
FUNCTIONS

- A function as series of instructions or group of statements with one specific purpose.
- A function is a program segment that carries out some specific, well defined task.
- A function is a self contained block of code that performs a particular task.

4.1 Types of functions

- C functions can be classified into **two** types,
 1. Library functions /pre defined functions /standard functions /built in functions
 2. User defined functions
- 1. Library functions /pre defined functions /standard functions/Built in Functions**
 - These functions are defined in the **library of C compiler** which are used frequently in the C program.
 - These functions are written by designers of c compiler.
 - C supports many built in functions like
 - Mathematical functions
 - String manipulation functions
 - Input and output functions
 - Memory management functions
 - Error handling functions
 - **EXAMPLE:**
 - `pow(x,y)`-computes x^y
 - `sqrt(x)`-computes square root of x
 - `printf()`- used to print the data on the screen
 - `scanf()`-used to read the data from keyboard.
- 2. User Defined Functions**
 - The functions written by the programmer /user to do the specific tasks are called user defined function(UDF's).
 - The user can construct their own functions to perform some specific task. This type of functions created by the user is termed as User defined functions.

Elements of User Defined Function

The Three Elements of User Defined function structure consists of :

- 1. Function Definition**
- 2. Function Declaration**
- 3. Function call**

1. Function Definition:

- A program Module written to achieve a specific task is called as function definition.
- Each function definition consists of two parts:
 - i. **Function header**
 - ii. **Function body**

General syntax of function definition

Function Definition Syntax	Function Definition Example
<pre>Datatype functionname(parameters) { declaration part; executable part; return statement; }</pre>	<pre>void add() { int sum,a,b; printf("enter a and b\n"); scanf("%d%d",&a,&b); sum=a+b; printf("sum is %d",sum); }</pre>

i. **Function header**

Syntax

datatype functionname(parameters)

- It consists of **three** parts

a) Datatype:

- ✓ **The data type can be int,float,char,double,void.**
- ✓ This is the data type of the value that the function is expected to return to calling function.

b) functionname:

- ✓ The name of the function.

-
- ✓ It should be a valid identifier.

c) parameters

- ✓ The parameters are list of variables enclosed within parenthesis.
- ✓ The list of variables should be separated by comma.

Ex: **void add(int a, int b)**

- In the above example the return type of the function is **void**
- the name of the function is **add** and
- The parameters are '**a**' and '**b**' of type integer.

ii. **Function body**

- The function body consists of the set of instructions enclosed between **{ and }** .
- The function body consists of following three elements:
 - a) **declaration part:** variables used in function body.
 - b) **executable part:** set of Statements or instructions to do specific activity.
 - c) **return :** It is a keyword, it is used to **return control back to calling function.**

If a function is not returning value then statement is:

return;

If a function is returning value then statement is:

return value;

2. Function Declaration

- The process of declaring the function before they are used is called as function declaration or function prototype.
- function declaration Consists of the data type of function, name of the function and parameter list ending with semicolon.

Function Declaration Syntax	
datatypefunctionname(type p1,type p2,.....type pn);	
Example	
int	add(int a, int b);
void	add(int a, int b);

Note: The function declaration should end with a semicolon ;

3. Function Call:

- The method of calling a function to achieve a specific task is called as function call.
- A function call is defined as **function name followed by semicolon.**
- A function call is nothing but invoking a function at the required place in the program to achieve a specific task.

Ex:

```
void main()
{
    add( ); // function call without parameter
}
```

Formal Parameters and Actual Parameters

- **Formal Parameters:**

- The variables defined in the **function header of function definition** are called **formal** parameters.
- All the variables should be separately declared and each declaration must be separated by **commas.**
- The formal parameters **receive the data from actual parameters.**

- **Actual Parameters:**

- The variables that are used when a function is invoked (in function call) are called **actual** parameters.
- Using actual parameters, the data can be transferred from calling function to the called function.
- The corresponding **formal** parameters in the **function definition** receive them.
- The **actual** parameters and **formal** parameters must match in number and type of data.

