UNIT–IV:

Functions: Introduction Designing structured programs, Declaring a function, Signature of a function, Parameters and return type of a function, passing parameters to functions, call by value,Passing arrays to functions, passing pointers to functions, idea of call by reference, Some C standard functions and libraries and Functions,
Recursion: Simple programs, such as Finding Factorial, Fibonacci series etc., Limitations of Recursive functions
Dynamic memory allocation: malloc, calloc, realloc, free, Allocating and freeing memory, Allocating memory for arrays of different data types

1. What is a function? Why we use functions in C language? Give an example.

Ans: Function in C:

A function is a block of code that performs a specific task. It has a name and it is reusable .It can be executed from as many different parts in a program as required, it can also return a value to calling program.

All executable code resides within a **function**. It takes input, does something with it, then give the answer. A C program consists of one or more functions.

A computer program cannot handle all the tasks by itself. It requests other program like entities called functions in C. We pass information to the function called **arguments** which specified when the function is called. A function either can return a value or returns nothing. Function is a subprogram that helps reduce coding.

Simple Example of Function in C

```
#include<stdio.h>
#include <conio.h>
int adition (int, int); //Function Declaration int
addition (int a, int b) //Function Definition
{
int r;
r=a + b;
return (r);
}
int main()
```

```
{
int z;
z= addion(10,3); //Function Call
printf ("The Result is %d", z);
return 0;
}
```

Output: The Result is 13

Why use function:

Basically there are **two reasons** because of which we use functions

1. Writing functions avoids rewriting the same code over and over. For example - if you have a section of code in a program which calculates the area of triangle. Again you want to calculate the area of different triangle then you would not want to write the same code again and again for triangle then you would prefer to jump a "section of code" which calculate the area of the triangle and then jump back to the place where you left off. That section of code is called „function'.

2. Using function it becomes easier to write a program and keep track of what they are doing. If the operation of a program can be divided into separate activities, and each activity placed in a different function, then each could be written and checked more or less independently. Separating the code into modular functions also makes the program easier to design and understand.

2. Distinguish between Library functions and User defined functions in C and Explain with examples.

Ans: Types of Function in C:

(i). Library Functions in C

C provides library functions for performing some operations. These functions are present in the c library and they are predefined.

For example sqrt() is a mathematical library function which is used for finding the square root of any number .The function scanf and printf() are input and output library function similarly we have strcmp() and strlen() for string manipulations. To use a library function we have to include some header file using the preprocessor directive #include.

For example to use input and output function like printf() and scanf() we have to include stdio.h, for math library function we have to include math.h for string library string.h should be included.

(ii). User Defined Functions in C

A user can create their own functions for performing any specific task of program are called user defined functions. To create and use these function we have to know these 3 elements.

1. Function Declaration
2. Function Definition
3. Function Call

1. Function declaration

The program or a function that calls a function is referred to as the calling program or calling function. The calling program should declare any function that is to be used later in the program this is known as the function declaration or function prototype.

2. Function Definition

The function definition consists of the whole description and code of a function. It tells that what the function is doing and what are the input outputs for that. A function is called by simply writing the name of the function followed by the argument list inside the parenthesis. Function definitions have two parts:

Function Header

The first line of code is called Function Header.

int sum(int x, int y)

It has three parts

(i). The name of the function i.e. sum

(ii). The parameters of the function enclosed in parenthesis

(iii). Return value type i.e. int

Function Body

Whatever is written with in { } is the body of the function.

3. Function Call

In order to use the function we need to invoke it at a required place in the program. This is known as the function call.

3. Write some properties and advantages of user defined functions in C?

Ans:

Properties of Functions

- Every function has a unique name. This name is used to call function from “main()” function.
- A function performs a specific task.
- A function returns a value to the calling program.

Advantages of Functions in C

- Functions has top down programming model. In this style of programming, the high level logic of the overall problem is solved first while the details of each lower level functions is solved later.

- A C programmer can use function written by others
- Debugging is easier in function
- It is easier to understand the logic involved in the program
- Testing is easier

4. Explain the various categories of user defined functions in C with examples?

Ans:

A function depending on whether arguments are present or not and whether a value is returned or not may belong to any one of the following categories:

(i) Functions with no arguments and no return values.

