Unit 1: Concept of data structures

- Introduction: data types, data structures and abstract data types
- Introduction to algorithms

What is data structure?

- Data structure is a way of organizing all data items and establishing relationship among those data items.
- Data structures are the building blocks of a program.

  Data structure mainly specifies the following four things:
  - Organization of data.
  - Accessing methods
  - Degree of associativity
  - Processing alternatives for information

To develop a program of an algorithm, we should select an appropriate data structure for that algorithm. Therefore algorithm and its associated data structures form a program.

Algorithm + Data structure = Program

A **static data structure** is one whose capacity is fixed at creation. For example, array. A **dynamic data structure** is one whose capacity is variable, so it can expand or contract at any time. For example, linked list, binary tree etc.

Abstract Data Types (ADTs)

An **abstract data type** is a data type whose representation is hidden from, and of no concern to the application code. The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. It is called "abstract" because it gives an implementation-independent view. The process of providing only the essentials and hiding the details is known as abstraction.

The user of **data type** does not need to know how that data type is implemented, for example, we have been using Primitive values like int, float, char data types only with the knowledge that these data type can operate and be performed on without any idea of how they are implemented. So a user only needs

to know what a data type can do, but not how it will be implemented. Think of ADT as a black box which hides the inner structure and design of the data type.

For example, when writing application code, we don't care how strings are represented: we just declare variables of type *String*, and manipulate them by using string operations.

Once an abstract data type has been designed, the programmer responsible for implementing that type is concerned only with choosing a suitable data structure and coding up the methods. On the other hand, application programmers are concerned only with using that type and calling its methods without worrying much about how the type is implemented.

 # ADT for Relation:-

```
/*value of definition */
Abstract type def <integer, integer> RTIONAL // value definition
Condition RATIONAL [1] = 0;     // denominator is not equal to zero;
```

cond^{n}.

```
/* operator definition * /
Abstract RATIONAL make rational (a, b)
Int a,b;
Precondition b! = 0;
Post condition make rational [0] == a;
                Make rational [1] == b;
Abstract RATIONAL add (a, b)
RATIONAL a,b;
Past condition  add [1] == a[1] * b[1];
                Add [0] == a[a] * b[1] + b[0] * a[1];
Abstract RATIONAL mult (a,b)
RATIOANL a, b;
Post condition   mult [0] ==a[a] * b[0];
                 Mult [1] == a[1] *b[1];
Abstract RATIONAL equal (a,b)
RATIONAL a,b;
Post condition equal == (a[0] * b[1] == b [0] * a[1]);
```

Classification of data structure:

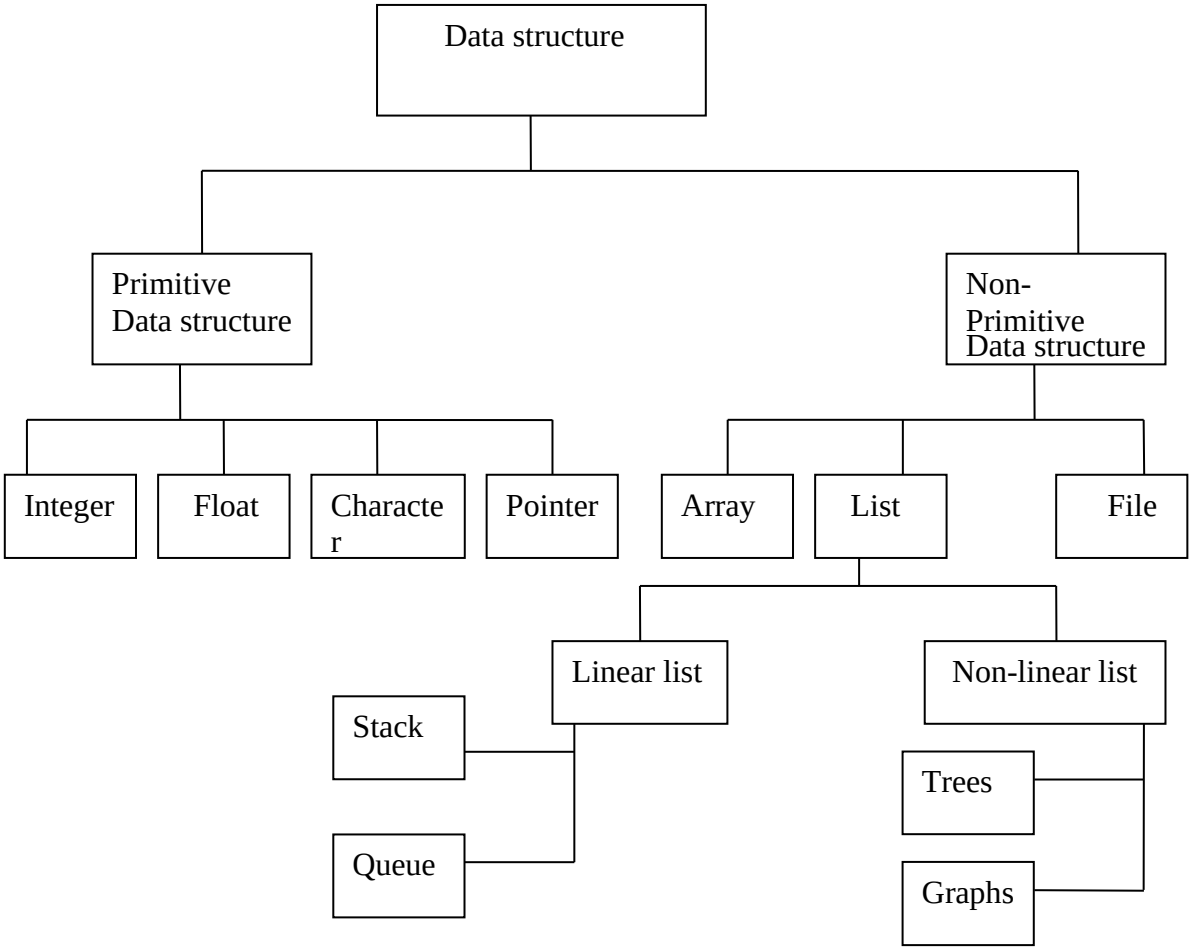


Fig:- Classification of data structure

Primitive data structure is a fundamental type of data structure that stores the data of only one type whereas the non-primitive data structure is a type of data structure which is a user-defined that stores the data of different types in a single entity

Array

- Array is a group of same type of variables that have common name
- Each item in the group is called an *element* of the array
- Each element is distinguished from another by an *index*
- All elements are stored contiguously in memory
- The elements of the array can be of any valid type- integers, characters, floatingpoint types or user-defined types

Types of Array:

1). One dimensional array:

The elements of the array can be represented either as a single column or as a single row.

