UNIT- V

Topics:

Functions – Function prototype, function return type, signature of a function, function arguments, call by value, Function call stack and Activation Records, Recursion v/s Iteration, passing arrays (single and multi-dimensional) to functions.

Storage classes- Automatic, Static, Register, External, Static and Dynamic linking implementation, C program memory (show different areas of C program memory and where different type of variables are stored), scope rules. –7 Hrs

Functions:

Function is a sub-program or module that performs a specific task. Every C program must have at least one function **main()** from where the program enters and starts executing.

Functions are of two kinds:

- **Inbuilt functions**
- **User defined functions**

Inbuilt functions:

Those functions which are already defined in the header files and are known to the compiler after inclusion of the header files. These are used by the program to perform a specific task such as input/output or some mathematical operation or string related operation etc.

Ex. printf(..),scanf(..), sqrt(..), pow(..), strcmp(..) etc.

User defined functions:

Those functions which are defined by the users and called inside the program to do a specific task that programmer expects.

Ex. to add two integers, to perform some mathematical operation which can't be done with an inbuilt function, to process strings as per the program requirement etc.

Ex. Following is general form of the C program that calls any function Func1().

```
void Func1() /*user defined function*/
{
        Executable statement(s);
}
void main()
{
        Executable statement(s);
        Func1();
        Executable statement(s);
}
```

ADVANTAGES OF USER DEFINED FUNCTIONS

1) User defined functions (UDF) helps to divide a large program into smaller modules which makes the programmer easy to understand and maintain the large programs.
2) Redundant/Repeated code in a program can be avoided by calling an UDF.
3) Programmers can easily identify and debug program (remove errors) easily.

Function Declaration

Every function in C program should be declared before it is used. Function declaration gives compiler information about function name type of arguments to be passed and the return type.

The syntax is as shown below:

return_type function_name(data_type argument1, data_type argument2……,data_type argumentN);

Function Call

Control of the program cannot be transferred to user-defined function unless it is called or invoked. The syntax to invoke a function inside a program:

function_name(argument1, argument2….argumentN);

or

variable_name= function_name(argument1, argument2….argumentN);

where the data type of **variable_name** above is same as return type of function.

Function Definition

Function definition contains programming codes to perform specific task.
The syntax is as shown below:

Data_type function_name(data_type argument1, data_type argument2…….,data_type argumentN)
{
//body of function
//return value – if Data type (return type) of the function is other than void
}

Example 1.

```
#include <stdio.h>
void Display();
void main()
{
        printf("Main program.\n");
        Display();
}
void Display()
{
        printf("I am inside the display function.\n");
}
```

Output:

Main program.
I am inside the display function.

In the above program the main program calls the function Display() with no arguments while the function displays message "I am inside the display function." After the message "Main program." is displayed by the main program.

Note before calling the function Display() compiler is informed about its prototype before the main function by declaring it and ending with a semicolon.

Example 2.

```
#include <stdio.h>
int Sum(int x, int y);
```

← **Function prototype**

```
void main()
{       int x, y, result;

                        Actual arguments
         x=77;
         y=103;
        result=Sum(x , y);
                                                Function Call
         printf("The sum of %d and %d is %d\n", x , y , result );
}
```

```
                        Return type
                                Formal arguments
        int Sum(int x, int y)
        {
                int temp;
                temp=x+y;       Returns a value to the calling program and
                                the value should be same as the return type.
                return(temp);
        }
```

Output:

The sum of 77 and 103 is 180

In the above program the function **Sum** is called by the main program by passing two arguments x and y of integer type. The function returns a value of integer type. Hence, its definition shows the return type as **int**. While the arguments in the main program called as actual arguments are sent by value i.e their actual value is passed. In the function definition the arguments passed are known as formal arguments and their data type and order of the arguments should remain same although their names while declaring again the arguments in the function definition may be changed.

