FUNCTIONS:

Definition: A function is a block of code/group of statements/self contained block of statements/ basic building blocks in a program that performs a particular task. It is also known as ***procedure*** or ***subroutine*** or **module**, in other programming languages.

To perform any task, we can create function. A function can be called many times. It provides *modularity* and code *reusability*.

Advantage of functions

1) **Code Reusability**

By creating functions in C, you can call it many times. So we don't need to write the same code again and again.

2) **Code optimization**

It makes the code optimized we don't need to write much code.

3) **Easily to debug the program.**

Example: Suppose, you have to check 3 numbers (781, 883 and 531) whether it is prime number or not. Without using function, you need to write the prime number logic 3 times. So, there is repetition of code.

But if you use functions, you need to write the logic only once and you can reuse it several times.

Types of Functions

There are two types of functions in C programming:

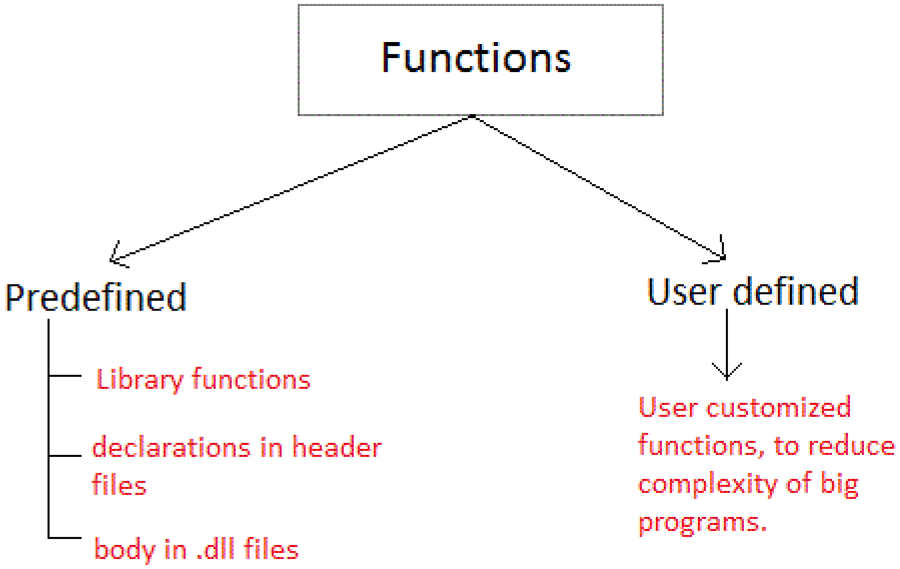
1. Library Functions: are the functions which are declared in the C header files such as scanf(), printf(), gets(), puts(), ceil(), floor() etc. You just need to include appropriate header files to use these functions. **These are already declared and defined in C libraries. links to be Remembered**

System defined functions are declared in header files

System defined functions are implemented in .dll files. (DLL stands for Dynamic Link Library).

To use system defined functions the respective header file must be included.

2. User-defined functions: are the functions which are created by the C programmer, so that he/she can use it many times. It reduces complexity of a big program and optimizes the code. Depending upon the complexity and requirement of the program, you can create as many user-defined functions as you want.


Functions
Predefined
Library functions
declarations in header files
body in .dll files
User defined
User customized functions, to reduce complexity of big programs.

ELEMENTS OF USER-DEFINED FUNCTINS:

In order to write an efficient user defined function, the programmer must familiar with the following three elements.

1 : Function Declaration. (Function Prototype).

2 : Function Call.

3 : Function Definition

Function Declaration. (Function Prototype).

A function declaration is the process of tells the compiler about a function name.

Syntax

return_type function_name(parameter/argument);

return_type function-name();

Ex : int add(int a,int b);

int add();

Note: At the time of function declaration function must be terminated with ;.

Calling a function/function call

When we call any function control goes to function body and execute entire code.

