MODULE – 3

Functions: Introduction using functions, Function definition, function declaration, function call, return statement, passing parameters to functions, scope of variables, storage classes, recursive functions.

Arrays: Declaration of arrays, accessing the elements of an array, storing values in arrays, Operations on arrays, passing arrays to functions, two dimensional arrays, operations on two-dimensional arrays, two dimensional arrays to functions, multidimensional arrays, applications of arrays.

Textbook: Chapter 11.1-11.10, 12.1-12.10, 12.12

Chapter 1 : Functions

1.1   Introduction using functions

C enables programmers to **break up a program into segments commonly known as functions**, each of which can be written more or less independently of the others.

C functions can be classified into two categories, namely **library functions and user defined functions.**

Library Functions: The functions that are not required to be written by us. eg: printf(), scanf(), sqrt().

User defined Functions: The functions that are to be developed by the user at the time of writing a program. eg: main(). These functions can later become a part of C library.

Differences between Library function and user defined function

| Library function	User defined function
Predefined, it is called when required	Defined by user whenever it is required with the three elements function call, function definition and function declaration
Name of the function is a key word	Name of the function is given by user
Ex-sqrt(), cos(), sin()	Ex- square_root

1.2   Why are functions needed?

- Dividing a program into separate well-defined functions facilitates *each function to be written and tested separately*. Below figure shows that the main() function calls other functions for dividing the entire code into smaller sections (or functions). This approach is referred to as the **top-down approach**.

- **Understanding, coding, and testing** multiple separate functions is far easier than doing it for one big function.

- If a big program has to be developed without the use of any function other than main() function, then **there will be countless lines** in the main() function and maintaining this program will be very difficult.

- All the libraries in C contain a set of functions that the programmers are free to use in their programs. This **speeds up program development**, by allowing the programmer to concentrate only on the code that he has to write.

- When a big program is broken into comparatively smaller functions, then different programmers working on that project can **divide the workload** by writing different functions.

- Like C libraries, programmers can also **write their functions and use them at different points** in the main program or in any other program that needs its functionalities.

Functions calling another functions
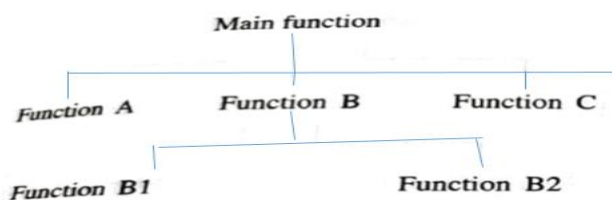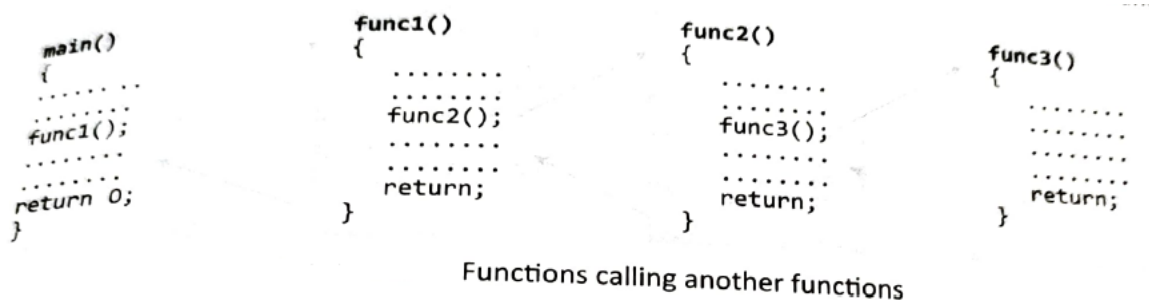
Top-down approach of solving a problem

Function func1() called twice from main()
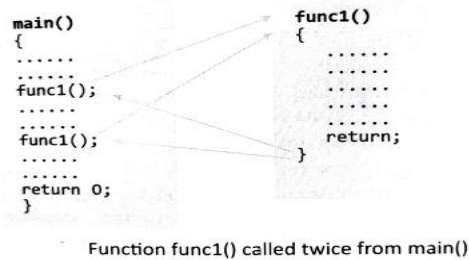
1.3 Using functions

While using functions we will be using the following terminologies:

- A function that uses another function g is known as the **calling function** and g is known as the **called function**.

- The inputs that a function takes are known as **arguments/ parameters**.

- When a called function returns some result back to the calling function, it is said to **return** that result.

- The parameter list in the called function are called **formal parameters**.

- The arguments using in function call are called **actual arguments**.

1.4 Elements of User-Defined Functions

In order to make use of a user-defined function, we need to establish three elements that are related to functions.

- Function definition
- Function call
- Function declaration / function prototype

The *function definition* is an written to implement the requirements of the function.

In order to use this function we need to invoke it at a required place in the program. This is known as *function call*. The program (or a function that calls the function is referred to as the calling program or calling function.)

The calling program should declare any function (like declaration of a variable) that is used to be used later in the program. This is known as the *function declaration or function prototype*.

1.4.1 Definition of Functions

A function definition, also known as function implementation shall include the following elements:

1) Function name
2) Function type / return type
3) List of parameters
4) Local variable declarations
5) Function statements
6) Return statement

All the six elements are grouped into two parts, namely,

1) Function header (first three elements)
2) Function body (second three elements)

General format of function definition

```
return_type function_name (parameter list)
{
        local variable declaration;
        executable statement1;
        executable statement2;
        …
        …
        return statement;
}
```

The first line return_type function_name(parameter list) is known as the function header and the statements within the opening and closing braces constitute the function body.

Function Header: The function header consists of three parts: the return data type, the function name and the formal parameter list.

name and type: The function type specifies the type of value(like float or double) that the function is expected to return to the program calling the function. If the return type is not explicitly specified C will assume that it is an integer type. If the function is not return anything,

we need to specify the return type as void. The value returned is the output produced by the function.

The function name is any valid C identifier and therefore must follow the same rules of formation as other variable names in C. The name should be appropriate to the task performed by the function. We should not duplicate library routine names or OS commands.

Formal parameter list: The parameter list declares the variables that will receive the data sent by the calling program. They are often referred to as formal parameters. The parameters are also known as arguments.

The parameter list contains declaration of variables separated by commas and surrounded by parenthesis.

Ex:

      float quadratic(int a,int b,int c)

      {…}

      int sum(int a,int b)

      {…}

Declaration of variables cannot be combined. (i.e) int sum(int a,b) is illegal

If a function does not receive values from a calling program (no formal parameters), we should use the keyword void between the parentheses.

      void printline(void)

      {

          …

      }

Can also be written like void printline().

Function body: The function body contains the declarations and statements necessary for performing the required task. The body enclosed in braces contains 3 parts in the order given below:

1) Local declarations that specify the variables needed by the function.
2) Function statements that perform the task of the function.
3) A return statement that returns the value evaluated by the function. [If the function does not return any value, we can omit the return statement but we should specify the return type as void.

Eg -1 int mul(int x,int y)

```
    {
            int p;          /*local variable*/
            p=x*y;          /*computes product*/
            return p;               /*returns the result*/
    }
```

Eg-2   void sum(int a,int b)

```
    {
            printf("sum=%d",a+b);                   /*  no local variables */
            return;                 /* optional */
    }
```

Return Values and their Types

While it is possible to pass to the called function any number of values, the called function can return one value per call, at the most.

- *return;* - returns nothing; When the return statement is encountered, the control is immediately passed back to the calling function. Eg –

    if(some_condition)

    return;

- *return(expression);* - returns the value of the expression to the calling function. Eg-

    ```
    int mul(intx,int y)
    {
            int p;
            p= x*y;
            return(p);
    }
    ```

returns p which is a product of x*y. We can also write *return(x*y)* by combining the statements.

1.4.2   Function calls (invocation of function)

A function can be called by simply using the function name followed by a list of actual parameters(arguments), if any, enclosed in parentheses.

Example:

```
int main()
{
        int y;
        y=mul(10,5);            /*function call*/
        printf("%d\n",y);
        return 0;
}
```

When the compiler encounters a function call, the control is transferred to the function mul(). This function is then executed line by line and a value is returned when a return statement is encountered. This value is assigned to y.
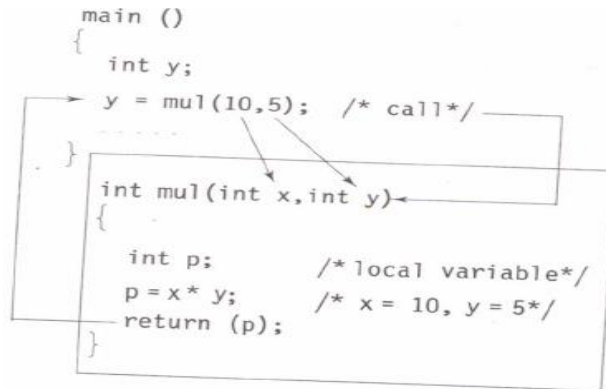
The function call sends two integer values 10 and 5 to the function.

        int mul(int x,int y) which are assigned to x and y respectively. The function computes the product x and y, assigns the result to the local variable p, and then returns the value 25 to the main where it is assigned to y again.

