LECTURE NOTE 16

RECURSION

Recursion is a process in which a problem is defined in terms of itself. In 'C' it is possible to call a function from itself. Functions that call themselves are known as **recursive** functions, i.e. a statement within the body of a function calls the same function. Recursion is often termed as 'Circular Definition'. Thus recursion is the process of defining something in terms of itself. To implement recursion technique in programming, a function should be capable of calling itself.

Example:

```
void main()
{
……….…….          /* Some statements*/
fun1();

……….…….          /* Some statements */
} void
fun1()

{
……….…….          /* Some statements */ fun1();
/*RECURSIVE CALL*/

……….…….          /* Some statements */
}
```

Here the function **fun1()** is calling itself inside its own function body, so fun1() is a recursive function. When main() calls fun1(), the code of fun1() will be executed and since there is a call to fun1() insidefun1(), again fun1() will be executed. It looks like the above program will run up to infinite times but generally a terminating condition is written inside the recursive functions which end this recursion. The following program (which is used to print all the numbers starting from the given number to 1 with successive decrement by 1) illustrates this:

```
void main()
{
 int a;
 printf("Enter a number");
 scanf("%d",&a);
 fun2(a);
}
int fun2(int b)
{
 printf("%d",b);
 b--;
 if(b>=1)    /* Termination condition i.e. b is less than 1*/
{
 fun2(b);
}
}
```

How to write a Recursive Function?

Before writing a recursive function for a problem its necessary to define the solution of the problem in terms of a similar type of a smaller problem.

Two main steps in writing recursive function are as follows:

(i). Identify the Non-Recursive part(base case) of the problem and its solution(Part of the problem whose solution can be achieved without recursion).

(ii). Identify the Recursive part(general case) of the problem(Part of the problem where recursive call will be made).

Identification of Non-Recursive part of the problem is mandatory because without it the function will keep on calling itself resulting in infinite recursion.

How control flows in successive recursive calls?

Flow of control in successive recursive calls can be demonstrated in following example:

Consider the following program which uses recursive function to compute the factorial of a number.

```
void main()
{
int n,f;
printf("Enter a number");
scanf("%d",&n);
f=fact(a);
printf("Factorial of %d is %d",n,f);
}
int fact(int m)
{
int a;
if (m==1)
  return (1);
else
  a=m*fact(m-1);
return (a);
}
```

In the above program if the value entered by the user is 1 i.e.**n**=1, then the value of **n** is copied into **m**. Since the value of **m** is 1 the condition 'if(m==1)' is satisfied and hence 1 is returned through return statement i.e. factorial of 1 is 1.
When the number entered is 2 i.e. n=2, the value of n is copied into m. Then in function fact(), the condition 'if(m==1)' fails so we encounter the statement a=m*fact(m-1); and here we meet

recursion. Since the value of **m** is 2 the expression (m*fact(m-1)) is evaluated to (2*fact(1)) and the result is stored in **a**(factorial of a). Since value returned by fact(1) is 1 so the above expression reduced to (2*1) or simply 2. Thus the expression m*fact(m-1) is evaluated to 2 and stored in **a** and returned to main(). Where it will print 'Factorial of 2 is 2'.

In the above program if **n**=4 then main() will call fact(4) and fact(4) will send back the computed value i.e. **f** to main(). But before sending back to main() fact(4) will call fact(4-1) i.e. fact(3) and wait for a value to be returned by fact(3). Similarly fact(3) will call fact(2) and so on. This series of calls continues until **m** becomes 1 and fact(1) is called, which returns a value which is so called as termination condition. So we can now say what happened for **n**=4 is as follows

fact(4) returns (4*fact(3))