Syntax : function-name();

function-name(parameter/argument);

return value/ variable = function-name(parameter/argument);

Ex : add(); // function without parameter/argument

add(a,b); // function with parameter/argument

c=fun(a,b); // function with parameter/argument and return values

Function definition

Function definition contains programming codes to perform specific task.

Syntax of function definition

```
return_type function_name(type(1) argument(1),..,type(n) argument(n))
{
        //body of function
}
```

Function definition has two major components:

1. Function declaration

Function **declaration** is the first line of function definition. When a function is called, control of the program is transferred to function **declaration**.

Syntax of function declarator

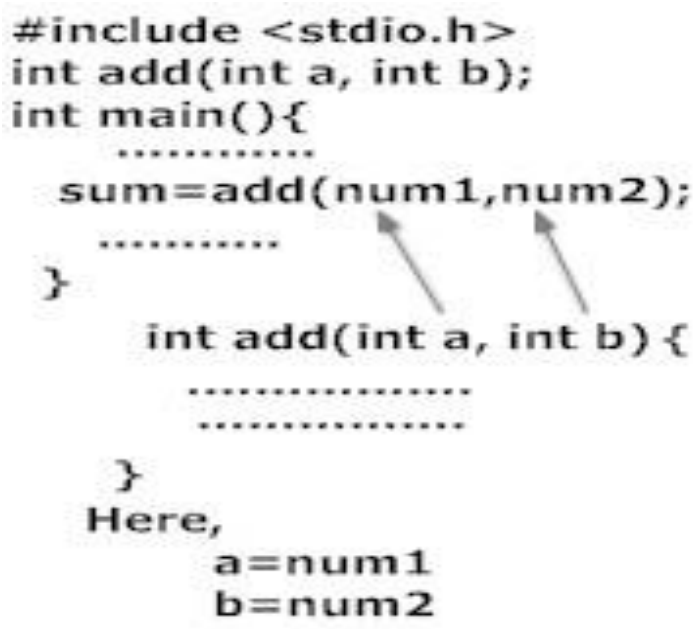
return_type function_name(type(1) argument(1),....,type(n) argument(n))

Syntax of function declaration and declaration are almost same except, there is no semicolon at the end of declaration and function declaration is followed by function body.

In above example, int add(int a,int b) in line 12 is a function decleration.

2. Function body

Function declaratory is followed by body of function inside braces.



```
#include <stdio.h>
int add(int a, int b);
int main(){
      ............
   sum=add(num1,num2);
     ..........
  }
        int add(int a, int b){
           ................
           ...............
     }
     Here,
            a=num1
            b=num2
```

Arguments that are passed in function call and arguments that are accepted in function definition should have same data type. For example:

If argument *num1* was of int type and *num2* was of float type then, argument variable *a* should be of type int and b should be of type float,i.e., type of argument during function call and function definition should be same.

A function can be called with or without an argument.

Return Statement

Return statement is used for returning a value from function definition to calling function.

Syntax of return statement

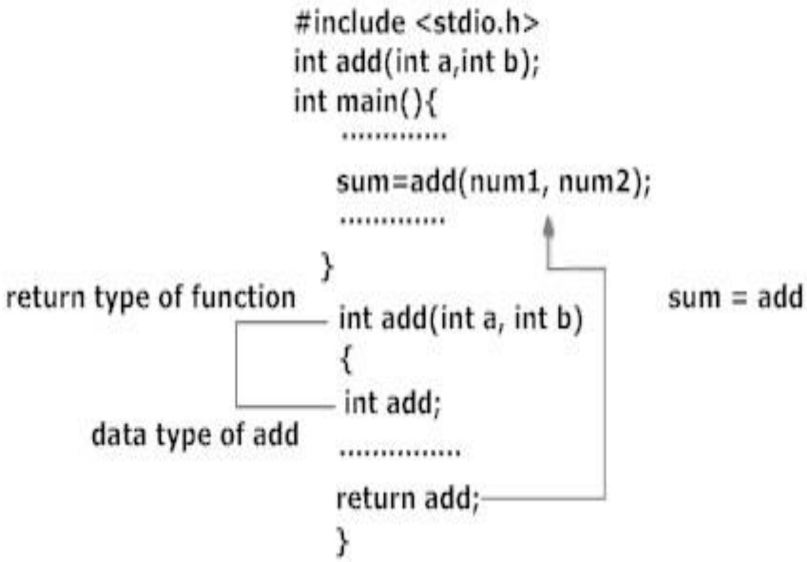
return (expression);

