Lecture # 9

Recursion

- The programs we have discussed in previous programming courses are generally structured as functions that call one another in a disciplined, hierarchical manner.
- For some problems, it is useful to have functions call themselves.

- A *recursive function* is a function that calls itself either directly or indirectly through another function.

- A recursive function is called to solve the problem. The function actually knows how to solve the simplest case (s) or so called Base Case (s).
- If the function is called with a base case, the function simply returns the result.
- If a function is called with more complex problem, the function divides the problem into two conceptual pieces:
  - A piece the function knows how to do and
  - A piece the function doesn't knows how to do.

- To make recursion feasible the latter piece (i.e. piece the function doesn't knows how to do) must resemble the original problem, but in the form of slightly simpler or slightly smaller version of the original problem.

- Because this new problem looks like the original problem the function launches (calls) a fresh (new) copy of itself to go to work on smaller problems. This is referred to as a recursive call and is also called *recursion step*.

Example : Factorial

- Given a positive integer n, n factorial is defined as the product of all integers between n and 1. for example…

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

  0! = 1

  so

n! = 1 if (n== 0)

n! = n * (n-1) * (n-2) * (n-3) …… * 1 if n> 0

- So we can present an algorithm that accepts integer n and returns the value of n! as follows.

```
int prod = 1, x, n;
cin>>n;
for( x = n; x > 0; x-- )
        prod = prod * x;
cout<<n<<" factorial is"<<prod;
```

- Such an algorithm is called *iterative* algorithm because it calls itself for the explicit (precise) repetition of some process until a certain condition is met.

- In above program n! is calculated like as follows.

0! = 1

1! = 1

2! = 2 * 1

3! = 3 * 2 * 1

4! = 4 * 3 * 2 * 1

- Let us see how the recursive definition of the factorial can be used to evaluate 5!.

1. 5! = 5 * 4!
2. 4! = 4 * 3!
3. 3! = 3 * 2!
4. 2! = 2 * 1!
5. 1! = 1 * 0!
6. 0! = 1

- In above each case is reduced to a simpler case until we reach the case 0! Which is defined directly as 1. we may therefore backtrack from line # 6 to line # 1 by returning the value computed in one line to evaluate the result of previous line.

Lets incorporate this previous process of backward going into an algorithm as follows.....

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

int factorial( int numb )
{
        if( numb <= 0 )
            return 1;
        else
            return numb * factorial( numb - 1 );
}//----------------------------------------
void main()
{
        clrscr();
        int n;
        cout<<" \n Enter no for finding its Factorial.\n";
        cin>>n;
        cout<<"\n Factorial of "<<n<<" is : "<<factorial( n );
        getch();

}// end of main().
```

- Above code reflects recursive algorithm of n!. In this function is first called by main( ) and then it calls itself until it calculates the n! and finally return it to main( ).

Efficiency of Recursion

- In general, a non recursive version of a program will execute more efficiently in terms of time and space than a recursive version. This is because the overhead involved in entering and exiting a new block (system stack) is avoided in the non recursive version.

- However, sometimes a recursive solution is the most natural and logical way of solving a problem as we will soon observe while studying different Tree *data structures*.