Function call types:

- **Call by value**
- **Call by reference**

Call by value: The function is called by sending arguments by their values i.e the values passed by the calling program through the variables are copied to the formal parameters.

Example 3.

```
#include <stdio.h>
int Max(int,int);   /* Function Prototype */
void main()
{       int big;
                printf("Main program.\n");
                big=Max(10,15);
                printf("The biggest number is %d",big);
}
int Max( int x, int y)
{
                if( x > y )
                        return x;
                else
                        return y;
}
```

Output:

The biggest number is 15

The values 10 and 15 in the main program are passed to the function **Max** are passed by values and are called as actual arguments/parameters. These values are copied to the formal parameters x and y. The function after processing returns x if x is greater than y or returns y if y is greater than x to the calling program.

Example 4.

```
#include <stdio.h>
void Display();
void main()
{
        printf("Main program.\n");
        Display();
}
void Display()
{
        printf("I am inside the display function.\n");
}
```

Example 5.

/* Program passes two arguments of integer type to the function Sum. The function Sum adds the two integers returns their sum after addition in the function to the main program.*/

```
#include <stdio.h>
int Sum(int x, int y);

void main()
{   int x,y,result;
    x=77;
    y=103;
    result=Sum(x , y);
     printf("The sum of %d and %d is %d\n",x,y,result);
}
int Sum(int x, int y)
{
    int temp;
    temp=x+y;
    return(temp);
}
```

Passing an array to a function:

Example 1.

```
#include <stdio.h>
int add(int num[ ], n)
{  int i,sum=0;
        for(i=0;i<n;i++)
                sum=sum+num[i];
   return sum; /* returns sum to the calling program */
}
void main()
{
         int n, sum,i,num[10];
        printf("Enter a positive integer: ");
        scanf("%d",&n);
        printf("Enter %d integers: ",n);
        for(i=0;i<n;i++)
                scanf("%d",&num[i]);
        sum=add(num,n);
        printf("\nSum is %d", sum);
}
```

Output:

```
Enter a positive integer: 4
Enter 4 integers: 9 3 77 16
Sum is 105
```

Example 2.

```
#include <stdio.h>
#include <ctype.h>
int    CountVowels ( char str[] );

void main()
{       char str[25];
        Int vowels;
        printf(“Enter a string:”);
        gets(str);
        vowels=CountVowels(str);
```

```
        printf("Number of vowels in %s is %d\n",str,vowels);
}

int CountVowels(char str[])
{
        int cnt=0,i=0;
        while(str[i]!='\0')
        {
                switch(tolower(str[i]))
                {
                        case 'a':
                        case 'e':
         case 'i':
                        case 'o':
                        case 'u':cnt++;
                                break;
                }
                i++;
        }
         return(cnt);
}
```

The above program accepts a string in the main program, then it send that string to the function **CountVowels**, which is an array. The function then after counting the vowels present in the string passed returns the count of the vowel thru that function.

The Call Stack is thus composed of any number of activation records that get added to the stack as new subroutines are added, and removed from the stack (usually) as they return.

The actual structure and order of elements is platform and even implementation defined.

Allows C programmers to debug their application and read memory dumps even in the absence of a debugger or debugging symbols.

RECURSION

- A function that calls itself is known as recursive function and this process is called recursion.
- Example-1: Program to find sum of first N natural numbers using recursion.

```
#include <stdio.h>
int add(int n)
{
if(n==0)
        return 0;
else
        return n+add(n-1); /*self call to function add() */
}
void main()
{
        int n, sum;
       printf("Enter a positive integer: ");
       scanf("%d",&n);
       sum=add(n);
       printf("\nSum is %d", sum);
}
```

Output:

```
Enter a positive integer: 10
Sum is 55
```

In the above program:

Function add(n) is called with the argument n. Inside the function add, it sums the value of n and then calls the same function add by decreasing the value of n. This process is repeated until the value of n is greater than 0.

When the value of n reaches zero, the function starts returning, and in the process it starts adding the number stored on the stack i.e from 1 to n.