Declaring one-dimensional array:

data_type array_name[size];

Following are some valid array declarations:

int age[15]; float weight[50]; int marks[100]; char section[12]; char name[10];

Following are some invalid array declarations in c:

int value[0]; int

marks[0.5]; int

number[-5];

Array Initialization (1-D):

The general format of array initialization is:

data_type array_name[size]={element1,element2,……..,element n};

for example:

int age[5]={22,33,43,24,55};

int weight[]={55,6,77,5,45,88,96,11,44,32};

float a[]={2,3.5,7.9,-5.9,-8};

char section[4]={'A','B','C','D'};

char name[10]="Bhupendra";

Example 1: A program to read n numbers and to find the sum and average of those numbers.

```
#include<stdio.h> void main()
{
    int a[100], i, n, sum=0; float avg;
    printf("Enter number of elements");
    scanf("%d",&n);
    printf("Enter %d numbers",n);
    for(i=0;i<n;i++)
```

```
    {                   scanf("%d",&a[i]);
                sum=sum+a[i];
                //sum+=a[i];
    } avg=sum/n;
    printf("sum=%d\n Average=%f", sum, avg);
}
```

Some common operations performed in one-dimensional array are:

- Creating of an array
- Inserting new element at required position
- Deletion of any element
- Modification of any element
- Traversing of an array
- Merging of arrays

Insertion of new element at required position: Let we have an array a[6]=(1,5,7,6,22,90};

Suppose we want to insert 20 in array a, at location with index 4, it means the elements 22 and 90 must shift 1 position downwards as follows.

a[0]	1
a[1]	5
a[2]	7
a[3]	6
a[4]	
a[5]	22
a[6]	
a[7]	90
a[8]	0
a[9]	
	0
	0
	0

a[0]	1
a[1]	5
a[2]	7
a[3]	6
a[4]	
	……..
a[5]	22
a[6]	
a[7]	9 0
a[8]	0
	0

a[0]	1
a[1]	
a[2]	5
a[3]	7
a[4]	
a[5]	6
a[6]	20
a[7]	
a[8]	22
a[9]	
	90
	0
	0
	0

Original array Elementsa[9] 0shifted downwards Array after insertion

C- code for above problem:

```
#include<stdio.h>
#include<conio.h> void main()
{
        int a[100], pos, nel, i;
        printf("Enter no of elements to be inserted");
        scanf("%d", &n);
        printf("Enter %d elements", n);
        for(i=0;i<n;i++)
        { scanf("%d", &a[i]);
        }
        printf("Enter position at which you want to insert new element");
        scanf("%d", &pos);
        printf("Enter new element");
        scanf("%d", &nel);
        for(i=n-1; i>=pos; i--)
        { a[i+1] = a[i];
        } a[pos]=nel; n++;
        printf("New array is:\n");
        for(i=0; i<n; i++)
        { printf("%d\t", a[i]);
        }
        getch();
}
```

Deletion of any element from an array:

Suppose we want to delete the element a[5]=90, then the elements following it were moved upward by one location as shown in fig. below:

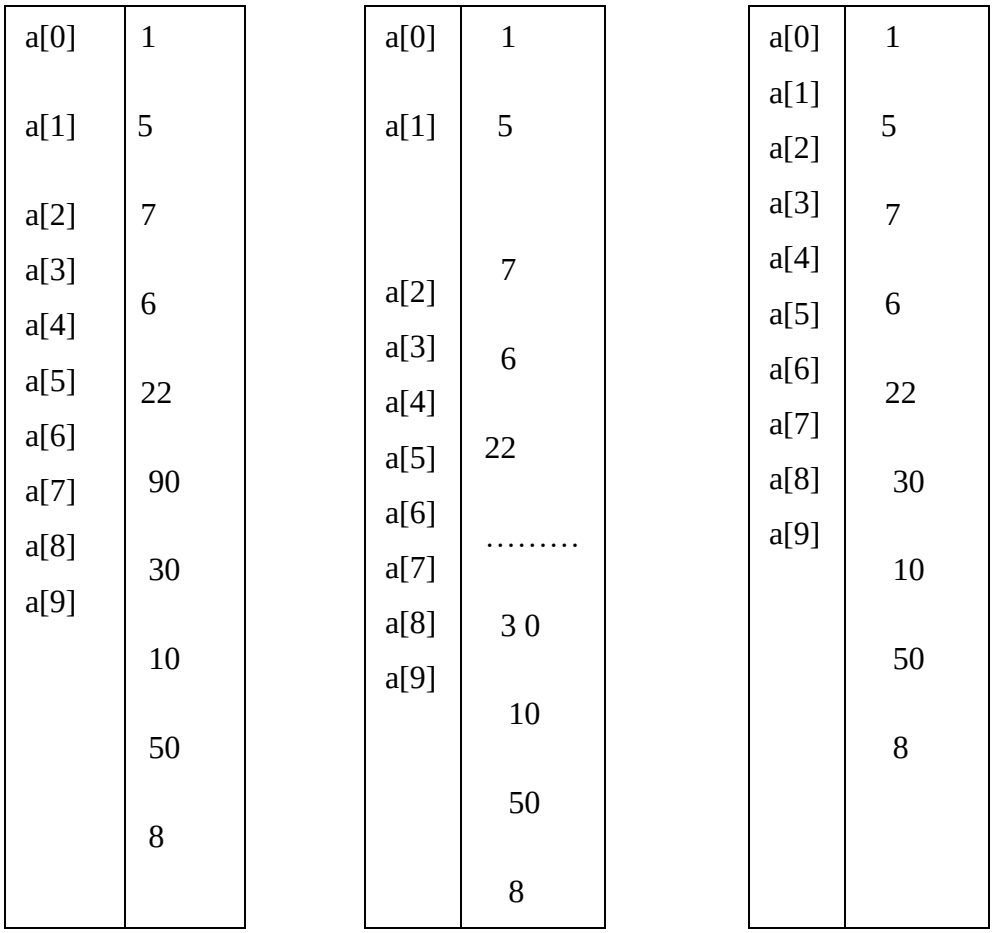


Fig: - Deleting an element from one-dimensional array

C- code for above problem:

```
#include<stdio.h>
#include<conio.h> void main()
{
    int a[100], pos, i; clrscr();
    printf("Enter no of elements to be inserted");
    scanf("%d", &n);
    printf("Enter %d elements", n);
    for(i=0;i<n;i++)
    { scanf("%d", &a[i]);
    }
    printf("Enter position at which you want to delete an element");
    scanf("%d", &pos);
    for(i=pos; i<n; i++)
    {
        a[i] = a[i+1];
```

```
        }
        n--;
        printf("New array is:\n");
        for(i=0; i<n; i++)
        {
                printf("%d\t", a[i]);
        }
 }
```

Modification of any element:

```
#include<stdio.h>
#include<conio.h> void main()
{
        int a[100], pos, nel,  i;
        printf("Enter no of elements to be inserted");
        scanf("%d", &n);
        printf("Enter %d elements", n);
        for(i=0;i<n;i++)
        { scanf("%d", &a[i]);
        }
        printf("Enter position at which you want to modify an element");
        scanf("%d", &pos);
        printf("Enter new element");
        scanf("%d", &nel);
        a[pos]=nel;
        printf("New array is:\n");
        for(i=0; i<n; i++)
        {
                printf("%d\t", a[i]);
        }
        getch();
}
```

Traversing of an array:

Traversing means to access all the elements of the array, starting from first element upto the last element in the array one-by-one.

C- code for above problem:

```
#include<stdio.h>
#include<conio.h> void main()
{
    int a[100], pos, i; clrscr();
    printf("Enter no of elements to be inserted");
    scanf("%d", &n);
    printf("Enter %d elements", n);
    for(i=0;i<n;i++)
    { scanf("%d", &a[i]);
    }
    printf("Traversing of the array:\n");
    for(i=0; i<n; i++)
    {
        printf("%d\t", a[i]);
    }
    getch();
}
```

Merging of two arrays:

Merging means combining elements of two arrays to form a new array. Simplest way of merging two arrays is the first copy all elements of one array into a third empty array, and then copy all the elements of other array into third array.

Suppose we want to merge two arrays a[6] and b[4]. The new array says c will be having (6+4) =10 elements as shown in figure below.

a[0]	1
a[1]	5
a[2]	7
a[3]	6
a[4]	12
a[5]	15

b[0]	20
b[1]	25
b[2]	30
b[3]	35

c[0]	1
c[1]	5
c[2]	7
c[3]	6
c[4]	12
c[5]	15
c[6]	20
c[7]	25
c[8]	30
c[9]	35

Fig: - Merging of two arrays

C- code for above problem:

```
#include<stdio.h>
#include<conio.h> void main()
{
    int a[6], b[4], c[10], i, j;
    printf("Enter elements of first array\n");
    for(i=0;i<6;i++)
    scanf("%d", &a[i]);
    printf("Enter elements of second array\n");
    for(i=0;i<4;i++)
    { scanf("%d", &a[i]);
    }
    for(i=0; i<6; i++)
    {
        c[i]=a[i];
    }
    j=i; // here i=j=6
    for(i=0; i<4; i++)
    {
        c[j]=a[i]; j++; }
    printf("The resulting array is:\n");
    for(i=0; i<10; i++)
```

```
    {
        printf("%d\t", c[i]);
    }
    getch();
}
```

Two-Dimensional array:

When we declare two dimensional array, the first subscript written is for the number of rows and the second one is for the column.

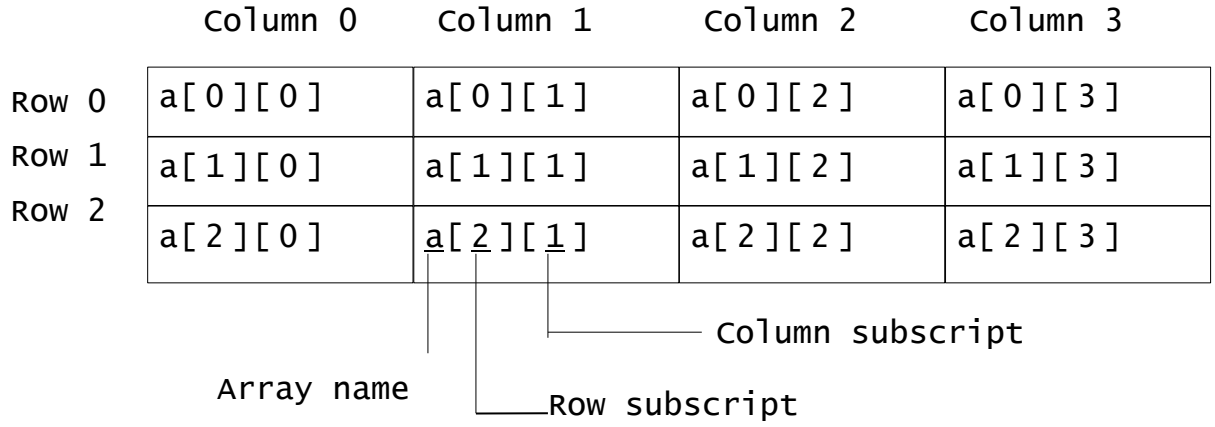
Declaration of 2- D array:

data_type array_name[row_size][column_size];

Example;

int a[3][4]; float b[10][10]; int this first example, 3 represents number of rows and 4 represents number of columns.