For example:

return a;

return (a+b);

In above example, value of variable add in add() function is returned and that value is stored in variable *sum* in main() function. The data type of expression in return statement should also match the return type of function.



```
#include <stdio.h>
int add(int a,int b);
int main(){
    .............
    sum=add(num1, num2);
    .............
  }
    int add(int a, int b)
    {
     int add;
    ..............
    return add;
    }
```

Parameters:

Parameters provide the data communication between the calling function and called function.

They are two types of parameters

1 : Actual parameters.

2 : Formal parameters.

1 : Actual Parameters : These are the parameters transferred from the calling function (main program) to the called function (function).

2 : Formal Parameters :These are the parameters transferred into the calling function (main program) from the called function(function).

- ❖ The parameters specified in calling function are said to be Actual Parameters.
- ❖ The parameters declared in called function are said to be Formal Parameters.
- ❖ The value of actual parameters is always copied into formal parameters.

```
Ex : main()
{
fun1( a , b ); //Calling function
}
fun1( x, y ) //called function
{
. . . . . .
}
```

Where

a, b are the Actual Parameters

x, y are the Formal Parameters

Difference between Actual Parameters and Formal Parameters

Actual Parameters	**Formal Parameters**
Actual parameters are used in calling function when a function is invoked.	Formal parameters are used in the function header of a called function.

Actual parameters can be constants, variables or expression.
Ex : c=add(a,b) **//variable**
c=add(a+5,b); **//expression.**
c=add(10,20); **//constants.**
	2 : Formal parameters should be only variable. Expression and constants are not allowed.
	Ex : int add(int m, int n); //**CORRECT**
	int add(int m+n,int n) //**WRONG**
	int add(int m,10); //**WRONG**

Actual parameters send values to the formal parameters. **Ex :** c=add(4,5);	Formal parameters receive values from the actual parameters.
Address of actual parameters can be sent to formal parameters	if formal parameters contains address, they should be declared as pointers.

3: Formal parameters receive values from the actual parameters.

1. Functions with no Parameters and no Return Values.

2: Functions with no Parameters and Return Values.

3: Functions with Parameters and no Return Values.

4: Functions with Parameters and Return Values.

1. Functions with no Parameters and no Return Values.

1. In this category, there is no data transfer between the calling function and called function.

2: But there is flow of control from calling function to the called function.

3: When no parameters are there, the function cannot receive any value from the calling function.

4: When the function does not return a value, the calling function cannot receive any value from the called function.

/*C program to check whether a number entered by user is prime or not using function with no arguments and no return value*/

```
#include <stdio.h>
void prime();
void main()
   {
  prime();      //No argument is passed to prime().
}
```

```
void prime()
{
/* There is no return value to calling function main(). Hence, return type of prime() is void */
    int num,i,flag=0;
    printf("Enter positive integer enter to check:\n");
    scanf("%d",&num);
    for(i=2;i<=num/2;++i){
        if(num%i==0){
            flag=1;
         }
    }
    if (flag==1)
        printf("%d is not prime",num);
    else
       printf("%d is prime",num);
    }
```