- The following steps helps to understanding in a better way the working

of the recursive function:

```
add(5)
=5+ add(4)
=5+4+ add(3)
=5+4+3+ add(2)
=5+4+3+2+ add(1)
=5+4+3+2+1+ add(0)
=5+4+3+2+1+0
=5+4+3+2+1
=5+4+3+3
=5+4+6
=5+10
=15
```

- Any recursive function must have an exit condition. In the above example when, n is equal to 0, there is no more recursive call, the recursion ends and functions starts returning the values on the stack by performing the desired mathematical operation.

  Example-2: Program to find the factorial of a positive integer.

```
#include <stdio.h>
int fact(int n)
{
if(n==0)
        return 1;
else
        return n*fact(n-1); /*self call to function fact(..) */
}
void main()
{
int n, Factorial;
printf("Enter a positive integer: ");
scanf("%d",&n);
Factorial=fact(n);
printf("\nFactorial of %d is %d", n,Factorial);
}
```

Output:
Enter a positive integer: 5
Factorial of 5 is 120

- The following steps helps to understanding in a better way the working of the recursive function:

fact(5)
=5*fact(4)
=5*4*fact(3)
=5*4*3*fact(2)
=5*4*3*2*fact(1)
=5*4*3*2*1*fact(0)
=5*4*3*2*1*1
=5*4*3*2*1
=5*4*3*2
=5*4*6
=5*24
=120

- In the above example when, n is equal to 0, there is no recursive call and recursion ends. The function starts returning the values on the stack by performing the desired mathematical operation.

Example-3: Program to find the Sum of first N natural numbers using recursion.

```
#include <stdio.h>
int Sum(int n)
{
if(n==0)
        return 0;
else
        return ( n+Sum(n-1)); /*self call to function Sum(..) */
}
void main()
{
int n, FinalSum;
printf("Enter a positive integer: ");
scanf("%d",&n);
```

```
FinalSum =Sum(n);
printf("\nThe sum of first %d natural numbers is %d", n, FinalSum);
}
```

Output:

Enter a positive integer: 10

The sum of first 10 natural numbers is 55

Storage classes

General Syntax:

storage_class var_data_type var_name;

1. **auto:** This is the default storage class for all the variables declared inside a function or a block. Hence, the keyword auto is rarely used while writing programs in C language. Auto variables can be only accessed within the block/function they have been declared and not outside them (which defines their scope). Of course, these can be accessed within nested blocks within the parent block/function in which the auto variable was declared. They are assigned a garbage value by default whenever they are declared.

```
void F1();
void main()
{ auto int x;
  printf("x=%d",x);   // Garbage Value
  F1();
}
voif F1()
{
  printf("x=%d",x);    //x is not accessible in the function F()
}
```

2. **extern:** Extern storage class tells us that the variable is defined elsewhere and not within the same block where it is used. Basically, the value is assigned to it in a different block and this can be overwritten/changed in a different block as well. So an extern variable is nothing but a global variable initialized with a desired value where it is declared, in order to be used elsewhere. It can be accessed within any function/block. Also, a normal global variable can be made extern as well by placing the 'extern' keyword before its declaration/definition in any function/block. This basically signifies

that we are not initializing a new variable but instead we are using/accessing the global variable only. The main purpose of using extern variables is that they can be accessed between two different files which are part of a large program.