1.4.3 Function Declaration/function prototype

Like variables, all functions in a C program must be declared, before they are invoked. It is also called as function prototype. It consists of four parts.

1) Function type(return type)
2) Function name
3) Parameter list
4) Terminating semicolon

        SYNTAX: **return_type function_name (parameter list);**

It is similar to function header except the terminating semicolon.

Eg- int mul(int m,int n); // function prototype

int mul(int,int);

mul(int,int);

When a function does not take any parameters and does not return any value, its prototypes is written as: **void display(void);**

1. *Add two numbers*

```
#include<stdio.h>
int add(int a,int b);
void main()
{
        int a,b,c;
        printf("Enter the value of a and b:");
        scanf("%d%d",&a,&b);
        c=add(a,b);
        printf("\n Addition of %d and %d is %d",a,b,c);
}
int add(inta,int b)
{
        int c;
        c=a+b;
        return c;
}
```

Output

Enter the values of a and b:4 5

Addition of 4 and 5 is 9

A prototype declaration may be placed in two places in a program.

1) Above all the functions (including main)

2) Inside a function definition.

When we place the declaration above all the functions (in the global declaration section), the prototype is referred as global prototype. Such declarations are available for all the functions in the program.

When we place it in a function definition (in the local declaration section), the prototype is called a local prototype. Such declarations are used by the functions containing them.

Prototype declarations are compulsory if the function is invoked before the function definition is placed. If the function definition is placed before it is invoked, then the prototype declarations are optional.

Location of function definition

1. Function definition can be written after main(). In this function declaration is compulsory.

   ```
   #include<stdio.h>
   int add(int a,int b);
   void main()
   {
           …
           …
   }
   int add(int a,int b)
   {
           …
           …
   }
   ```

2. Function definition can be written before main(). In this function declaration is not compulsory.

   ```
   #include<stdio.h>
   int add(int a,int b)
   {
           …
           …
   }
   ```

```
void main()
{
        …
        …
}
```

1.5 Parameter Passing techniques

The technique used to pass data from one function to another is known as parameter passing. Parameter passing can be done in two ways:

- Pass by value (also known as call by value).
- Pass by Pointers/Reference (also known as call by pointers).

1.5.1 Pass by value

In *pass by value,* values of actual parameters are copied to the variables in the parameter list of the called function. **Any changes in the formal arguments will not be reflected in the actual arguments.**

Example : To swap two numbers

```
void swap(int a,int b);
void main()
{
        int a,b;
        printf("enter two numbers");
        scanf("%d%d",&a,&b);
        printf("before swapping");
        printf("a=%d,b=%d",a,b);
        swap(a,b);
        printf("after swapping");
        printf("a=%d,b=%d",a,b);
}
void swap(int p,int q)
{
        int t;
```

```
        t=p;
        p=q;
        q=t;
        printf("in function");
        printf("a=%d,b=%d",a,b);


}
```

Output:

enter two numbers 5 4

before swapping

a=5,b=4

in function

a=4,b=5

after swapping

a=5,b=4

Advantages

The passing arguments can be variables, constants or expression

Disadvantages

Additional storage space

Lot of time to copy

Performance penalty if function is called many times

1.5.1 Pass by pointers

In *pass by pointers* (also known as pass by address), the memory addresses of the variables is passed rather than the copies of values being sent to the called function. **Any changes in the formal arguments will be reflected in the actual arguments.**

Example : To swap two numbers

```
        void swap(int *p,int *q);
        void main()
        {
                int a,b;
                printf("enter two numbers");
```

```
        scanf("%d%d",&a,&b);
        printf("before swapping");
        printf("a=%d,b=%d",a,b);
        swap(&a,&b);
        printf("after swapping");
        printf("a=%d,b=%d",a,b);
}
void swap(int *p,int *q)
{
        int t;
        t=*p;
        *p=*q;
        *q=t;
}
```

Output:

enter two numbers 5 4

before swapping

a=5,b=4

after swapping

a=4,b=5

Advantages

Time and space efficiency

Change is reflected in the calling function

Used to return multiple values (pass those as arguments by reference)

Disadvantages

Difficult to tell whether the argument is meant for input, output or both.

Call by Value	*Call by Reference*
Copy of values are passed from calling to called function	*Addresses are passed from calling to called function*
Any changes in the formal arguments will not be reflected in the actual	*Any changes in the formal arguments will be reflected in the actual*

arguments.   *arguments.*

Execution of the program is slow   *Execution of the program is fast*

Program   *Program*

1.6. Category of Functions

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

Category 1: Functions with no arguments and no return values

Category 2: Functions with arguments and no return values

Category 3: Functions with arguments and one return value

Category 4: Functions with no arguments but a return value

Category 5: Functions that return multiple values

1.6.1. No Arguments and no return values

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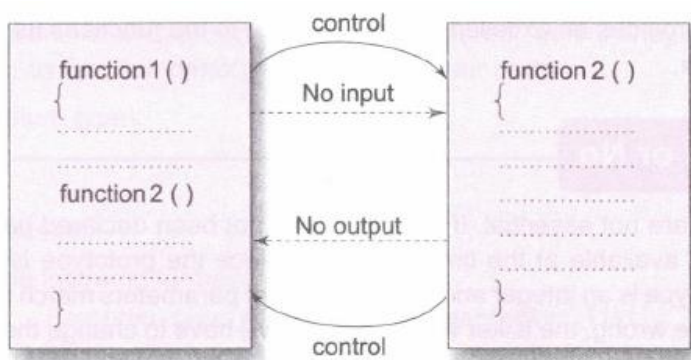
No data communication between functions

```
void add();
void main()
{
     add();
}
void add()
{
     int a,b,c;
```

printf("enter two numbers");

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

c=a+b;

printf("result of addition is %d",c);

}

1.6.2. *Arguments but no return values*

There is data communication between the calling function and the called function with arguments but no return value.
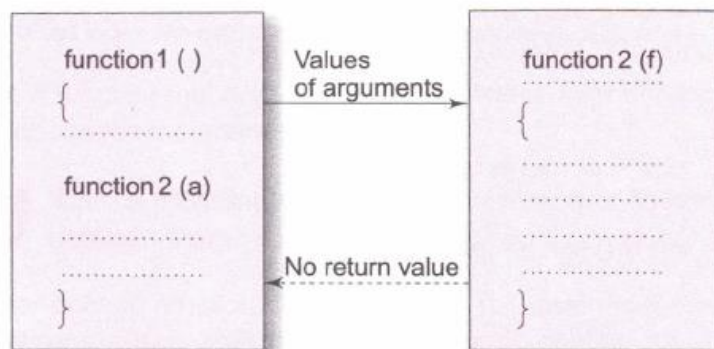
One-way data communication

The actual and formal arguments should match in number, type and order.
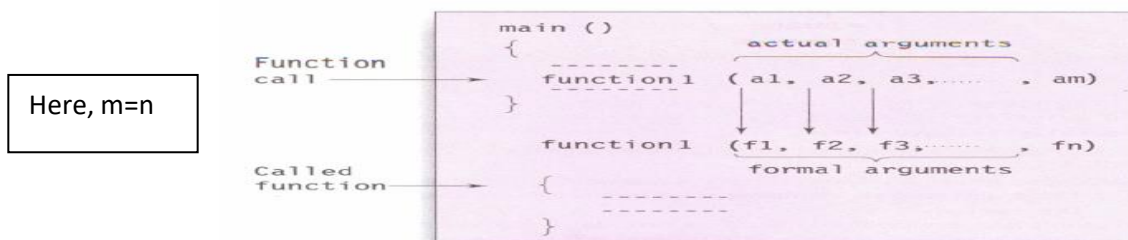
Arguments matching between the function call and the called function Example 2

void add(int a,int b);

void main()

{

int a,b;

printf("enter two numbers");

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

add(a,b);

}

```
void add(int a,int b)
{
        int c;
        c=a+b;
        printf("result of addition is %d",c);
}
```

1.6.3. Arguments with one return value

To assure a high degree of portability between programs, a function should generally be coded without involving any I/O operations.

There is data communication between calling and called function and vice versa. Such functions will have two-way data communication.

Two-way data communication between functions

```
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("result of addition is %d",c);
}
int add(int a,int b);
{
        int c;
        c=a+b;
        return c;
}
```
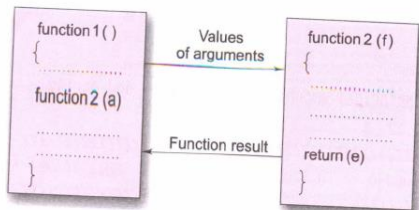
1.6.4. No arguments and but returns a value

There is no data communication between calling function to called function but there is communication between called function to calling function.
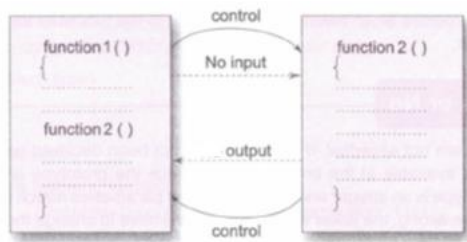

```
int add();
void main()
{
    int c;
    c=add();
    printf("result of addition is %d",c);
}
int add()
{
    int a,b;
    printf("enter two numbers");
    scanf("%d%d",&a,&b);
    c=a+b;
    return c;
}
```

1.6.5 Functions that returns multiple values

The return statement can return only one value. Suppose we want more information from function, we can achieve this using the arguments using call by pointers/reference or using arrays.