(ii) Functions with arguments and no return values.

(iii) Functions with arguments and return values.

(iv) Functions with no arguments and return values.

(i) Functions with no arguments and no return values:-

When a function has no arguments, it does not return any data from calling function. When a function does not return a value, the calling function does not receive any data from the called function. That is there is no data transfer between the calling function and the called function.

Example

```
#include   <stdio.h>
#include  <conio.h>
void printmsg()
{
printf ("Hello ! I Am A Function .");
}
int main()
{
printmsg();
return 0;
}
```

Output : Hello ! I Am A Function .

(ii) Functions with arguments and no return values:-

When a function has arguments data is transferred from calling function to called function. The called function receives data from calling function and does not send back any values to calling function. Because it doesn‟t have return value.

Example

```
#include<stdio.h>
#include <conio.h>
void add(int,int);

void main()
{
int a, b;

printf(“enter value”);

scanf(“%d%d”,&a,&b);

add(a,b);
}

void add (intx, inty)
{
int z ;
z=x+y;

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

output : enter values 2 3

```
        The sum = 5
```

(iii) Functions with arguments and return values:-

In this data is transferred between calling and called function. That means called function receives data from calling function and called function also sends the return value to the calling function.

Example

```
#include<stdio.h>
#include <conio.h>
int add(int, int);
main()
{
 int a,b,c;

printf("enter value");

scanf("%d%d",&a,&b);

c=add(a,b);
printf ("The sum =%d",c);
 }
int add (int x, int y)
{
int z;
z=x+y;

return z;
}
```

output : enter values 2 3

```
          The sum = 5
```

(iv) Function with no arguments and return type:-

When function has no arguments data cannot be transferred to called function. But the called function can send some return value to the calling function.

Example

```
#include<stdio.h>
#include <conio.h>
int add( );
main()
{
 int c;

c=add();
printf ("The sum =%d",c);
 }
int add ()
{
int x,y,z;

printf("enter value");

scanf("%d%d",&a,&b);

z=x+y;

return z;
}
```

Output: enter values 2 3

```
        The sum = 5
```

5. Explain the Parameter Passing Mechanisms in C-Language with examples.

Ans:

Most programming languages have 2 strategies to pass parameters. They are

(i) pass by value(Call by Value)

(ii) pass by reference (Call by Reference)

(i) Pass by value (or) call by value :-

In this method calling function sends a copy of actual values to called function, but the changes in called function does not reflect the original values of calling function.

Example program:

```
#include<stdio.h>

void fun1(int, int);

void main( )

{

int a=10, b=15;

fun1(a,b);

printf("a=%d,b=%d", a,b);

}

void fun1(int x, int y)

{

x=x+10;

y= y+20;

}
```

Output: a=10 b=15

The result clearly shown that the called function does not reflect the original values in main function.

(ii) Pass by reference (or) call by address :-

In this method calling function sends address of actual values as a parameter to called function, called function performs its task and sends the result back to calling function. Thus, the changes in called function reflect the original values of calling function. To return multiple values from called to calling function we use pointer variables.

Calling function needs to pass „&" operator along with actual arguments and called function need to use „*" operator along with formal arguments. Changing data through an address variable is known as indirect access and „*" is represented as indirection operator.

Example program:

```
#include<stdio.h>

void fun1(int,int);

void main( )

{

int a=10, b=15;

fun1(&a,&b);

printf("a=%d,b=%d", a,b);

}

void fun1(int *x, int *y)

{

*x = *x + 10;
*y = *y + 20;

}
```

Output: a=20 b=35

The result clearly shown that the called function reflect the original values in main function. So that it changes original values.

6. Differentiate actual parameters and formal parameters.

Ans:

Actual parameters	**Formal parameters**
The list of variables in calling function is known as **actual parameters**.	The list of variables in called function is known as **formal parameters**.
Actual parameters are variables that are declared in function call.	Formal parameters are variables that are declared in the header of the function definition.
Actual parameters are passed without using type	Formal parameters have type preceeding with them.

main( )
{ .….
function_name (actual parameters);
…..
}
	return_type function_name(formal parameters)
	{ .….
	function body;
	…..
	}

Formal and actual parameters must match exactly in type, order, and number.

Formal and actual parameters need not match for their names.

7. Explain in detail about nesting of functions with example.

Ans:

Nesting of functions

The process of calling a function within another function is called nesting of function

Syntax:-