• Think, two-dimensional arrays as tables/matrices arranged in rows and columns • Use first subscript to specify row no and the second subscript to specify column no.



Array Initialization (2-D):

The general format of array initialization is:

data_type array_name[row_size][col_size]={element1,element2,……..,element n}; **for example:**

```
    int a[2][3]={33,44,23,56,77,87};
```

or int a[2][3]={{33,44,23},

{56, 77, 87}};

Example 1: A program to find addition of any two matrices by using function

```
#include<stdio.h>
#include<conio.h>
void display(int [][], int, int);   //function prototype
void main() {
        int a[10][10], b[10][10],c[10][10],i, j, r, c;
printf("Enter size of a matrix");
scanf("%d%d", &r,&c);
        printf("Enter elements of first matrix\n");
        for(i=0;i<r;i++)
        {
                for(j=0;j<c;j++) {
                    scanf("%d",&a[i][j]);
                }
        }
         printf("Enter elements of second matrix\n");
 for(i=0;i<r;i++)
        { for(j=0;j<c;j++) {
                    scanf("%d",&b[i][j]);
                }
        }
// finding sum for(i=0;i<r;i++)
        { for(j=0;j<c;j++)
                {
                        c[i][j]=a[i][j]+b[i][j];
                }
        }
        printf("The first  matrix is\n");
        display(a, r, c);  //function call
        printf("The second  matrix is\n");
        display(b, r, c);  //function call
        printf("The resulting  matrix is\n");
        display(c, r, c);  //function call
        getch();
}
```

```
void display(int d[10][10], int r, int c)    //function definition
{
        int i, j;
        for(i=0;i<r;i++)
        { for(j=0;j<c;j++)
                {
                        printf("%d\t", d[i][j]);
                }
                printf("\n");
        }
}
```

Example 2: A program to find transposition of a matrix by using function

```
#include<stdio.h>
void display(int [][], int, int);   //function prototype
void main() {
        int a[10][10], t[10][10],i, j, r, c;
printf("Enter no of rows and no of columns");
scanf("%d%d",&r,&c);
printf("Enter elements of a matrix\n");
for(i=0;i<r;i++)
        { for(j=0;j<c;j++) {
                    scanf("%d",&a[i][j]);
                }
        }
//finding transpose of a matrix for(i=0;i<r;i++)
        { for(j=0;j<c;j++)
                { t[i][j]=a[j][i];
                }
        }
        printf("the original matrix is\n");
        display(a, r, c);              //function call
        printf("the transposed matrix is\n");
        display(t, r, c);              //function call
}
```

```
void display(int x[][], int r, int c)          //function definition
 {
            int i, j;
            for(i=0;i<r;i++)
            { for(j=0;j<c;j++)
                  {
                        printf("%d\t",x [i][j]);
                  }
                  printf("\n");
            }
 }
```

Implementation of a two-dimensional array:

A two dimensional array can be implemented in a programming language in two ways:

- Row-major implementation
- Column-major implementation

Row-major implementation:

Row-major implementation is a linearization technique in which elements of array are reader from the keyboard row-wise i.e. the complete first row is stored, and then the complete second row is stored and so on. For example, an array a[3][3] is stored in the memory as shown in fig below:

a[0][0]	a[0][1]	a[0][2]	a[1][0]	a[1][1]	a[1][2]	a[2][0]	a[2][1]	a[2][2]

complete second column is stored, and so on. For example an array a [3] [3] is stored in the memory as shown in the fig below:

a[0][0]	a[1][0]	a[2][0]	a[0][1]	a[1][1]	a[2][1]	a[0][2]	a[1][2]	a[2][2]

Row 1 Row 2 Row 3

Column-major implementation:

In column major implementation memory allocation is done column by column i.e. at first the elements of the complete first column is stored, and then elements of

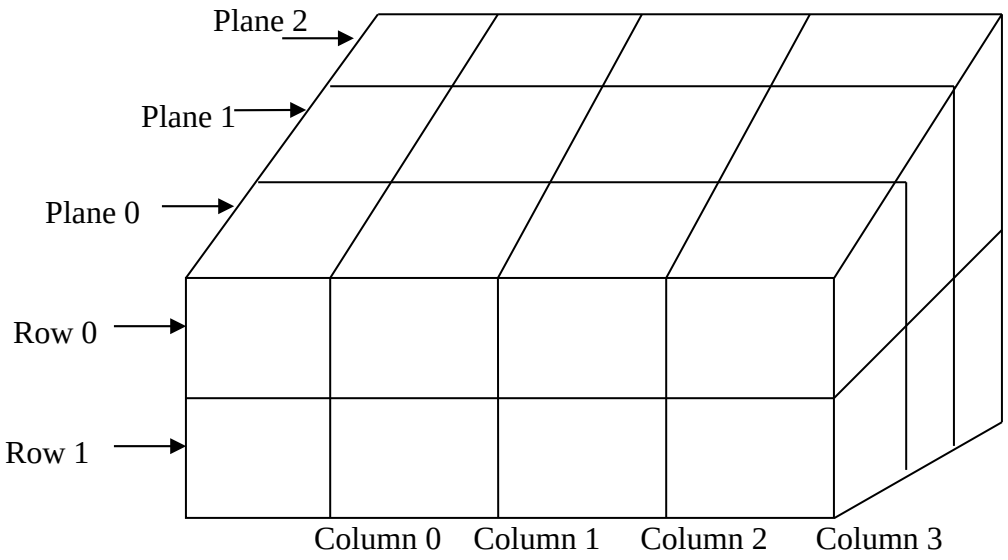
Column 1 column 2 column 3

Multi-Dimensional array

C also allows arrays with more than two dimensions. For example, a three dimensional array may be declared by int a[3][2][4];

Here, the first subscript specifies a plane number, the second subscript a row number and the third a column number.

