FUNCTIONS

Functions are the source code procedures that comprise a C program. They follow the general form;

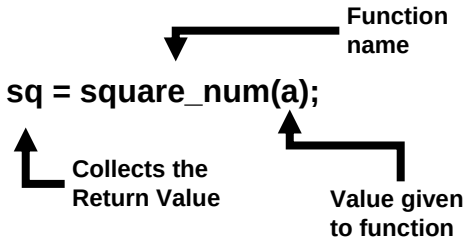
```
return_type function_name(parameter_list)
{
    statements
}
```

The return_type specifies the data type that will be returned by the function; char, int, double, void etc. C assumes that a function returns an integer value by default, if you don't mention a return type, its capable of returning an integer.

The code within a C function is invisible to any other C function, and jumps may not be made from one function into the middle of another, although functions may call other functions. Also, functions cannot be defined within functions, only within source modules. In other words, there is only one entry to a function but there can be multiple returns.

The following function computes the square of a number

```
main()
{
   int a,sq;
   printf("\nEnter the number ");
   scanf("%d",&a);
   sq = square_num(a);
   printf("\nThe square of %d is %d ",a,sq);
}
int square_num(int num)
{
   return (num*num);
}
```



Parameters may be passed to a function either by value, or by reference. If a parameter is passed by value, then only a copy of the current value of the parameter is passed to the function. A parameter passed by reference however, is a pointer to the actual parameter that may then be changed by the function. Its just like creating a Xerox copy of the notes, you can spoil the Xerox copy and original will remain unchanged but if u write on the original copy, you are modifying the contents.

Function prototypes

Prototypes for functions allow the C compiler to check that the type of data being passed to and from functions is correct. This is very important to prevent data overflowing its allocated storage space into other variables areas. A function prototype is placed at the beginning of the program, after any preprocessor commands, such as #include <stdio.h>, and before the declaration of any functions. It is written as

```
float square(float x);
main()
{
------
-----
}
float square(int x)
{
```

```
--
--
}
```

RECURSION

C allows a function to call another function, you have already done this while calling printf() scanf() etc from main(). A question that comes to mind is that "is it possible for a function to call itself in place of any other function"? The following scenario explains it

```
main()
{
    printf("\nInside main() function ");
    main(); /*call main function */
}
```

The output of this program is it enters an infinite execution sequence and prints *Inside main()* again and again.

Recursion is defined as a function calling itself. It is in some ways similar to a loop because it repeats the same code, but it requires passing in the looping variable and being more careful. Many programming languages allow it because it can simplify some tasks, and it is often more elegant than a loop. **This program will not continue forever, however**. The computer keeps function calls on a stack and once too many are called within ending, the program will terminate. Why not write a program to see how many times the function is called before the program terminates?