```
main()
{
………..
Function1();
……….
}
Function1();
{
 …………
Function2();
…………
}
Function2();
{
…………
Function3();
…………
}
Function3();
{
………….
}
```

main () can call Function 1() where Function1 calls Function2() which calls Function3() and so on

Ex:

```
float ratio (int, int,  int);

 int difference (int, int);

void main()

{

int a,b,c,d;

printf(" enter three numbers");

scanf("%d%d%d", &a, &b, &c );

d= ratio(a,b,c);

printf ("%f\n" ,d);

}

float ratio(int x,int y,int z)

{

int u ;

u=difference(y,z);

if (u) return(x/(y-

z)); else

return (0.0);

}

int difference (int p, int q)

{

if(p!=q)
return 1;
else

return 0;

}
```

main reads values a,b,c and calls ratio() to calculate a/(b-c) ratio calls another function difference to test whether (b-c) is zero or not this is called nesting of function.

8. How to Pass Array Individual Elements to Functions? Explain with example program.

Ans:

Arrays with functions:

To process arrays in a large program, we need to pass them to functions. We can pass arrays in two ways:

1) pass individual elements

2) pass the whole array.

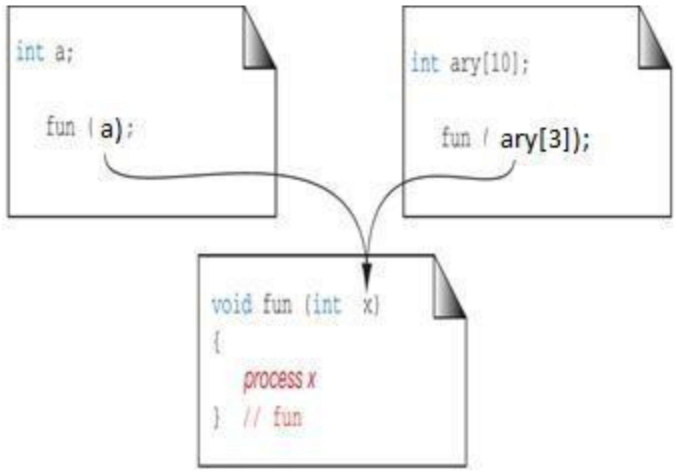
Passing Individual Elements:

One-dimensional Arrays:

We can pass individual elements by either passing their data values or by passing their addresses. We pass data values i.e; individual array elements just like we pass any data value .As long as the array element type matches the function parameter type, it can be passed. The called function cannot tell whether the value it receives comes from an array, a variable or an expression.

Program using call by value

```
void func1( int a);

void main()

{

int a[5]={ 1,2,3,4,5};

func1(a[3]);

}

void func1( int x)

{

printf("%d",x+100);

}
```



Two-dimensional Arrays:

The individual elements of a 2-D array can be passed in the same way as the 1-D array. We can pass 2-D array elements either by value or by address.

Ex: Program using call by value

```
void fun1(int a)

main()

{

int a[2][2]={1,2,3,4};

fun1(a[0][1]);

}

void fun1(int x)

{

printf("%d",x+10);

}
```

9. How can we pass the Whole Array to Functions? Explain with example program.

Ans: Passing an entire array to a function:

One-dimensional array:

To pass the whole array we simply use the array name as the actual parameter. In the called function, we declare that the corresponding formal parameter is an array. We do not need to specify the number of elements.

Program :

```
void fun1(int a[])

void main()

{

fun1(a);

}

void fun1( int x[])

{

int i, sum=0;

for(i=0;i<5;i++)

sum=sum+a[i];

printf("%d",sum);

}
```

Two-dimensional array:

When we pass a 2-D array to a function, we use the array name as the actual parameter just as we did with 1-D arrays. The formal parameter in the called function header, however must indicate that the array has two dimensions.

Rules:

- The function must be called by passing only the array name.
- In the function definition, the formal parameter is a 2-D array with the size of the second dimension specified.

Program:

```
void fun1(int a[][2])
void main()

{

int a[2][2]={1,2,3,4};

fun1(a);

}

void fun1(int x[][2])

{

int i,j;

for(i=0;i<2;i++)

for(j=0;j<2;j++)

printf("%d",x[i][j]);

}
```

10. Explain how pointers are used as function arguments.

Ans: When we pass addresses to a function, the parameter receiving the addresses should be pointers. The process of calling a function using pointers to pass the address of variables is known as "call by reference". The function which is called by reference can change the value of the variable used in the call.

Example :-

```
void main()

{

int x;

x=50;

change(&x);      /* call by reference or address */

printf("%d\n",x);

}

change(int *p)

{
  *p=*p+10;

}
```

When the function change () is called, the address of the variable x, not its value, is passed into the function change (). Inside change (), the variable **p** is declared as a pointer and therefore **p** is the address of the variable x. The statement,

```
*p = *p + 10;
```

Means ―add 10 to the value stored at the address **p‖.** Since **p** represents the address of **x**, the value of x is changed from 20 to 30. Thus the output of the program will be 30, not 20.

Thus, call by reference provides a mechanism by which the function can change the stored values in the calling function.

11. Write a ‗C' function using pointers to exchange the values stored in two locations in the memory. (Or)

Write a C program for exchanging of two numbers using call by reference mechanism.

Ans:

Using pointers to exchange the values

- Pointers can be used to pass addresses of variables to called functions, thus allowing the called function to alter the values stored there.
- Passing only the copy of values to the called function is known as **"call by value".**
- Instead of passing the values of the variables to the called function, we pass their addresses, so that the called function can change the values stored in the calling routine. This is known as **"call by reference",** since we are referencing the variables.
- Here the addresses of actual arguments in the calling function are copied into formal arguments of the called function. Here the formal parameters should be declared as pointer variables to store the address.

program

```
#include<stdio.h>

void swap (int, int);

void main( )

{

int a=10, b=15;
swap(&a, &b);

printf("a=%d,b=%d", a,b);

}

void swap(int *x, int *y)

{

int temp;

temp = *x;

*x = *y;

*y = temp;

}
```

Output: **a = 15 b=10**

12.What is recursive function? Write syntax for recursive functions.

Ans:

Recursion

Recursion is a process by which a function calls itself repeatedly, until some specified condition has been satisfied.

When a function calls itself, a new set of local variables and parameters are allocated storage on the stack, and the function code is executed from the top with these new variables. A recursive call does not make a new copy of the function. Only the values being operated upon are new. As each recursive call returns, the old local variables and parameters are removed from the stack, and execution resumes immediately after the recursive call inside the function.

The main advantage of recursive functions is that we can use them to create clearer and simpler versions of several programs.

Syntax:-

A function is **recursive** if it can call itself; either directly:

```
void f( )
{
f( );
}
```

(or) indirectly:

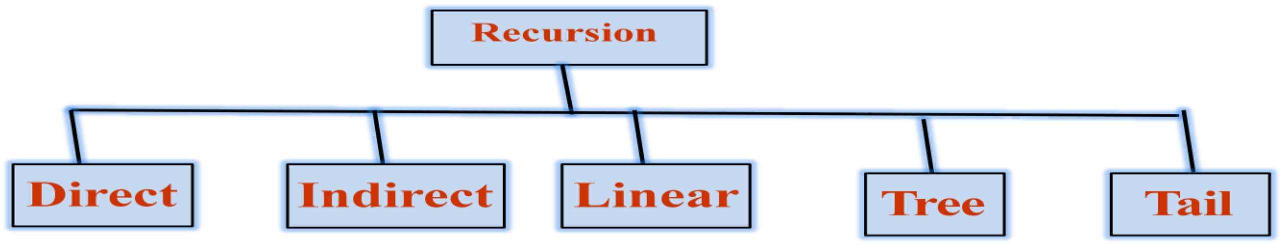
```
void f( )
{
g( );
}

void g( )
{
f( );
}
```

Recursion rule 1: Every recursive method must have a **base case** -- a condition under which no recursive call is made -- to prevent infinite recursion.

Recursion rule 2: Every recursive method must make progress toward the base case to prevent infinite recursion

13.Explain about different types of recursive functions with examples

Ans: ***Types Of Recursion***



Any recursive function can be characterized based on: whether the function calls itself directly or (direct or indirect). Whether any operation is pending at each recursive call (tail-recursive or not). the structure of the calling pattern (linear or tree-recursive).

Direct Recursion

A function is said to be *directly* recursive if it explicitly calls itself.

<u>Example</u>