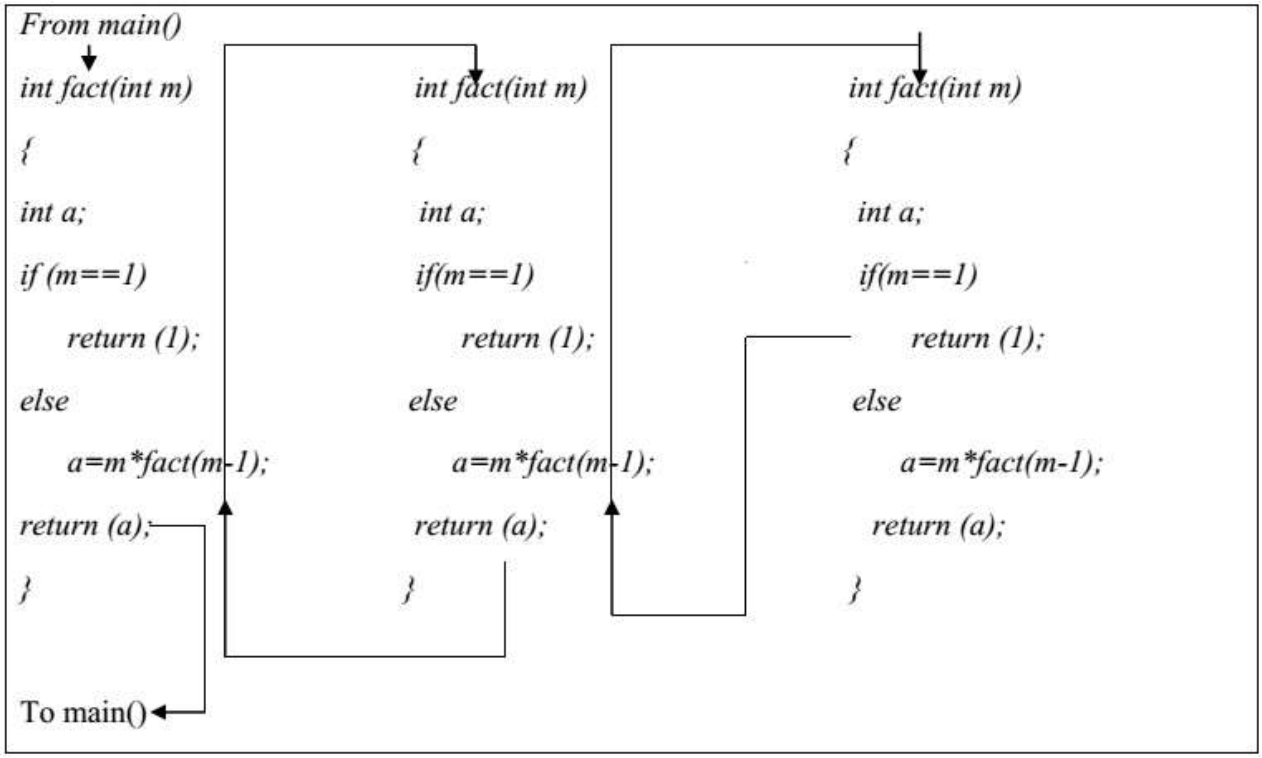
fact(3) returns (3*fact(2))

fact(2) returns (2*fact(1))

fact(1) returns (1)

So for **n**=4, the factorial of 4 is evaluated to 4*3*2*1=24.

For **n**=3, the control flow of the program is as follows:



Winding and Unwinding phase

All recursive functions work in two phases- winding phase and unwinding phase.

Winding phase starts when the recursive function is called for the first time, and ends when the termination condition becomes true in a call i.e. no more recursive call is required. In this phase a function calls itself and no return statements are executed.

After winding phase unwinding phase starts and all the recursive function calls start returning in reverse order till the first instance of function returns. In this phase the control returns through each instance of the function.

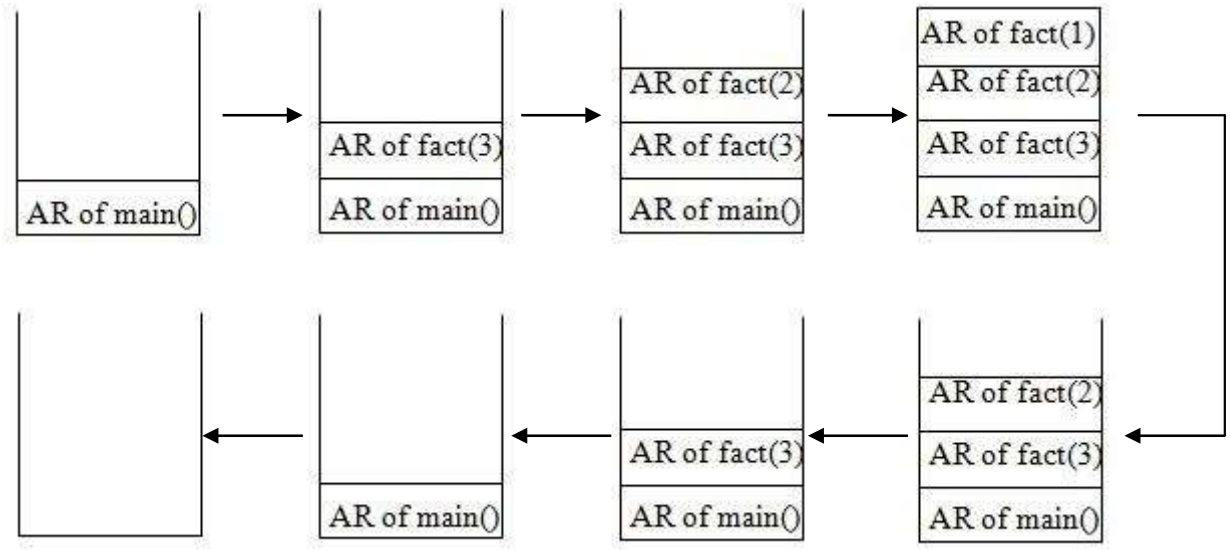
Implementation of Recursion

We came to know that recursive calls execute like normal function calls, so there is no extra technique to implement recursion. All function calls(Whether Recursive or Non-Recursive) are implemented through run time stack. Stack is a Last In First Out(LIFO) data structure. This means that the last item to be stored on the stack(PUSH Operation) is the first one which will be deleted(POP Operation) from the stack. Stack stores Activation Record(AR) of function during run time. Here we will take the example of function fact() in the previous recursive program to find factorial of a number.