Actual Parameters	Formal Parameters
Actual parameters are also called as argument list . Ex: add(m,n)	Formal parameters are also called as dummy parameters . Ex: int add(int a, int b)
The variables used in function call are called as actual parameters	The variables defined in function header are called formal parameters
Actual parameters are used in calling function when a function is called or invoked Ex: add(m,n) Here, m and n are called actual parameters	Formal parameters are used in the function header of a called function. Example: int add(int a, int b) { } Here, a and b are called formal parameters.
Actual parameters sends data to the formal parameters Example:	Formal parameters receive data from the actual parameters.

- **Differences between Actual and Formal Parameters**

4.2 Categories of the functions

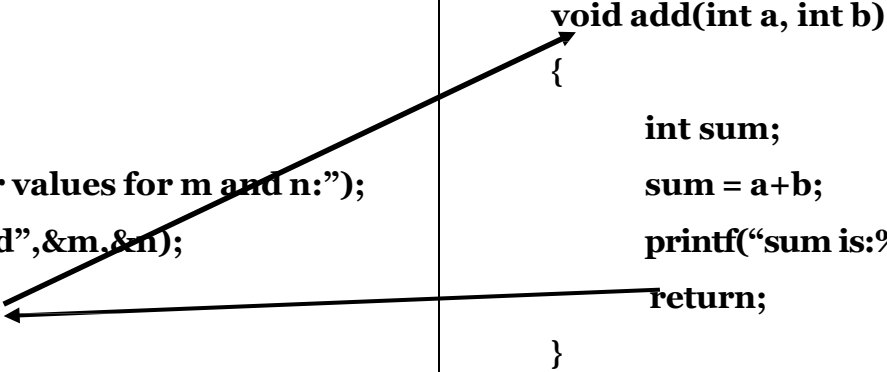
- 1. Function with no parameters and no return values**
- 2. Function with no parameters and return values.**
- 3. Function with parameters and no return values**
- 4. Function with parameters and return values**

1. Function with no parameters and no return values

1. Function with no parameters and no return values (void function without parameter)	
Calling function	Called function
<pre> /*program to find sum of two numbers using function*/ #include<stdio.h> void add(); void main() { add(); } </pre>	<pre> void add () { int sum; printf("enter a and b values\n"); scanf("%d%d",&a,&b); sum=a+b; printf("\n The sum is %d", sum); return; } </pre>
<ul style="list-style-type: none"> ✓ In this category no data is transferred from calling function to called function, hence called function cannot receive any values. ✓ In the above example,no arguments are passed to user defined function add(). ✓ Hence no parameter are defined in function header. ✓ When the control is transferred from calling function to called function a ,and b values are read,they are added,the result is printed on monitor. ✓ When return statement is executed ,control is transferred from called function/add to calling function/main. 	

2. Function with parameters and no return values

(void function with parameter)

Calling function	Called function
<pre>/*program to find sum of two numbers using function*/ #include<stdio.h> void add(int m, int n); void main() { int m,n; printf("enter values for m and n:"); scanf("%d %d",&m,&n); add(m,n); }</pre>	<pre>void add(int a, int b) { int sum; sum = a+b; printf("sum is:%d",sum); return; }</pre> 

- ✓ In this category, **there is data transfer** from the calling function to the called function using parameters.
- ✓ **But there is no data transfer from** called function to the calling function.
- ✓ The values of actual parameters m and n are copied into formal parameters a and b.
- ✓ The value of a and b are added and result stored in sum is displayed on the screen in called function itself.

3. Function with no parameters and with return values

Calling function

```
/*program to find sum of two numbers  
using function*/
```

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

```
int add();
```

```
void main()
```

```
{
```

```
    int result;
```

```
    result=add();
```

```
    printf("sum is:%d",result);
```

```
}
```

Called function

```
int add() /* function header */
```

```
{
```

```
    int a,b,sum;
```

```
    printf("enter values for a and  
b:");
```

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

```
    sum= a+b;
```

```
    return sum;
```

```
}
```

- ✓ In this category **there is no data transfer from the calling function to the called function.**
- ✓ But, there is data transfer from called function to the calling function.
- ✓ No arguments are passed to the function add(). So, no parameters are defined in the function header
- ✓ When the function returns a value, the calling function receives one value from the called function and assigns to variable result.
- ✓ The result value is printed in calling function.

