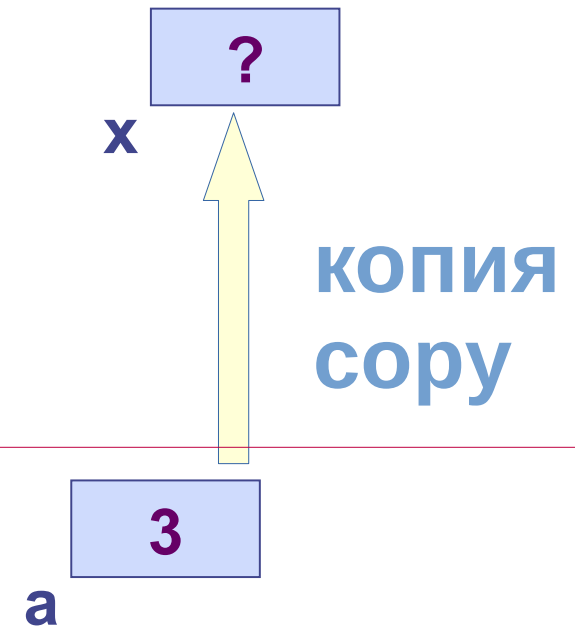


# Передача аргументов

- Аргументы передаются только по значению (копии)

```
void inc (int x) {  
    x = x + 1;    // x++;  
    // a = a + 1; // нельзя, не видно  
}
```

```
int main ( ) {  
    int a = 3;  
    inc (a);  
    inc (a);  
    printf("%d\n", a);    // ??  
}
```



# Когда копий не хватает?

- изменить передаваемое значение  
(два передаваемых значения)

```
void swap (int x, int y) { ... }
```

```
int main( ) {
```

```
    int a = 2, b = 7;
```

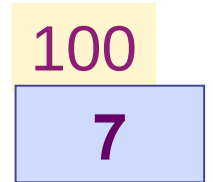
```
    swap(a, b);           // a = 7; b = 2;
```

```
}
```

- вернуть несколько значений  
из минут с 0:00 получить часы и минуты  
`h, m = mm2hm(135);` // в C так нельзя!
- Что делать?

# Адрес переменной

- Пусть переменная  $x$  (размером в несколько байт) начинается с байта с номером 100 и там записано число 7.
- **Адрес** — это номер байта (100, тоже число)  $x$
- Переменная может быть расположена в нескольких байтах подряд.

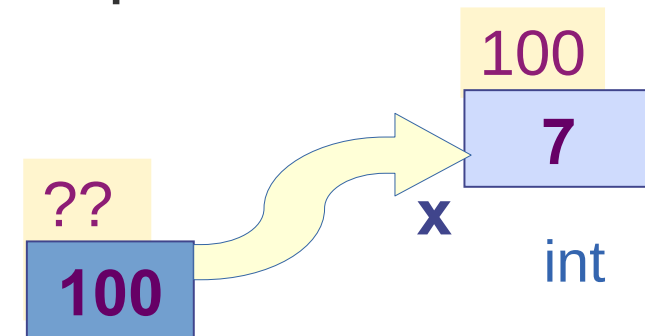


**Адрес переменной** — номер ее первого байта.

- Чтобы узнать по адресу переменной, какое число в ней хранится, нужно еще знать тип переменной:
  - размер переменной
  - как интерпретировать биты (int, float, unsigned)

