Functions

A function is a complete program in itself, similar to C main( ) function except that the name main( ) is a keyword and it should be replaced by any name.

Types of functions

- **Pre-defined / library functions**
- **User defined function**

Library Functions:

Output/ Input function: stdio.h

printf(), scanf(), getch(), getche(), getchar(), putchar(), puts(), gets()

Character function: string function- string.h

tolower(), toupper(), isalpha, isdigit, islower, isupper

String function

strcmp(), strcpy(), strlen(),strrev()

Log Function

log(), $\log_{10}$() a=20 log(a)

Exponential function: exp() 12345000000000 12345 +E09

Absolute function: abs()

Mathematical function:

sqrt(), round(), ceil(), floor() ceil(123.45) 124

User defined function:

A function is a self-contained block of statements that carries out some specific, well defined task specified by the user and is called as user defined function or simply a function. A C program can contain one or more than one functions. The one out of which must be main( ), from which the execution of program begins with. There is no UD function that can work independently rather it has to be called within the main( ) function.

Syntax

```
<Return type>  function name(arguments)  pass
{
   Stmt block
   return()
}
```

where <return type>—> is the type of value to be returned by the function, if no value is returned then keyword void should be used.

Function name is an user defined name of the function. The function can be called from another function by this name.

arguments --> is a list of parameters. This can be omitted also by leaving the parameters empty ().

The statement block enclosed within the opening brace and closing brace is known as the function body.

Features:

1. C program is a collection of one or more functions.

2. A function gets called when the function name is followed by a semicolon

```
void main( )
{
  sum( ) ;
}
```

3. A function is defined when function name is followed by a pair of braces in which one or more statements may be present.

```
sum( )
{
  Stmt block;
}
```

4. Any function can be called from any other function.

5. A function once written, can be called any number of times.

```
main()
{
  sum();
  sum();
}
```

6. The order of function called and the order of function defined may be different.

```
main()
{
  sum1();
  sum2();
  sum3();
}
sum3()
```

```
{
  stmt block:
}
sum1()
{
  stmt block:
}
sum2()
{
  stmt block:
}
```

7. A function can call itself also which is called recursion.

Example

```
void main( )
{
  clrscr( );
  printf (" \ n Hello, How are you ?");
  message( ) ; /* function call */
  printf (" n Do not Worry I am back in main");
  getch( );
}
message( ) /* function definition */
{
  printf (" n Where are you moving out ?");
}
```

In this program, the function message( ) is called. As the program starts executing, it prints first line as it is. In the second line function message( ) is

being called, that means control is passing to the function message( ) and the activity of the main( ) is temporarily suspended. message( ) function print its message and control is back to the main( ) function and it will print the last message and runs out of main( ) function *i.e.* the program ends.

Advantages of function

1. **Logical Clarity The use of user defined functions allow a large program to be broken down into a number of smaller, self-contained components, each of which has some unique, identifiable purpose. Large programs are very difficult to understand. So the program becomes easier to write and debug and their logic becomes much more clearer.**

2. **Avoids repetition Using function avoids repetition of the same instructions again and again. Different programs can use a single function. In a program same function can be called at different places, avoids the repetition. It saves both time and space.**
3. **Promotes portability The functions can be written that arc independent of system dependent features. Thus it promotes the portability.**
4. **Saves memory space Different data set can be transferred to the function each time when it is accessed. Thus it saves a lot of memory space by allowing same set of statements to be used for different set of data.**
5. **Modularization Decomposing a large program into smaller modules makes the designing of the program easier and faster. Different persons can work on different modules which can be further combined to make the large program complete.**

6. **Reduced length** The length of the source program can be reduced by using functions at appropriate places in a program.