```
int Func( int n)
{
if(n==0)  retrun n;
return (Func(n-1));
}
```

Indirect Recursion

A function is said to be *indirectly* recursive if it contains a call to another function which ultimately calls it.

<u>Example</u>

```
int Func1(int n)
{
if(n==0)  return n;
return Func2(n);
}
int Func2(int x)
{
return Func1(x-1);
}
```

These two functions are indirectly recursive as they both call each other.

Tail Recursion

A recursive function is said to be *tail recursive* if no operations are pending to be performed when the recursive function returns to its caller. That is, when the called function returns, the returned value is immediately returned from the calling function. Tail recursive functions are highly desirable because they are much more efficient to use as in their case, the amount of information that has to be stored on the system stack is independent of the number of recursive calls.

Example

```
int Fact(n)
{
return Fact1(n, 1);
}
int Fact1(int n, int res)
{
if (n==1)  return res;
return Fact1(n-1, n*res);
}
```

Linear Recursion

A recursive function is said to be *linearly* recursive when no pending operation involves another recursive call to the function. For example, the factorial function is linearly recursive as the pending operation involves only multiplication to be performed and does not involve another call to Fact.

Example:

```
int Fact(int)
{
if(n==1)  retrun 1;
return (n * Fact(n-1));
}
```

Tree Recursion

A recursive function is said to be *tree recursive* (or *non- linearly* recursive) if the pending operation makes another recursive call to the function. For example, the Fibonacci function Fib in which the pending operations recursively calls the Fib function.

Example:

```
int Fibonacci(int num)
{
if(num <= 2)
return 1;
return ( Fibonacci (num - 1) +  Fibonacci(num – 2));
}
```

14.What are the limitations of Recursion?

Ans:

1. Recursive solutions may involve extensive overhead because they use function calls.
2. Each function call requires push of return memory address,parameters, returned results, etc. and every function return requires that many pops.
3. Each time we make a call we use up some of our memory allocation. If the recursion is deep that is, if there are many recursive calls then we may run out of memory.
4. Recursion is implemented using system stack. If the stack space on the system is limited, recursion to a deeper level will be difficult to implement.
5. Aborting a recursive process in midstream is slow and sometimes nasty.
6. Using a recursive function takes more memory and time to execute as compared to its non-recursive counterpart.
7. It is difficult to find bugs, particularly when using global variables.
8. Recursion uses more processor time.

15.What are the advantages of Recursion?

Ans:

1. Recursive solutions often tend to be shorter and simpler than non- recursive ones.
2. Code is clearer and easier to use
3. Recursion represents like the original formula to solve a problem.
4. Follows a divide and conquer technique to solve problems
5. In some (limited) instances, recursion may be more efficient

16.Write a program to find factorial of a number using recursion.

Ans:

```
#include<stdio.h>
int fact(int);
main()
{
int n,f;
printf("\n Enter any
number:"); scanf("%d",&n);
f=fact(n);
printf("\n Factorial of %d is %d",n,f);
}
int fact(int n)
{
int f;
if(n==0||n==1) //base case
f=1;
else
f=n*fact(n-1); //recursive case
return f;
}
```

Output:- Enter any number: 5

```
        Factorial of 5 is 120
```

17. Differentiate between recursion and non- recursion.

Ans:

Recursion:- Recursion is a process by which a function calls itself repeatedly, until some specified condition has been satisfied.

When a function calls itself, a new set of local variables and parameters are allocated storage on the stack, and the function code is executed from the top with these new variables. A recursive call does not make a new copy of the function. Only the values being operated upon are new. As each recursive call returns, the old local variables and parameters are removed from the stack, and execution resumes immediately after the recursive call inside the function.

Ex:-

