UNIT IV

** this is memory management done in a system. Though it is not in your syllabus, it is important, so mentioned in content here.

When the term memory is used in programming, it generally refers the RAM, the main memory of the system. When the program is compiled, the compiler decides how much memory is required for running that application. When the program execution starts, the application gets reserved with some memory in RAM to have its code loaded and memory allocated for the data members (variables).

The application's memory is logically divided into four parts.

➔ Code part
➔ Static / global variables part
➔ Stack memory
➔ Heap memory

The machine code, in case of C, the executable file that has extension ".exe" is loaded into **CODE PART** of the application's memory.

The static variables and global variables of the program are allocated in a special memory reserved for them.

The memory part in which the local variables are allocated with memory is called **STACK**. The memory in stack is managed by the operating system itself. The programmer need not specify explicitly how much memory is to be allocated and this memory is released automatically soon after the program execution completed. The stack size is fixed based on number of variables declared. Once program execution starts, the stack size can not be either increased / decreased.

The memory part in which the memory is assigned dynamically is called **HEAP MEMORY**. This dynamic memory is managed with pointers. The memory assigned using malloc() etc functions, reserves the memory in the heap memory of application's reserved memory. The heap size can be modified. It can be increased / decreased and also released.

The dynamic memory managed in HEAP, must be allocated explicitly and must be released explicitly. If the memory is not released explicitly, then that memory cannot be further used for any other purpose. And the heap size will be decreasing for all programs that affect all the processes. This case is called "MEMORY LEAK". A good programmer should not permit "memory leak" in the program. So the dynamic memory assigned in HEAP must be released explicitly.

POINTERS

POINTERS: The Pointer is a special data type used to manipulate values by holding address of the memory location. The pointer is declared with an **ASTERISK**. The pointers can be used in two ways, either by allocating memory dynamically to the pointer or by storing address of other static memory.

Ex:

```
void main(){
        int a=5;
        int *p;
        p=&a;          //storing address of static memory (variable)
        printf("%d",*p);    //will display 5
}
```

The pointer manipulation is bit different from normal operations done on variables. As the pointer points to memory addresses, when the pointer is manipulated with asterisk, they manipulate the values in that addresses and when these are manipulated without asterisk, they manipulate the addresses. When the address in the pointer is modified, the pointer points to a new location whose address is stored within.

Ex:

```
void main(){
        int a,b,*p,*q;
        a=5;
        b=10;
        p=&a;
        q=&b;
        printf("a=%d ; b=%d; *p=%d; *q=%d",a,b,*p,*q);
}
```

Now the above will display "a=5; b=10; *p=5; *q=10;"

Ex:

```
void main(){
        int a,b,*p,*q;
        a=5;
        b=10;
        p=&a;
        q=&b;

        p=q;  //pointer manipulated without *
```

```
        printf("a=%d ; b=%d; *p=%d; *q=%d",a,b,*p,*q);
}
```
Now the above will display "a=5; b=10; *p=10; *q=10;"

When the pointer is manipulated without *, so the address held in the pointer "q" is copied into the pointer "p". So the pointer "p" now points to the new location. And the value in the location pointed by "p" is not modified.

Now the above will display "a=5; b=10; *p=10; *q=10;"

Ex:

```
void main(){
        int a,b,*p,*q;
        a=5;
        b=10;
        p=&a;
        q=&b;

        *p=*q;        //pointer manipulated with *

        printf("a=%d ; b=%d; *p=%d; *q=%d",a,b,*p,*q);
}
```
Now the above will display "a=10; b=10; *p=10; *q=10;"

When the pointer is manipulated with * , the values in the locations pointed by the pointers are manipulated. The variable "a" is assigned with value pointed by the pointer "q" and the value is assigned "NOT THROUGH variable" but through the POINTER. So the value at location pointed by pointer "p" is modified, i.e. the value of "a" become 10.
Now the above will display "a=10; b=10; *p=10; *q=10;"

When a pointer is declared and not assigned with any address, it is suggested to store a null value in that pointer, so that it won't point to a garbage address. The stdio.h library has a pre-defined macro NULL, that can be assigned to the pointer.
Ex: int *p; p=NULL;

Q. Explain how to use pointers in C

The pointers can be used to point to large memory like arrays.

```
void main(){
        int a[]={5,10,15,20,25};
        int *p;
```

```
        p=&a[0];
        printf("%d",*p) ;   //will display 5
        p++;
        printf("%d",*p);    //will display 10
}
```

The pointer arithmetic operation is different from normal arithmetic operation. It is done on number of bytes based on data type of the pointer. The int* when increased by 1 it gets increased by 2 bytes and the float* will increase by 4 bytes. In any case, the pointer moves to next element in the memory.

