

# Функции в С

## Общий вид функции

```
тип_возвращаемого_значения Имя_Функции (список_аргументов) {  
    «тело функции»  
}
```

- в С имена функций должны быть уникальны
  - ☞ в С++ каждая функция должна иметь уникальную сигнатуру вызова
- функции нельзя определять внутри других функций
- для каждого аргумента функции надо указывать его тип:  
`fun(double a, double b, int j)` // правильный список  
`fun(double a, b, int j)` // неправильный
  - ☞ если тип не указан, то считается, что это `int`; **ошибка в С23 и С++**
- если функция не имеет никаких параметров, можно указать `int fun(void)` или оставить скобки пустыми `int fun()`
  - ☞ в С23 и С++ эти два объявления полностью эквивалентны

## В функцию передаются значения переменных

👉 аргументы функции это локальные переменные которые получают значения в момент вызова этой функции

### Пример

```
void fun1(int a) { // функция fun1 ничего не возвращает
    a++;
}
int main() {
    int a = 5;
    fun1(a); // a++ ?
    printf("fun1: a= %d \n",a); // fun1: a= 5
}
```

## Оператор `return` и возвращаемое значение

- 1 возвращает «объект», если тип возвращаемого значения не `void`
- 2 обеспечивает немедленный выход из функции

## Обратите внимание

- использовать возвращаемое значение необязательно, например `printf()` возвращает число напечатанных символов
- при отсутствии `return` функция завершается по достижению закрывающей скобки `}`
  - 👉 если функция должна была что-то вернуть, то в этом случае возвращается произвольное значение, «мусор»
- в `main()` отсутствие `return` эквивалентно `return 0;` (C99, C++)

## Объявление и определение функций

- *Определение функции* — вся функция вместе с исполняемым кодом
- *Объявление функции или прототип* — только информация о функции: список параметров и тип возвращаемого значения

### 👉 Задача прототипа — описать интерфейс функции

```
// * declaration of functions (prototypes)
double fun(double x);
int sqr(int x);
// * call functions
int main() {
    printf(" fun(2) = %f\n",fun(2)); // fun(2) = 2.236068
    printf("  sqr(2) = %d\n",sqr(2)); // sqr(2) = 4
}
// * function definitions can follow their declarations
double fun(double x) { return sqrt(1+x*x); }
int sqr(int x) { return x*x; }
```

# Заголовочные файлы (header files)

👉 Прототипы библиотечных функций, «интерфейс», содержатся в заголовочных файлах с расширением: `.h` → `header`

## Заголовочные файлы стандартной библиотеки C

```
#include <math.h>      // математические функции
#include <stdio.h>      // ввод-вывод
#include <stdlib.h>     // функции общего назначения
```

👉 В C++ заголовочные файлы стандартной библиотеки не имеют расширения: `#include <cmath>`

👉 Сами функции откомпилированы и собраны в библиотеки и подключаются на стадии компоновки (линковки):

```
> clang prog.c -lm
```

# Указатель как аргумент функции

- в функцию адрес переменной передают через указатель:

```
void fun2(int* a) { (*a)++; } // function with pointer
int a=5;
fun2(&a);                      // a++
printf("fun2: a= %d\n",a);     // fun2: a= 6
```

- в результате функция меняет переменную находящуюся вне функции  
☞ тем не менее в функцию передается значение адреса переменной

## Функция `scanf()`: почему надо передавать адреса?

- `scanf` «сохраняет» вводимые с клавиатуры данные в переменные ⇒ аргументы `scanf()` это указатели, а передавать надо адреса:

```
int i,j;
scanf("%d %d",&i,&j); // читаем из stdin в i и j
```

- ☞ в C++ имеется версия `scanf` с ссылочными аргументами

## Пример: функция swap()

```
void swap(int* x, int* y) { // переменные x и y обмениваются значениями
    int tmp = *x;
    *x = *y;
    *y = tmp;
}

int main() { // проверка
    int i = 2;
    int j = 40;
    printf("before swap: i=%d, j=%d\n",i,j); // before swap: i=2, j=40
    swap(&i,&j);
    printf("after swap: i=%d, j=%d\n",i,j); // after swap: i=40, j=2
}
```

# Передача массива в функцию

## «Три» способа передать массив

```
void func1(int x[10]) { ... } // (1)
```

```
void func2(int x[]) { ... } // (2)
```

```
void func3(int* x) { ... } // (3)
```

☞ Эти объявления тождественны: в функцию передается не сам массив, а адрес первого элемента

## ☞ Размер массива надо передавать отдельно

```
double sum(int dim, double ar[]) {  
    // int dim = sizeof(ar)/sizeof(ar[0]); <- Ошибка, так не работает!  
    double s = 0.;  
    for(int i = 0; i < dim; i++) { s += ar[i]; }  
    return s;  
}
```