Defining a Function

The definition of function starts with declaring its prototype which consists of the name of the function, the data type of the return value and the data type of the arguments. The body of the function is enclosed within braces if the function is returning a value there must be at least one return statement.

```
float sum (float x, float y)     float sum(float,float)
{
  float s;
  s = x + y;
  return (s);
}
```

Features of function definition

1. The first statement of the function definition contains return type, name and the formal arguments.
2. If the return type of the function is not specified, then it is assumed to be of type int.
3. The arguments used in function definition are called formal arguments. These should be of same in type as well as number as used in function call. Formal arguments cannot be constants or expressions.
4. Rules for naming a function are same as for variable name.
5. The body of the function should be enclosed in braces.
6. The return type must be the same as mentioned in the function call.

Function declaration and prototypes

A function to be used in a program must have its prototype declared.

<data type of return value> <name> (data types of arguments);

where <type> is a type of value to be returned by the function

<name> is user defined name of the function

argument -> is a list of parameters

; --> The semicolon marks the end of prototype declaration

Eg.

```
float area(float );
void message( ) ;
int sum(int x, int y);
```

Therefore a function is declared by declaring its prototype.

Function call:

A function can be accessed by specifying its name, followed by a list of arguments enclosed in parentheses and separated by commas.

```
sum(a,b)
```

If the function call does not require any arguments, an empty pair of parentheses must follow the name of the function.

```
sum();
```

The arguments within the function call are called actual parameters or actual arguments. The actual arguments may be constants, variables or more complex expressions. Each actual argument must be of the same type as its corresponding formal arguments.

Eg:

```
sum (a, b);
```

where sum-->is the name of the function,

a, b-->are the actual arguments within parentheses.

There will be no return value in this function call and it ends with semicolon(;)

If the function returns a value, then the function call is often written as an assignment statement *Eg:*

a = area (rad);

or s = sum (nl, n2);

These functions will return a value and assign to a and s respectively.

The return statement

Syntax:

return (exp);

where exp can be a constant, variable or an expression.

Purposes :

1. return statement immediately transfers the control back to the calling program.

2. It returns the value after return to the calling program.

3. The limitation of return is it can return only one value.

e.g.

```
big (int a, int b)
{
  if (a > b)
    return(a);
 else
   return (b);
}
```

In this function, if the value of a is greater than b then a will be returned otherwise b will be returned to calling function.

Actual and formal arguments:

The arguments of the calling functions are called Actual arguments and the arguments of the called function is called formal or dummy arguments. These

are called formal arguments because they represent the names of data items that are transferred into the function from the calling function of the program. The formal arguments are local to the function because they are not recognized outside the function.

Eg:

```
main()
{
  int n1,n2,s;
  ....
  s=sum(n1,n2);              //actual parameter
}
Int sum(int a,int b)                  //formal parameter
{
  .....
}
```

Categories of functions:

1. Function with no arguments and no return value.
2. Function with arguments but no return value.
3. Function with no arguments but return a value.
4. Function with arguments and return value.

Functions with no arguments and no return value: In this type of function, main program will not send any arguments to the function and also the function will not return any value to the main program.

```
/* Program with no arguments but no return */
#include <stdio.h>
#include <conio.h>
void sum (); /* Declaration */
void main( )
{
  sum ();
  getch( );
}
void sum ()
{
  int a, b, sum;
  clrscr( );
  printf ("Enter the value of a & b");
  scanf ("%d%d", &a, &b");
```

```
  sum= a+b;
  printf ("sum = %d", sum);
}
```

Functions with arguments and no return value: In this category of the function or sub program, the main program will send any argument to the function but function sub-program will not return some value to the main program. The data can be transferred only from the function block to calling function. The values a, b are entered in the called function from the main function, calculations are performed in the function and after that result is calculated.

```
/* Program with arguments but no return */
#include <stdio.h>
#include <conio.h>
void sum (int, int); /* Declaration */
void main( )
{
   int a, b;
  clrscr( );
  printf ("Enter the value of a & b");
  scanf ("%d%d", &a, &b");
  sum (a, b);  actual parameter
  getch( );
}
void sum (int x, int y)   formal parameter
{
   int sum;
   sum= x+y;
```

```
  printf ("sum = %d", sum);
}
```