However C does allow an arbitrary number of dimensions. For example, a sixdimensional array may be declared by int b[3][4][6][8][9][2];



Show that an array is an ADT:

Let A be an array of type T and has n elements then it satisfied the following operations:

- CREATE(A): Create an array A
- INSERT(A,X): Insert an element X into an array A in any location
- DELETE(A,X): Delete an element X from an array A
- MODIFY(A,X,Y): modify element X by Y of an array A
- TRAVELS(A): Access all elements of an array A
- MERGE(A,B): Merging elements of A and B into a third array C

Thus by using a one-dimensional array we can perform above operations thus an array acts as an ADT.

Structure:

A structure is a collection of one or more variables, possibly of different types, grouped together under a single name.

An array is a data structure in which all the members are of the same data type. Structure is another data structure in which the individual elements can differ in type. Thus, a single structure might contain integer elements, floating-point elements and character elements. The individual structure elements are referred to as *members.* Defining a structure: A structure is defined as

struct *structure_name*

{ member 1; member 2;

……..

member n;

};

We can define a structure to hold the information of a student as follows: **struct** Student

{ **c**har name[2];

int roll;

char sec;

float marks;

};

Structure variable declaration:

struct Student s1, s2, s3;

We can combine both template declaration and structure variable declaration in one statement.

Eg,

struct Student

{ **c**har name[2];

int roll;

char sec;

float marks;

} **s1, s2, s3;**

Accessing members of a structure:

There are two types of operators to access members of a structure. Which are:

- Member operator (dot operator or period operator (.))
- Structure pointer operator (->).

Structure initialization:

Like any data type, a structure variable can be initialized as follows: **struct** Student

{ **c**har name[20];

int roll; char sec;

float marks;

};

struct Student s1={"Raju", 22, 'A', 55.5};

The s1 is a structure variable of type Student, whose members are assigned initial values. The first member (name[20[) is assigned the string "Raju", the second member (roll) is assigned the integer value 22, the third member (sec) is assigned the character 'A', and the fourth member (marks) is assigned the float value 55.5.

Example: program illustrates the structure in which read member elements of structure and display them.

```
#include<stdio.h> void main()
{
    struct Student
    { char name[20];
        int roll;
        char sec;
        float marks;
    };
    struct Student s1;
    clrscr();
    printf("Enter the name of a student");
    gets(s1.name);
    printf("Enter the roll number of a student");
    scanf("%d",&s1.roll);
```

```
    printf("Enter the section of a student");
    scanf("%c",&s1.sec);
    printf("Enter the marks obtained by the student");
    scanf("%f",&s1.marks);
    //displaying the records
    printf("Name=%s\n Roll number =%d\n Section=%c\n Obtained marks=%f",s1.name, s1.roll, s1.sec,
    s1.marks);
}
```

Structures within Structures:

Structures within structures mean nesting of structures. Study the following example and understand the concepts.

Example: the following example shows a structure definition having another structure as a member. In this example, person and students are two structures. Person is used as a member of student.(person within Student)

```
#include<stdio.h>
struct Person
{ char name[20]; int age;
};
struct Student
{
    int roll;
    char sec;
    struct Person p;
};
void main()
{
    struct Student s;
    printf("Enter the name of a student");
    gets(s.p.name);
    printf("Enter age"); scanf("%d",&s.p.age);
    printf("Enter the roll number of a student");
    scanf("%d",&s.roll);
    printf("Enter the section of a student");
```

```
        scanf("%c",&s.sec);
        //displaying the records
        printf("Name=%s\n Roll number =%d\n Age=%d\n Section=%c\n",s.p.name, s.roll, s.p.age, s.sec);
}
```

Passing entire structures to functions:

```
#include<stdio.h>
void display(struct student);
struct student
{ char name[20]; int age; int
        roll; char sec;
};
 void main()
{
        struct student s; int i;
        printf("Enter the name of a student");
         gets(s.name);
        printf("Enter age");
        scanf("%d",&s.age);
        printf("Enter the roll number of a student");
        scanf("%d",&s.roll);
        printf("Enter the section of a student");
        scanf("%c",&s.sec);
        display(s);   //function call }
void sisplay(struct student st)
{
        //displaying the records
        printf("Name=%s\n Roll number =%d\n Age=%d\n Section=%c\n",st.name, st.roll, st.age, st.sec);
}
```

Unions:

Both **structure** and **unions** are used to group a number of different variables together. Syntactically both structure and unions are exactly same. The main difference between them is in storage. In structures, each member has its own memory location but all members of union use the same memory location which is equal to the greatest member's size.

Declaration of union:

The general syntax for declaring a union is:

```
union  union_name
{ data_type   member1;      data_type
              member2;      data_type
              member3;
……….
……….
data_type     memberN;
};
```

We can define a **union** to hold the information of a student as follows:

```
union Student
{ char name[2];
      int  roll;  char  sec;
      float marks;
};
```

union variable declaration: union Student s1, s2, s3; we can combine both template declaration and **union** variable declaration in one statement. Eg, **union** Student

```
{ char name[2];
      int roll; char sec; float marks;
} s1, s2, s3;
```

Differences between structure and unions

1. The amount of memory required to store a structure variable is the sum of sizes of all the members. On the other hand, in case of a union, the amount of memory required is the same as member that occupies largest memory.

```
#include<stdio.h> #include<conio.h>
struct student
{
      int     roll_no;     char
      name[20];
};
union employee
```

```
{
    int ID; char name[20];
}; void main()
{
struct student s; union
employee e;
printf("\nsize of s = %d bytes",sizeof(s));  // prints 22 bytes
printf("\nSize of e = %d bytes",sizeof(e)); // prints 20 bytes
getch();
}
```

Pointers

A pointer is a variable that holds address (memory location) of another variable rather than actual value. Also, a pointer is a variable that points to or references a memory location in which data is stored. Each memory cell in the computer has an address that can be used to access that location. So, a pointer variable points to a memory location and we can access and change the contents of this memory location via the pointer. Pointers are used frequently in C, as they have a number of useful applications. In particular, pointers provide a way to return multiple data items from a function via function arguments.