Suppose fact() is called from main() with argument 3 i.e.

*fact(3); /*From main()*/*

Now will see how the run time stack changes during the evaluation of factorial of 3.



The following steps will reveal how the above stack contents were expressed:

First main() is called, so PUSH AR of main() into the stack. Then main() calls fact(3) so PUSH AR of fact(3). Now fact(3) calls fact(2) so PUSH AR of fact(2) into the stack. Likewise PUSH AR of fact(1). After the above when fact(1) is completed, POP AR of fact(1), Similarly after completion of a specific function POP its corresponding AR. So when main() is completed POP AR of main(). Now stack is empty.

In the winding phase the stack content increases as new Activation Records(AR) are created and pushed into the stack for each invocation of the function. In the unwinding phase the Activation Records are popped from the stack in LIFO order till the original call returns.

Examples of Recursion

Q1. Write a program using recursion to find the summation of numbers from 1 to n.

Ans: We can say 'sum of numbers from 1 to n can be represented as sum of numbers from 1 to n-1 plus n' i.e.

Sum of numbers from 1 to n = n + Sum of numbers from 1 to n-1

= n + n-1 + Sum of numbers from 1 to n-2

= n+ n-1 + n-2 + ... +1

The program which implements the above logic is as follows:

```
#include<stdio.h>

void main()

{

int n,s;

 printf("Enter a number");

 scanf("%d",&n);

 s=sum(n);

 printf("Sum of numbers from 1 to %d is %d",n,s);

}
```

```
 int sum(int m) int r;

if(m==1)

        return (1);

else

        r=m+sum(m-1);/*Recursive Call*/

        return r;

}
```

Output:

Enter a number 5
15

Q2. Write a program using recursion to find power of a number i.e. n^m.

Ans: We can write,

$n_m = n*n_{m-1}$

$=n*n*n^{m-2}$

$=n*n*n*$……m times $*n^{m-m}$

The program which implements the above logic is as follows:

```
#include<stdio.h>

int power(int,int);

 void main()

{

        int n,m,k;

        printf("Enter the value of n and m");

        scanf("%d%d",&n,&m);
```

```
        k=power(n,m);
        printf("The value of n^m for n=%d and m=%d is %d",n,m,k);
}
int power(int x, int y)
{
        if(y==0)
        {
                return 1;
        }
        else
        {
                return(x*power(x,y-1));
        }
}
```

Output:

Enter the value of n and m

3

5

The value of n^m for n=3 and m=5 is 243

Q3.Write a program to find GCD (Greatest Common Divisor) of two numbers.

Ans: The GCD or HCF (Highest Common Factor) of two integers is the greatest integer that divides both the integers with remainder equals to zero. This can be illustrated by Euclid's remainder Algorithm which states that GCD of two numbers say x and y i.e.

GCD(x, y) = **x** if y is 0

= **GCD(y, x%y)** otherwise

The program which implements the previous logic is as follows:

```
#include<stdio.h>
int GCD(int,int);
void main()
{
 int a,b,gcd;
printf("Enter two numbers");
scanf("%d%d",&a,&b);
gcd=GCD(a,b);
printf("GCD of %d and %d is %d",a,b,gcd);
}
int GCD(int x, int y)
{
if(y==0)
return x;
 else
 return GCD(y,x%y);
}
```

Output:

Enter two numbers 21

35

GCD of 21 and 35 is 7

Q4:Write a program to print Fibonacci Series upto a given number of terms.

Ans: Fibonacci series is a sequence of integers in which the first two integers are 1 and from third integer onwards each integer is the sum of previous two integers of the sequence i.e.

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …….....

The program which implements the above logic is as follows:

```
#include<stdio.h>
int Fibonacci(int);
void main()
{
 int term,i;
printf("Enter the number of terms of Fibonacci Series which is going to be printed");
scanf("%d",&term);
 for(i=0;i<term;i++)
{
printf("%d",Fibonacci(i));
}
}
int Fibonacci(int x)
{
if(x==0 || x==1)
return 1;
else
return (Fibonacci(x-1) + Fibonacci(x-2));
}
```

Output:

Enter the number of terms of Fibonacci Series which is going to be printed 6

1 1 2 3 5 8 13