# Передача структур функциям

- 1 Передача всей структуры целиком
- 2 Передача указателя на структуру

## 1 Передача всей структуры целиком:

👉 в функции создается копия структуры

```
void Print_element(struct element e) {  
    printf(" element %s\n"  
          " number %d\n"  
          " atomic weight %f\n", e.name, e.number, e.A);  
}
```

```
Print_element(He);
```

```
Output> element Helium
```

```
Output> number 2
```

```
Output> atomic weight 4.002600
```

## ② Передача указателя на структуру: 👉 наиболее общий, рекомендуемый способ

```
void read_element(struct element* e) {  
    printf(" Enter the name of the element> ");  
    scanf("%s",e->name);  
    printf(" Enter the number of the element> ");  
    scanf("%i",&e->number);  
    printf(" Enter the weight of the element> ");  
    scanf("%lf",&e->A);  
}
```

```
read_element(&dump);
```

```
Enter the name of the element> Rhenium
```

```
Enter the number of the element> 75
```

```
Enter the weight of the element> 186.207
```

```
Print_element(dump);
```

```
Output> element Rhenium
```

```
Output> number 75
```

```
Output> atomic weight 186.207000
```

# Несколько возвращаемых значений

Как сделать, что бы функция вернула более одного значения?

## ❶ Использовать указатели (массивы) в аргументах

```
// function return array with random numbers:
void rand_arr(int n, int* arr) {
    for(int i = 0; i < n; i++) { arr[i] = rand(); }
}

int ar[100] = {[0]=0}; // zero array with designated initializer
rand_arr(2,ar);
rand_arr(2,&ar[10]);
for(i = 0; i < 3; i++) {
    printf("ar[%i]= %11d ar[%i]= %11d\n",i,ar[i],10+i,ar[10+i]);}
```

ar[0]=	16807	ar[10]=	1622650073
ar[1]=	282475249	ar[11]=	984943658
ar[2]=	0	ar[12]=	0

## ② Вернуть структуру

```
struct Array {  
    size_t n;  
    double* x;  
};  
  
struct Array uniarr(size_t n) { // fill by uniform distribution  
    struct Array tmp;  
    tmp.n = n;  
    tmp.x = (double*) malloc(n*sizeof(double));  
    for ( size_t i = 0; i < n; ++i )  
        tmp.x[i] = (double)rand() / RAND_MAX;  
    return tmp;  
}  
  
struct Array arr = uniarr(5);  
for ( size_t i = 0; i < arr.n; ++i ) {  
    printf("arr[%zu]= %f\n",i,arr.x[i]); // arr[0]= 0.532767 ...  
}
```

# Встраиваемые (inline) функции

## Оптимизация вызова функции

- ✓ Предполагает, что вызов `inline` функции будет максимально быстрым: код функции будет вставляться на место вызова
- ✓ Синтаксически нет различия в вызове обычной и `inline` функции

— пример inline функции —

```
static inline double Sqr(double x) { return x*x; }
```

👉 `static` для функции означает, что ее область видимости – файл где она определена; такую функцию можно целиком поместить в заголовочный файл

👉 `inline` функцию нельзя включить в библиотеку

👉 следует использовать для действительно маленьких функций, иначе из-за «раздувания» кода быстродействие всей программы может упасть

# Глобальные переменные и функции

- \* Стоит ли использовать глобальные переменные как «дополнительные» аргументы функции или ее возвращаемые значения?

## Следует избегать использовать глобальные переменные!

- ☞ Затрудняют понимание программы:
  - скрывают явные связи между функциями и переменными
  - невозможно установить правила использования
- ☞ «Нарушают» параллелизм исполнения

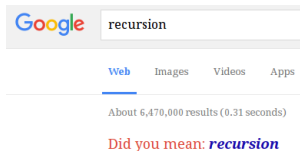
## Глобальные переменные имеет смысл использовать:

- В небольших, изолированных программах
- Слишком велики затраты чтобы избежать использование глобальных переменных: простые вещи следует делать просто

# Рекурсия

От латинского *recursio* – круговорот

Рекурсивная функция – функции вызывающая сама себя



Пример: рекурсия для гамма-функции  $\Gamma(z) = \Gamma(z+1)/z$

```
double gamma_recursion(double z) {
    if( z < 10.5 ) {
        if( fabs(z) < 2.e-16) {
            return HUGE_VAL; // +oo
        }
        return gamma_recursion(z+1)/z;
    }
    return exp(log_gam(z)); // Stirling approximation for large z
}
```

## Преимущества рекурсии

- простота написания некоторых алгоритмов

## Недостатки рекурсивных функций в C

- более медленное выполнение по сравнению с циклом
- может вызвать «переполнение стека»: **Stack overflow**