Example : To swap two numbers using pointers

```
void swap(int *p,int *q);
void main()
```

```
        {
                int a,b;
                printf("enter two numbers");
                scanf("%d%d",&a,&b);
                printf("before swapping");
                printf("a=%d,b=%d",a,b);
                swap(&a,&b);
                printf("after swapping");
                printf("a=%d,b=%d",a,b);
        }
        void swap(int *p,int *q)
        {
                int t;
                t=*p;
                *p=*q;
                *q=t;
        }
```

> **Output:**
>
> enter two numbers 5 4
>
> before swapping
>
> a=5,b=4
>
> after swapping
>
> a=4,b=5

Differences between actual and formal arguments (Arguments are also called as parameters)

Actual arguments	*Formal arguments*
Used in calling function	*Used in called function*
May be variable names, expressions or constants	*Must be variable name*
add(5,10)	*int add(int a, int b)*
add(a,b)	
add(a+3,b+2)	

<u>*Note:*</u> *The actual and formal arguments should match in number, type and order.*

1.7 SCOPE, VISIBILITY AND LIFETIME OF VARIABLES

Visibility - refers to the accessibility of a variable from the memory.

Longevity or Lifetime refers to the period during which a variable retains a given value during

the execution of a program (alive).

Scope of a variable is defined as the region or boundary of the program in which the variable is visible.

Global scope: Variables defined outside the block have global scope.

Local scope: Variable defined the block have local scope.

Global variable: defined before all functions in the global area of the program.

Memory is allocated for these variables only once and all the allocated memory is initialized to 0. These variables can be accessed by any function and are active throughout the program.

Local variable: defined within the function.

Differences between local and global variable

LOCAL VARIABLE	GLOBAL VARIABLE
Declared in the local scope	Declared in the global scope
Declared within the function	Declared outside all the functions
Also called Internal Variables	Also called External Variables
When declared in the local scope, garbage values are present in those variables	When declared in the global scope, automatically initializes to zero

Example:

```
#include<stdio.h>
int c;                  // Global Variable
void add(int a,int b)
{
    c=a+b;
}
void main()
{
    int a=10,b=20;      // Local variable
    add(a,b);
    printf("a=%d,b=%d,c=%d",a,b,c);
}
```

1.8 STORAGE CLASSES IN C

Every variable in C has two properties: type and storage class. Type refers to data type of

variable. Storage class refers to how long it stays in existence.

There are four types of storage classes:

1) automatic
2) external
3) static
4) register

Automatic: Variables declared inside the function body are automatic by default. These variables are known as Local variables. These variables does not have scope outside the function. The keyword auto is used but it is used very rarely.

<u>**Example:** **auto int a; or int a;**</u>

External: External variables are accessed by any function. They are known as global variables as well. Variables declared in the global scope i.e outside the functions. In a case of a large program, containing more than one file, keyword extern is used in file 2 to indicate that the variable specified is global variable and declared in another file.

<u>**Example:** **extern int a;**</u>

Register: Register variables are automatic variables (Local variables) which gets stored in a register rather than a RAM (primary memory) for faster access.

<u>**Example:** **register int a;**</u>

Static: Static variables are variables, which gets initialized only for the first call, after which the updated value is used.

Example:

```
int increment()
{
        static int count=0;
        count++;
        return count;
}
void main()
{
        int val1,val2;
        val1=increment();
```

val2=increment();

printf("\n value1=%d, value2=%d",val1,val2);

        }

Output: value1=1, value2=2

1.9. Recursion

When a called function calls another function a process of 'chaining' occurs. *Recursion* is a special case of this process, where a function calls itself.

Two important elements of recursion is **general case and base case.**

General case (Recursive case) : This statement reduces the size of the problem(i.e) subset of the same problem.

Base case : This statement terminates the problem.

A program should have at least one base case and one general case.

Let us see how recursion works in finding factorial of a number.

Recursive definition of factorial(n)

factorial(n) = 1, if n=0   **base case**

factorial(n) = n*factorial(n-1), otherwise   **general case**

5!=5*4!

  =5*4*3!

  =5*4*3*2!

  =5*4*3*2*1!

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

  =5*4*3*2*1*1

  =120

Write a program to find factorial of positive integers using recursion.

```
#include<stdio.h>
int factorial(int n);
void main()
{
        int n,res;
        printf("\n enter the value of n");
        scanf("%d",&n);
```

```
        res=factorial(n);

        printf("factorial of %d is %d",n,res);

}

int factorial(int n)

{

        if(n==0)

                return 1;

        else

                return(n*factorial(n-1));

}
```

<u>Output</u>

factorial of 5 is 120

Tracing

factorial(3)=3*factorial(2)

        =3*2*factorial(1)

        =3*2*1*factorial(0)

        =3*2*1*1

        =6

Write a program to print Fibonacci series up to n using recursion.

Recursive definition of fibonacci(n)

fibonacci(n) = 0, if n=0   **base case**

fibonacci(n) = 1, if n=1   **base case**

fibonacci(n) = fibonacci(n-1)+fibonacci(n-2), otherwise.   **General case**

Program

```
#include<stdio.h>

int fib(int n);

void main()

{

        int n,res,i;

        printf("\n enter the value of n");
```

```
        scanf("%d",&n);
        for(i=0;i<n;i++)
        {
                res=fib(i);
                printf("%d\t",res);
        }
}
int fib(int n)
{
        if(n==0)
                return 0;
        else if(n==1)
                return 1;
        else
                return(fib(n-1)+fib(n-2));
}
```

Output:

Enter the value of n 5

0 1 1 2 3

Tracing when n=5

fib(0)=0

fib(1)=1

fib(2)=fib(1)+fib(0)=1+0=1

fib(3)=fib(2)+fib(1)=fib(1)+fib(0)+fib(1)=1+0+1=2

fib(4)=fib(3)+fib(2)=fib(2)+fib(1)+fib(1)+fib(0)=fib(1)+fib(0)+fib(1)+fib(1)+fib(0)=1+0+1+1+0

=3

Write a program to find the GCD of 2 numbers using recursion.

Recursive definition of gcd(m,n)

gcd(m,n)=n, if m%n=0   **base case**

gcd(m,n) = gcd(n,m%n), Otherwise **General case**

```
#include<stdio.h>
int gcd(int m,int n);
int main()
{
        int num1,num2,res;
        printf("enter two numbers");
        scanf("%d%d",&num1,&num2);
        res=gcd(num1,num2);
        printf("GCD is %d",res);
        return 0;
}
int gcd(int m,int n)
{
        int rem;
        rem=m%n;
        if(rem==0)
                return n;
        else
                return(gcd(n,rem));
}
```

m	n	rem
20	30	20
30	20	10
20	10	0
returns 10(n)		

1.10 Nesting of Functions

C permits nesting of functions freely, main can call function1, which calls function2, which calls function3, and so on.

Write a program to find the binomial coefficient using recursion.

```
#include<stdio.h>
int factorial(int n);
int ncr(int n,int r);
void main()
{
        int n,r,res;
        printf("Enter the value of n and r(r<=n):");
```

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

        res=ncr(n,r);

        printf("binomial coefficient is %d",res);

}

int factorial(int n)

{

        if(n==0)

                return 1;

        else

                return(n*factorial(n-1));

}

int ncr(int n,int r)

{

        return(factorial(n)/(factorial(r)*factorial(n-r)));

