Function is an independent group of statements that performs a specific task. When any program is very long or same code is repeating many times then we try to cut the program in different modules or blocks (called function) so that whole program became more understandable, easier to debug (error checking) and size of code will be lesser.

Classification of Function: -

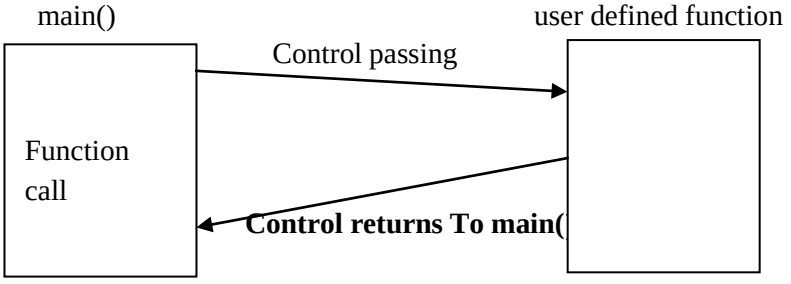
Functions can be classified into two categories

- **Library Function**: - This type of functions are defined by the language and provided along with the compile. For example printf(), scanf(), clrscr() and getch().
- **User defined function**: - The functions defined by the users are called user defined function. The main() and any other user defined function.

Functions have several advantages:

- Less code duplication – easier to read / update programs
- Simplifies debugging – each function can be verified separately
- Reusable code – the same functions can be used in different programs (e.g. printf).

How Functions works: - The basic building block of a program is function. Every C program is collection of one or more functions. Any C program must contains main() function because the execution of a program starts at main().



Example 1: Simple programs that add two integers using user defined function.

```
int add (int, int);                          //Function Prototype

void main()
{
        int sum, a,b;
        printf("Enter two numbers:\n");
        scanf("%d%d",&a,&b);
        sum= add(a,b);                       // Function Calling
        printf("sum=%d",sum);}


int add (int a, int b)                       //Function Header
{
        int c;                               // Function  Body
        c=a+b;
        return(c);                           //return statement
}
```

In this program main() itself is a function and it call the function **add(a,b)**. When **main()** calls the function **add(a,b)**, the control passes to the function **add(a,b)** and the activity of **main ()** is temporarily suspended while **add(a,b)** function starts its execution. When the **add(a,b**) function finished its execution, the control returns at the exact same point of **main()** from where **add(a,b)** was called.
Here main( ) is referred as **calling** function and add(a,b) is referred as **called** function.

Arguments: - values passed to a function are called arguments or paraeters.

a) **Formal Arguments**: - the variables declared in the function header are called Formal Arguments.
b) **Actual Arguments**: - the variables that are passed n a function call are called Actual Arguments.

Components of User Defined Function: -

Any user defined function must have following components.

a) Function Prototype
b) Function call
c) Function Definition

a) **Function Prototype** :- the function prototype declaration is a single statement written before any function definition and is terminated by a semicolon. It gives some basic information about the function to the compiler such as return type, name of the function and the type and number of arguments needed by a function. General syntax

<Reurn type><Function Type>(Data_Type, Data_Type,……. ,Data_Type);

Example:- int add (int, int);

b) **Function call**: - to utilize the function services, the function have to be called from main() or some other functions. General syntax

<Function name>(parameter list)

c) **Function Definition**: - function definition involves implementation of the function or writing the code of the function. It consist of

- Function Header
  - return type
  - name of the function
  - List of parameters with data type
- Function Body
  - Block of statements
  - Return statement

Categories of user defined function: - based on the return type and arguments afunction can be classified into four categories.

1. Function with no argument and no return value
2. Function with some argument and no return value
3. Function with no argument and with return value
4. Function with some argument and return value

1. **Function with no argument and no return value**: - function with no argument means it doesn't receive any data from the calling function. Function with no return value means(return type void) that the controls come back to called function without any value.