«Плохая» рекурсия — числа Фибоначчи:  $f_0 = 1; f_1 = 1; f_{n+1} = f_n + f_{n-1}$

```
unsigned int fibonacci(unsigned int n) {  
    return (n < 2) ? 1 : fibonacci(n-1) + fibonacci(n-2);  
}
```

**fibonacci(46) – время выполнения ~ 20 секунд ...**

```
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584  
fibonacci(46) = 1836311903
```



# Стандартная библиотека C: variadic functions

## Функции с переменным числом аргументов <stdarg.h>

- Обычная функция имеет фиксированное число аргументов
- Вариативные функции позволяют работать с произвольным, заранее неизвестным, числом аргументов, например `printf`, `scanf`
- Декларируются с помощью «многоточия» в списке аргументов  
📌 **многоточие всегда последний аргумент в функции**

```
double average(int n, ...); // average of 'n' arguments
```

- Вызов не отличается от вызова обычных функций:

```
double x2 = average(2, 4.,5.);           // average of 2 numbers  
double x5 = average(5, 4.,5.,-1.,2.,3.); // average of 5 numbers
```

- Программируются с помощью макросов, определенных в <stdarg.h>  
`va_list`, `va_start`, `va_arg`, `va_end`

## Пример вариативной функции average()

```
#include <stdio.h>
#include <stdarg.h> // macros: va_list, va_start, va_arg, va_end
double average(int n, ...) {
    va_list va;      // holds the information for other macros
    va_start(va,n);  // to get the address first "variadic arg"
    double sum = 0;
    for(int i = 0; i < n; i++) {
        sum += va_arg(va,double); // accesses the next arg
    }
    va_end(va); // ends traversal of args
    return sum/n;
}

int main() {
    double x2 = average(2, 4.,5.);
    printf(" x2= %f\n",x2);          // x2= 4.500000
    double x5 = average(5, 4.,5.,-1.,2.,3.);
    printf(" x5= %f\n",x5);          // x5= 2.600000
}
```

## Обратите внимание

- Должен быть хотя бы один фиксированный аргумент (до C23)
  - 👉 в примере: `average(int n, ...);`
- Число вариативных аргументов обычно передается как фиксированный аргумент
- Если тип аргументов заранее неизвестен, он должен передаваться либо через фиксированные аргументы, либо в самом списке
- Соответствие реального и ожидаемого числа аргументов и их типов — задача программиста:

```
average(2, 4,5);    // 0.000000 after 4 and 5 must be dots
average(3, 4.,5.); // 3.000000 number of parameters must be 2
```

## В C23 можно написать функцию без фиксированного аргумента

```
void average( ... ) { // no required parameter, C23
    va_list args;
    va_start( args ); // no second argument in va_start macros, C23
```

# Указатель на функцию

## Зачем нужны указатели на функции?

- ☞ Передача исполняемого кода в качестве одного из параметров функции

## Типичные проблемы решаемые с помощью указателей на функции:

- задание функции сравнения в алгоритмах сортировки и поиска:

```
void qsort(void *base, size_t nmemb, size_t size,  
          int(*compar)(const void *, const void *) )
```

- численные расчеты: интегрирование, минимизация и тому подобное:

```
int gsl_integration_qag(const gsl_function * f, double a, double b,  
                       double epsabs, double epsrel, ...)
```

- callback-functions (функции обратного вызова):

```
button = XmCreatePushButton(toplevel, "button", args, 1);  
XtAddCallback(button, XmNarmCallback, my_callback, NULL);
```



## Синтаксис

- Задать, определить, указатель на функцию можно так:

```
возвращаемый_тип (*имя) (перечисление типов аргументов)  
int (*pf)(); // pf - указатель на функцию без параметров  
void (*pf1)(double) = NULL; // pf1 - один аргумент  
double (*pf2)(int,int) = NULL; // pf2 - два аргумента
```

- Упростить запись можно с помощью оператора `typedef`:

```
typedef double (*FUNCPTR)(double, double);  
// теперь FUNCPTR - «новый тип» для указателя на функцию:  
FUNCPTR pf2 = NULL; // define pointer to function
```

-  стандарт запрещает преобразование указателей на функции в указатели на данные, например в `void*`
-  преобразование указателя на функцию в указатель на функцию другого типа приводит к неопределенному поведению при вызове

👉 Имена функций можно использовать как адреса этих функций

## Примеры

```
double Rad(double x, double y) { return sqrt(x*x+y*y); }
```

\_\_\_\_\_ присваивание адреса функции указателю \_\_\_\_\_

```
typedef double (*FUNCPTR)(double, double);  
FUNCPTR pf2 = &Rad; // assignment using address operator  
FUNCPTR pf2s = Rad; // short form  
double (*pf2n)(double,double) = Rad; // assignment without typedef
```