```
void main( )
{
int n=5;
fact( n);
}
int fact( )
{
if(n==0 || n==1)
  return 1;
else
return(n*fact(n-1));
}
```

Non-Recursion:-

Using looping statements we can handle repeated statements in „C‟. The example of non recursion is given below.

Syntax:-

```
void main( )
{
int n=5;
res = fact(n);
printf("%d",res);
}
int fact( )
{
for(i=1;i<=n;i++)
{
f=f+1;
}
return f;
}
```

Differences:

- Recursive version of a program is slower than iterative version of a program due to overhead of maintaining stack.
- Recursive version of a program uses more memory (for the stack) than iterative version of a program.
- Sometimes, recursive version of a program is simpler to understand than iterative version of a program.

18. Write short notes on Towers of Hanoi problem

Ans:

1. It consists of three poles, and a number of disks of different sizes which can slide onto any pole.
2. The puzzle starts with the disks in a neat stack in ascending order of size on one pole, the smallest at the top, thus making a conical shape.
3. The objective of the puzzle is to move the entire stack to another pole, obeying the following rules:
   - Only one disk must be moved at a time.
   - Each move consists of taking the upper disk from one of the poles and sliding it onto another pole, on top of the other disks that may already be present on that pole.
   - No disk may be placed on top of a smaller disk.

Algorithm:

Input:- Input of discs in Tower of Hanoi , specification of SOURCE as from the piller and DEST as to piller,AUX as the intemediate piller.

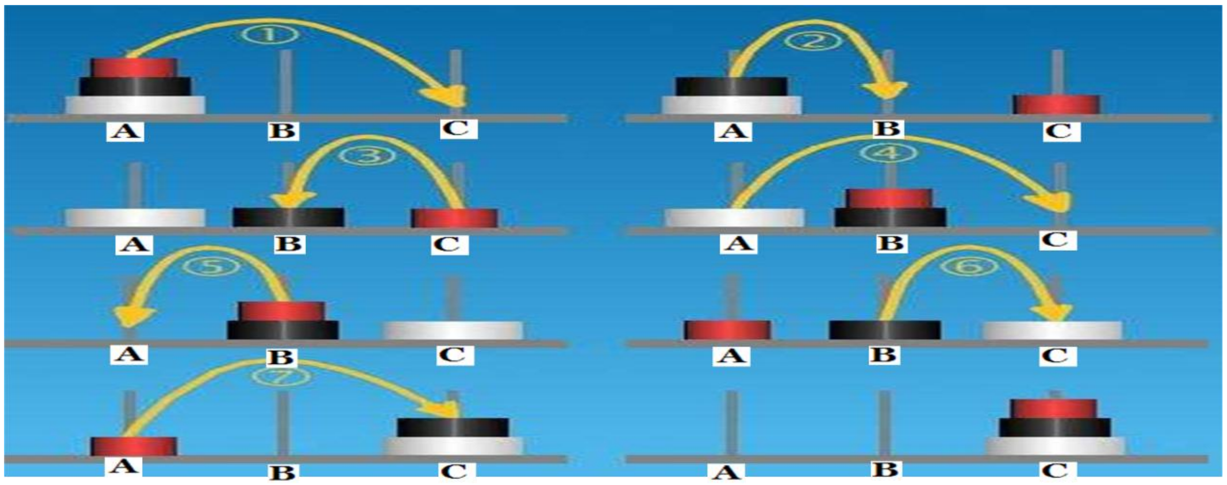
Output :- Steps of moves of N discs from piller SOURCE to DEST piller.

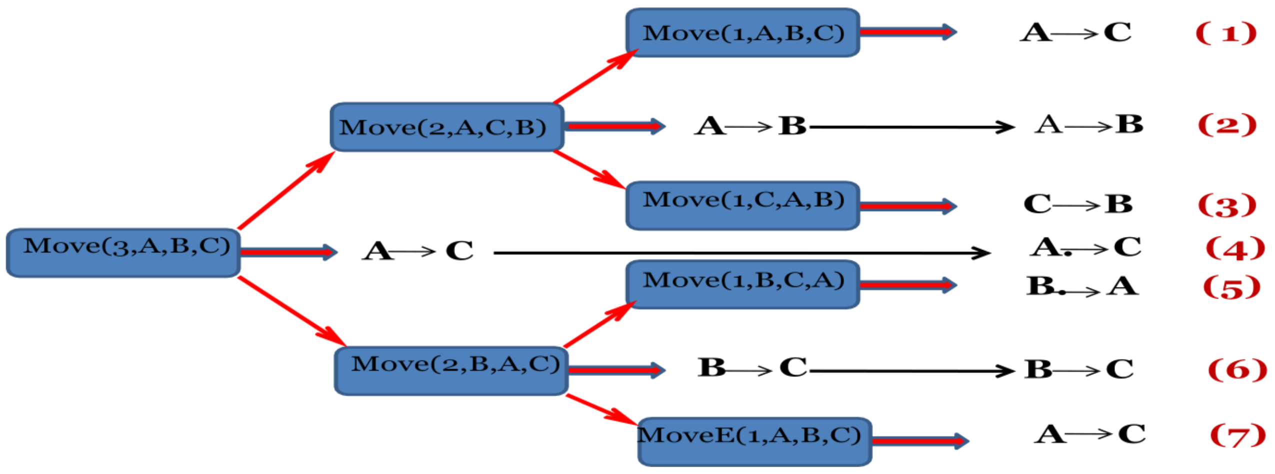
Steps