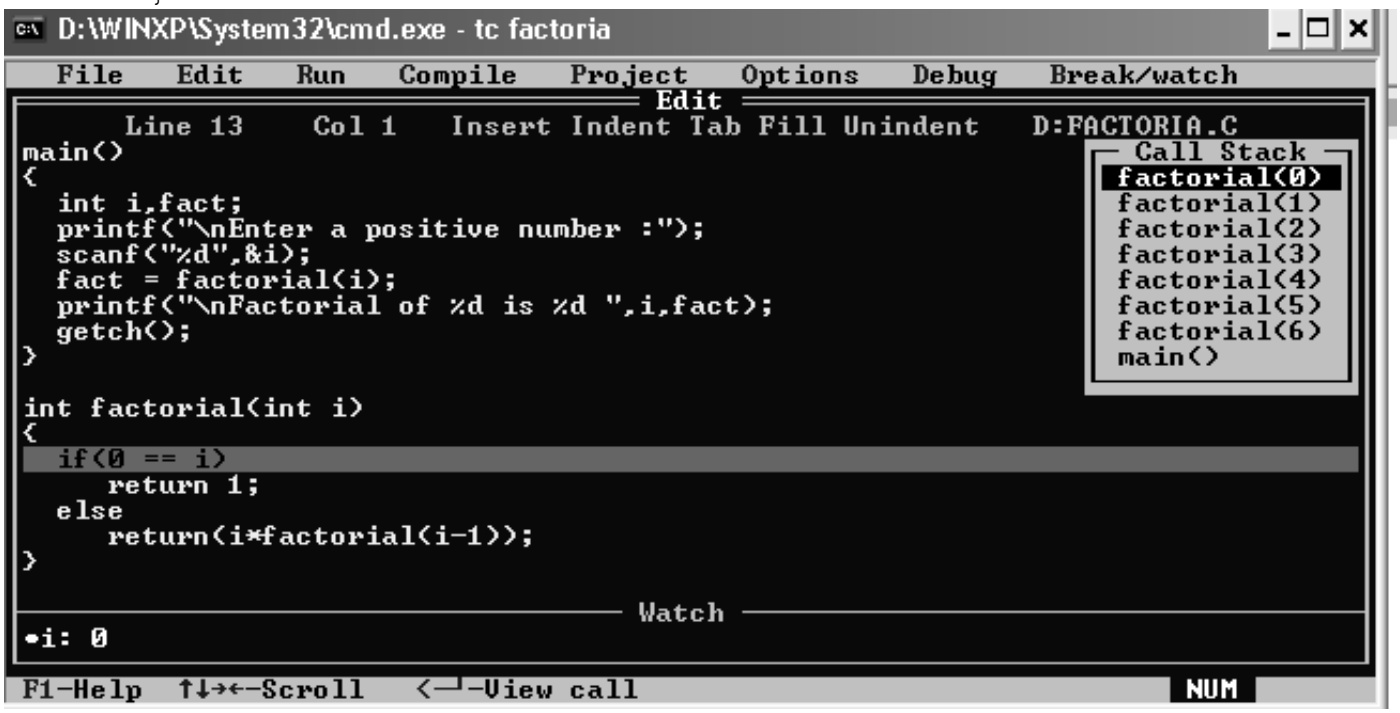
```
main()
{
  printf("\nInside Main()");
  recurse(1);
}

int recurse(int i)
{
  printf("\n%d",i);
  recurse (i+1);
}
```

The last value is the count of function call and it depends from the computer to the computer and the number of programs currently executing on that computer effects its value. The above two programs illustrate the infinite recursion. Recursion is used in a controlled manner. Normally, a recursive function will have a variable that performs a similar action; one that controls when the function will finally exit. This is often called the **'end condition'** of the function. Basically, it is an if-statement that checks some variable for a condition (such as a number being less than zero, or greater than some other number) and if that condition is true, it will not allow the function to call itself again. (Or, it could check if a certain condition is true, normally the contrary of its end condition, and only then allow the function to call itself). Following program illustrates use of recursion for computing factorial.

```
main()
{
  int i,fact;
  printf("\nEnter a positive number :");
  scanf("%d",&i);
  fact = factorial(i);
  printf("\nFactorial of %d is %d ",i,fact);
  getch();
}
int factorial(int i)
{
  if(0 == i)
       return 1;
```

```
        else
             return(i*factorial(i-1));
    }
```



Observe Call Stack towards the right of window. It is read from bottom to top as it's a STACK

Return Statement

Quite often while working with functions we encounter the return statement. The return statement terminates the execution of a function and returns control to the calling function. Execution resumes in the calling function at the point immediately following the call. A return statement can also return a value to the calling function.
return (*expression*) or an empty return return

The value of *expression*, if present, is returned to the calling function. If *expression* is omitted, the return value of the function is undefined. The expression, if present, is converted to the type returned by the function. If the function was declared with return type void, a return statement containing an expression generates a warning and the expression is not evaluated.

If no return statement appears in a function definition, control automatically returns to the calling function after the last statement of the called function is executed. In this case, the return value of the called function is undefined. If a return value is not required, declare the function to have void return type; otherwise, the default return type is int.

Many programmers use parentheses to enclose the *expression* argument of the return statement. However, C does not require the parentheses.