P1.c:

```
extern int var=10;
void main()
{ int x;
  printf("x=%d",x);   // Garbage Value
  F1();
}
```

P2.c:

```
extern int var;
void main()
{ int x=100;
  x=x+var;    //var is accessible in the program P2.c although it is declared
              // in P1.c
  printf("x=%d",x);
  F1();
}
```

3. **static**: This storage class is used to declare static variables which are popularly used while writing programs in C language. Static variables have a property of preserving their value even after they are out of their scope! Hence, static variables preserve the last value that was assigned in their scope. Hence, they are initialized only once and exist till the termination of the program. Thus, no new memory is allocated because they are not re-declared. Their scope is local to the function to which they were defined. Global static variables can be accessed anywhere in the program. By default, they are assigned the value 0 by the compiler.

```
#include <stdio.h>
void F1(int x)
{
  static int var=10;
  printf("\nx=%d, var=%d\n",x,var);
```

```
    var++;
    x++;
  }

  void main()
  { int x=100,i;
    for(i=0;i<3;i++)
        F1(x);
  }
```

Output:

x=100, var=10
x=100, var=11
x=100, var=12

4. **register**: This storage class declares register variables which have the same functionality as that of the auto variables. The only difference is that the compiler tries to store these variables in the register of the microprocessor if a free register is available. This makes the use of register variables to be much faster than that of the variables stored in the memory during the runtime of the program. If a free register is not available, these are then stored in the memory only. Usually few variables which are to be accessed very frequently in a program are declared with the register keyword which improves the execution time of the program. An important and interesting point to be noted here is that we cannot obtain their address because they are not stored in the memory.

Ex. Register variable is declared similar to an auto variable except that it is declared with the keyword **register** before the data type.

register int rvar=26;

Storage Specifier	**Storage Area**	**Initial Value**	**Scope**	**Life**
Auto	Stack	Garbage	Within block	End of block
extern	Data Segment	Zero	Global multiple files	End of program
static	Data Segment	Zero	Within block	End of program
Register	CPU Register	Garbage	Within block	End of block

Activation record

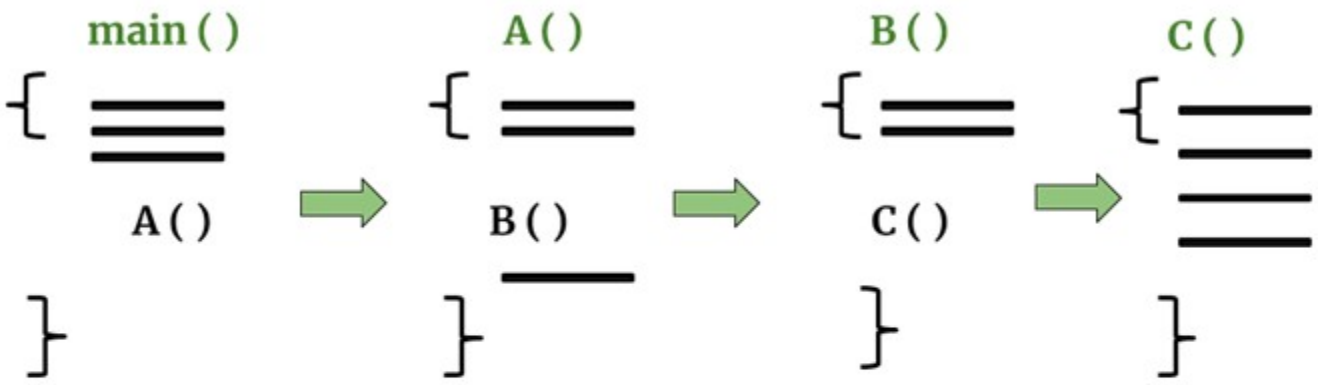
Activation record is used to manage the information needed by a single execution of a procedure (function). An activation record is pushed into the stack when a procedure (function) is called and it is popped when the control returns to the caller function.

Activation record is another name for Stack Frame. It's the data structure that composes a call stack. It is generally composed of:

- Locals to the callee
- Return address to the caller
- Parameters of the callee

ACTIVATION RECORDS

Control stack or runtime stack is used to keep track of the live procedure (**function(s)**) activations i.e the procedures whose execution have not been completed. A procedure name (**function**) is pushed on to the stack when it is called (activation begins) and it is popped when it returns (activation ends). Information needed by a single execution of a procedure is managed using an **activation record** or **frame**. When a procedure is called, an activation record is pushed into the stack and as soon as the control returns to the caller function the activation record is popped.