```
if N>0 then
        move(N-1 ,SOURCE ,AUX ,DEST)
           SOURCE ->DEST  (move from SOURCE to DEST)
        move(N-1 ,AUX ,DEST ,SOURCE)
end if
stop
```

Example:

For N=3 , how will this recursion solve the problem as shown





19. Write a C program to implement Towers of Hanoi problem using recursion

Ans:

```
#include<stdio.h>
void towers(int n,char source,char dest,char aux);
void main()
{
   int n;
   printf("enter no.of Discs");
   scanf("%d",&n);
   towers(n,'A','C','B');
}
void towers(int n,char source,char dest,char aux)
{
   if(n==1)
   {
    printf("\nMove Disk from %c -> %c\n",source,dest);
   }else
   {
   towers(n-1,source,aux,dest);
   printf("\nMove Disk from %c -> %c\n",source,dest);
   towers(n-1,aux,dest,source);
    }}
```

20. Write a program to calculate GCD of two numbers using recursion

Ans:

```
#include<stdio.h>

int gcd(int,int);

main()

{

int a,b;

printf("\n Enter any two

numbers:"); scanf("%d%d",&a,&b);

printf("\nGCD=%d",gcd(a,b));

}

int gcd(int a,int b)

{

if(b==0) //base case

return a;

else

return gcd(b,a%b); //recursive case

}
```

21. Write a program to generate Fibonacci series using recursive functions.

Ans:

```
#include<stdio.h>

#include<conio.h>

void main()

{

 int f,f1,f2,n,i,res;

printf("enter the limit:");

scanf("%d",&n);

printf("The fibonacci series is:");

for(i=0;i<n;i++)

{

 printf("%d\t", fib(i));

}

}

int fib(int i)

{

if(i==0)

return 0;

else if(i==1)

return 1;

else

return(fib(i-1)+fib(i-2));

}
```

22. Differentiate between static and dynamic memory allocation.

Explain about static and dynamic memory allocation techniques.

Ans: Memory can be reserved for the variables either during the compilation time or during execution time. Memory can be allocated for variables using two different techniques:

1. Static allocation

2. Dynamic allocation

1) **Static allocation**: If the memory is allocated during compilation time itself, the allocated memory space cannot be expanded to accommodate more data or cannot be reduced to accommodate less data.

In this technique once the size of the memory is allocated it is fixed. It cannot be altered even during execution time .This method of allocating memory during compilation time is called static memory allocation.

2) **Dynamic allocation**: Dynamic memory allocation is the process of allocating memory during execution time. This allocation technique uses predefined functions to allocate and release memory for data during execution time.

Static memory allocation	Dynamic memory allocation
It is the process of allocating memory at compile time.	It is the process of allocating memory during execution of program.
Fixed number of bytes will be allocated.	Memory is allocated as and when it is needed.
The memory is allocated in memory stack.	The memory is allocated from free memory pool (heap).
Execution is faster	Execution slow
Memory is allocated either in stack area or data area	Memory is allocated only in heap area
Ex: Arrays	Ex: Dynamic arrays, Linked List, Trees

23. Explain about malloc( ) and calloc( ) dynamic memory management functions with an example.

Ans: The function malloc( ) allocates a block of **size** bytes from the free memory pool (heap).

It allows a program to allocate an exact amount of memory explicitly, as and when needed.