- So the programmer must use **appropriate pointer** in the program, the int* can be used to point only to int memory.

```
void main(){
        int a[]={5,10,15,20,25};
        int *p;
        p=&a[0];
        printf("%d",*p) ;   //will display 5
        printf("%d", *p+1 );        //will display 6
}
```

Here it displays 6, as pointer has higher priority, first *p is evaluated and +1 done.

The priority can be changed using ( ).

```
void main(){
        int a[]={5,10,15,20,25};
        int *p;
        p=&a[0];
        printf("%d",*p) ;           //will display 5
        printf("%d",* (p+1) );      //will display 10
}
```

Here first p+1 is done and then * applied to that new address. In p+1, as "p" is an int pointer, it will increase by 2 bytes, and points to next element.

When we declare an array in C, the C runtime creates a pointer with same name of the array and stores the base address of array into that pointer.

So in above example, when the array int a[] is declared, the RE has a pointer with name "a" itself. That built in pointer can be manipulated as of normal pointer.

```
void main(){
        int a[]={5,10,15,20,25};
        int i;
```

```
        for(i=0;i<5;i++)
                printf("%d  ",*(a+i) );
}
```

Conclusion : in C or C++, we can declare arrays and use either array or pointer syntax and also, we can declare pointers and use array or pointer syntaxes. The difference is arrays are static memory and pointers support dynamic memory.

Q. Explain about pointer arithmetic operations
Q. Explain about Arrays and pointers.

Memory allocation in C programs:

In C, language, the memory management can be done in two ways.

1. Static memory management :
   The memory once assigned at beginning of the program is called static memory and the static memory is allocated with memory in the stack and its size is fixed. It cannot be modified with its size and can not be released.
   Ex: variables, arrays, structure / union variables etc.

2. Dynamic memory management: the memory that can be assigned during running of program is called dynamic memory. This need not be assigned at beginning of program. This can be assigned once program execution starts. So it is called dynamic memory. The dynamic memory can be increased / decreased with its size and also released. The dynamic memory is managed with in heap of the application's memory.

The dynamic memory is managed with the help of pointers. Functions related with dynamic memory management are defined in the library "alloc.h" library.

malloc() : this function allocates specified number of bytes memory and returns its base address that can be held by a pointer.

This function must be type casted based on type of pointer used.

Ex:

```
int *p;
p= (int*) malloc( sizeof(int) *5 );
```

here the sizeof(int) returns memory required for 1 integer variable and it is multiplied by 5, gives 10. The malloc() allocates 10 bytes memory and returns

the base address of the reserved memory that is type casted as “int*” and held by the pointer “p”. Now the pointer “p” points to 5 integers memory.

calloc() : it is similar to malloc(), but the malloc() leaves garbage values in the reserved memory and the calloc() initializes all elements with zeros.

realloc() : is used to increase or decrease the size of memory reserved, but it is not much used, as it leads to data loss.

free() : this function is used release the memory assigned to the pointer.
Ex: free(p);

The memory assigned once to a pointer with malloc() or calloc() or realloc() is reserved with in heap of the memory, and it must be released with free(). If not released, then the reserved memory can not be further used by the OS, and this is called “MEMORY LEAK” situation.

A pointer when freed, it still holds the garbage address, and mis-behaves. Such pointer is called **DANGLING** pointer. A good programmer should not leave dangling pointers in program. So once a pointer is freed, then it must be assigned with NULL.

Q. explain about dynamic memory management in C
Q. what are different functions related with dynamic memory management

---

Functions

Function Prototype, definition and calling : When a software project is developed, it includes lakhs of lines of code. Writing thousands of lines of code within one main() function leads to confusion and gives burden on system.

Almost all the high level languages provide a facility of dividing programs into modules called functions. A function is piece of code identified with a name.

A function has two aspects, a definition and a call. The function definition includes the process to be executed and the function call is invocation of definition. A function defined once can be called number of times.
Advantages of functions:

- ➔ Increase in modularity of program
- ➔ Increase in readability of program
- ➔ Easy identification of errors
- ➔ Easy debugging of code
- ➔ Reusability of code

Function declaration / prototyping:

In C, when a function defined later the call, i.e. when a function is called first and defined later, then it must be given with forward declaration, called prototyping. The forward declaration is an indication to Runtime Environment that a function is called prior its definition.

Syntax : return_type fun_name(parameters) ;

The forward declaration can be done either globally or within main() definition in declaration area. If it is declared within main() definition, then it can be called only within main(). If declared globally, then it can be called anywhere in program.