Pointer Declaration

Pointer variables, like all other variables, must be declared before they may be used in a C program. We use asterisk (*) to do so. Its general form is:

*data-type *ptrvar;*

For example, *int* ptr;* float *q; char *r;

This statement declares the variable *ptr* as a pointer to *int*, that is, *ptr* can hold address of an integer variable.

Reasons for using pointer:

- A pointer enables us to access a variable that is defined outside the function.
- Pointers are used in dynamic memory allocation.
- They are used to pass array to functions.
- They produce compact, efficient and powerful code with high execution speed.
- The pointers are more efficient in handling the data table.

- They use array of pointers in character strings result in saving of data storage space in memory. Sorting stings using pointer is very efficient.
- With the help of pointer, variables can be swapped without physically moving them.
- Pointers are closely associated with arrays and therefore provides an alternate way to access individual array elements.

Pointer initialization:

Once a pointer variable has been declared, it can be made to point to a variable using an assignment statement as follows:

```
int marks; int
*marks_pointer;
Marks_pointer=&marks;
```

Passing (call) by Value and Passing (call) by Reference Arguments can generally be passed to functions in one of the two ways:

- Sending the values of the arguments (pass by value)
- Sending the addresses of the arguments (pass by reference)

Pass by value: In this method, the value of each of the actual arguments in the calling function is copied into corresponding formal arguments of the called function. With this method the changes made to the formal arguments in the called function have no effect on the values of actual arguments in the calling function. The following program illustrates 'call by value'.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a,b;
void swap(int, int );
clrscr();
a = 10; b = 20;
swap(a,b);
    printf("a = %d\tb = %d",a,b);
    getch();
}
```

```
void swap(int x, int y)
{
        int t;
        t = x; x = y; y
        = t;
        printf("x = %d\ty = %d\n",x,y);
}
```

The output of the above program would be *x = 20*
y = 10 a = 10 b = 20

Note that values of **a** and **b** are unchanged even after exchanging the values of x and y.

Pass by reference: In this method, the addresses of actual arguments in the calling function are copied into formal arguments of the called function. This means that using these addresses we would have an access to the actual arguments and hence we would be able to manipulate them. The following program illustrates this fact.

```
#include<stdio.h>
#include<conio.h>
void main()
{ int a,b;
void swap(int*, int*);
        clrscr();
a = 10; b = 20;
swap(&a,&b);
printf("a = %d\tb = %d",a,b);
        getch();
}
void swap(int *x, int *y)
{
        int t; t = *x; *x
        = *y; *y = t;
        printf("x = %d\ty = %d\n",*x,*y);
}
```

The output of the above program would be *x = 20*
y = 10 a = 20 b = 10

Note: We can use call by reference to return multiple values from the function.

Pointers and Arrays

An array name by itself is an address, or pointer. A pointer variable can take different addresses as values. In contrast, an array name is an address, or pointer, that is fixed.

Pointers and One-dimensional Arrays

In case of one dimensional array, an array name is really a pointer to the first element in the array. Therefore, if x is a one-dimensional array, then the address of the first array element can be expressed as either &x[0] or simply x. Moreover, the address of the second array element can be expressed as either &x[1] or as (x+1), and so on. In general, the address of array element (x+i) can be expressed as either &x[i] or as (x+i). Thus we have two different ways to write the address of any array element: we can write the actual array element, preceded by an ampersand; or we can write an expression in which the subscript is added to the array name.

Since, &x[i] and (x+i) both represent the address of the ith element of x, it would seem reasonable that x[i] and *(x+i) both represent the contents of that address, i.e., the value of the ith element of x. The two terms are interchangeable. Hence, either term can be used in any particular situation. The choice depends upon your individual preferences. For example,

/* Program to read n numbers in an array and display their sum and average */

```
#include<stdio.h>
#include<conio.h>
#define SIZE 100
void main()
{
        float a[SIZE],sum=0,avg;
        Int n,i;
        printf("How many numbers?");
        scanf("%d",&n);
        printf("Enter numbers:\n");
        for(i=0;i<n;i++)
        {
                scanf("%f",(a+i)); // scanf("%f",&a[i]);
                sum=sum+*(a+i); //sum=sum+a[i];
        }
```

```
    avg=sum/n;
    printf("Sum=%f\n",sum);
    printf("Average=%f",avg);
    getch();
}
```

/* using pointer write a program to add two 3 × 2 matrices and print the result in matrix form */

```
#include<stdio.h>
#include<conio.h>
#define ROW 3
#define COL 2
void main()
{
int a[ROW][COL],b[ROW][COL],i,j,sum;
    clrscr();
    printf("Enter elements of first matrix:\n");
    for(i=0;i<ROW;i++)
    { for(j=0;j<COL;j++)
    scanf("%d", (*(a+i)+j));
        printf("\n");
    }
    printf("Enter elements of second matrix:\n");
    for(i=0;i<ROW;i++)
    { for(j=0;j<COL;j++)
    scanf("%d", (*(b+i)+j));
        printf("\n");
    }
    printf("Addition matrix is:\n");
    for(i=0;i<ROW;i++)
    {   for(j=0;j<COL;j++)
        {
            sum = *(*(a+i)+j)+*(*(b+i)+j);
            printf("%d\t",sum);
        }
        printf("\n");
```