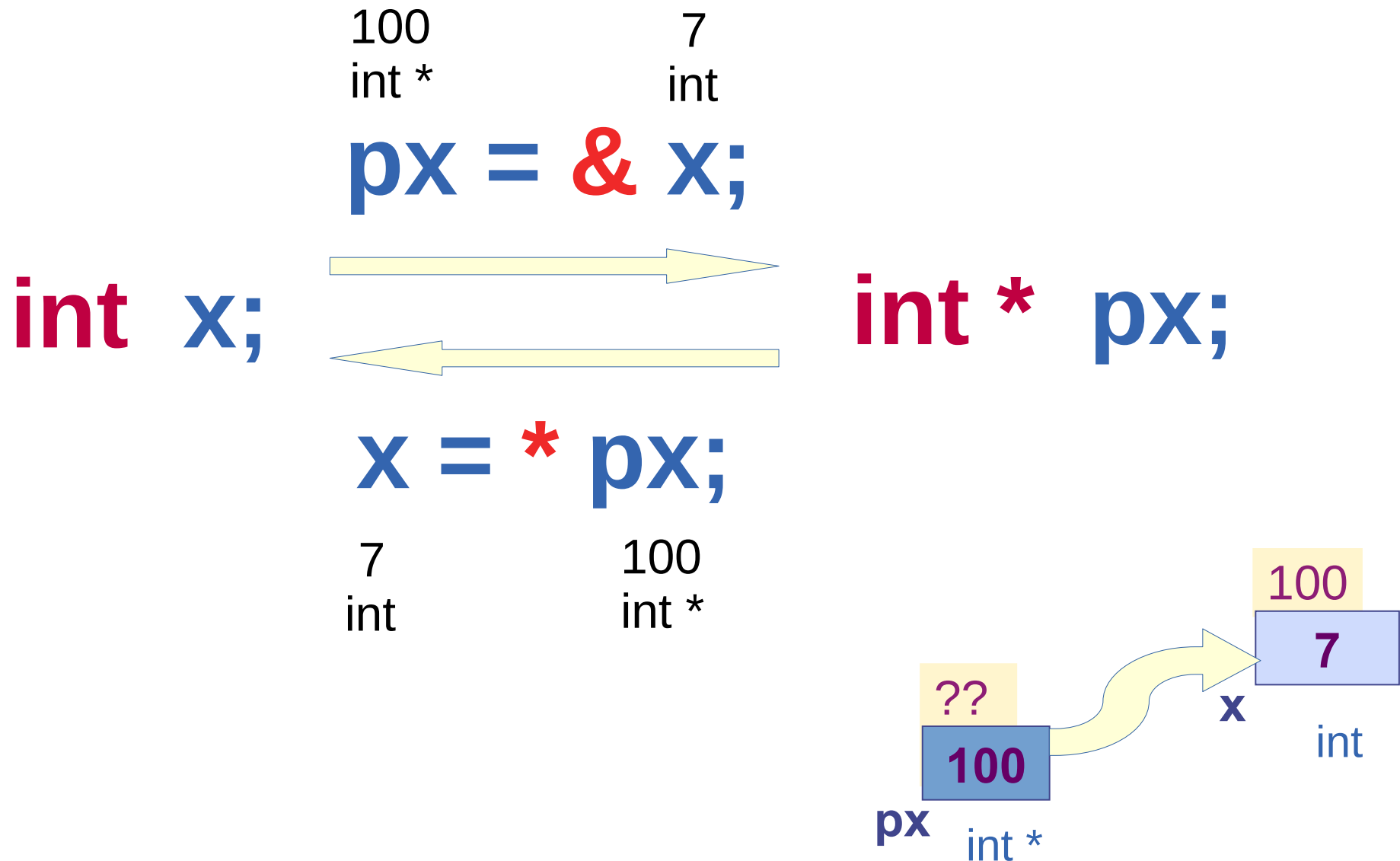
# Тип "указатель на ..."

- `2 * 3` // умножение
- `int x = 7;`  
`int * px;` // переменная px типа указатель на int  
// pointer to int  
`px = &x;` // & - вычислить адрес переменной



- `* px = 23;` // \* - разыменование `px` `int *` `x = 23;`  
// записать `int` по адресу, который хранится в `px`  
// потому что переменная `px` типа `int *`  
`* px = * px + 4;` `x = x + 4;`