Functions with no argument but return a value

In this category of the function or sub program, main program will not send any argument to the function but function sub-program will send (return) some value to the main program. The data can be transferred only from the function block to calling function. The values a, b are entered in the function, calculation are performed in the function and after that result is calculated and it is returned to the calling function.

```
/* with no argument but return a value */
#include <stdio.h>
#include <conio.h>
void main( )
{
  int sum( ); /* Function Declaration*/
  int result;
  result = sum( );
  printf(" sum is = %d", result);
  getch( );
}
int sum( ) /* Function Definition*/
{
  int a, b, c;
clrscr();
  printf (" \n Enter the value of a and b");
  scanf ("%d%d", &a, &b);
```

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

Function with arguments and a return value : In this category of the function, the main program or the calling program will send arguments to the called program and at the end of the called program the function will send value back to the main program. Data communication takes place between both the calling portion of a program and the function block.

```
/* Program with arguments and with return */
#include <stdio.h>
#include <conio.h>
void main( )
{
  int sum (int, int); /* Declaration */
  int a, b, result;
  clrscr( );
  printf ("Enter the value of a and b");
  scanf ("%d%d", &a, &b);
  result= sum (a, b);
  printf ("sum = %d", result);
  getch( );
}
int sum (int a, int b) /* Definition */
{
  int c;
  c= a+b;
  return (c);
```

```
}
```

Passing arguments to a function:

- The two way communication between the various functions can be achieved through parameters or arguments and the return statement.
- The set of arguments defined in a function are called formal or dummy arguments whereas the set of corresponding arguments sent by the calling function are called actual arguments.
- The actual arguments must correspond in number, type and order with formal arguments specified in the function definition.
- The actual arguments can be constants, variables, expressions or array names.

Two ways

- Call the value
- Call the reference

Call by value

In this type of parameter passing, only the values are passed from the calling function to the called function, therefore this technique is called as call by value.

Write a program to calculate factorial of a given integer number by call by value function

```
/* Factorial of a given number */
#include <stdio.h>
#include <conio.h>
void main( )
{
  long fact (int); /* Function prototype is must if we want to
  int num; return value other than int */
  long f;
```

```
   clrscr( );
   printf (" \n Enter a number");
   scanf ("%d", &num);
   f = fact (num);
   printf (" \n Factorial of %d = %ld", num, f);
   getch( );
}
long fact (int n)
{
  int i;
  long fl = 1;
  for ( i = 1; i < = n ; i ++)
    fl = fl * i;
  return(f1);
}
```

output :

Enter a number 5

Factorial of 5 = 120

Call by Reference

If the changes made to the formal parameters be reflected to the corresponding actual parameters then we should use call by reference method of parameter passing. This technique passes the addresses or references of the actual parameters to the called function. Thus the actual and formal parameters share the same memory locations. This can be achieved by applying an address operator (&) to the actual parameters in the function call and value at address operator (*) to the formal parameters in the function definition.

Write a program to exchange the contents of two variables by using call by reference method

```
/* Exchange two variables by call by reference */
#include <stdio.h>
#include <conio.h>
exchange(int*, int*);
void main( )
{
  int a, b ;
  clrscr( );
  printf (" \n Enter two numbers");
  scanf ("%d%d", &a, &b);
  exchange (&a, &b); /* call by reference */
  printf (" \n The exchanged contents are %d %d", a, b);
  getch( );
}
exchange (int *x, int *y)
{
  int temp;
  temp = *x;
 *x = *y;
 *y = temp;
}
```

