

# Основы алгоритмизации и программирования

## Лекция 11

### Функции пользователя

# Функции пользователя

С увеличением объема программы ее код становится все более сложным. Одним из способов борьбы со сложностью любой задачи является ее **разбиение на части**. В языке **Си**, как и в любом языке программирования высокого уровня, задача может быть разбита на более простые подзадачи при помощи подпрограмм-функций. После этого программу можно рассматривать в более укрупненном виде – на уровне взаимодействия созданных подпрограмм. Использование подпрограмм в коде программы и ведет к упрощению ее структуры.

Разделение программы на подзадачи позволяет также избежать **избыточности кода**, поскольку функцию записывают один раз, а вызывать ее на выполнение можно многократно из разных точек программы. Кроме того, **упрощается процесс отладки** программы, содержащей подпрограммы. Часто используемые функции можно помещать в отдельные библиотеки

Следующим шагом в повышении уровня абстракции программы является **группировка функций** и связанных с ними данных в **отдельные файлы (модули)**, компилируемые отдельно. Получившиеся в результате компиляции объектные модули объединяются в исполняемую программу с помощью компоновщика. Разбиение на модули уменьшает время перекомпиляции и облегчает процесс отладки, скрывая несущественные детали за интерфейсом модуля, что позволяет отлаживать программу по частям.

# Функции пользователя

**Функция** – это именованная последовательность инструкций, выполняющая какое-либо законченное действие.

Разделение программы на максимально обособленные части (подпрограммы) является довольно сложной задачей, которая должна решаться на этапе проектирования программы.

В языке **Си** любая подпрограмма является функцией, представляющей собой отдельный программный модуль, к которому можно обратиться, чтобы передать через параметры исходные данные и получить один или несколько результатов его работы.

В отличие от других языков программирования высокого уровня в языке **Си** нет разделения на подпрограммы-процедуры и подпрограммы-функции, здесь вся программа строится только из функций.

# Декларация функции

Как и любой объект программы на языке **Си**, пользовательские функции необходимо **декларировать**. Объявление функции пользователя, т.е. ее декларация, выполняется в двух формах – в **форме описания (объявления)** и в **форме определения**, т.е. **любая пользовательская функция должна быть объявлена и определена**.

**Описанием функции** является декларация ее прототипа, который сообщает компилятору о том, что далее будет приведено ее полное определение (**текст**), т.е. **реализация**.

**Объявление функции** (прототип, заголовок) задает ее свойства – идентификатор, тип возвращаемого значения (если такое имеется), количество и типы параметров.

В стандарте языка используется следующий формат декларации (объявления) функций

*тип\_результата* **ID\_функции** (*список*);

В **списке** перечисляются типы параметров данной функции, причем идентификаторы переменных в круглых скобках прототипа указывать необязательно, т.к. компилятор языка их не обрабатывает

# Декларация функции

**Описание прототипа** дает возможность компилятору проверить соответствие типов и количества параметров при фактическом вызове этой функции.

Пример

Объявление функции ***fun***, которая имеет три параметра типа ***int***, один параметр типа ***double*** и возвращает результат типа ***double***:

```
double fun(int, int, int, double);
```

Каждая функция, вызываемая в программе, должна быть определена (только один раз). **Определение функции** – это ее полный текст, включающий заголовок и код.

Полное определение (реализация) функции

```
тип_результата ID_функции(список параметров)
{
    код функции
    return выражение;
}
```

# Декларация функции

**Тип результата** определяет тип выражения, значение которого возвращается в точку ее вызова при помощи оператора ***return выражение;***. **Выражение** преобразуется к **типу\_результата**, указанному в заголовке функции и передается в точку вызова. Тип возвращаемого функцией значения может быть любым базовым типом, а также указателем на массив или функцию. Если функция не должна возвращать значение, указывается тип ***void***. В данном случае оператор ***return*** можно не ставить. Из функции, которая не описана как ***void***, необходимо возвращать значение, используя оператор ***return***. Если тип функции не указан, то по умолчанию устанавливается тип ***int***.

**Список параметров** состоит из перечня типов и идентификаторов параметров, разделенных запятыми. Список параметров определяет объекты, которые требуется передать в функцию при ее вызове

В определении и в объявлении одной и той же функции **типы и порядок следования параметров должны совпадать**. Тип возвращаемого значения и типы параметров совместно определяют тип функции

Функция может **не иметь параметров**, но круглые скобки необходимы в любом случае. Если у функции отсутствует список параметров, то при декларации такой функции желательно в круглых скобках указать ***void***.