# адрес (pointer), значение (value)

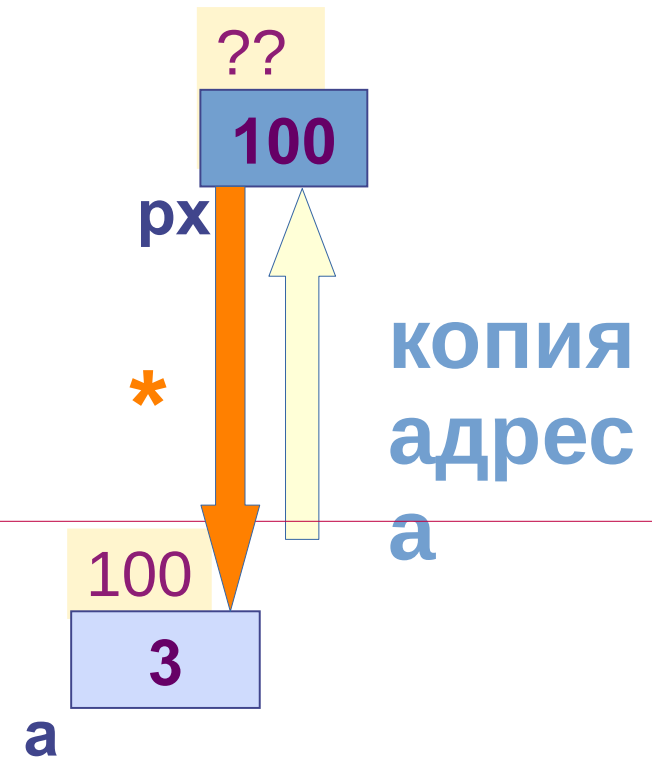


# Передаем копию адреса

- Аргументы передаются только по значению

```
void inc (int * px) {  
    *px = *px + 1;    // (*px)++;  
}
```

```
int main ( ) {  
    int a = 3;  
    inc (&a);  
    inc (&a);  
    printf("%d\n", a);    // ??  
}
```



# Вернуть часы и минуты

- Возвращать 2 значения нельзя
- Можно передать адреса 2 переменных, по которым сохранить результаты.
- ```
void mm2hm (int mm, int * ph, int * pm) {  
    // mm — минут с 00:00  
    // ph, pm — указатели на результат часы и минуты  
    * ph = .... ;  
    * pm = ... ;  
}  
  
int main ( ) {  
    int h, m;  
    mm2hm ( 135, &h, &m);
```




# Глобальные переменные

- определены вне всяких блоков и функций
- **Область видимости** – от декларации до конца файла
- область можно расширить
- поэтому пишут в начале файла
- **Время жизни** = время выполнения этой программы
- **начальное значение** = 0

Что будет выведено на печать?

```
int x;  
void foo ( ) {  
    x++;  
}  
int main ( ) {  
    foo ()  
    foo ();  
    printf ("x=%d\n", x);  
  
    return 0;  
}
```

x 

# Что будет напечатано?

- ```
void foo ( ) {  
    int x = 2;  
    x++;  
    printf ("x=%d\n", x);  
}  
  
int main ( ) {  
  
    foo ( );  
    foo ( );  
    return 0;  
}
```

x 2

# static

- **static** int x;
- *Область видимости*
  - блок для локальных
  - не более файла для НЕ локальных
- **Время жизни** = время выполнения этой программы
- **начальное значение** = 0
- иницируется **один** раз
-

# Что будет выведено?

```
void foo ( ) {  
    static int x;  
    x++;  
    printf ("x=%d\n", x);  
}  
  
int main ( ) {  
    foo ( );  
    foo ( );  
    return 0;  
}
```

x 0

# Что будет выведено?

```
void foo ( ) {  
    static int x = 2;  
    x++;  
    printf ("x=%d\n", x);  
}  
  
int main ( ) {  
    foo ( );  
    foo ( );  
    return 0;  
}
```

x

2

# Заключение

- Функции

```
float foo (float x, int n); // прототип
```

```
float foo (float x, int n) { // реализация
```

```
    float res;
```

```
    res = x * n;
```

```
    return res;
```

```
}
```

```
int main ( ) {
```

```
    float z;
```

```
    z = foo (3.5, -10);      // ВЫЗОВ
```

```
    return 0;
```

```
}
```

# Переменные

Переменные	Область видимости	Время жизни	<code>int x;</code>
Локальные	до конца блока	до конца блока	мусор
Аргументы функций	до конца функции	1 вызов функции	<code>foo(3)</code>
Глобальные	до конца файла в другом файле: <b><code>extern int x;</code></b>	выполнение программы	<b>0</b>
<code>static</code>	до конца файла	выполнение программы	<b>0</b>

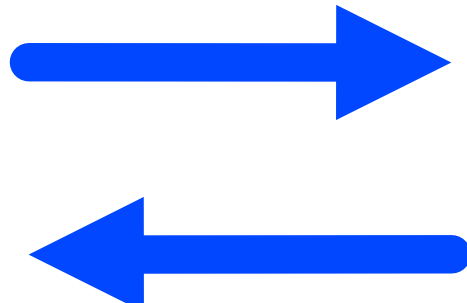
- Аргументы в функцию передаются **по значению** (копии)



# Указатели

**&** операция взятия указателя

`int x = 1;`      `p = &x;`      `int * p;`



`*p = *p + 3`

**\*** операция чтения/записи в память по указателю

**int \*** тип данных