Function with no arguments but return value

```
/*C program to check whether a number entered by user is prime or not using function with
    no arguments but having return value */
#include <stdio.h>
int input();
int main(){
    int num,i,flag = 0;
    num=input();     /* No argument is passed to input() */
    for(i=2; i<=num/2; ++i){
    if(num%i==0){
        flag = 1;
        break;
    }
    }
    if(flag == 1)
        printf("%d is not prime",num);
    else
        printf("%d is prime", num);
    return 0;
}
int input(){   /* Integer value is returned from input() to calling function */
    int n;
    printf("Enter positive integer to check:\n");
    scanf("%d",&n);
    return n;
}
```

Function with arguments and no return value

```
#include<stdio.h>
#include<conio.h>
void sum(int a,int b);
void main()
{
int m,n;
printf("Enter m and n values:");
scanf("%d%d",&m,&n);
sum(m,n);
getch();
}
        void sum(int a,int b)
        {
        int c;
        c=a+b;
        printf("sum=%d",c);
                    }
```

Functions with Parameters and Return Values.

```
#include<stdio.h>
#include<conio.h>
int sum(int a,int b);
void main()
{
int m,n,c;
printf("Enter m and n values");
scanf("%d%d",&m,&n);
c=sum(m,n);
printf("sum=%d",c);
}
int sum(int a,int b)
{
int c;
c=a+b;
return c;
}
```

PASSING PARAMETERS TO FUNCTIONS:

There are two ways to pass value or data to function in C language: ***call by value*** and ***call by reference*.** Original value is **not modified in call by value** but it is **modified in call by reference.**

C provides two mechanisms to pass parameters to a function.

1 : Pass by value (OR) Call by value.
2 : Pass by reference (OR) Call by Reference.

Pass by value (OR) Call by value :

When a function is called with actual parameters, the values of actual parameters are copied into formal parameters. If the values of the formal parameters changes in the function, the values of the actual parameters are not changed. This way of passing parameters is called pass by value or call by value.

OR

In call by value, value being passed to the function is locally stored by the function parameter in stack memory location. If you change the value of function parameter, it is changed for the current function only. It will not change the value of variable inside the caller method such as main().

Experiments #01:

```
#include<stdio.h>
#include<conio.h>
void swap(int ,int );
void main()
{
int i,j;
printf("Enter i and j values:");
scanf("%d%d",&i,&j);
printf("Before swapping:%d%d\n",i,j);
swap(i,j);
printf("After swapping:%d%d\n",i,j);
}
        void swap(int a,int b)
        {
        int temp;
        temp=a;
        a=b;
        b=temp;
}
```

OUTPUT

```
Enter i and j values: 10 20
Before swapping: 10 20
                         After swapping: 10 20
```

Pass by reference (OR) Call by Reference:

In pass by reference, a function is called with addresses of actual parameters. In the function header, the formal parameters receive the addresses of actual parameters. Now the formal parameters do not contain values, instead they contain addresses. Any variable if it contains an address, it is called a pointer variable. Using pointer variables, the values of the actual parameters can be changed. This way of passing parameters is called call by reference or pass by reference.

```
#include<stdio.h>
#include<conio.h>
void swap(int *,int *);
void main()
{
int i,j;
printf("Enter i and j values:");
scanf("%d%d",&i,&j);
printf("Before swapping:%d%d\n",i,j);
swap(&i ,&j);
printf("After swapping:%d%d\n",i,j);
}
void swap(int *a,int *b)
{
int temp;
temp=*a;
*a=*b;
*b=temp;
```

```
    }
```

Output

```
Enter i and j values: 10 20
Before swapping: 10 20
After swapping: 20 10
```

Difference between Call by value and Call by reference

Call by value	**Call by Reference**
1 : When a function is called the values of variables are passed	1: When a function is called the address of variables is passed.
2: Change of formal parameters in the function will not affect the actual parameters in the calling function.	**2:** The actual parameters are changed since the formal parameters indirectly manipulate the actual parameters.
3: Execution is slower since all the values have to be copied into formal parameters.	**3:** Execution is faster since only address is copied.

Standard Functions

The standard functions are built-in functions. In C programming language, the standard functions are declared in header files and defined in .dll files. In simple words, the standard functions can be defined as "the readymade functions defined by the system to make coding more easy". The standard functions are also called as library functions or pre-defined functions.