Eg

```
void print_msg();
void main()
{
    print_msg();
}
void print_msg()
{
    printf("Hello World");
}
```

2. **Function with some argument and no return value**: - function with some argument means it passes some data from the calling function to the called function. Function with no return value means(return type void) that the controls come back to called function without any value.

```
void add (int, int);                    //Function Prototype

void main()
{
        int a,b;
        printf("Enter two numbers:\n");
        scanf("%d%d",&a,&b);
        add(a,b);                       // Function Calling
        }


void add (int a, int b)                 //Function Header
{
        int c;                          // Function  Body
        c=a+b;
        printf("Sum=%d",c);
}
```

3. **Function with no argument and with return value**: - - function with no argument means it doesn't receive any data from the calling function. But the control returns back to called function with some specific type of value.

Eg

```
int add ( );                            //Function Prototype

void main()
{
        int sum;
        sum=add();                      // Function Calling
        printf("Sum=%d",sum);
        }


int add ( )                             //Function Header
{
        int c, a,b;
        printf("Enter two numbers:\n");
        scanf("%d%d",&a,&b);            // Function  Body
```

```
        c=a+b;
        return(c);                    //return statement
    }
```

4. **Function with some argument and return value**: - this type of function passes arguments to the calling function as well as returns values when the control returns to the caller.
   Eg Same as Example 1

Example : Write a C function to find factorial of a positive number.

```
#include <stdio.h>

int factorial(int);
void main()
    { int n, fact;
      clrscr();
      printf("Enter any number : ");
      scanf("%d",&n);
      fact=factorial(n);
      printf("The fatorial of %d is %d ",n,fact);
     getch();

    }
 int factorial(int n)
    {  int f=1;
      for(int i=1;i<=n;i++)
            f=f*i;
      return f;
    }
```

Example : write a c function to calculate $x^3+y^3+z^3$

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

int find_pow(int,int,int);

void main()
```

```
{
  int x,y,z,power;
  printf("enter three no.s ");
  scanf("%d%d%d",&x,&y,&z);
  power=find_pow(x,y,z);
  printf("power=%d",power);
}
int find_pow(int a, int b,int c)
{
  int p;
  p=(pow(a,3)+pow(b,3)+pow(c,3));
  return(p);
}
```

Example : Write a C function of power(a,b) to calculate the value of a raised to b.

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

int power(int,int);
void main()
{
 int a,b,pow;
clrscr();
 printf("enter the numbers");
 scanf("%d%d",&a,&b);
 pow=power(a,b);
 printf("power=%d",pow);
 getch();
}

int power(int x,int y)
{
  int i,p=1;
  for(i=1;i<=y;i++)
              p=p*x;
  return(p);
}
```

Example: - Write a function to calculate GCD of two given number.

```
int gcd(int, int);
```

```
void main()
{ int a, b, result;
   printf("Enter the two numbers to find their GCD: ");
   scanf("%d%d", &a, &b);
   result = gcd(a, b);
   printf("The GCD of %d and %d is %d.\n", a, b, result);
}

int gcd(int a, int b)
{    while (a != b)
   {  if (a > b)
         a=a - b;
      else
          b=b - a;
    }

   return a;
}
```

Passing Arguments to Function:- In 2 different methods parameter or arguments can be passed from calling function to the called function.

Call by value: - In this mode function calling, the values of the actual arguments of the calling function are transferred to the formal arguments of the called function. C makes different copies of the function arguments. So two different copies of the function arguments exist ---one inside calling function and another inside called function. So changes taking place in the formal arguments will not reflected to the corresponding actual argument.

Example:- swapping value of two variable using call by value.

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

int swap(int , int);                    // Prototype Declaration of function

main( )
{
        int a = 10, b = 20 ;
        swap(a,b);                      // call by value and a and b are actual parameters
```

```
        printf ( "\na = %d b = %d", a, b ) ;
        getch();
}


//Function defination
int swap( int x, int y )                    // x and y are formal parameters
{
        int  t  ;
        t = x ;
        x = y ;
        y = t ;
        printf ( "\nx = %d y = %d", x, y ) ;
}
```

Call by Address/call by reference: - In this mode function calling, the addresses of the actual arguments of calling function are transferred to the formal arguments of called function. When we pass the arguments by reference, only one copy of the arguments is kept in the memory. Any changes that take place in the formal argument will be immediately reflected to the corresponding actual argument.

Example:- swapping value of two variable using call by address.