}
```

CHAPTER 2 - ARRAYS

2.1 Introduction to Arrays

The fundamental data types, namely char, int, float, and double are constrained by the fact that a variable of these types can store only one value at any given time. Therefore, they can be used only to handle limited amounts of data.

In many applications, however, we need to handle a large volume of data in terms of reading, processing and printing. To process such large amounts of data, we need a powerful data structure that would facilitate efficient storing, accessing and manipulation of data items.

C supports a derived data type known as **array** that can be used for such applications.

Disadvantages of arrays:

   i)   Size should be declared in advance.
   ii)   If the number of elements is less than the size, lot of space is wasted.

Array:

An array is a derived data type used to store a collection of similar (same type or homogenous) data items, stored contiguously in memory under a single name.

Instead of declaring individual variables, such as number0, number1, ..., and number99, we declare one array variable such as number and use number[0], number[1], and ..., number[99] to represent individual variables. A specific element in an array is accessed by an index. An index is an integer positive number that starts with 0.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.
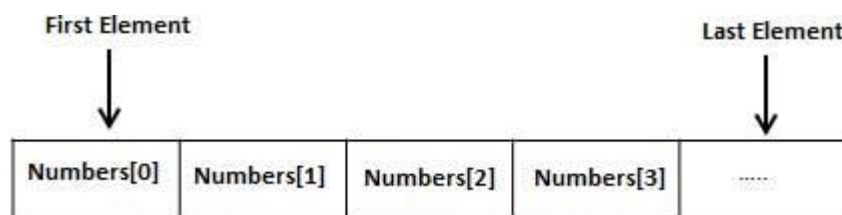
Array name refers to the starting address of the array. Arrays are also called as subscripted value.

Some examples where the concepts of an array can be used:
- List of employees in an organization.
- List of customers and their telephone numbers.

use arrays to represent not only simple lists of values but also tables of data in two or three or more dimensions.

Types of arrays

- One dimensional arrays (1-D)
- Two dimensional arrays(2-D)
- Multi-dimensional arrays

ONE DIMENSIONAL ARRAYS

A linear list of data items of same type which are stored contiguously in memory is called one dimensional array (1-D array).

A List of items can be given one variable name using only one subscript and such a variable is called a single subscripted variable or a one dimensional array.

2.2 DECLARATION OF ONE DIMENSIONAL ARRAYS (DECLARATION OF ARRAYS)

The general form of array declaration is : **type array_name[size];**

Declaration of array specifies 3 things

1) The **type** specifies the type of element that will be contained in the array such as int, float, double or char.

2) **array_name** : to identify the array.

3) **size** indicates the maximum number of elements that can be stored inside the array.

For example,

1) float height[50];

 Declares the height to be an array containing 50 real elements. Any subscripts 0 to 49 are valid.

2) int a[10];

 Declares **a** as an array to contain a maximum of 10 integer constants. Remember:

 • Any reference to the arrays outside the declared limits would not necessarily cause an error. Rather, it might result in unpredictable program results.

 • The size should be an integer constant.

3) char name[10];

 Declares the name as a character array (string) variable that can hold a maximum of 10 characters.

int   number[4];

The computer reserves four storage locations as shown below:

number[0]   First element

number[1]   Second element

number[2]   Third element

number[3]   Fourth element

The elements of array are referenced by an index (also known as subscript). Subscript is an ordinal number which is used to identify an element of the array.

Why index starts at 0: An index refers to a relative position of an element from the first element. So , for the fourth element the index is 3 because there are 3 elements before it. For the first element, no element is present before it. So, the index is 0.

2.3 ACCESSING THE ELEMENTS OF AN ARRAY

For accessing an individual element of the array, the array index must be used. For example, to access the fourth element of the array, we must write arr[3].

To process all the elements of the array, we will use for loop:

```
int i,marks[10];
for(i=0;i<10;i++)
{
        marks[i]=-1;
}
```

Calculating the Address of Array Elements:

For example, if we want to represent a set of four numbers, say (10,20,30 ,40) ,by an array variable number, then we may declare the variable number as follows:

int   number[4];

And the computer reserves four storage locations as shown below:

number[0]

number[1]

number[2]

number[3]

This would cause the array number to store the values as shown below:

number[0]	10
number[1]	20
number[2]	30
number[3]	40

Total allocated memory=size*sizeof(datatype)

=4*sizeof(int) **[sizeof(int)=2 bytes in a 16-bit machine]**

=4*2=8 bytes

If the starting address is 8000.

Address of an element=starting address+(index*sizeof(datatype))

Address of number[2]=8000+2*sizeof(int)

> **=8000+2*2**

> **=8004**

Memory layout

number[0]**8000**	10
number[1]**8002**	20
number[2]**8004**	30
number[3]**8006**	40

2.4 STORING VALUES IN ARRAYS

After an array is declared, its elements must be initialized. Otherwise, they will contain "garbage".

An array can be initialized at either of the following stages:

- At compile time
- At run time

Compile Time Initialization

we can initialize the elements of arrays in the same way as the ordinary variables when they are declared.

The general form of initialization of arrays is:

1) **Full initailization**

    type array-name[size] = {list of values};

    The values in the list are separated by commas. For example, the statement

    int a[3] = { 1,2,3 }; will assign a[0]=1,a[1]=2,a[2]=3 with size 3.

2) **Partial initailization**

Compile time initialization may be partial. That is, the number of initializers may be less than declared size. In such cases. the remaining elements are inilialized to zero, if the array type is numeric and NULL if the type is char.

 For example,

 int number [5] = {10, 20}; will initialize the first two elements to 10 and 20 respectively, and the remaining elements to 0.

Similar the declaration. char city [5] = {'B'}; will initialize the first element to 'B' and the remaining four to NULL.

3) **Without size**

The size may be omitted. In such cases, the compiler allocates enough space for all initialize elements.

For example, the statement int a[ ] = {1,1,1,1}; will declare the **a** array to contain four elements with initial values 1. This approach works fine as long as we initialize every element in the array.

4) **With character constants**

Character arrays may be initialized in a similar manner. Thus, the statement

char name [ ] = { 'J' , 'o ' , ' h ', ' n ', '\0'}; declares the name to be an array of five characters, initialized with the string "John" ending with the null character.

5) **With string**

we can assign the string literal directly as under: char name [] = "John"; In this size is five. Here null character need not be explicitly specified.

If we have more initializers than the declared size, the compiler will produce an error. That is, the statement int number [3] = {10, 20, 30, 40}; will not work. It is illegal in C.

Run Time Initialization

An array can be explicitly initialized at run time. This approach is usually applied for initializing large arrays.

int a [3] ;

scanf("%d%d%d", &a[0],&a[1].,&a[2]);

(or)

int i,a [3] ;

for(i=0;i<3;i++)

        scanf("%d", &a[i]);

will initialize array elements with the values entered through the keyboard.

2.5 OPERATIONS ON ARRAYS

The following are the operations done on arrays

1) Traversing an array

2) Inserting an element in an array

3) Deleting an element in any array

4) Merging two arrays

5) Searching an element in an array

6) Sorting an array in ascending or descending order

1) *Traversing an Array*

Traversing an array means accessing each and every element of the array for a specific purpose.

1. **Input and Output an array elements of size n**

   ```c
   void main()
   {
           int n,i,a[10];
           printf("enter size");
           scanf("%d",&n);
           printf("enter the elements");
           for(i=0;i<n;i++)
                   scanf("%d",&a[i]);
           printf("The elements are");
           for(i=0;i<n;i++)
                   printf("%d\t",a[i]);
   }
   ```

2. **Biggest element in an array**

   ```c
   void main()
   {
           int n,i,a[10],big;
           printf("enter size");
           scanf("%d",&n);
           printf("enter the elements");
           for(i=0;i<n;i++)
                   scanf("%d",&a[i]);
           big=a[0];
           for(i=0;i<n;i++)
           {
                   if(a[i]>big)
   ```

```
                big=a[i];
        }
        printf("The biggest element is %d ",big);
}
```

3. **Sum of array elements**

```
void main()
{
        int n,i,a[10],sum=0;
        printf("enter size");
        scanf("%d",&n);
        printf("enter the elements");
        for(i=0;i<n;i++)
                scanf("%d",&a[i]);
        for(i=0;i<n;i++)
        {
                sum=sum+a[i];
        }
        printf("The sum  is %d ",sum);
}
```

4. **Sum of squares**

   Otherwise the question for the same program will be given as under:

   Write a program using a single-subscripted variable to evaluate the following expressions:

   $$Total = \sum_{i=1}^{10} x_i^2$$

   the values of x1 ,x2, ... are read from the terminal.

```
void main()
{
        int n,i,a[10],sum=0;
```

```
        printf("enter size");
        scanf("%d",&n);
        printf("enter the elements");
        for(i=0;i<n;i++)
                scanf("%d",&a[i]);
        for(i=0;i<n;i++)
        {
                sum=sum+a[i]*a[i];
        }
        printf("The sum  is %d ",sum);
}
```

5. **Add two array elements**

```
void main()
{
        int n,i,a[10],b[10],c[10];
        printf("enter size");
        scanf("%d",&n);
        printf("enter the elements of a array");
        for(i=0;i<n;i++)
                scanf("%d",&a[i]);
        printf("enter the elements of b array");
        for(i=0;i<n;i++)
                scanf("%d",&b[i]);
        for(i=0;i<n;i++)
                c[i]=a[i]+b[i];
        printf("The resultant  array is");
        for(i=0;i<n;i++)
                printff("%d",c[i]);
}
```

2) *Inserting an Element in an Array*

It means adding a new data element to an already existing array. Inserting an element at the end of array of **size n** is easy. If a is the array, a[n]=num and n=n+1.

Inserting in the middle of the array, say pos is also possible, for which we have to move all elements that have index greater than pos one position towards right and create space for the new element. Then we increment n by 1.
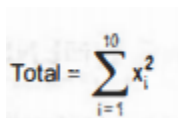
```
void main()
{
        int n,i,a[10],pos,num;
        printf("\n Enter size");
        scanf("%d",&n);
        printf("enter the elements");
        for(i=0;i<n;i++)
                scanf("%d",&a[i]);
        printf("\n Enter num to insert");
        scanf("%d",&num);
        printf("\n Enter position to insert");
        scanf("%d",&pos);
        for(i=n-1;i>=pos;i--)
                a[i+1]=a[i];
        a[pos]=num;
        n++;
        printf("The resultant  array is");
        for(i=0;i<n;i++)
                printff("\t %d",a[i]);
}
```

3) *Deleting an Element in an Array*

It means deleting a data element from an existing array. Deleting an element at the end of array of **size n** is easy. Simply decrement n as n=n-1.

Deletion in the middle of the array is also possible, for which we have to move all elements that have index greater than pos one position towards left. Then we decrement n by 1.