Output : Enter two numbers 10 20

The exchanged contents are 20 10

END

Write a program to find the sum of the following series using a function

```
/* Find the sum of series 1 + 2 + 3 + ....n */
#include <stdio.h>
#include <conio.h>
void main( )
{
  int n, sum;
  clrscr( );
  printf (" \n Enter last term of the series");
  scanf ("%d", &n);
  sum = series (n);
  printf (" \n sum of series = %d", sum);
  getch( );
}
series (int last)
{
   int 1, s = 0;
   for (1 = 1; i < = last; i ++)
       s+ = i;
   return (s);
}
```

Write a program to find the sum of digits of an integer number by using function.

```
/* sum of digits of a number */
#include <stdio.h>
#include <conio.h>
```

```
void main( )
{
int s ;
long n ;
clrscr( );
printf (" \n Enter a number");
scanf ("%ld", &n);
s = sumdig (n)
printf (" \n Sum of digits of Vold = %d", n, s);
getch( );
}
sumdig (long num)
{
int d, sum = 0;
while (num > 0)
{
d = num % 10;
num = num/ 10;
sum + = d;
}
return (sum);
}
```

Output : Enter a number 12345

Sum of digits of 12345 = 15

Write a program to compute Binomial coefficient ${}^{n}C_{r}$ using function

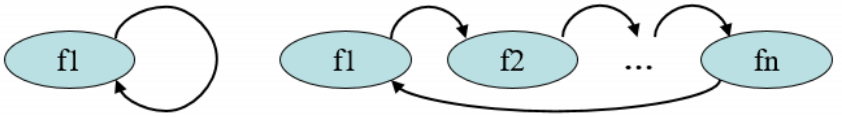
```
/* Compute Binomial coefficient */
#include <stdio.h>
#include <conio.h>
void main( )
{
  int ncr;
  long n, r;
  clrscr( );
  printf ("\n Enter value of n and r");
  scanf ("%ld%ld", &n, &r);
  ncr = fact (n) / (fact (r) * fact (n-r));
  printf (" n Value of ncr = %d", ncr);
  getch( );
}
fact (int num)
{
   int i;
   long f = 1;
   for (i = 1; i < = num ; i ++)
       f = f * i;
   return (f);
}
```

Recursion

Recursion is derived from the word recur which means repetition. In C, it is possible for the function to call itself. Recursion is the process by virtue of which a function or a procedure can call itself.

Therefore, the recursive function is

- **a kind of function that calls itself, or**
- **a function that is part of a cycle in the sequence of function calls.**



Eg:

```
fun()
{
  fun( );
}
```

This means that function fun( ) is calling itself.

There are two basic requirements for recursion :

1. The function must call itself again and again.

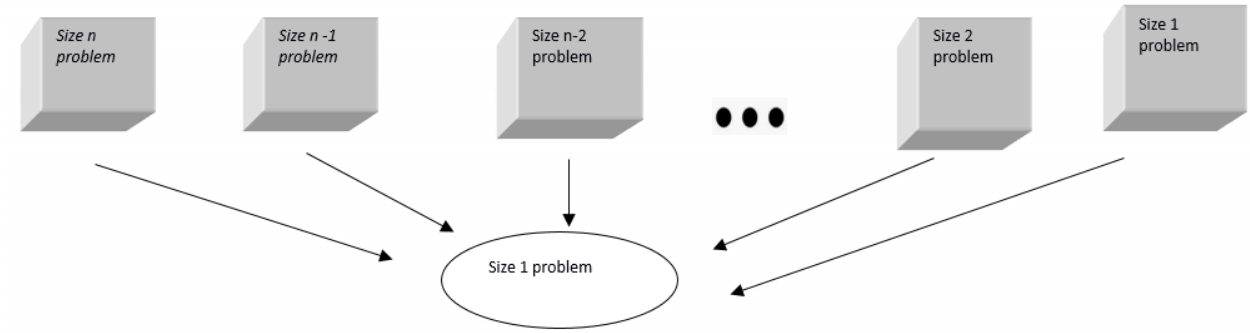
2. The function must have an exit condition.