\_\_\_\_\_ вызов функции по указателю \_\_\_\_\_

```
double res = (*pf2)(1,10); // full form  
double result = pf2(3,4); // short form
```

## Массив указателей на функции (таблица функций)

\_\_\_\_\_ массив из десяти указателей \_\_\_\_\_

```
double (*pf[10])(int,int);
```

\_\_\_\_\_ typedef значительно упрощает жизнь \_\_\_\_\_

```
typedef double (*FIIPTR) (int,int)
FIIPTR pf[10]; // более привычная запись
```

## Указатель на функцию как аргумент функции

\_\_\_\_\_ функция `qsort` с указателем на функцию типа `ComparF` \_\_\_\_\_

```
typedef int(*ComparF)(const void *, const void *);
void qsort(void *base,size_t nmemb,size_t size, ComparF my_comp) {
    ...
    my_comp(adrl,adr2); // вызов функции
    ...
}
```

## Пример с указателем на функцию: вычисление $\int_0^1 f(x)dx$

❶ имя функции фиксировано: `fun(x)` (указателей на fun нет)

```
double fun(double x); // prototype of function
double Integral() {
    int Nsteps = 100;
    double h = 1./(double)Nsteps;
    double sum = 0.5*(fun(0) + fun(1));
    for(int i = 1; i < Nsteps; i++) {
        double x = h*i;
        sum += fun(x);
    }
    return h*sum;
}
double fun(double x) { return cos(x); } // definition
printf("Int_0^1(cos(x))=%f\n",Integral()); // Int_0^1(cos(x))=0.841464
```



② используем указатель на функцию

```
typedef double (*FPTR)(double);  
double Integral(FPTR fun) { ... } // use 'fun' as an argument
```

```
// ВЫЗОВ
```

```
printf("Int_0^1(cos(x)) = %f\n",Integral(cos));
```

```
Result> Int_0^1(cos(x)) = 0.841464
```

```
printf("Int_0^1(sin(x)) = %f\n",Integral(sin));
```

```
Result> Int_0^1(sin(x)) = 0.459694
```

- 3 Если требуется интегрировать функцию с дополнительными параметрами:

$$f(z) = \prod_{n=2}^5 g(n, z), \quad g(n, z) = \int_0^1 \sin(x^n + z) dx$$

## Общая идея

☞ в `Integral()` и в подынтегральную функцию добавляется еще один аргумент `void* userdata`, который используется для передачи дополнительных параметров

- общий вид подынтегральной функции:

```
typedef double (*FPTR)(double, void*);
```

- вид функции интегрирования:

```
double IntegralU(FPTR fun, void* userdata) {  
    // вызов fun(x) заменяется на fun(x,userdata)  
}
```

Для подынтегральной функции пишется «функция обертка»

```
double Integrand(double x, void* userdata) {  
    double* arg = (double*) userdata;  
    double y = arg[0]; // 'n' parameter  
    double z = arg[1]; // 'z' parameter  
    return sin(pow(x,y)+z);  
}
```

```
double userdata[] = {0,3}; // n=0 and z=3 parameters  
double prod = 1.;  
for(int i = 2; i <= 5; i++) {  
    userdata[0] = i; // change 'n' for each integration  
    prod *= IntegralU(Integrand,userdata); // Вызов интегратора  
}  
printf("#prod_{n=2}^{5} #int_{0}^{1} (sin(x^{n}+3)) = %e\n", prod);
```

$$\prod_{n=2}^5 \int_0^1 (\sin(x^n + 3)) = 1.875599e - 05$$

Дополнительные слайды

## Передача двумерного массива в функцию

Адрес элемента:  $a[i][j] = *(a + i * (\text{max value of } j) + j)$

👉 В функции обязательно должен быть указан размер правого измерения

## Пример: печать 2D-массива `int mtx[][5];`

```
void print_mtx(int Nrow, int Ncol, int mtx[][5]) {  
    for(int i = 0; i < Nrow; i++) {  
        for(int j = 0; j < Ncol; j++) {  
            printf("%3d ", mtx[i][j]);  
        }  
        printf("\n");  
    }  
}
```

## Передача многомерного массива в функцию

```
void func1(int Matrix[][8][3][4][3]) { ... }
```

## Обычная и встраиваемая версии функции

- В C (но не в C++) можно иметь две версии функции — встраиваемую и обычную

### Для таких случаев общая стратегия:

- Определяем в заголовочном файле `inline function` и включаем этот файл везде, где требуется эта функция
- В одном единственном `c`-файле добавляем объявление: `external function`

- 1 в заголовочном файле `header.h`:  

```
inline double Sqr(double x) {return x*x;}
```
- 2 в `main.c` -файле:  

```
#include "header.h"  
extern double Sqr(double x); /* prototype */
```
- 3 в остальных `.c` -файлах:  

```
#include "header.h"
```