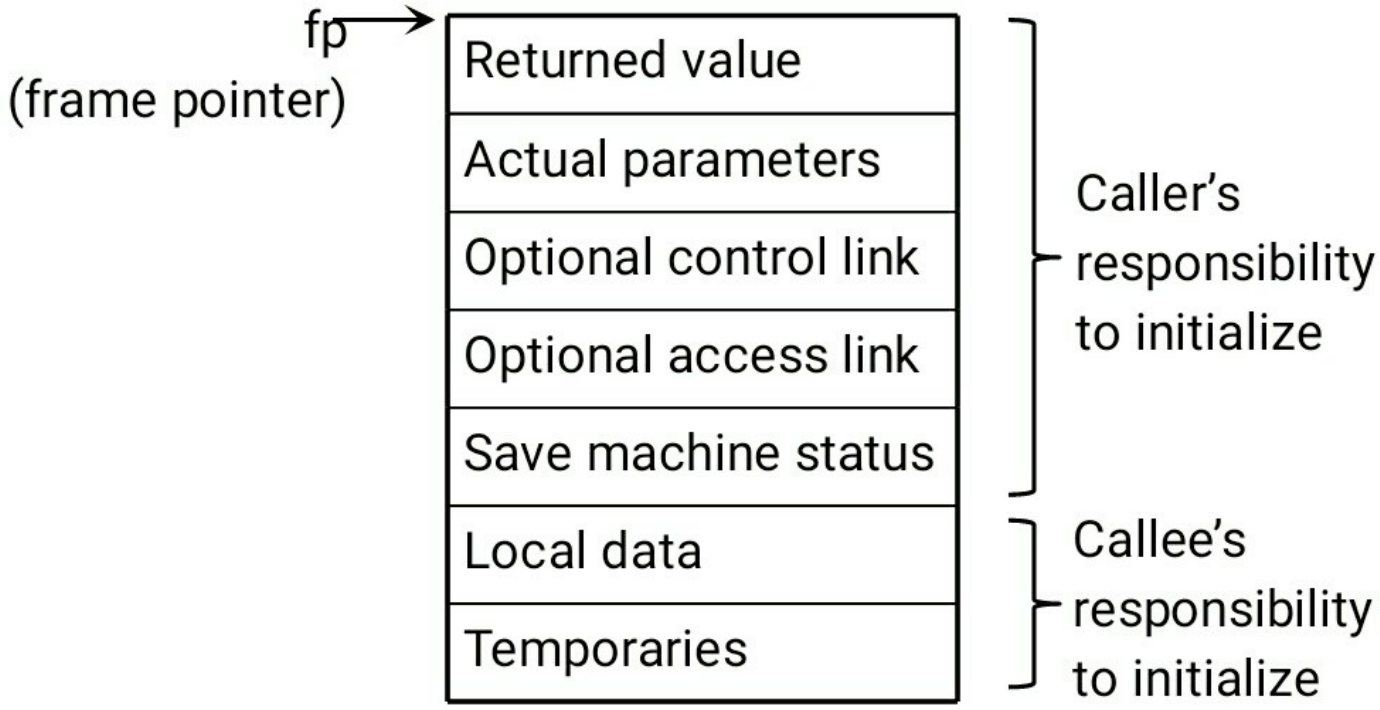
Before going to A() activation record of main is created when A() is called.

Before going to B() activation record of main and A() in stack.

Before going to C() activation record of main and B() is pushed in stack.