Function Definition:

The Function definition includes the process to be done. The entire process to be done is grouped using braces and identified with a name. When the function is called the process given in definition executes.

Syntax:

```
return_type fun_name( formal arguments)
{
        Process to be done;
}
```

Function call:

The function call is invocation or activation of the definition. A function defined once, can be called number of times.

Syntax : fun_name(actual arguments) **;**

Q. Explain about Function Prototype, definition and calling
Q. Give a note on Functions

Passing arguments:

The functions are provided with ( ) to pass values from one function into another, from function call into function definition. The arguments given at function call are called actual arguments and the arguments at definition are called formal arguments. The actual arguments can be either variable names or values. The formal arguments are variables with individual declarations.

During program execution the values of actual arguments are copied into formal arguments, and the formal arguments are processed with in definition without affecting actual arguments.

So, the number, data type and sequence of formal arguments must be matching with that of actual arguments.

Return statement:

The function definition after doing process can send values back from called function into calling function with the "return" keyword. The values can be returned for further use.

➔ The return can return only one value at a time
➔ The return is last statement to be executed in a function
➔ A function can have only one return executed.

The function's return type is data type specification of the value being returned by the function. In C, the Runtime's default data type is int. So, if no return type is specified, the C RE takes the return type as int. if the function won't return any value, then the return type must be specified as "void".
As the main() won't return any value, it is generally returned on "void main()".

Ex:

```
int add(int a, int b) ;
void main()
{       int a,b,c;
        printf("Enter two numbers:");
        scanf("%d%d",&a,&b);
        c=add(a,b);
        printf("sum is %d",c);
}
int add(int x, int y)
{
        return  x+y;
}
```

Q. Explain about passing arguments and return type of a function

---

Nesting of functions:

Standard C does not support nested functions, meaning you cannot define a function inside the body of another function. Functions in C must be defined at the global scope.

However, some compilers, notably GCC (GNU Compiler Collection), offer an extension to the C language that allows for nested functions. This feature is non-standard and therefore not portable across different compilers or environments.

Example of Nested Function (GCC-specific):

```
#include <stdio.h>
```

```
void outerFunction() {
   // Nested function (GCC extension)
   void innerFunction() {
      printf("Hello from inner function!\n");
   }
   innerFunction(); // Calling the nested function
}

int main() {
   outerFunction();
   return 0;
}
```

Important Considerations:

Non-standard: Nested functions are a GCC extension and not part of the C standard. Code using them will not be portable to compilers that do not support this extension.

Scope: Nested functions can access local variables and parameters of the enclosing function, similar to lexical scoping.

Limited Use: Due to their non-standard nature and potential for reduced portability, nested functions are generally discouraged in C programming unless there is a very specific and compelling reason to use them within a GCC-only environment.

Q. Explain about nested functions.

Categories of functions:

In C language, functions are generally classified into two major categories based on how they are created and how they communicate data.

1. Based on where the function is defined

(a) Library Functions : These functions are predefined in C header files. We simply include the header file and use them.

Examples:

printf() and scanf() → <stdio.h>

sqrt(), pow() → <math.h>

strlen(), strcpy() → <string.h>

(b) User-Defined Functions : These are functions created by the programmer. Used to improve code modularity, readability, and reusability.

Example:

```
int sum(int a, int b) {
   return a + b;
}
```

2. Based on arguments and return type

C functions can also be categorized based on whether they take inputs (arguments) and whether they return a value.

(a) Function with No Arguments and No Return Value
Used when the function performs a task but does not need input or give output.
Example:

```
void display() {
   printf("Hello!");
}
```

(b) Function with Arguments but No Return Value
Takes input but does not return anything.
Example:

```
void add(int a, int b) {
   printf("%d", a + b);
}
```

(c) Function with No Arguments but Returns a Value
Does not take input but returns a result.

Example:

```
int getNumber() {
   return 10;
}
```

(d) Function with Arguments and Return Value
Most commonly used and Takes input and returns a result.

Example:

```
int multiply(int x, int y) {
   return x * y;
}
```

Q. Explain about types of functions.

Recursion (Basic Concept only):

Recursion is the process of repeating items in a self-similar way. In programming languages, if a program allows you to call a function inside the same function, then it is called a recursive call of the function

The C programming language supports recursion, i.e., a function to call itself. But while using recursion, programmers need to be careful to define an exit condition from the function otherwise it will go into an infinite loop.

Recursive functions are very useful to solve many mathematical problems, such as calculating the factorial of a number, generating Fibonacci series, etc.