***void main(void){ ... }***

# Декларация функции

В функции может быть несколько операторов ***return***, но может и не быть ни одного (тип ***void*** – это определяется потребностями алгоритма). В последнем случае возврат в вызывающую программу происходит после выполнения последнего оператора кода функции.

## Пример

Функция, определяющая наименьшее значение из двух целочисленных переменных:

```
int min (int x, int y){  
    return (x<y) ? x : y;  
}
```

Функции, **возвращающие значение**, желательно использовать в правой части выражений языка **Си**, иначе возвращаемый результат будет утерян.


В языке **Си** каждая **функция** – это отдельный блок программы, вход в который возможен только через вызов данной функции

# Вызов функции

Для **вызова функции** в простейшем случае нужно указать ее имя, за которым в круглых скобках через запятую перечислить список передаваемых ей **аргументов**. Вызов функции может находиться в любом месте программы, где по синтаксису допустимо выражение того типа, который формирует функция.

Простейший вызов функции имеет следующий формат:

***ID\_функции (список аргументов);***



В качестве аргументов можно использовать **константы, переменные, выражения** (их значения перед вызовом функции будут определены компилятором)

**Аргументы** в списке вызова **должны совпадать** со списком параметров вызываемой функции **по количеству и порядку следования**, а типы аргументов при передаче в функцию будут преобразованы, если это возможно, к типу соответствующих им параметров

Связь между функциями осуществляется через аргументы и возвращаемые функциями значения. Ее можно осуществить также через внешние, глобальные переменные



# Вызов функции

**Глобальные переменные доступны всем функциям**, где они не описаны как локальные переменные. Использовать их для передачи данных между функциями довольно просто, но тем не менее этого делать не рекомендуется. **Необходимо стремиться к тому, чтобы функции в программе были максимально независимыми и чтобы их интерфейс полностью определялся прототипами этих функций**

**Функции могут располагаться в исходном файле в любом порядке**, при этом исходная программа может размещаться в нескольких файлах.

Все величины, описанные **внутри функции, 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.

Если этого требуется избежать, при объявлении локальных переменных используется модификатор ***static***

# Вызов функции

Пример

```
#include <stdio.h>
void f1(int);
void main(void)
{
    f1(5);
}

void f1(int i)
{
    int m=0;
    puts(" n m p ");
    while (i-->0) {
        static int n = 0;
        int p = 0;
        printf(" %d %d %d \n", n++, m++, p++);
    }
}
```

# Вызов функции

## Пример

Статическая переменная ***n*** будет создана в сегменте данных оперативной памяти и инициализируется нулем только один раз при первом выполнении оператора, содержащего ее определение, т.е. при первом вызове функции ***f1***. Автоматическая переменная ***m*** инициализируется при каждом входе в функцию. Автоматическая переменная ***p*** инициализируется при каждом входе в блок цикла.

В результате выполнения программы получим:

<b><i>n</i></b>	<b><i>m</i></b>	<b><i>p</i></b>
<b>0</b>	<b>0</b>	<b>0</b>
<b>1</b>	<b>1</b>	<b>0</b>
<b>2</b>	<b>2</b>	<b>0</b>
<b>3</b>	<b>3</b>	<b>0</b>
<b>4</b>	<b>4</b>	<b>0</b>

# Передача аргументов в функцию

В языке **Си** аргументы при стандартном вызове функции передаются **по значению**. Это означает, что в стеке, как и в случае локальных данных, выделяется место для формальных параметров функции. В выделенное место при вызове функции заносятся значения фактических аргументов, при этом проверяется соответствие типов и при необходимости выполняются их преобразования. При несоответствии типов выдается диагностическое сообщение. Затем функция использует и может изменять эти значения в стеке.

При **выходе из функции** измененные значения теряются, т.к. время жизни и зона видимости локальных параметров определяется кодом функции. Вызванная функция не может изменить значения переменных, указанных как фактические аргументы при обращении к данной функции.

В случае необходимости функцию можно использовать для изменения передаваемых ей аргументов. В этом случае в качестве аргумента необходимо в вызываемую функцию передавать не значение переменной, а ее адрес.

При передаче по адресу в стек заносятся копии адресов аргументов, а функция осуществляет доступ к ячейкам памяти по этим адресам и может изменить исходные значения аргументов. Для обращения к значению аргумента-оригинала используется операция «\*».