Activation record comprises of the following fields:

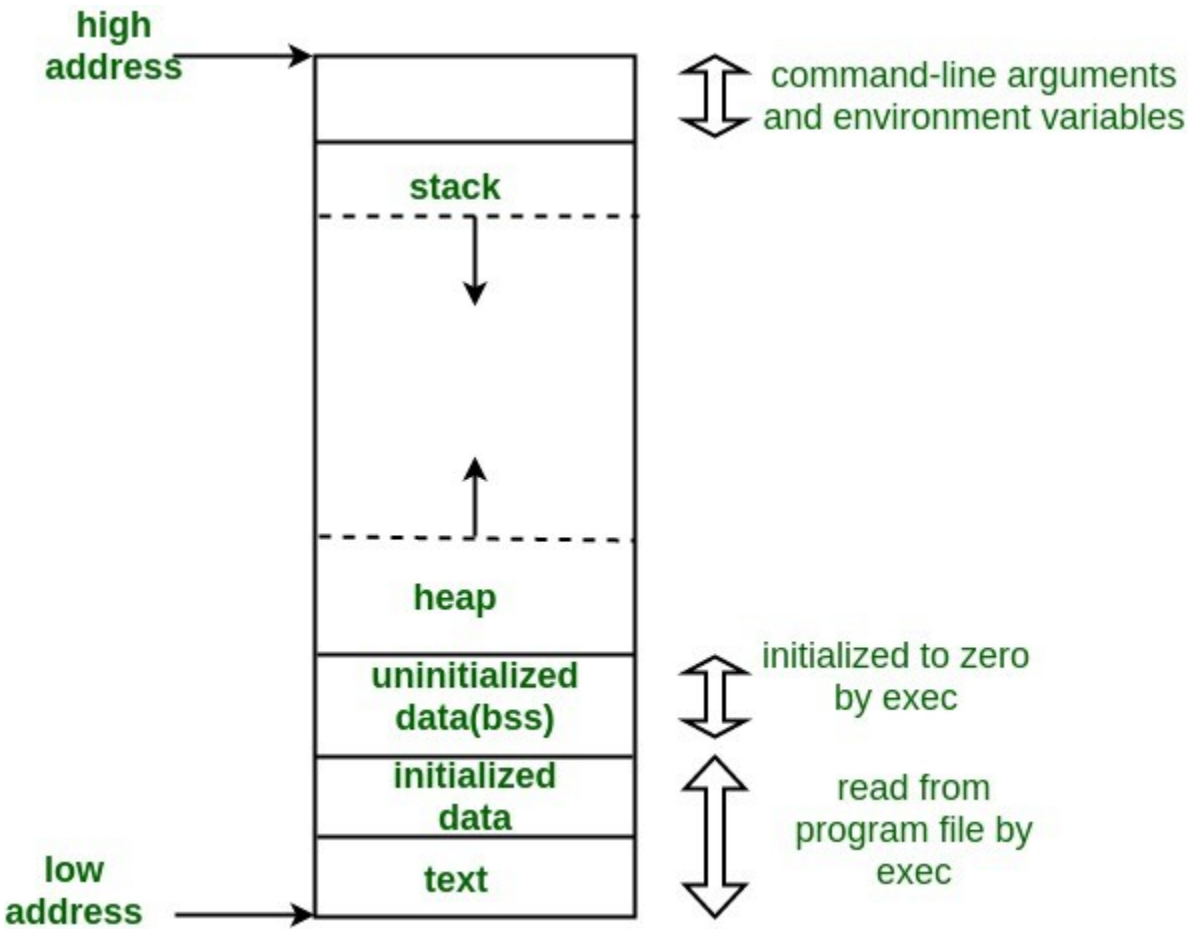
- **Local variables:** holds the data that is local to the execution of the procedure.
- **Temporary values**: stores the values that arise in the evaluation of an expression.
- **Saved machine registers:** holds the information about status of machine just before the function call.
- **Access link (optional):** refers to non-local data held in other activation records.
- **Control link (optional):** points to activation record of caller.
- **Return value:** used by the called procedure to return a value to calling procedure
- **Actual parameters**



Memory map of C program

A typical memory representation of C program consists of following sections.