In C when we use standard functions, we must include the respective header file using #include statement. For example, the function printf() is defined in header file stdio.h (Standard Input Output header file). When we use printf() in our program, we must include stdio.h header file using #include<stdio.h> statement.

Header File	Purpose	Example Functions
stdio.h	Provides functions to perform standard I/O operations	printf(), scanf()
conio.h	Provides functions to perform console I/O operations	clrscr(), getch()
math.h	Provides functions to perform mathematical operations	sqrt(), pow()
string.h	Provides functions to handle string data values	strlen(), strcpy()
stdlib.h	Provides functions to perform general functions	calloc(), malloc()
time.h	Provides functions to perform operations on time and date	time(), localtime()
ctype.h	Provides functions to perform - testing and mapping of character data values	isalpha(), islower()
setjmp.h	Provides functions that are used in function calls	setjump(), longjump()
signal.h	Provides functions to handle signals during program execution	signal(), raise()
assert.h	Provides Macro that is used to verify assumptions made by the program	assert()
locale.h	Defines the location specific settings such as date formats and currency symbols	setlocale()
stdarg.h	Used to get the arguments in a function if the arguments are not specified by the function	va_start(), va_end(), va_arg()
errno.h	Provides macros to handle the system calls	Error, errno
float.h	Provides constants related to floating point data values	
limits.h	Defines the maximum and minimum values of various variable types like char, int and long	
stddef.h	Defines various variable types	
graphics.h	Provides functions to draw graphics.	circle(), rectangle()

STANDARD “C“ LIBRARY FUNCTIONS

1 : stdio.h

2 : stdlib.h

3 : string.h

4 : math.h

5 : ctype.h

6 : time.h

STANDARD I/O LIBRARY FUNCTIONS <STDIO.H>

Functions	DataType	Purpose
printf()	int	Send data items to the standared output device.
scanf()	int	Enter data items from the standard input device.
gets(s)	char	Enter string s from the standard input device.
getc(f)	int	Enter a string character from file f.
getchar()	int	Enter a single character from the standard input device.
putc(c,f)	int	Send a single character to file f.
puts(s)	int	Send string s to the standard output device.
putchar(c)	int	Send a single character to the standard output device.
fgetc(f)	int	Enter a single character from file f.
fgets(s,I,f)	char	Enter string s, containing I characters, from file f.
fprintf(f)	int	Send data items to file f.
fscanf(f)	int	Enter data items from file f.
fputc(c,f)	int	Send a single character to file f.
fputs(s,f)	int	Send string s to file f.
fread(s,il,i2,f)	int	Enter i2 data items, each of size i1 bytes, from file f.
fclose(f)	int	Close file f, return 0 if file is successfully closed.

STANDARD LIBRARY FUNCTIONS <STDLIB.H>

Functions	DataType	Purpose
abs(i)	int	Return the absolute value of i.
atof(s)	double	Convert string s to a double-precesion quantity.
calloc(u1,u2)	void*	Allocate memory for an array having u1 elements, each of length u2 bytes. Return a pointer to the beginning of the allocated space.
exit(u)	void	Close all files and buffers, and terminate the program.
free(p)	void	Free a block of allocated memory whose beginning is indicated by p.
malloc(u)	void*	Allocate u bytes of memory.
rand()	int	Return a random positive integer.
realloc(p,u)	void*	Allocate u bytes of new memory to the pointer variable p, return a pointer to the beginning of the new memory space.
system(s)	int	Pass command string s to the operating system.
srand(u)	void	Initialize the random number generator.