```
void main()
```

```
{
        int n,i,a[10],pos,num;
        printf("\n Enter size");
        scanf("%d",&n);
        printf("enter the elements");
        for(i=0;i<n;i++)
                scanf("%d",&a[i]);
        printf("\n Enter position to delete");
        scanf("%d",&pos);
        for(i=pos;i<n-1;i++)
                a[i]=a[i+1];
        n--;
        printf("The resultant  array is");
        for(i=0;i<n;i++)
                printff("\t %d",a[i]);
}
```

4) ***Merging Two Arrays***

Merging two arrays in a third array means first copying the contents of first array into the third array and then copying the contents of the second array into the third array. Hence, the merged array contains contents of the first array followed by the contents of the second array.

```
void main()
{
        int a1[10],a2[10],a3[20],n1,n2,i,m,index=0;
        printf("\n Enter size of array1");
        scanf("%d",&n1);
        printf("\n Enter the elements of first array");
        for(i=0;i<n1;i++)
                scanf("%d",&a1[i]);
        printf("\n Enter size of array2");
        scanf("%d",&n2);
        printf("\n Enter the elements of second array");
```

```
for(i=0;i<n2;i++)
        scanf("%d",&a1[i]);
for(i=0;i<n1;i++)
{
        a3[index]=a1[i];
        index++;
}
for(i=0;i<n2;i++)
{
        a3[index]=a2[i];
        index++;
}
m=n1+n2;
printf("The resultant  array is");
for(i=0;i<m;i++)
        printff("\t %d",a3[i]);
}
```

5) *Searching for a value in an array*

Searching means to find whether a particular value is present in the array or not.

Two methods

1) Linear search
2) Binary Search

Linear Search

Linear search is a very simply search algorithm were search is made over all items one by one.

Every item is checked and if a match is found then that particular index is returned otherwise the search continues till the end of the array.

Algorithm

Linear Search

 Step 1: read n and initialize found=0

Step 2: read array elements A

Step 3: read the key item to be searched

Step 4: for i=0 till n-1

if A[i] = key

then

        print element found at index i;

found=1;

end if

end for

step 5: if found=0

then

        print element not found

step 6: stop

WORKING

Key = 4

↓

10   12   3   5   11   4   2   23   43   29

        ↓

10   12   3   5   11   4   2   23   43   29

            .

                        .

                        .

                            ↓

10   12   3   5   11   4   2   23   43   29
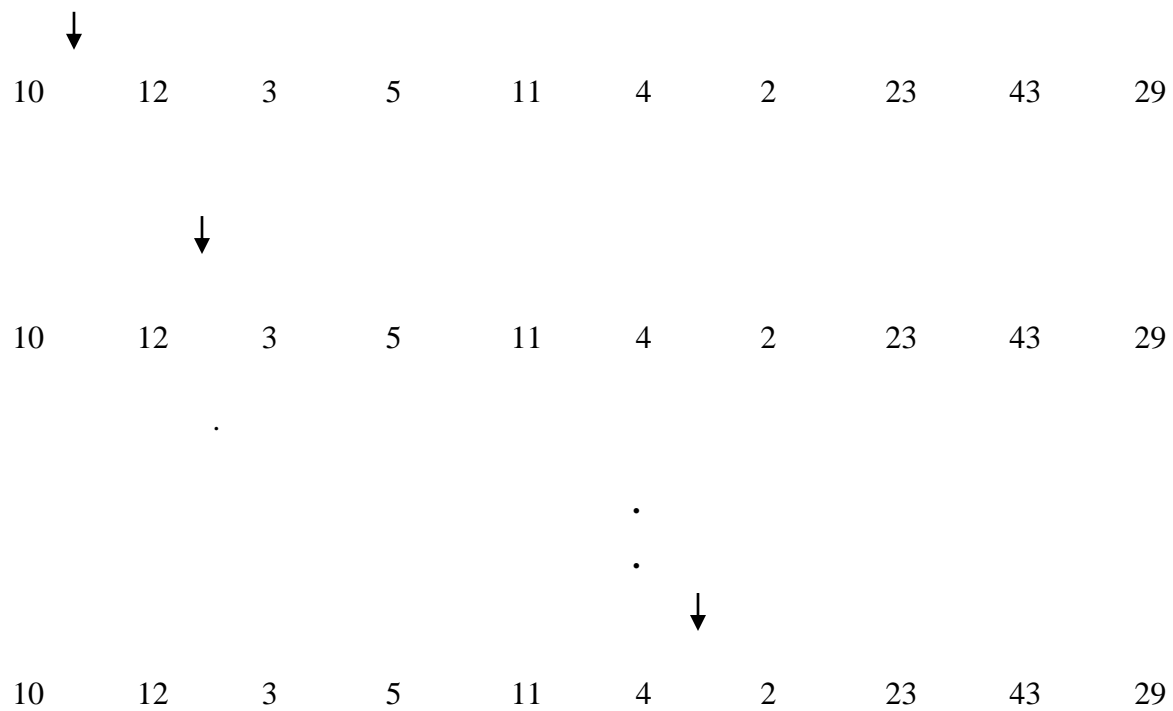
program

```c
#include<stdio.h>
int main( )

{
        int n,a[20],found=0,i,key;
        printf("enter the number of elements n");
        scanf("%d",&n);
        printf("enter the array elements");
        for(i=0;i<n;i++)
        {
                scanf("%d",&a[i]);
        }
        printf("enter the key to be searched\n");
        scanf("%d",&key);
        for(i=0;i<n;i++)
        {
                if(a[i]==key)
                {
                        printf("element found at index %d\n",i);
                        found=1;
                        break;
                }
        }
        if(found==0)
        {
                printf("element not found");
        }
}
```
Output:

1: enter the number of elements n 5

enter the array elements 12 18 2 4 21

enter the key to be searched

4

element found at index 3

2:

enter the number of elements n 5

enter the array elements 12 18 2 4 21

enter the key to be searched

100

element not found

Advantages

Linear search is simple and very easy to understand and implement

It does not require the data in the array to be stored in any particular order.

DISADVANTAGES

It is inefficient

For example, if we have 1000 elements and the key element is found at last index then it looks 1000 elements in order to find the last element.

Binary Search

Binary search looks for a particular item by comparing the middle most item of the sorted array. If a match occurs, then the index of item is returned.

If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item.

Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the sub-array reduces to zero.

Algorithm

Binary Search

Step 1: read n and set found=0

Step 2: read array elements A in ascending Order

Step 3: read the key item to be searched

Step 4: set low=0

      set high=n-1

Step 5: while key not found and low<=high

  do

       mid=(low+high)/2

       if A[mid] = key

       set found=1

       if a[mid] < key

       set low = mid +1

       if a[mid] > key

       set high = mid -1

end while

 step 6: if found =1

then

       print element found at position mid+1

else

print element not found

step 7: stop.

working

For a binary search to work, it is mandatory for the target array to be sorted.

The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.

First, we shall determine half of the array by using this formula

Mid=(low+high)/2

Here it is, (0 + 9) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.
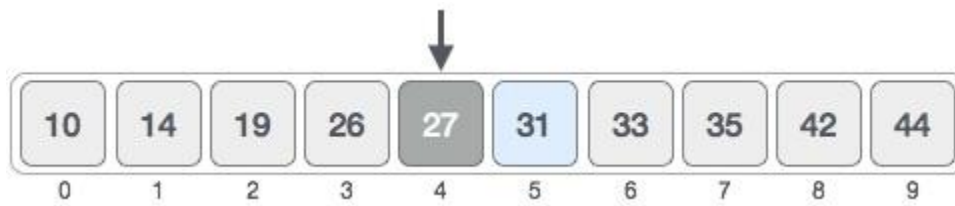
Now we compare the value stored at location 4, with the value being searched, i.e. 31.

We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.

We change our low to mid + 1 and find the new mid value again.

Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.

The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location.

Hence, we calculate the mid again. This time it is 5.
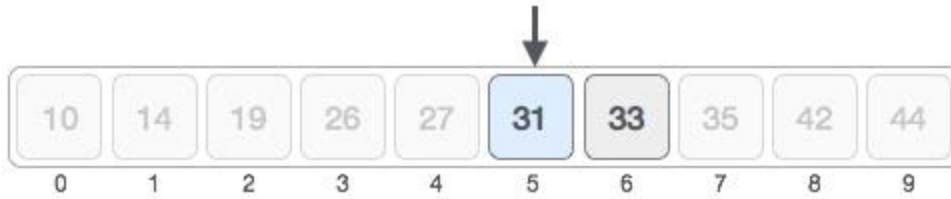
We compare the value stored at location 5 with our target value. We find that it is a match.

We conclude that the target value 31 is stored at location 5.

Program