```
}
getch(); }
```

/*Sum of two matrix using dynamic memory allocation*/

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
void read(int**, int, int);
void write(int**, int, int);
void main()
{
        int **a; int **b;
        int **s;
        int r,c,i,j; clrscr();
        printf("Enter no of row and columns of a matrix\n");
        scanf("%d%d",&r,&c);
        for(i=0;i<r;i++)
        {
            *(a+i)=(int*)malloc(sizeof(int)*c);
            *(b+i)=(int*)malloc(sizeof(int)*c);
            *(s+i)=(int*)malloc(sizeof(int)*c);
        }
        printf("Enter elements of first matrix");
        read(a,,r,c);
        printf("Enter elements of Second matrix");
        read(b,,r,c);
        for(i=0;i<r;i++) {
          for(j=0;j<c;j++)
          {
           *(*(s+i)+j)=*(*(a+i)+j)+*(*(b+i)+j);
           }
        }
        printf("Matrix A is:\n\n");
        write(a,r,c);
        printf("Matrix B is:\n\n");
```

```
        write(b,r,c);
        printf("Sum ofmatrix A and B is:\n\n");
        write(s,r,c); getch();
}
void read(int **x,,int r,int c)
{       int i,j;
        for(i=0;i<r;i++)
        {
                for(j=0;j<c;j++)
                        { scanf("%d",*(x+i)+j);
                }
        }
}
void write(int**y,int r,int c)
{
        int i,j;
        for(i=0;i<r;i++)
        {
                for(j=0;j<c;j++)
                {
                        printf("%d\t",*(*(y+i)+j));
                }
                printf("\n");
        }
}
```

Algorithm

- Concept and definition
- Design of algorithm
- Characteristic of algorithm
- Big O notation

Algorithm:

An algorithm is a precise specification of a sequence of instructions to be carried out in order to solve a given problem. Each instruction tells what task is to be done. There should be a finite number of instructions in an algorithm and each instruction should be executed in a finite amount of time.

Properties of Algorithms:

- **Input:** A number of quantities are provided to an algorithm initially before the algorithm begins. These quantities are inputs which are processed by the algorithm.
- **Definiteness:** Each step must be clear and unambiguous.
- **Effectiveness:** Each step must be carried out in finite time.
- **Finiteness:** Algorithms must terminate after finite time or step
- **Output:** An algorithm must have output.
- **Correctness:** Correct set of output values must be produced from the each set of inputs.

Write an algorithm to find the greatest number among three numbers:

Step 1: Read three numbers and store them in X, Y and Z

Step 2: Compare X and Y. if X is greater than Y then go to step 5 else step 3

Step 3: Compare Y and Z. if Y is greater than Z then print "Y is greatest" and go to step 7 otherwise go to step 4

Step 4: Print "Z is greatest" and go to step 7

Step 5: Compare X and Z. if X is greater than Z then print "X is greatest" and go to step 7 otherwise go to step 6

Step 6: Print "Z is greatest" and go to step 7

Step 7: Stop

Lab work No:1

#Write a program using user defined functions to sum two 2-dimensional arrays and store the sum of the corresponding elements into third array and print all the three arrays with its values at the corresponding places (not in a single row), get the transpose of the third matrix and print

```
#include<stdio.h> #include<conio.h>
void read(int[10][10], char, int, int);
```

```
void write(int[10][10],int , int);
void main()
{
        int a[10][10];
        int b[10][10];
        int s[10][10];
        int t[10][10];
        int r,c,i,j;
        clrscr();
        printf("Enter no of row and columns of a matrix\n");
        scanf("%d%d",&r,&c);
        read(a,'a',r,c);
        read(b,'b',r,c);
        for(i=0;i<r;i++)
        {
          for(j=0;j<c;j++)
          {
                s[i][j]=a[i][j]+b[i][j];
           }
        }
        printf("Matrix      A      is:\n\n");
        write(a,r,c);
        printf("Matrix      B      is:\n\n");
        write(b,r,c);
        printf("Sum of matrix A and B is:\n\n");
        write(s,r,c);
        for(i=0;i<r;i++)
        {
          for(j=0;j<c;j++)
          { t[i][j]=s[j][i];
           }
        }
        printf("Transpose of tesultant matrix is:\n");
        write(t,r,c); getch();
}
```

```
void read(int x[10][10],char ch,int r,int c)
{
  int i,j;
  printf("Enter elements of a matrix[%c]\n",ch);
for(i=0;i<r;i++)
  {
          for(j=0;j<c;j++)
          { scanf("%d",&x[i][j]);
    }
  }
}

void write(int y[10][10],int r,int c)

{
 int i,j;
 for(i=0;i<r;i++)
 {
   for(j=0;j<c;j++)
   {
     printf("%d\t",y[i][j]);
   }
   printf("\n");
 }
}
```

Lab work No:2

Write a program using an array to perform the following tasks:(use switch case for menu)

a. Insert element into an array at specified position

b. Delete element from an array

c. Traversing

d. Searching a particular element in the array

```
#include<stdio.h>
#include<conio.h>
void insert(int [100], int*);
```

```
void delet(int [100], int*);
void traverse(int [100], int*);
void searching(int [100], int*);
void main()
{
        int a[100],nel,pos,i;
        int n;
        int choice;
        printf("Enter no of elements to be inserted");
        scanf("%d", &n);
        printf("Enter  %d  elements",  n);
        for(i=0;i<n;i++)
        { scanf("%d", &a[i]);
        } do
        {       printf("\nmanu for program:\n");
                printf("1:insert\n2:delete\n3:Traverse\n4:searching\n5:exit\n");
                printf("Enter your choice\n");
                scanf("%d",&choice);
                switch(choice)
                {
                        case 1:
                        insert(a,&n);
                        break;
                        case 2:
                                delet(a,&n);
                                break;
                        case 3:
                                traverse(a,&n);
                                break;
                        case 4:
                                searching(a,&n);
                                break;
                        case 5:
                                exit(1);
                                break;
```

```
                default:
                        printf("Invalied choice");
            }
        }while(choice<6);
    }

    void insert(int a[100], int *n)
    {
        int pos, nel, i;
        printf("Enter position at which you want to insert new element");
        scanf("%d", &pos);
        printf("Enter new element");
        scanf("%d", &nel);
        for(i=*n-1; i>=pos; i--)
        {
            a[i+1] = a[i];
        }       a[pos]=nel;
        *n=*n+1;
        printf("New array is:\n");
        for(i=0; i<*n; i++)
        {
            printf("%d\t", a[i]);
        }
     }

void delet(int a[100], int *n)
{
    int pos, i;
    printf("Enter position at which you want to delete an element");
    scanf("%d", &pos);
    for(i=pos; i<*n; i++)
    {
        a[i] = a[i+1];
    }
    *n=*n-1;
    printf("New array is:\n");
```

```
    for(i=0; i<*n; i++)
    {
        printf("%d\t", a[i]);
    }
}

    void traverse(int a[100], int *n)
    {
    int i;
    printf("Elements of array are:\n");
    for(i=0;i<*n;i++)
    {
        printf("%d\t",a[i]);
    }
 }

    void searching(int a[100], int *n)
    {
        int k,i;
        printf("Enter searched item");
        scanf("%d",&k);
        for(i=0;i<*n;i++)
        {
            if(k==a[i])
            { printf("******successful search******"); break;
            }
        } if(i==*n)
        printf("**********unsuccessful search*********");
   }
```

