

# Функции

## Общий вид функции

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

- имена функций должны быть уникальны
- функции нельзя определять внутри других функций, все функции «глобальны», область видимости — вся программа
- 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` — не рекомендуется!
- если функция не имеет никаких параметров, можно указать `void` или оставить скобки пустыми: `int getTime()`

## В функцию передаются только 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 «число», 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, возвращаемое значение будет неопределенно
- если в `main()` отсутствие `return`, это эквивалентно `return 0;` (C99)

## 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);
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)

👉 Интерфейс для библиотечных функций описывается с помощью прототипов функций, которые помещают в заголовочные файлы 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
```

👉 Сами функции откомпилированы и собраны в библиотеки, например подключение математической библиотеки: `> clang prog.c -lm`

# Указатель как аргумент функции

- чтобы передать в функцию адрес переменной, надо декларировать аргумент функции как указатель:

```
void fun2(int* a) { (*a)++; } // function with pointer
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» данные вводимые с клавиатуры 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
```

## 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 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]) { ... }
```

# Передача структур функциям

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

## 1 Передача всей структуры целиком: внимание, структуры копируются!

```
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 в аргументах (массивы)

```
// 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 Tuple{
    int n;
    double x;
};

// function return the quotient and remainder of a/b
struct Tuple divide(int a, int b) {
    struct Tuple tmp;
    tmp.n = a/b;                                // целая часть
    tmp.x = (double)a/((double)b - (double) tmp.n; // остаток
    return tmp;
}

struct Tuple t = divide(17,3);
printf("17/3 = %i + %f\n",t.n,t.x); // 17/3 = 5 + 0.666667
```

# 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 или возвращаемые значения?

**Следует избегать использовать глобальные переменные!**

- Затрудняют понимание программы: скрывают явные связи между функциями и переменными, невозможно установить правила использования

 Делают невозможным параллелизм исполнения

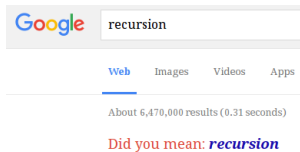
**Когда глобальные переменные имеет смысл использовать?**

- Небольшие, изолированные программы
- 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; // a large value, possibly +oo  
        }  
        return gamma_recursion(z+1)/z;  
    }  
    return exp(log_gam(z)); // Stirling approximation for big 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
```

# 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`:

```
// определим FUNCPTR - «новый тип» для указателя на функцию:
typedef double (*FUNCPTR)(double, double);
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 (таблица) указателей на функции

- массив указателей на «однотипные» функции задается в виде:

```
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

- Задание функции 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(adr1,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); // вместо прототипа функции fun  
double Integral(FPTR fun) { ... }
```

```
// 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);
```

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