```c
#include<stdio.h>
void main()
{
        int n,a[20],found=0,i,key,low,high,mid;
        printf("enter the number of elements n");
        scanf("%d",&n);
        printf("enter the array elements in ascending order");
        for(i=0;i<n;i++)
        {
                scanf("%d",&a[i]);
        }
        printf("enter the key to be serached\n");
        scanf("%d",&key);
        low=0;
        high=n-1;
        while(found!=1 && low<=high)
        {
                mid=(low+high)/2;
```

```
                if(a[mid]==key)
                        found=1;
                else if(a[mid]<key)
                        low=low+1;
                else
                        high=high+1;
        }
        if(found==1)
        {
                printf("element found at position %d\n",mid+1);
        }
        else
        {
                printf("element not found");
        }
}
```

Output:

enter the number of elements n10

enter the array elements in ascending order10 20 30 40 50 60 70 80 90 100

enter the key to be searched

70

element found at position 7

enter the number of elements n10

enter the array elements in ascending order10 20 30 40 50 60 70 80 90 100

enter the key to be searched

76

Element not found

Advantages

Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.

Disadvantages

Binary search works only for sorted list.

6) *Sorting an array in ascending or descending order*

Arranging the elements of an array either in ascending or in descending order is called sorting an array.

Bubble sort

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order.

Working

We take an unsorted array for our example.

14	33	27	35	10

Bubble sort starts with very first two elements, comparing them to check which one is greater.

14	33	27	35	10

In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.

14	33	27	35	10

We find that 27 is smaller than 33 and these two values must be swapped.

14	33	27	35	10
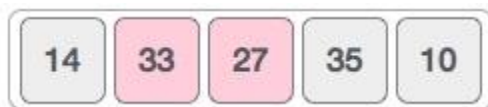
The new array should look like this −

| 14 | 27 | 33 | 35 | 10 |

Next we compare 33 and 35. We find that both are in already sorted positions.

| 14 | 27 | 33 | 35 | 10 |

Then we move to the next two values, 35 and 10.

| 14 | 27 | 33 | 35 | 10 |

We know then that 10 is smaller 35. Hence they are not sorted.

| 14 | 27 | 33 | 35 | 10 |

We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this −

| 14 | 27 | 33 | 10 | 35 |

To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this −

| 14 | 27 | 10 | 33 | 35 |

Notice that after each iteration, at least one value moves at the end.

| 14 | 10 | 27 | 33 | 35 |
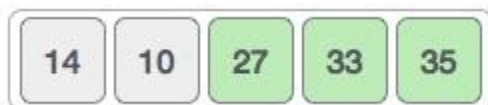
And when there's no swap required, bubble sorts learns that an array is completely sorted.
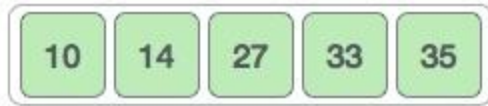
Algorithm

Step 1: read n

Step 2: read array elements A in unsorted order

Step 3: print unsorted array elements A

Step 4: for i=0 till n-1

do

      for j=0 till n-i-1

do

if A[j] > A[j+1]

      swap (A[j] , A[j+1]

end if

end for

end for

step 6: print the sorted array elements A

step 7: stop

program

```
#include<stdio.h>
int main()
{
        int n,a[20],i,j, temp;
        printf("enter the number of elements n");
        scanf("%d",&n);
        printf("enter the array elements in any order");
        for(i=0;i<n;i++)
        {
                scanf("%d",&a[i]);
        }
```

```
        printf("the unsorted elements are \n");

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

        {

                printf ("%d\t",a[i]);

        }

        for(i=0;i<n-1;i++)

        {

                for(j=0;j<n-i-1;j++)

                {

                        if(a[j]>a[j+1])

                        {

                                temp=a[j];

                                a[j]=a[j+1];

                                a[j+1]=temp;

                        }


                }

        }

        printf("\n the sorted elements are\n");

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

        {

                printf ("%d\t",a[i]);

        }

}
```

output

enter the number of elements n 5

enter the array elements in any order 50 20 40 10 30

the unsorted elements are

50   20   40   10   30

 the sorted elements are

10   20   30   40   50

Advantage

Easy to understand

Easy to implement

In place, no external memory is needed

Disadvantage

It does more element assignments than its counter part.

2.6 PASSING ARRAYS TO FUNCTIONS

Like the values of simple variables, it is also possible to pass the values of an array to a function. To pass a one-dimensional an array to a called function, it is sufficient to list the name of the array, *without any subscripts, and the size of the array as arguments.*

In C, the name of the array represents the address of its first element. By passing the array name, we are, in fact, passing the address of the array to the called function. The array in the called function now refers to the same array stored in the memory. Therefore, any changes in the array in the called function will be reflected in the original array. Passing addresses of parameters to the functions is referred to as *pass by address* (or pass by pointers).

However, this does not apply when an individual element is passed on as argument. Note that we cannot pass a whole array by value as we did in the case of ordinary variables.

1. The function must be called by passing only the name of the array.
2. In the function definition, the formal parameter must be an array type; the size of the array does not need to be specified (optional).
3. The function prototype must show that the argument is an array.

Write a program that uses a function to sort an array of integers using bubble sort.

A program to sort an array of integers using the function bubblesort(). Its output clearly shows that a function can change the values in an array passed as an argument.

Program