STRING LIBRARY FUNCTIONS <STRING.H>

Functions	DataType	Purpose
strlen()	int	Finds length of string
strlwr()	char	Converts a string to lowercase
strupr()	char	Converts a string to uppercase
strcat()	char	Appends one string at the end of another
strcpy()	char	Copies a string into another
strcmp()	char	Compares two strings
strrev()	char	Reverses string

MATH LIBRARY FUNCTIONS <MATH.H>

Functions	DataType	Purpose
acos(d)	double	Return the arc cosine of d.
atan(d)	double	Return the arc tangent of d.
asin(d)	double	Return the arc sine of d.
ceil(d)	double	Return a value rounded up to the next higher integer.
cos(d)	double	Return the cosine of d.
cosh(d)	double	Return the hyperbolic cosine of d.
exp(d)	double	Raise e to the power d.
fabs(d)	double	Return the absolute value of d.
floor(d)	double	Return a value rounded down to the next lower integer.
labs(l)	long int	Return the absolute value of l.
log(d)	double	Return the natural logarithm of d.
pow(d1,d2)	double	Return d1 raised to the d2 power.
sin(d)	double	Return the sine of d.
sqrt(d)	double	Return squre root of d.
tan(d)	double	Return the tangent of d.

CHARACTER LIBRARY FUNCTIONS <CTYPE.H>

Functions	DataType	Purpose
isalnum(c)	Int	Determine if argument is alphanumeric. Return nonzero value if true, 0 otherwise.
isalpha(c)	Int	Determine if argument is alphabetic. Return nonzero value if true, 0 otherwise.
isascii(c)	Int	Determine if argument is an ASCII character,. Return nonzero value if true, 0 otherwise.
isdigit(c)	Int	Determine if argument is a decimal digit. Return

		nonzero value if true, 0 otherwise.
isgraph(c)	Int	Determine if argument is a graphic printing ASCII Character. Return nonzero value if true, 0 otherwise.
islower(c)	Int	Determine if argument is lowercase. Return nonzero value if true, 0 otherwise.
isprint(c)	Int	Determine if argument is a printing ASCII character. Return nonzero value if true, 0 otherwise.
isspace(c)	Int	Determine if argument is a whitespace character. Return nonzero value if true, 0 otherwise.
isupper(c)	Int	Determine if argument is uppercase. Return nonzero value if true, 0 otherwise.
toascii(c)	Int	Convert value of argument to ASCII
tolower(c)	Int	Convert letter to lowercase
toupper(c)	Int	Convert letter to uppercase.

Recursion

- When function is called within the same function, it is known as recursion in C.
- A function that calls itself, and doesn't perform any task after function call, is known as tail recursion. In tail recursion, we generally call the same function with return statement.

Features:

- There should be at least one if statement used to terminate recursion.
- It does not contain any looping statements.

Advantages:

- It is easy to use.
- It represents compact programming structures.