4. Function with parameters and with return values	
Calling function	Called function
<pre> /*program to find sum of two numbers using function*/ #include<stdio.h> int add(); void main() { int result,m,n; printf("enter values for m and n:"); scanf("%d %d",&m,&n); result=add(m,n); printf("sum is:%d",result); } </pre>	<pre> int add(int a, int b) /* function header */ { int sum; sum= a+b; return sum; } </pre>
<ul style="list-style-type: none"> ✓ In this category, there is data transfer between the calling function and called function. ✓ When Actual parameters values are passed, the formal parameters in called function can receive the values from the calling function. ✓ When the add function returns a value, the calling function receives a value from the called function. ✓ The values of actual parameters m and n are copied into formal parameters a and b. ✓ Sum is computed and returned back to calling function which is assigned to variable result. 	

4.3 Passing parameters to functions or Types of argument passing

The different ways of passing parameters to the function are:

- ✓ **Pass by value or Call by value**
- ✓ **Pass by address or Call by address**

1. Call by value:

- In call by value, the values of actual parameters are copied into formal parameters.
- The formal parameters contain only a copy of the actual parameters.
- So, even if the values of the formal parameters changes in the called function, the values of the actual parameters are not changed.
- The concept of call by value can be explained by considering the following program.

Example:

```
#include<stdio.h>

void swap(int a,int b);

void main()
{
    int m,n;
    printf("enter values for a and b:");
    scanf("%d %d",&m,&n);
    printf("the values before swapping are m=%d n=%d \n",m,n);
    swap(m,n);
    printf("the values after swapping are m=%d n=%d \n",m,n);
}

void swap(int a, int b)
{
    int temp;
    temp=a;
    a=b;
    b=temp;
}
```

- Execution starts from function main() and we will read the values for variables m and n, assume we are reading 10 and 20 respectively.
- We will print the values before swapping it will print 10 and 20.
- The function swap() is called with actual parameters m=10 and n=20.
- In the function header of function swap(), the formal parameters a and b receive the values 10 and 20.
- In the function swap(), the values of a and b are exchanged.

-
- But, the values of actual parameters m and n in function main() have not been exchanged.
 - The change is not reflected back to calling function.

2. Call by Address

- In Call by **Address**, when a function is called, the addresses of actual parameters are sent.
- In the called function, the formal parameters should be declared as pointers with the same type as the actual parameters.
- The addresses of actual parameters are copied into formal parameters.
- Using these addresses the values of the actual parameters can be changed.
- This way of changing the actual parameters indirectly using the addresses of actual parameters is known as pass by address.

Example:

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

```
void swap(int a,int b);
```

```
void main()
```

```
{
```

```
    int m,n;
```

```
    printf("enter values for a and b:");
```

```
    scanf("%d %d",&m,&n);
```

```
    printf("the values before swapping are m=%d n=%d \n",m,n);
```

```
    swap(&m,&n);
```

```
    printf("the values after swapping are m=%d n=%d \n",m,n);
```

```
}
```

```
void swap(int*a, int*b)
```

```
{
```

```
    int temp;
```

```
    temp=*a;
```

```
    *a=*b;
```

```
    *b=temp;
```

```
}
```

NOTE:

Pointer: A pointer is a variable that is used to store the address of another variable.

Syntax: datatype *variablename;

Example: int *p;

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

```
void main()
```

```
{
```

```
    int a, *p;
```

```
    p=&a;
```

```
}
```

In the above program **p** is a **pointer** variable, which is storing the address of variable **a**.

Differences between Call by Value and Call by reference

Call by Value	Call by Address
When a function is called the values of variables are passed	When a function is called the addresses of variables are passed
The type of formal parameters should be same as type of actual parameters	The type of formal parameters should be same as type of actual parameters, but they have to be declared as pointers .
Formal parameters contains the values of actual parameters	Formal parameters contain the addresses of actual parameters.