# Передача аргументов в функцию

## Пример

Функция, в которой меняются местами значения  $x$  и  $y$ :

```
void zam(int *x, int *y)
{
    int t = *x;
    *x = *y;
    *y = t;
}
```

Участок программы с обращением к данной функции:

```
void zam (int*, int*);
void main (void)
{
    int a=2, b=3;
    printf(" a = %d , b = %d\n", a, b);
    zam (&a, &b);
    printf(" a = %d , b = %d\n", a, b);
}
```

При таком способе передачи данных в функцию их значения будут изменены, т.е. на экран монитора будет выведено

**$a = 2, b = 3$**

**$a = 3, b = 2$**

# Передача аргументов в функцию

Если требуется запретить изменение значений какого-либо параметра внутри функции, то в его декларации используют атрибут ***const***, например:

```
void f1(int, const double);
```

Рекомендуется указывать ***const*** перед всеми параметрами, изменение которых в функции не предусмотрено. Это облегчает, например, отладку программы, т.к. по заголовку функции видно, какие данные в функции изменяются, а какие нет

# Операция typedef

Любому типу данных, как стандартному, так и определенному пользователем, можно задать новое имя с помощью операции ***typedef***

```
typedef тип новое_имя ;
```

Введенный таким образом новый тип используется аналогично стандартным типам, например, введя пользовательские типы:

***typedef unsigned int*** ***UINT***; – здесь ***UINT*** – новое имя;

***typedef char*** ***M\_s*** [101]; – здесь ***M\_s*** – тип пользователя, определяющий строки, длиной не более 100 символов.

Декларации объектов введенных типов будут иметь вид:

<b><i>UINT</i></b> i, j;	→	две переменные типа <b><i>unsigned int</i></b> ;
<b><i>M_s</i></b> str[10];	→	массив из 10 элементов, в каждом из которых можно хранить по 100 символов.

# Указатели на функции

В языке **Си** идентификатор функции является константным указателем на начало функции в оперативной памяти и не может быть значением переменной. Но имеется возможность декларировать указатели на функции, с которыми можно обращаться как с переменными (например, можно создать массив, элементами которого будут указатели на функции).

Как и любой объект языка **Си**, указатель на функции необходимо декларировать. Формат объявления указателя на функции следующий:

***тип (\*переменная-указатель)(список параметров);***

т.е. декларируется указатель, который можно устанавливать на функции, возвращающие результат указанного *типа* и которые имеют указанный ***список параметров***. Наличие первых круглых скобок обязательно, так как без них – это декларация функции, которая возвращает указатель на результат.

Пример

```
double (*p_f)(char, double);
```

Объявление данного вида говорит о том, что декларируется указатель ***p\_f***, который можно устанавливать на функции, возвращающие результат типа *double* и имеющие два параметра: первый – символьного типа, а второй – вещественного типа



# Указатели на функции

Идентификатор функции является константным указателем, поэтому для того чтобы установить переменную-указатель на конкретную функцию, достаточно ей присвоить ее идентификатор:

***переменная-указатель = ID\_функции;***

Например, имеется функция с прототипом: **double f1(char, double);** тогда операция

***p\_f = f1;***

установит указатель *p\_f* на данную функцию.

Вызов функции после установки на нее указателя выглядит так:

***(\*переменная-указатель)(список аргументов);***

или

***переменная-указатель (список аргументов);***

После таких действий кроме стандартного обращения к функции:

***ID\_функции(список аргументов);***

появляется еще два способа вызова функции:

***(\*переменная-указатель)(список аргументов);***

или

***переменная-указатель (список аргументов);***

Последняя запись справедлива, так как *p\_f* также является адресом начала функции в оперативной памяти.

# Указатели на функции

## Пример

Для нашего примера к функции ***f1*** можно обратиться следующими способами:

<b><i>f1</i></b> ('z', 1.5);	– обращение к функции по имени (ID);
<b>(* p_f)</b> ('z', 1.5);	– обращение к функции по указателю;
<b>p_f</b> ('z', 1.5);	– обращение к функции по ID указателя.

**Основное назначение указателей на функции** – это обеспечение возможности передачи идентификаторов функций в качестве параметров в функцию, которая реализует некоторый вычислительный процесс, используя формальное имя вызываемой функции.