Ex :

```
#include <stdio.h>

int factorial(int);

void main() {
   int num;

   printf("Enter a number: ");
   scanf("%d", &num);
   printf("Factorial of %d = %lld", num, factorial(num));
}
int factorial(int n) {
   if (n == 0 || n == 1)
      return 1;
   return n * factorial(n - 1);
}
```

Q. Explain recursion with an example code.

Parameter passing by address & by value:

The functions when called, they can be passed either variables or addresses. Based on what is passed, the function calls are classified into two types.

Function call by value:

When calling functions, the local variables of calling functions are passed as arguments to called function as arguments and in called function, they are

received as formal arguments and processed. The process done in the called function will not affect the values in actual arguments.

Ex:

```
void swap(int,int);
void main()
{
        int a,b;
        a=5;
        b=10;
        printf("before swapping a=%d b=%d\n",a,b);
        swap(a,b);
        printf("after swapping a=%d b=%d\n",a,b);
}
void swap(int a, int b)
{
        int t;
        t=a;
        a=b;
        b=t;
        printf("in swap function a=%d b=%d\n",a,b);
}
```

Output:

```
before swapping a=5 b=10
in swap function  a=10 b=5
after swapping a=5 b=10
```

though the function swapped, it is not affected in actual arguments.

Function call by reference:

To make sure the functions make do the process on values stored in actual arguments, the programmer can pass address of actual arguments and receive them into the pointers declared as formal arguments and to do process through pointers. Though the pointers are declared in called function, as they hold address of actual arguments, when the process is done through pointers, they affect the actual arguments.

Ex:

```
void swap(int*, int*);
void main()
{
        int a,b;
        a=5;
        b=10;
        printf("before swapping a=%d b=%d\n",a,b);
        swap(&a,&b);
        printf("after swapping a=%d b=%d\n",a,b);
}
void swap(int *a, int *b)
{
        int t;
        t=*a;
        *a=*b;
        *b=t;
        printf("in swap function a=%d b=%d\n",*a,*b);
}
```

Output:

```
before swapping a=5 b=10
in swap function  a=10 b=5
after swapping a=10 b=5
```

Here as the pointers in called function points to the address of actual arguments, the actual arguments are processed.

Q. Explain function call by value and call by reference.

Local and Global variables:

SCOPE of variables:

The scope of variables are based on the location they are declared. The variables based on location of declaration, are classified into two types.

- ➔ Local variables: the variables declared with in a function definition are called local variables to that function. These have their scope only within that function. So they can be accessed in that function only.

They are not accessible in other functions of the program, so the functions are given with ( ) to pass them as arguments.

➔ Global variables: the variables declared outside of all the functions are called global variable and they have their scope to entire program, so they can be accessed in all the functions. As these are accessible in all the functions, there is a chance of miss-usage of them, that leads to data corruption. So to provide security to data, it is suggested to programmers not to use global variables or reduce usage of global variables.

STORAGE CLASSES: The C has a concept called storage classes, that specifies the lifetime of the variable and its scope. The different storage classes are as follows.

➔ Auto variables : it is default storage class of C. when a variables is declared with data type keyword, it is by default auto variable. If it is declared as local variable, it has its scope with in that function and if declared as global, then it has its scope in entire program. When declared these will have garbage values.
Ex: auto int a;

➔ Extern variables: In olden days, the programs used to be done in other languages like Cobol, Pascal etc. if there is a requirement of using code written in C, to be used in other language programs, then the code in C must be declared with keyword “extern”. In C, the variables and also functions can be declared as “extern”.

-> Ex: extern int x;
-> extern void disp()
{
    Printf(“extern function”);
}

➔ Register variables: during program execution, the values need to be transferred from main memory into CPU registers (processor cache), to undergo process. This data transfer between RAM and registers consumes much time. To reduce this time consumption, the variables

can be directly declared with in CPU registers with the keyword "register".

Though the time consumption of data transfer is reduced with register variables, the performance of the processor is also reduced. While programming, generally the variables that need to change their values frequently for number of times are declared as register variables. If CPU registers are not free, then the OS declares them as normal variables with in main memory.

In programming, as the loop controlling variables change their values frequently for number of times, only these are declared as register variables and as generally only integer type variables are used for loop control, in C, only int data type variables can be declared as register variables.

-> ex: register int i;

➔ Static variables: when a variable is declared as static within a function, the variable preserves its last update value even when it goes out of scope. It preserves its value and that can be used for the subsequent calls of the same function. Generally to reduce the usage of global variables, the programmers prefer using static variables.

-> ex : static int x;

Q. Explain scope of variables and storage classes?