Lab work No:3

Write a program to multiply two m*n matrices using dynamic memory allocation. Hint c[i][j]+=a[i][k]*b[[k][j];

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

```
void read(int**,int,int);
void write(int**,int,int);
void main()
{
        int **a; int **b; int **m;
        int r1,c1,r2,c2,i,j,k;
        printf("Enter no of row and columns of first matrix\n");
        scanf("%d%d",&r1,&c1);
        printf("Enter no of row and columns of second matrix\n");
        scanf("%d%d",&r2,&c2);
        for(i=0;i<r1;i++)
        {
                *(a+i)=(int*)malloc(sizeof(int)*c1);
                *(m+i)=(int*)malloc(sizeof(int)*c2);
        }
        for(i=0;i<r2;i++)
        {
                *(b+i)=(int*)malloc(sizeof(int)*c2);

        }
        printf("Enter elements of first matrix”);
        read(a,r1,c1);
        printf("Enter elements of Second matrix\n”);
        read(b,r2,c2);
        for(i=0;i<r1;i++)
        {
          for(j=0;j<c2;j++)
          {
                *(*(m+i)+j)=0;
                for(k=0;k<c1;k++)
                {

             (*(*(m+i)+j))+=*(*(a+i)+k)*(*(*(b+k)+j));
           }
         }
        }
```

```
        printf("Matrix A is:\n\n");
        write(a,r1,c1);
        printf("Matrix B is:\n\n");
        write(b,r2,c2);
        printf("product of matrix A and B is:\n\n");
        write(m,r1,c2);
        getch();
}
void read(int **x,int r,int c)
{
        int i,j;            for(i=0;i<r;i++)
        {
                for(j=0;j<c;j++)
                        { scanf("%d",*(x+i)+j);
                }
        }
}

void write(int**y,int r,int c)
{
   int i,j;
   for(i=0;i<r;i++)
        {
                for(j=0;j<c;j++)
                {
                        printf("%d\t",*(*(y+i)+j));
                }
                 printf("\n");
        }
}
```

Big Oh (O) notation:

When we have only asymptotic upper bound then we use O notation. A function f(x)=O(g(x)) (read as f(x) is big oh of g(x)) if there exists two positive constants c and x_0 such that for all x >= x_0, f(x) <= c*g(x)

The above relation says that g(x) is an upper bound of f(x)

O(1) is used to denote constants.

Example:

f(x)=5x3+3x2+4 find big oh(O) of f(x)

solution:

f(x)= 5x3+3x2+4<= 5x3+3x3+4x3 if x>0

<=12x3

=>f(x)<=c.g(x) where c=12

and g(x)=x3

Thus by definition of big oh O(f(x))=O(x3)

Big Omega (Ω) notation:

Big omega notation gives asymptotic lower bound. A function f(x) =Ω (g(x)) (read as g(x) is big omega of g(x)) iff there exists two positive constants c and x_0 such that for all x >= x_0, 0 <= c*g(x) <= f(x).

The above relation says that g(x) is a lower bound of f(x).

Big Theta (Θ) notation:

When we need asymptotically tight bound then we use notation. A function f(x) = (g(x)) (read as f(x) is big theta of g(x)) iff there exists three positive constants c_1, c_2 and x_0 such that for all x >= x_0, c_2*g(x) <= f(x) <= c_1*g(x)

The above relation says that f(x) is order of g(x)

Example:

f(n) = $3n^2$ + 4n + 7 g(n) = n^2 , then prove that

f(n) = (g(n)).

Proof: let us choose c_1, c_2 and n_0 values as 14, 1 and 1 respectively then we can have,

f(n) <= c_1*g(n), n>=n_0 as $3n^2$ + 4n + 7 <= 14*n^2 , and f(n) >= c_2*g(n), n>=n_0 as $3n^2$ + 4n + 7 >= 1*n^2 for all n >= 1(in both cases).

So c_2*g(n) <= f(n) <= c_1*g(n) is trivial. Hence f(n) =

Q (g(n)).

Example : Fibonacci Numbers

Input: *n*

Output: n^{th} Fibonacci number.

Algorithm: assume *a* as first(previous) and *b* as second(current) numbers fib(n)

```
{
        a = 0, b= 1, f=1 ;
        for(i = 2 ; i <=n ; i++)
        {
                f = a+b ; a=b ;
                b=f ;
        }
        return f ;
}
```

Efficiency:

Time Complexity: The algorithm above iterates up to n-2 times, so time complexity is O(n).

Space Complexity: The space complexity is constant i.e. O(1).

Example : Bubble sort

Algorithm

```
BubbleSort(A, n)
{
        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;
                    }
                }
            }
}
```

Time Complexity:

Inner loop executes for (n-1) times when i=0, (n-2) times when i=1 and so on:

Time complexity = (n-1) + (n-2) + (n-3) + ……. +2 +1

= $O(n^2)$