```
Ptr=(cast_type *)malloc (byte_ size);
```

Ptr is a pointer of type **cast_type**.The malloc returns a pointer to an area of memory with size **byte_size**.The parameter passed to malloc() is of the type **byte_size**. This type is declared in the header file **alloc.h**. **byte_size** is equivalent to the unsigned int data type. Thus, in compilers where an int is 16 bits in size, malloc() can allocate a maximum of 64KB at a time, since the maximum value of an unsigned int is 65535.

Return value:

- ✓ On success, i.e., if free memory is available, malloc() returns a pointer to the newly allocated memory. Usually, it is generic pointer. Hence, it should be typecast to appropriate data type before using it to access the memory allocate.
- ✓ On failure, i.e., if enough free memory does not exist for block, malloc() returns NULL. The constant NULL is defined in stdio.h to have a value zero. Hence, it is safe to check the return value.

Ex: 1) malloc(30); allocates 30 bytes of memory and returns the address of byte0.

2) malloc(sizeof(float)); allocates 4 bytes of memory and returns the address of byte0.

2) calloc( ) — allocates multiple blocks of memory

The function calloc( ) has the following prototype:

```
Ptr= (cast_type *)calloc(n,ele_ size);
```

calloc( ) provides access to the C memory heap, which is available for dynamic allocation of variable-sized blocks of memory.

Unlike malloc(), the function calloc( ) accepts two arguments: **n** and **ele_size**. The parameter **n** specifies the number of items to allocate and **ele_size** specifies the size of each item.

Return value:

- ✓ On success, i.e., if free memory is available, calloc( ) returns a pointer to the newly allocated memory. Usually, it is generic pointer. Hence, it should be typecast to appropriate data type before using it to access the memory allocated.
- ✓ On failure, i.e., if enough free memory does not exist for block, calloc( ) returns NULL. The constant NULL is defined in stdio.h to have a value zero. Hence, it is safe to verify the return value before using it.

Ex: 1) calloc(3,5); allocates 15 bytes of memory and returns the address of byte0.

2) malloc(6,sizeof(float)); allocates 24 bytes of memory and returns the address of byte0.

24. Explain about free( ) and realloc( ) allocation functions with an example?

Ans: **i) realloc( ) — grows or shrinks allocated memory**

The function realloc( ) has the following prototype:

```
Ptr= realloc(ptr, newsize);
```

The function realloc() allocates new memory space of size **newsize** to the pointer variable ptr and returns a pointer to the first byte of the new memoty block**.**

ptr is the pointer to the memory block that is previously obtained by calling malloc(), calloc() or realloc(). If **ptr** is NULL pointer, realloc() works just like malloc().

Return value:

- ✓ On success, this function returns the address of the reallocated block, which might be different from the address of the original block.
- ✓ On failure, i.e., if the block can‟t be reallocated or if the size passed is 0, the function returns NULL.

The function realloc() is more useful when the maximum size of allocated block cann‟t be decided in advance.

Ex: int *a;

a=(int *) malloc(30); //first 30 bytes of memory is allocated.

a=(int *) realloc(a,15); //later the allocated memory is shrink to 15 bytes.

ii) free( ) — de-allocates memory

The function free( ) has the following prototype:
The function free( ) de-allocates a memory block pointed by **ptr.**

```
free(ptr);
```

ptr is the pointer that is points to allocated memory by malloc( ), calloc( ) or realloc(). Passing an uninitialized pointer, or a pointer to a variable not allocated by malloc( ), calloc() or realloc() could be dangerous and disastrous.

Ex: int *a;

a=(int *) malloc(30); //first 30 bytes of memory is allocated.

free(a); //de-allocates 30 bytes of memory.