Scope and Life time of a variable

Scope of a variable is defined as the region or boundary of the program in which the variable is visible. There are two types

(i) Global Scope

(ii) Local Scope

i. Global Scope:

- The variables that are defined outside a block have global scope.
- That is any variable defined in global area of a program is visible from its definition until the end of the program.
- For Example, the variables declared before all the functions are visible everywhere in the program and they have global scope.

ii. Local Scope

- a. The variables that are defined inside a block have local scope.
- b. They exist only from the point of their declaration until the end of the block.
- c. They are not visible outside the block.

❖ Life Span of a variable

- The life span of a variable is defined as the period during which a variable is active during execution of a program.

For Example

- ❖ The life span of a global variable is the life span of the program.
- ❖ The life span of local variables is the life span of the function, they are created.

❖ Storage Classes

- *There are following storage classes which can be used in a C Program:*

- i. *Global variables*
- ii. *Local variables*
- iii. *Static variables*
- iv. *Register variables*

i. Global variables:

- These are the variables which are defined before all functions in global area of the program.
- Memory is allocated only once to these variables and initialized to zero.
- These variables can be accessed by any function and are alive and active throughout the program.
- Memory is deallocated when program execution is over.

e.g

```
#include<stdio.h>
int x;
main()
{
x=10;
printf("%d",x);
printf("x=%d",fun1());
printf("x=%d",fun2());
printf("x=%d",fun3());
}
```

ii. Local variables(automatic variables)

-
- These are the variables which are defined within a functions.
 - These variables are also called as automatic variables.
 - The scope of these variables are limited only to the function in which they are declared and cannot be accessed outside the function.
 - e.g

```
#include<stdio.h>
main()
{
    int m=1000;
    func2();
    printf("%d\n",m);
}
func1()
{
    int m=10;
    printf("%d\n",m);
}
```

iii. *Static variables*

- The variables that are declared using the keyword static are called static variables.
- The static variables can be declared outside the function and inside the function. They have the characteristics of both local and global variables.
- Static can also be defined within a function.

Ex: *static int a,b;*

```
#include<stdio.h>
main()
{
    int l;
    for (l=1;l<=3;l++)
        stat();
    stat()
    {
        static int x=0;
        x=x+1;
        printf("x=%d\n",x);
    }
}
```

iv. *Register variables*

-
- Any variables declared with the qualifier register is called a register variable.
 - This declaration instructs the compiler that the variable under use is to be stored in one of the registers but not in main memory.
 - Register access is much faster compared to memory access. Ex:
register int a;

Recursion

- Recursion is a method of solving the problem where the solution to a problem depends on solutions to smaller instances of the same problem.
- **Recursive function is a function that calls itself during the execution.**
- Consider Example for finding factorial of 5

Factorial(5)=n*fact(n-1)

```
return 5 * factorial(4) = 120
└─ return 4 * factorial(3) = 24
    └─ return 3 * factorial(2) = 6
        └─ return 2 * factorial(1) = 2
            └─ return 1 * factorial(0) = 1
```

$1 * 2 * 3 * 4 * 5 = 120$

Example 1.

/**** Factorial of a given number using Recursion *****/**

#include<stdio.h>

int fact(int n);

void main()

```
{
int num,result;
printf("enter number:");
scanf("%d",&num);
result=fact(num);
printf("The factorial of a number is: %d",result);
}
```

int fact(int n)

```
{
    if(n==0)
        return 1;
    else
        return (n*fact(n-1));
}
```

```
}  
output :  
enter number:5  
The factorial of a number is:120
```

Fibonacci Sequence:

The Fibonacci Sequence is the series of numbers:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

The next number is found by adding up the two numbers before it.

- The 2 is found by adding the two numbers before it (1+1)
- The 3 is found by adding the two numbers before it (1+2),
- And the 5 is (2+3),
- and so on!

Example 2.

/**** Factorial of a given number using Recursion *****/**

```
#include<stdio.h>  
int fibonacci(int);  
void main ()  
{  
    int n,f;  
    printf("Enter the value of n?");  
    scanf("%d",&n);  
    f = fibonacci(n);  
    printf("%d",f);  
}  
int fibonacci (int n)  
{  
    if (n==0)  
    {  
        return 0;  
    }  
    else if (n == 1)  
    {  
        return 1;  
    }  
    else  
    {  
        return fibonacci(n-1)+fibonacci(n-2);  
    }  
}
```