# Указатели на функции

## Пример

Написать функцию вычисления суммы **sum**, обозначив слагаемое формальной функцией **fun(x)**. При вызове функции суммирования передавать через параметр реальное имя функции, в которой задан явный вид слагаемого. Например, пусть требуется вычислить две суммы:

$$S_1 = \sum_{i=1}^{2n} \frac{x}{5} \quad \text{и} \quad S_2 = \sum_{i=1}^n \frac{x}{2}$$

Поместим слагаемые этих сумм в пользовательские функции **f1** и **f2** соответственно. При этом воспользуемся операцией **typedef**, введя пользовательский тип данных: указатель на функции **p\_f**, который можно устанавливать на функции, возвращающие результат **double** и имеющие один параметр типа **double**.

Тогда в списке параметров функции суммирования достаточно будет указать фактические идентификаторы функций созданного типа **p\_f**.

# Указатели на функции

Пример

```
...
typedef double (*p_f)(double);
double sum(p_f, int, double);           // Декларации прототипов функций
double f1(double);
double f2(double);
void main(void)
{
    double x, s1, s2;
    int n;
    puts (" Введите кол-во слагаемых n и значение x: ");
    scanf ("%d %lf ", &n, &x);
    s1 = sum (f1, 2*n, x);
    s2 = sum (f2, n, x);
    printf("\n\t N = %d , X = %lf ", n, x);
    printf("\n\t Сумма 1 = %lf\n\t Сумма 2 = %lf ", s1, s2);
}
```

# Указатели на функции

Пример

/\* Первый параметр функции суммирования – формальное имя функции, введенное с помощью *typedef* типа \*/

```
double sum(p_f fun, int n, double x) {  
    double s=0;  
    for(int i=1; i<=n; i++)  
        s+=fun(x);  
    return s;  
}
```

//\_\_\_\_\_ Первое слагаемое \_\_\_\_\_

```
double f1(double r) {  
    return r/5.;  
}
```

//\_\_\_\_\_ Второе слагаемое \_\_\_\_\_

```
double f2(double r) {  
    return r/2.;  
}
```

# Рекурсивные функции

**Рекурсивной** (самовызываемой или самовызывающей) называют функцию, которая прямо или косвенно вызывает сама себя.

Возможность прямого или косвенного вызова позволяет различать **прямую** или **косвенную** рекурсии. **Функция называется косвенно рекурсивной** в том случае, если она содержит обращение к другой функции, содержащей прямой или косвенный вызов первой функции. В этом случае по тексту определения функции ее рекурсивность (косвенная) может быть не видна. Если в функции **используется вызов этой же функции**, то имеет место **прямая рекурсия**, т.е. функция по определению рекурсивная.



**Рекурсивные алгоритмы** эффективны в задачах, где рекурсия использована в самом определении обрабатываемых данных. Поэтому изучение рекурсивных методов нужно проводить, вводя динамические структуры данных с рекурсивной структурой.

# Рекурсивные функции

**В рекурсивных функциях необходимо выполнять следующие правила.**

При каждом вызове в функцию передавать модифицированные данные.

На каком-то шаге должен быть прекращен дальнейший вызов этой функции, это значит, что рекурсивный процесс должен шаг за шагом упрощать задачу так, чтобы для нее появилось нерекурсивное решение, иначе функция будет вызывать себя бесконечно

После завершения очередного обращения к рекурсивной функции в вызывающую функцию должен возвращаться некоторый результат для дальнейшего его использования

# Рекурсивные функции

## Пример

Заданы два числа  $a$  и  $b$ , большее из них разделить на меньшее, используя рекурсию.

...

```
double proc(double, double);
```

```
void main (void)
```

```
{
```

```
    double a,b;
```

```
    puts(" Введи значения a, b : ");
```

```
    scanf("%lf %lf", &a, &b);
```

```
    printf("\n Результат деления : %lf", proc(a,b));
```

```
}
```

```
//----- Функция -----
```

```
double proc( double a, double b) {
```

```
    if ( a< b ) return proc ( b, a );
```

```
        else return a/b;
```

```
}
```



# Рекурсивные функции

## Пример

Заданы два числа  $a$  и  $b$ , большее из них разделить на меньшее, используя рекурсию.

```
...
double proc(double, double);
void main (void)
{
    double a,b;
    puts(" Введи значения a, b : ");
    scanf("%lf %lf", &a, &b);
    printf("\n Результат деления : %lf", proc(a,b));
}

//----- Функция -----
double proc( double a, double b) {
    if ( a< b ) return proc ( b, a );
    else return a/b;
}
```

Если  $a$  больше  $b$ , условие, поставленное в функции, не выполняется и функция **proc** возвращает нерекursивный результат.