```
void bubblesort(int a[ ],int n);
void main()
{
```

```
        int i,a[10];
        printf("enter n");
        scanf("%d",&n);
        printf("enter elements");
        for(i=0;i<n;i++)
                scanf("%d",&a[i]);
        printf("before sorting\n");
        for(i = 0; i< n; i++)
                printf("%d\t ",a[i]);
        bubblesort (a,n);
        printf("after sorting\n");
        for(i = 0; i<n; i++)
                printf("%d\t",a[i]);
}
void bubblesort(int a[ ],int n)
 {
        int i, j, t;
        for(i=0;i<n-1;i++)
        {
                for(j=0;j<n-1-i;j++)
                {
                        if(a[j] >a[j+1])
                        {
                        t=a[j];
                        a[j]=a[j+1];
                        a[j+1] = t;
                        }
                }
        }
}
```

Output

enter n 5

enter elements 40 90 73 81 35

before sorting

40 90 73 81 35

after sorting

35 40 73 81 90

2.7 TWO DIMENSIONAL ARRAYS

It is a table of data items of similar datatype (same datatype) with two subscripts stored contiguously in memory under a single name.

Declaration of Two-dimensional arrays:

type array_name [row_size][column_size);

Two-dimensional arrays are stored in memory, as shown below.

The first index selects the row and the second index selects the column within that row.

Therefore, a two-dimensional m x n array is an array(m x n Matrix) that contains mx n data elements and each element is accessed using two subscripts, i and j where i<= m and j<= n.

Ex - int marks [3] [5];

A two-dimensional array called marks is declared that has m(3) rows and n(5) columns. The first element of the array is denoted by marks[0][0], the second element as marks[0][1], and so on. Here, marks [0][0] stores the marks obtained by the first student in the first subject, marks [1][0] stores the marks obtained by the second student in the first subject, and so on.

Rows \ Columns	Col 0	Col 1	Col 2	Col 3	Col 4
Row 0	marks[0][0]	marks[0][1]	marks[0][2]	marks[0][3]	marks[0][4]
Row 1	marks[1][0]	marks[1][1]	marks[1][2]	marks[1][3]	marks[1][4]
Row 2	marks[2][0]	marks[2][1]	marks[2][2]	marks[2][3]	marks[2][4]

Two-dimensional array

marks[0] –	marks[0]	marks[1]	marks[2]	marks[3]	marks[4]

marks[1] –	marks[0]	marks[1]	marks[2]	marks[3]	marks[4]

marks[2] –	marks[0]	marks[1]	marks[2]	marks[3]	marks[4]

Representation of two-dimensional array marks[3][5]

There are two ways of storing a two-dimensional array in the memory. The first way is the row major order and the second is the column

major order. Let us see how the elements of a 2D array are stored in a row major order. Here, the elements of the first row are stored before the elements of the second and third rows.

(0,0) (0,1) (0,2) (0,3) (1,0) (1,1) (1,2) (1,3) (2,0) (2,1) (2,2) (2,3)
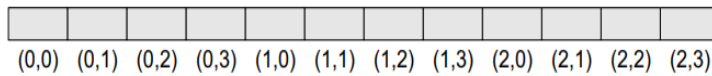
Elements of a 3 × 4 2D array in row major order

00	01	02	03
10	11	12	13
20	21	22	23

(0,0) (1,0) (2,0) (3,0) (0,1) (1,1) (2,1) (3,1) (0,2) (1,2) (2,2) (3,2)

Elements of a 4 × 3 2D array in column major order

Total allocated memory=row_size* col_size *sizeof(datatype)
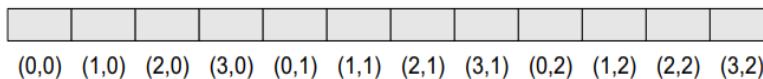
=3*5*sizeof(int) **[sizeof(int)=2 bytes in a 16-bit machine]**

=3*5*2=30 bytes

If the starting address is 8000.

Address of an element=starting address+ sizeof(int)[col_size(row_index)+col_index]

Address of marks[2][4]=8000+2[5(2)+4]

=8000+2*14

=8028

Initialization Of Two Dimensional Arrays

After an array is declared, its elements must be initialized. Otherwise, they will contain "garbage". An array can be initialized at either of the following stages:

• At compile time

• At run time

Compile Time Initialization

we can initialize the elements of arrays in the same way as the ordinary variables when they are declared.

1) **Full initialization**

The initialization is done row by row (row major order)

type array-name[row-size][col-size] = {list of values};

The values in the list are separated by commas. For example, the statement

int a[3][2]={1,2,3,4,5,6}; will assign a[0][0]=1, a[0][1]=2, a[1][0]=3, a[1][1]=4, a[2][0]=5, a[2][1]=6 with size 6.

2) **Full initialization** (by surrounding the elements of the each row by braces)

int a[3][2]={{1,2},{3,4},{5,6}};

For example, the statement will assign a[0][0]=1, a[0][1]=2, a[1][0]=3, a[1][1]=4, a[2][0]=5, a[2][1]=6 with size 6.

3) **Full initialization (in the form of matrix)**

int a[3][2]={

{1,2},

{3,4},

{5,6}

};

For example, the statement will assign a[0][0]=1, a[0][1]=2, a[1][0]=3, a[1][1]=4, a[2][0]=5, a[2][1]=6 with size 6.

4) **Partial initailization**

Compile time initialization may be partial. That is, the number of initializers may be less than declared size. In such cases. the remaining elements are inilialized to zero,if the array type is numeric and NULL if the type is char.

int a[3][2]={

{1,2},

{3}

};

For example, the statement will assign a[0][0]=1, a[0][1]=2, a[1][0]=3, a[1][1]=0, a[2][0]=0, a[2][1]=0 with size 6.

5) **Without size**

The size may be omitted. In such cases, the compiler allocates enough space for all initialize elements.

For example, the statement int a[ ][2] = {1,1,1,1,1,1}; will declare the **a** array to contain six elements with initial values 1. This approach works fine as long as we initialize every element in the array.

row-size=total number of elements/col-size

row-size=6/2=3

When the array is completely initialized with all values, explicitly, we need not specify the size of the first dimension.

column size is mandatory.

Run Time Initialization (Accessing the elements of 2D arrays)

An array can be explicitly initialized at run time. This approach is usually applied for initializing large arrays.

In case of one-dimensional arrays, we used a single for loop to vary the index i, so that all the elements could be scanned. Since the two-dimensional array contains two subscripts, we will use two for loops to scan the elements. The first for loop will scan each row in the 2D array and the second for loop will scan individual columns for every row in the array.

int a [3][2] ;

scanf("%d%d%d", &a[0][0],&a[0][1],&a[1][0],&a[1][1],&a[2][0],&a[2][1]);

(or)

int a [3][2] ;

for(i=0;i<3;i++)

{

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

      {

            scanf("%d", &a[i][j]);

      }

}

will initialize array elements with the values entered through the keyboard.

for(i=0;i<row-size;i++)

{

      for(j=0;j<col-size;j++)

      {

            scanf("%d", &a[i][j]);

      }

}

Programs:

1. **Input and Output a matrix of size m*n**

```
void main()
{
        int m,n,i,j,a[10][10];
        printf("enter row size and column size");
        scanf("%d%d",&m,&n);
        printf("enter the elements");
        for(i=0;i<m;i++)
        {
                for(j=0;j<n;j++)
                {
                        scanf("%d",&a[i][j]);
                }
        }
        printf("The elements are");
        for(i=0;i<m;i++)
        {
                printf("\n");
                for(j=0;j<n;j++)
                {
                        printf("%d\t",&a[i][j]);
                }
        }
}
```

2. **Biggest element in a 2-D array**

```
void main()
{
        int m,n,i,j,a[10][10],big;
        printf("enter row size and column size");
        scanf("%d%d",&m,&n);
        printf("enter the elements");
```

```
            for(i=0;i<m;i++)
            {
                    for(j=0;j<n;j++)
                    {
                            scanf("%d",&a[i][j]);
                    }
            }
            big=a[0][0];
            for(i=0;i<m;i++)
            {
                    for(j=0;j<n;j++)
                    {
                            if(a[i][j]>big)
                                    big=a[i][j];
                    }
            }
                    printf("The biggest element is %d ",big);
    }
```

3. Multiplication table

Write a program to compute and print a multiplication table for number 1 to 5 as shown below.

	1	2	3	4	5
1	1	2	3	4	5
2	2	4	6	8	10
3	3	6	.	.	.
4	4	8	.	.	.
5	5	10	.	.	25

The program uses a two-dimensional array to store the table values. Each value is calculated using the control variables of the nested for loops as follows:

product(i][j] = row * column

where i denotes rows and j denotes columns of the product table.

PROGRAM TO PRINT MULTIPLICATION TABLE

```c
int main()
{
        int   i,j, product[10][10] ;
        for ( i = 1 ; i <= 10 ; i++ )
        {
                for( j = 1 ; j <= 10 ; j++ )
                {
                        product[i][j] = i*j;
                }
        }
        for( i = 1 ; i <= 10 ; i++ )
        {
                printf("\n") ;
                for( j = 1 ; j <= 10 ; j++ )
                {
                        printf("%d\t",product[i][j]);
                }
        }
}
```

2.8 OPERATIONS ON TWO-DIMENSIONAL ARRAYS

Two-dimensional arrays can be used to implement the mathematical concept of matrices. In mathematics, a matrix is a grid of numbers, arranged in rows and columns. Thus, using two dimensional arrays, we can perform the following operations on an m×n matrix:

1. ***Transpose*** Transpose of an m x n matrix A is given as a n x m matrix B, where $B_{i,j} = A_{j,i}$.

   ```c
   #include <stdio.h>
   int main()
   {
           int a[10][10],b[10][10],m,n,i,j;
           printf("Enter rows and columns of matrix: ");
           scanf("%d%d",&m,&n);
   ```

```c
printf("\nEnter elements of matrix:\n");
for(i=0; i<m; i++)
{
        for(j=0; j<n; j++)
        {
        scanf("%d", &a[i][j]);
        }
}
printf("\nEntered Matrix: \n");
for(i=0; i<m; i++)
{
        printf("\n");
        for(j=0; j<n; j++)
        {
        printf("%d\t ", a[i][j]);
        }
}
for(i=0; i<n; i++)
{
        for(j=0; j<m; j++)
        {
        b[i][j] = a[j][i];
        }
}
printf("\nTranspose of Matrix:\n");
for(i=0; i<n; i++)
{
        printf("\n");
        for(j=0; j<m; j++)
        {
        printf("%d\t",b[i][j]);
```

```
                    }
               }
          return 0;
     }
```

```
Enter rows and columns of matrix: 3 2

Enter elements of matrix:
1 2 3 4 5 6

Entered Matrix:

1          2
3          4
5          6
Transpose of Matrix:

1         3         5
2         4         6
_____
```

2. **_Sum_** Two matrices that are compatible with each other can be added together, storing the result in the third matrix. Two matrices are said to be compatible when they have the same number of rows and columns. The elements of two matrices can be added by writing: $C_{i,j} = A_{i,j} + B_{i,j}$.

```c
void main()
{
        int m,n,i,j,a[10][10],b[10][10,c[10][10];
        printf("Enter rows and columns of matrix: ");
        scanf("%d%d",&m,&n);
        printf("\nEnter elements of matrix A:\n");
        for(i=0; i<m; i++)
        {
                for(j=0; j<n; j++)
                {
                        scanf("%d", &a[i][j]);
                }
        }
        printf("\nEnter elements of matrix B:\n");
        for(i=0; i<m; i++)
        {
```

```
                for(j=0; j<n; j++)
                {
                        scanf("%d", &b[i][j]);
                }
        }
        for(i=0; i<m; i++)
        {
                for(j=0; j<n; j++)
                {
                        c[i][j]=a[i][j]+b[i][j];
                }
        }
        printf("\nResultant Matrix: \n");
        for(i=0; i<m; i++)
        {
                printf("\n");
                for(j=0; j<n; ++j)
                {
                        printf("%d\t ", c[i][j]);
                }
        }
}
```

3. ***<u>Difference</u>*** Two matrices that are compatible with each other can be subtracted, storing the result in the third matrix. Two matrices are said to be compatible when they have the same number of rows and columns. The elements of two matrices can be subtracted by writing: $C_{i,j} = A_{i,j} - B_{i,j}$

```
        void main()
        {
                int m,n,i,j,a[10][10],b[10][10,c[10][10];
                printf("Enter rows and columns of matrix: ");
                scanf("%d%d",&m,&n);
```