```
#include <stdio.h>

void swap(int*, int*);

void main()
{
        int x, y;
        printf("Enter the value of x and y\n");
        scanf("%d%d",&x,&y);
        printf("Before Swapping\nx = %d\ny = %d\n", x, y);
        swap(&x, &y);
        printf("After Swapping\nx = %d\ny = %d\n", x, y);
}
void swap(int *a, int *b)
{
        int temp;
```

```
        temp = *b;
        *b  = *a;
        *a  = temp;
}
```

Difference between call by value and call by address/reference:-

call by value	call by address/reference
The values of the actual arguments of the calling function are transferred to the formal arguments of the called function.	The addresses of the actual arguments of calling function are transferred to the formal arguments of called function.
Any changes that take place in the formal argument will not reflect to the corresponding actual argument.	Any changes that take place in the formal argument will be immediately reflected to the corresponding actual argument.
Two different copies of the function arguments are kept in the memory.	Only one copy of the function arguments is kept in the memory
This method consumes more memory space wrt. call by reference.	This method consumes less memory space wrt. call by value. .

Recursion: - If a statement within the body of a function calls the function itself until a specific base condition is satisfied--- this is called **recursion**. A function is called **recursive** function if there exists a statement in its body for the function call itself.

Advantages: -

1. Algorithms are easier to understand.
2. Codes are shorter in recursive function.

Disadvantage: -

1. The computer can run out of memory if the recursion calls are not checked.
2. I is not efficient in term of speed and execution time.

Difference between recursion and iteration

Recursion	Iteration.
1. Recursion is a technique of defining any terms by itself.	1. It is a process of executing a statement or a set of statements repeatedly, until some specific condition.
2. All problems don't have recursive solution.	2. Any recursive problem can be solved iteratively.

3. There must be exclusive if-statement inside recursive function, specifying stopping condition.	3. Iteration involves clear cut steps initiation, condition, execution and updation.
4. Recursion is actually a worse option for solving simple problem.	4. Iteration is more efficient in terms of memory utilization and execution time.

Example: - Write a function to find the factorial of a number using recursion.

```
#include<stdio.h>

int fact(int);       //prototype

void main(){
  int num,f;
  printf("\nEnter a number: ");
  scanf("%d",&num);
  f=fact(num);
  printf("\nFactorial of %d is: %d",num,f);
  return 0;
}

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

Explanation of the above program: -

First time fact() is called from main(), with n=3. From here, since the n not equal to 1, the if block is skipped and fact() is called again with the argument (n-1) i.e. 2. The recursive call is terminated when n becomes 1.

Example: - Write a recursive function to calculate GCD of two given number.

```
int gcd(int, int);

void main()
{  int a, b, result;
   printf("Enter the two numbers to find their GCD: ");
   scanf("%d%d", &a, &b);
```

```
    result = gcd(a, b);
    printf("The GCD of %d and %d is %d.\n", a, b, result);
}

int gcd(int a, int b)
{   while (a != b)
    {  if (a > b)
          return gcd(a - b, b);
       else
          return gcd(a, b - a);
     }

    return a;
}
```

Example: - Write a recursive function for Fibonacci series.

```
#include<stdio.h>

int Fibonacci(int);
main()
{
  int n, i = 0, c;
  scanf("%d",&n);
  printf("Fibonacci series\n");
  for ( c = 1 ; c <= n ; c++ )
  {
    printf("%d\n", Fibonacci(i));
    i++;
  }
   return 0;
}

int Fibonacci(int n)
{
  if ( n == 0 || n == 1)
    return n;
  else
    return ( Fibonacci(n-1) + Fibonacci(n-2) );
}
```