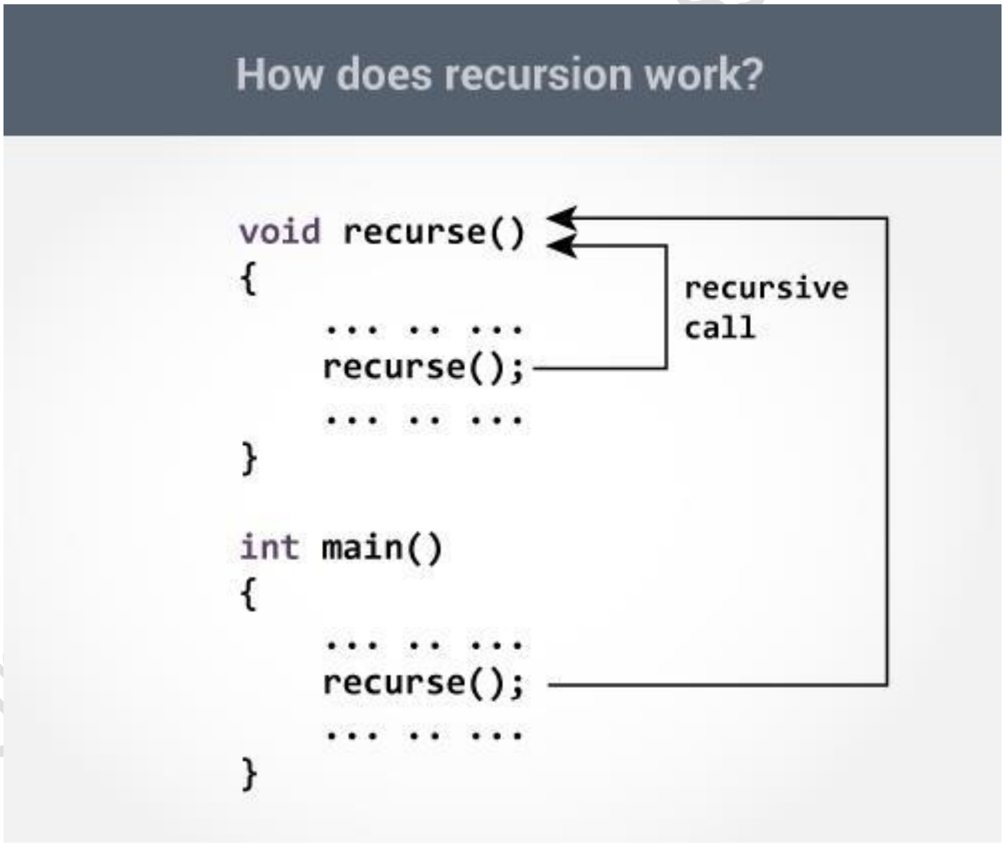
Disadvantages:

- It is slower than that of looping statements because each time function is called.

Note: while using recursion, programmers need to be careful to define an exit condition from the function; otherwise it will go into an infinite loop. Recursive functions are very useful to solve many mathematical problems, such as calculating the factorial of a number, generating Fibonacci series, etc.

Example of recursion.

```
recursionfunction()
{
recursionfunction();//calling self function
}
```



How does recursion work?

```
void recurse()
{
     ... .. ...
     recurse();
     ... .. ...
}

int main()
{
     ... .. ...
     recurse();
     ... .. ...
}
```

Experiments:#01: Write a C program to find sum of first n natural numbers using recursion. Note: Positive integers are known as natural number i.e. 1, 2, 3....n

```
#include <stdio.h>
int sum(int n);
int main(){
    int num,add;
    printf("Enter a positive integer:\n");
    scanf("%d",&num);
    add=sum(num);
    printf("sum=%d",add);
}
int sum(int n){
    if(n==0)
       return n;
    else
       return n+sum(n-1);    /*self call  to function sum() */
}
```

```
OUTPUT
5
15
```

In, this simple C program, sum() function is invoked from the same function. If *n* is not equal to 0 then, the function calls itself passing argument 1 less than the previous argument it was called with. Suppose, *n* is 5 initially. Then, during next function calls, 4 is passed to function and the value of argument decreases by 1 in each recursive call. When, n becomes equal to 0, the value of n is returned which is the sum numbers from 5 to 1.

For better visualization of recursion in this example:

```
sum(5)
=5+sum(4)
=5+4+sum(3)
=5+4+3+sum(2)
```

=5+4+3+2+sum(1)

=5+4+3+2+1+sum(0)

=5+4+3+2+1+0

=5+4+3+2+1

=5+4+3+3

=5+4+6

=5+10

=15

Experiments:#02 Source code to Calculate H.C.F using recursion

```
#include <stdio.h>
int hcf(int n1, int n2);
int main()
{
  int n1, n2;
  printf("Enter two positive integers: ");
  scanf("%d%d", &n1, &n2);
  printf("H.C.F of %d and %d = %d", n1, n2, hcf(n1,n2));
  return 0;
}
int hcf(int n1, int n2)
{
   if (n2!=0)
     return hcf(n2, n1%n2);
   else
     return n1;
}
```

Assignments

Q1.Program to print Fibonacci Series using recursion

Q2.Find Factorial of Number Using Recursion