```
        printf("\nEnter elements of matrix A:\n");

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

        {

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

                {

                        scanf("%d", &a[i][j]);

                }

        }

        printf("\nEnter elements of matrix B:\n");

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

        {

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

                {

                        scanf("%d", &b[i][j]);

                }

        }

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

        {

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

                {

                        c[i][j]=a[i][j]-b[i][j];

                }

        }

        printf("\nResultant Matrix: \n");

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

        {

                printf("\n");

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

                {

                        printf("%d\t ", c[i][j]);

                }
```

```
        }
    }
```

Product Two matrices can be multiplied with each other if the number of columns in the first matrix is equal to the number of rows in the second matrix. Therefore, m x n matrix A can be multiplied with a p x q matrix B if n=p. The dimension of the product matrix is m x q. The elements of two matrices can be multiplied by writing: $C_{i,j} = \sum A_{i,k} \times B_{k,j}$ for k=1 to n. #include<stdio.h>

```c
#include<stdlib.h>
void main()
{
        int a[10][10],b[10][10],c[10][10];
        int m,n,p,q,i,j,k;
        printf("Enter the order of the matrix A\n");
        scanf("%d%d",&m,&n);
        printf("Enter the order of the matrix B\n");
        scanf("%d%d",&p,&q);
    if(n!=p)
    {
        printf("\n multiplication not possible");
        exit(0);
    }
        else
        {
    printf("Enter the elements of matrix A...\n");
        for(i=0;i<m;i++)
    {
            for(j=0;j<n;j++)
                {
            scanf("%d",&a[i][j]);
                }
        }
```

```c
        printf("Enter the elements of matrix B...\n");
        for(i=0;i<p;i++)
          {
                        for(j=0;j<q;j++)
                        {
             scanf("%d",&b[i][j]);
          }
      }
    printf("\n Matrix A \n");
      for(i=0;i<m;i++)
          {
              printf("\n");
                  for(j=0;j<n;j++)
                      {
                      printf("%d\t",a[i][j]);
                  }
              }
          printf("\n Matrix B \n");
      for(i=0;i<p;i++)
          {
              printf("\n");
                  for(j=0;j<q;j++)
                      {
                      printf("%d\t",b[i][j]);
                  }
              }


      for(i=0;i<m;i++)
          {
                  for(j=0;j<q;j++)
              {
```

```
                    c[i][j]=0;

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

                    c[i][j]=c[i][j]+a[i][k]*b[k][j];

              }

          }

     printf("\n Product of A and B matrices : MATRIX C\n");

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

          {

             printf("\n");

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

                    {

                    printf("%d\t",c[i][j]);

                    }

              }

     }

  }
```

Test cases

Test No	Input Parameters	Expected Output	Obtained Output
1	Enter the order of the matrix A 2 2 Enter the order of the matrix B 2 2 Enter the elements of matrix A... 1 2 3 4 Enter the elements of matrix B... 5 6 7 8	MATRIX A 1 2 3 4 MATRIX B 5 6 7 8 Product of A and B matrices : MATRIX C 19 22 43 50	MATRIX A 1 2 3 4 MATRIX B 5 6 7 8 Product of A and B matrices : MATRIX C 19 22 43 50
2	Enter the order of the matrix A 2 3 Enter the order of the matrix B 2 2 Enter the elements of matrix A... 1 2 3 4 5 6 Enter the elements of matrix B... 5 6 7 8	multiplication not possible	multiplication not possible

2.9 PASSING TWO-DIMENSIONAL ARRAYS TO FUNCTIONS

Like simple arrays, we can also pass two-dimensional arrays to functions. The approach is similar to one-dimensional arrays. The rules are simple.

1. The functions must be called by passing only the array name.
2. In the function definition, we must indicate that the array has two-dimensions by including two sets of brackets.
3. The prototype declaration should be similar to the function header.

The function given below calculates the average of values in a two-dimensional matrix

```c
float calculate(int a[10][10],int m,int n);
void main()
{
        int m,n,i,j,a[10[10],sum;
        float avg;
        printf("enter the row size and column size");
        scanf("%d%d",&m,&n);
        printf("enter elements");
        for(i=0;i<m;i++)
        {
                for(j=0;j<n;j++)
                {
                        scanf("%d",&a[i][j]);
                }
        }
        sum= calculate(a,m,n);
        avg=(float) sum/(m*n);
        printf("sum=%d",sum);
        printf("average=%f",avg);
}
float calculate(int a[10][10],int m,int n)
{
        int i,j;
```

```
        float s=0;

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

        {

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

                {

                        s=s+a[i][j];

                }

        }

        return s;

    }
```

Output:

enter the row size and column size 3 2

enter elements 1 2 3 4 5 6

sum=21

average=3.5

2.10 MULTIDIMENSIONAL ARRAYS

A multi-dimensional array in simple terms is an array of arrays. As we have one index in a one dimensional array, two indices in a two-dimensional array, in the same way, we have n indices in an n-dimensional array or multi-dimensional array. Conversely, an n–dimensional array is specified using n indices. An n-dimensional m1 x m2 x m3 x … x m_n array is a collection of m1 x m2 x m3 x … x m_n elements.

In a multi-dimensional array, a particular element is specified by using n subscripts as $A[I_1][I_2][I_3]...[I_n]$, where $I_1 <=M_1$, $I_2 <=M_2$, $I_3 <=M_3$, $...I_n <=M_n$

A multi-dimensional array can contain as many indices as needed and as the requirement of memory increases with the number of indices used. However, in practice, we hardly use more than three indices in any program. Below Figure shows a three-dimensional array. The array has three pages, four rows, and two columns.

Three-dimensional array
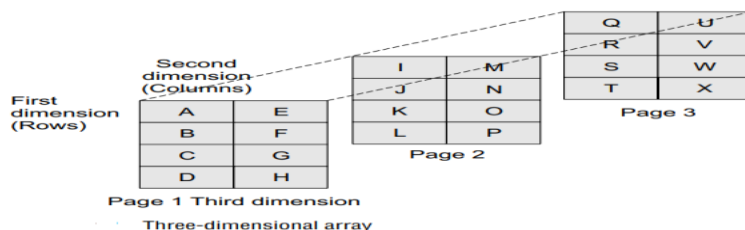
Write a program to read and display a 2 x 2 x 2 array.

```c
#include <stdio.h>

int main()
{
        int a [2][2][2], i, j, k;

        printf("\n Enter the elements of the matrix");

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

        {

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

                {

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

                        {

                                scanf("%d", &a[i][j][k]);

                        }

                }

        }

        printf("\n The matrix is: ");

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

        {

                printf(("\n");

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

                {

                        printf(("\n");

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

                        {

                                printf("%d\t", &a[i][j][k]);

                        }

                }

        }
}
```
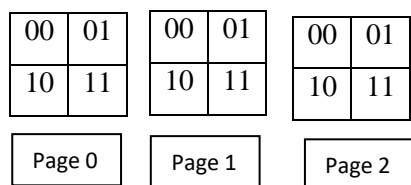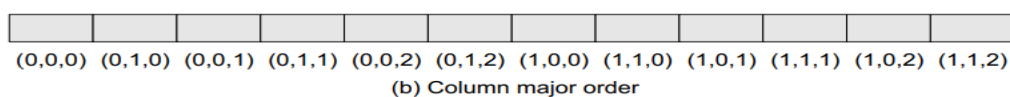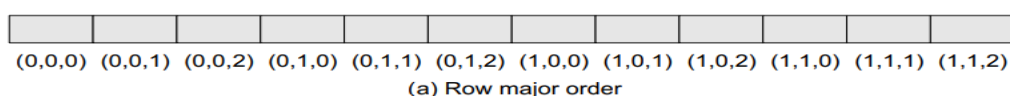
Example

Consider a three-dimensional array defined as int A[2][2][3]. Calculate the number of elements in the array. Also, show the memory representation of the array in the row major order and the column major order.

Solution

00	01
10	11

Page 0

00	01
10	11

Page 1

00	01
10	11

Page 2

A three-dimensional array consists of pages. Each page, in turn, contains m rows and n columns.

(0,0,0)	(0,0,1)	(0,0,2)	(0,1,0)	(0,1,1)	(0,1,2)	(1,0,0)	(1,0,1)	(1,0,2)	(1,1,0)	(1,1,1)	(1,1,2)

(a) Row major order

(0,0,0)	(0,1,0)	(0,0,1)	(0,1,1)	(0,0,2)	(0,1,2)	(1,0,0)	(1,1,0)	(1,0,1)	(1,1,1)	(1,0,2)	(1,1,2)

(b) Column major order

The three-dimensional array will contain 2 x 2 x 3 = 12 elements.

2.11 APPLICATIONS OF ARRAYS

- Arrays are widely used to implement mathematical vectors, matrices, and other kinds of rectangular tables.

- Many databases include one-dimensional arrays whose elements are records.

- Arrays are also used to implement other data structures such as strings, stacks, queues, heaps, and hash tables.

- Arrays can be used for sorting elements in ascending or descending order.