Problems Suitable for Recursive Functions

- **One or more simple cases of the problem have a straightforward solution.**
- **The other cases can be redefined in terms of problems that are closer to the simple cases.**
- **The problem can be reduced entirely to simple cases by calling the recursive function.**

```
If this is a simple case
    solve it
else
   redefine the problem using recursion
```

Splitting a Problem into Smaller Problem

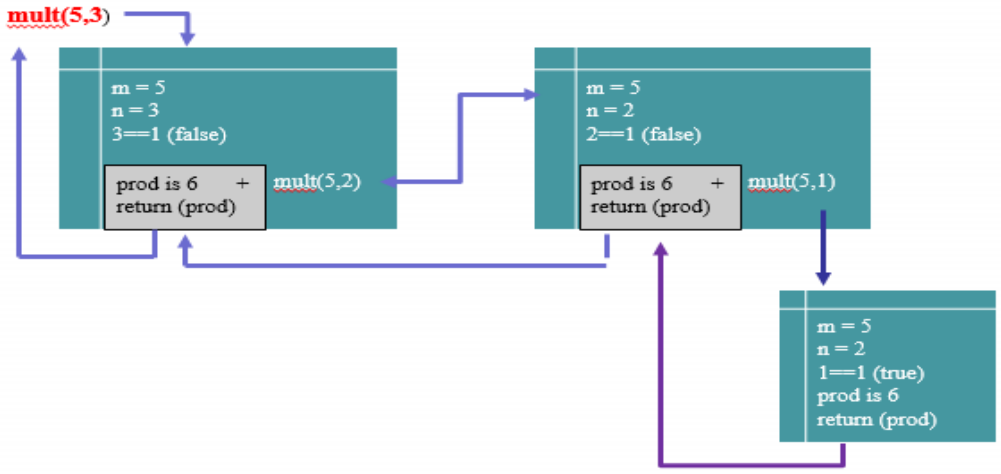


- **Assume that the problem of size 1 can be solved easily (i.e., the simple case).**
- **We can recursively split the problem into a problem of size 1 and another problem of size n-1.**

Example

Implementation of Multiplication using Addition

```
int mult(int m, int n)//
{
 int prod;
 if(n==1)
  prod=m;  // m*1=m, simple case
else
  prod=m+mult(m,n-1);  //m*n=m+m*(n-1), recursive step
return(prod);
}
```

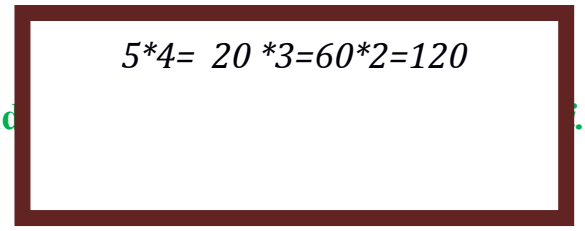


Recursive approach to find the factorial of n number

n! = n * (n-1)!

So function f(n) is defined in terms of f(n-1). This type of definition is known as recursive definition. Now by the same definition the factorial of n-1 can be defined as

(n-1)! = (n-1) * (n-2)!



We can continue this process till the end ***i.e.*** **0! which is equal to 1.**

Thus algorithm for calculating the factorial n is expressed in terms of an algorithm for calculating the factorial n-1 and so on.

```
if n = 0
   fact=1
else
   fact (n) = n * fact (n-1)          if n >0
```

Here the function fact has been defined

Write a program to compute factorial of a number using recursion.

```
/* Factorial of a number using recursion */
#include <stdio.h>
#include <conio.h>
int fact(int); //prototype
void main( )
{
  int num;
  clrscr( );
  printf (" \n Enter the number");
  scanf ("%d", &num);   4
  printf (" \n The factorial of %d = %d", num, fact (num));
```

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

Output

```
Enter any number 4
The factorial of 4 = 24
```

END