1. Text segment
2. Initialized data segment
3. Uninitialized data segment
4. Stack
5. Heap



A Typical memory layout of a running program (process)

1. Text Segment:
A text segment, also known as a code segment or simply as text, is one of the sections of a program in an object file or in memory, which contains executable instructions. As a memory region, a text segment may be placed below the heap or stack in order to prevent heaps and stack overflows from overwriting it. Usually, the text segment is sharable so that only a single copy needs to be in memory ***for frequently executed programs***, such as text editors, the C compiler, the shells, and so on. Also, the text segment is often ***read-only***, to ***prevent a program from accidentally modifying its instructions***.

2. Initialized Data Segment:
Initialized data segment, usually called the Data Segment. A data segment is a portion of virtual address space of a program, which contains the global variables and static variables that are initialized by the programmer.
Note that, data segment is not read-only, since the values of the variables can be altered at run time. This segment can be further classified into initialized read-only area and initialized read-write area.

For instance the global string defined by char s[] = “hello world” in C and a C statement like int var=1 outside the main (i.e. global) would be stored in initialized read-write area. And a global C statement like const char* string = “hello world” makes the string literal “hello world” to be stored in initialized read-only area and the character pointer variable string in initialized read-write area.

Ex: static int i = 10 will be stored in data segment and global int i = 10 will also be stored in data segment

3. Uninitialized Data Segment:

Uninitialized data segment, often called the “bss” segment, named after an ancient assembler operator that stood for “***block started by symbol***.” Data in this segment is initialized by the kernel to arithmetic 0 before the program starts executing.
For instance a variable declared **static int i;** would be contained in the BSS segment. For instance a global variable declared **int j;** would be contained in the BSS segment.

4.Stack:
The stack area grows in the opposite direction; when the stack pointer meets the heap pointer, implies free memory is exhausted. (With modern large address spaces and virtual memory techniques they may be placed almost anywhere, but they still typically grow opposite directions.)
On the standard PC x86 computer architecture, it grows toward address zero; on some other architectures it grows in the opposite direction. The set of values pushed for ***one function call*** is termed a “***stack frame***”;

Stack, where ***automatic variables are stored, along with information*** that is saved each time a function is called. Each time a function is called, ***the address of where to return to and certain information about the caller’s environment, such as some of the machine registers, are***

saved on the stack. The newly called function then allocates room on the stack for its automatic and temporary variables. The best example is a recursive function in C. Each time a recursive function calls itself, a new stack frame is used, so one set of variables doesn't interfere with the variables from another instance of the function.

5. Heap:

Heap is the segment where dynamic memory allocation usually takes place. The heap area begins at the end of the BSS segment and grows to larger addresses from there. The Heap area is managed by ***malloc / calloc***, ***realloc*** and ***free*** functions. The Heap area is shared by all shared libraries and dynamically loaded modules in a process.

Static and Dynamic Linking of Libraries:

Let's understand the life cycle of a typical program right from writing source code to its execution.