Пусть теперь условие выполнилось, тогда функция **proc** обращается сама к себе, аргументы в вызове меняются местами и последующее обращение приводит к тому, что условие вновь не выполняется и функция возвращает нерекursивный результат.

# Рекурсивные функции

## Пример

Функция для вычисления факториала *неотрицательного* значения  $k$ .

```
double fact (int k) {  
    if ( k < 1 ) return 1;  
    else  
        return k * fact ( k - 1);  
}
```

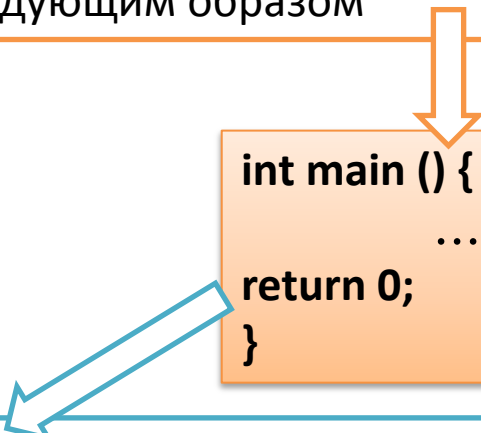
Для нулевого значения параметра функция возвращает **1** ( $0! = 1$ ), в противном случае вызывается та же функция с уменьшенным на 1 значением параметра и результат умножается на текущее значение параметра. Тем самым для значения параметра  $k$  организуется вычисление произведения

$$k * (k-1) * (k-2) * \dots * 3 * 2 * 1 * 1$$

Последнее значение «1» – результат выполнения условия  $k < 1$  при  $k = 0$ , т.е. последовательность рекурсивных обращений к функции *fact* прекращается при вызове *fact*(0). Именно этот вызов приводит к последнему значению «1» в произведении, так как последнее выражение, из которого вызывается функция, имеет вид:  $1 * \text{fact}(1 - 1)$ .

# Параметры командной строки функции **main**

В стандарте **ANSI** функция **main** возвращает целочисленный результат, т.е. используется следующим образом



```
int main () {  
    ...  
    return 0;  
}
```

Оператор **return** возвращает операционной системе код завершения функции, причем значение **0** трактуется как нормальное завершение, остальные значения воспринимаются как ошибки.

Функция **main** может быть определена с параметрами, которые передаются из внешнего окружения, например, из командной строки. Во внешнем окружении действуют свои правила представления данных, а точнее, все данные представляются в виде строк **СИМВОЛОВ**.

# Параметры командной строки функции `main`

Для передачи этих строк в функцию `main` используются два параметра, общепринятые (необязательные) идентификаторы которых `argc` и `argv`:

```
int main (int argc, char *argv[]) ...
```



Параметр `argc` имеет тип `int`, его значение формируется из анализа командной строки и равно количеству слов в командной строке, включая и имя вызываемой функции.

Параметр `argv` – это массив указателей на строки, каждая из которых содержит одно слово из командной строки. Если слово должно содержать **символ пробел**, то при записи его в командную строку оно **должно быть заключено в кавычки**

Функция `main` может иметь и третий параметр `argp`, который служит для передачи параметров операционной системы (ОС), в которой выполняется программа, в этом случае ее заголовок имеет вид

```
int main (int argc, char *argv[], char *argp[])
```

# Параметры командной строки функции `main`

Операционная система поддерживает передачу значений для параметров ***argc***, ***argv***, ***argp***, а пользователь отвечает за передачу и использование фактических аргументов функции ***main***.

## Пример

Программа печати фактических аргументов, передаваемых из ОС в функцию ***main*** и параметров оперативной системы

```
int main ( int argc, char *argv[], char *argp[])
{
    int i;
    printf ("\n Program Name %s", argv[0]);
    for( i=1; i <=argc; i++)
        printf ("\n Argument %d = %s", i, argv[i]);
    printf ("\n OC parametrs: ");
    while (*argp) {
        printf ("\n %s", *argp);
        argp++;
    }
    return 0;
}
```

Очевидно, что оформленная таким образом функция ***main()*** может вызываться рекурсивно из любой функции