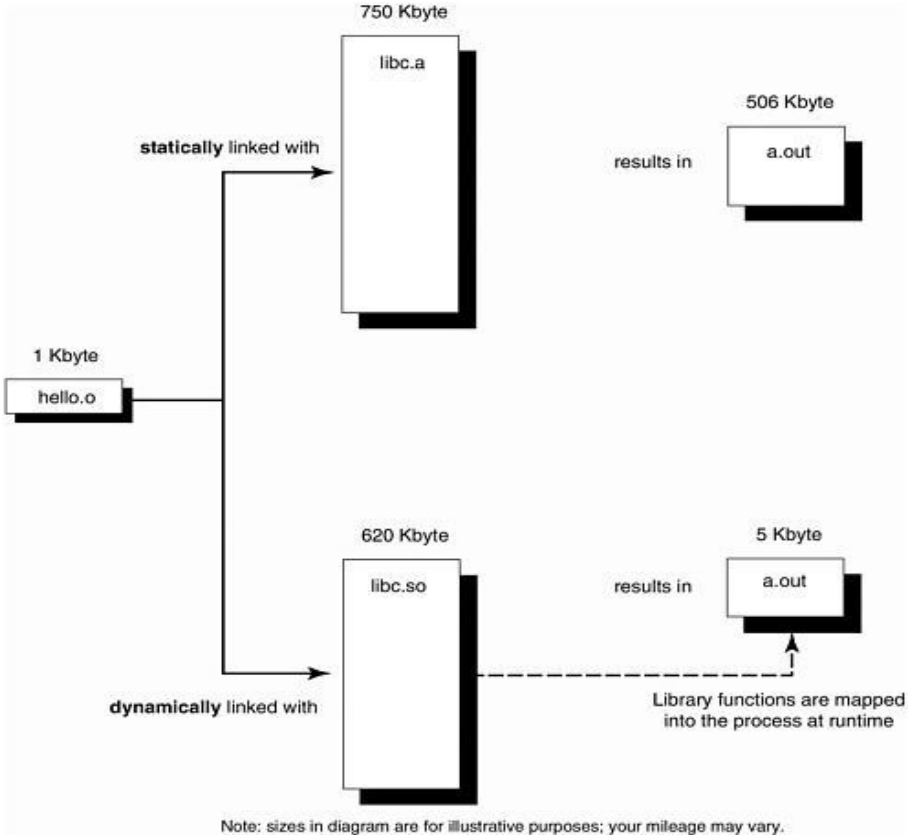
A program is first written using any editor of programmer's choice in form of a text file, then it has to be compiled in order to translate the text file into object code that a machine can understand and execute.

Program life cycle (**write -> compile -> link -> load -> execute**).

The program we write might make use of other programs, or libraries of programs. These other programs or libraries must be assembled together with the program written in order to execute it and linking (part of the compilation) is the process of bringing these external programs together for its successful execution. Static and dynamic linking are two processes of collecting and combining multiple object files in order to create a ***single executable file***.

Linking can be performed at both ***compile time***, when the source code is translated into machine code; and ***load time***, when the program is loaded into memory by the loader, and even at run time, by application programs. This is performed by programs called ***linkers***. Linkers are also called ***link editors***. Linking is performed as the last step in compiling a program.

After linking, for execution program must be moved into memory. In doing so, the ***addresses to the data and instructions*** must be assigned final for its execution.



The figure below illustrates the process in a linking statically and dynamically linked libraries and the final size of the file on compiling and executing using both the methods.

Static Linking	Dynamic Linking
Static linking is the process of copying all library modules used in the program into the final executable image.	In dynamic linking the names of the external libraries (shared libraries) are placed in the final executable file while the actual linking takes place at ***run time*** when both executable file and libraries are placed in the memory.
Static linking is performed by programs called ***linkers*** as the last step in compiling a program. Linkers are also called ***link editors***.	Dynamic linking is performed at run time by the ***operating system***.
Statically linked files are significantly larger in size because external programs are also combined into the executable files.	In dynamic linking only one copy of shared library is kept in memory. This drastically reduces the size of executable programs. Hence, saves memory.
In static linking, if any of the external programs has changed then the program needs to be recompiled and re-linked again otherwise the changes are not reflected in existing executable file.	In dynamic linking this is not the case and individual shared modules can be updated and recompiled. There is no need to recompile and relink the program.
Statically linked program takes constant load time every time it is loaded into the memory for execution.	In dynamic linking load time might be reduced if the shared library code is already present in memory.
Programs that use statically-linked libraries are usually faster than those that use shared libraries.	Programs that use shared libraries are usually slower than those that use statically-linked libraries.
In statically-linked programs, all code is contained in a single executable module. Therefore, they never run into compatibility issues.	Dynamically linked programs are dependent on a compatible library. If a library is changed (for ex., a new compiler release may change a library), then applications might have to be reworked to be made compatible with the new version of the library. If a library is removed from the system, programs using that library will no longer work.

Difference between Static Linking and Dynamic Linking.

========== End of UNIT- V ==========