Discipline Courses-I
Semester-I
Paper: Programming Fundamentals
Unit-III
Lesson: Arrays
Lesson Developer: Harita Mehta
College/Department: Acharya Narendra Dev College, University of Delhi

Table of Contents

Introduction

Applications may demand to store group of similar data items which are least expected to change with time. Such groups of similar data items are preferred to be stored in the form of an **Array** Data type. For example, marks of students, salary of employees, and so on. An array is a collection of same type of data items which are stored in consecutive memory locations under a common name. The array name is followed by pairs of square brackets. An integer number written inside these square brackets is the size of array i.e. number of elements that this array can store. If a single pair of square brackets is present after the array name, then it is called a **one dimensional array**. If two pairs of square brackets are present after the array name, then it is called a **two dimensional array**. A **matrix** is a two dimensional array written in the form of rows and columns. In this case, the size of the array is the product of two integers mentioned inside two pairs of square brackets. Similarly, three dimensional arrays, four dimensional arrays and so on exist. In general, arrays with dimension greater than one are called **multi dimensional arrays**. They can be regarded as array of arrays. But these higher dimensional arrays are rarely encountered. In this chapter, you will learn how to declare, initialize, traverse and print the elements of one dimensional array, two dimensional array and n dimensional arrays. The chapter focuses on various operations that can be performed on these arrays. Further, light is thrown on how data in a multi dimensional array is stored in a computer memory which is essentially one dimensional in nature.

Computer systems are often used to store large amount of data from which specific information must be retrieved according to some search criterion. **Searching** refers to the operation of finding the location of the given data item in a collection of items such as arrays. The search is said to be successful or unsuccessful depending on whether the element that is to be searched is found or not. The problem of searching may be linked to real world situations. For example, let us consider a telephone directory which contains names address and telephone numbers of a large number of people. Given a name, the task here can be to find out the telephone number and address of that particular person. Various searching algorithms are available. Actually, in order to facilitate fast searching, the efficient storage of data is an important issue. If the data is stored in an array, then efficient searching is obtained by well known search techniques namely, **Linear Search** and **Binary Search**. This chapter covers these search techniques in detail. Further, performance of these search techniques is discussed.

In computer science, sorting is one of the major operations regularly used in almost all the applications. **Sorting** is an arrangement of data items in a sequential order according to an ordering criterion. For Example, if we work with a student file, we might be interested in sorting the student names in alphabetical order. Ordering or sorting data in an increasing or decreasing fashion according to some linear relationship among data items is of fundamental importance. For example, efficient sorting is important in optimizing the use of other algorithms such as search algorithms viz. binary search works on sorted array elements only. Actually, retrieval of information is made easier when the data is stored in some predefined order. Various sort techniques are available. This chapter covers **Bubble sort, Selection sort** and **Insertion sort** in detail. Further, performance of these sorting algorithms is discussed.

Arrays can be passed as argument in a function call by use of one of the three methods available viz. **call by reference**, **call by value** and **input/output argument**. It can also be returned from a function. The chapter explains this concept by use of various examples. Finally, the chapter focuses on multi dimensional arrays in detail.

Learning Objective

After reading this chapter you should be able to:

- Declare, initialize and traverse one dimensional arrays
- Use numeric and character arrays
- Use static and dynamic arrays
- Perform various operations on one dimensional arrays
- Declare, initialize and print elements of a two dimensional array
- Data storage forms of a two dimensional array
- Common operations on a matrix
- Searching an array for a value using
    - Linear Search
    - Binary Search
- Understand the importance of sorted data.
- Categories of sort algorithms
- Concept of Bubble sort, Selection sort and Insertion sort
- Analysis of these sorts
- Passing array as an argument in a function call using call by value, call by reference and input/output argument forms.
- Returning an array from a function
- Declare, initialize and print the elements of a multi dimensional array
- Data storage patterns in multi dimensional arrays

7.1 One Dimensional Array

7.1.1 Fundamentals of One Dimensional Arrays

A Data Structure is a way of organizing data that specifies:-

1. A set of data elements i.e. a data object and
2. A set of operations which may be legally applied to the elements of this data object.

For Example: The data object integers along with description of how arithmetic operations like +,-,x,/ can be applied on it, constitute a data structure definition.

In any programming language, a data structure is known as a data type for defining variables of that type in order to create many instances. C++ has rich collection of data types. These data types can be broadly classified as:

1. Primary Data Types
2. Secondary Data Types

C++ Data Types	
Primary Data Types	**Secondary Data Types**
Char	**Array**
Int	**Pointer**
Float	**Structure**
Double	**Union**
Void	**Enum**

Table 7.1 Data Types in C++

One of the commonly used secondary data type is an array. An **array** is a collection of same type of data items which are stored in consecutive memory locations under a common name. Individual array elements are identified by an integer index. The number of elements in the array is called its **dimension or size or length**. In C++, the index begins at zero and is always written inside square brackets. For example: If A is the name of an array then its elements are accessed using A[0], A[1], A[2] etc.

A One dimensional array has the following components:

1. a name
2. a type: this is the data type of all array elements.
3. an extent: this is the range of the indices of array elements.

Indices must be integers within the range. The smallest and the largest indices are referred as the lower bound (LB) and the upper bound (UB), respectively.

An Array size or dimension is calculated by the formula:

Dimension = UB - LB+1

For Example, given an array A[5],
then

UB = 4,
LB = 0 and
Dimension = UB - LB + 1
= 4 -0+1
= 5

The array representation is as follows:

A[0]	A[1]	A[2]	A[3]	A[4]
ML1	ML2	ML3	ML4	ML5

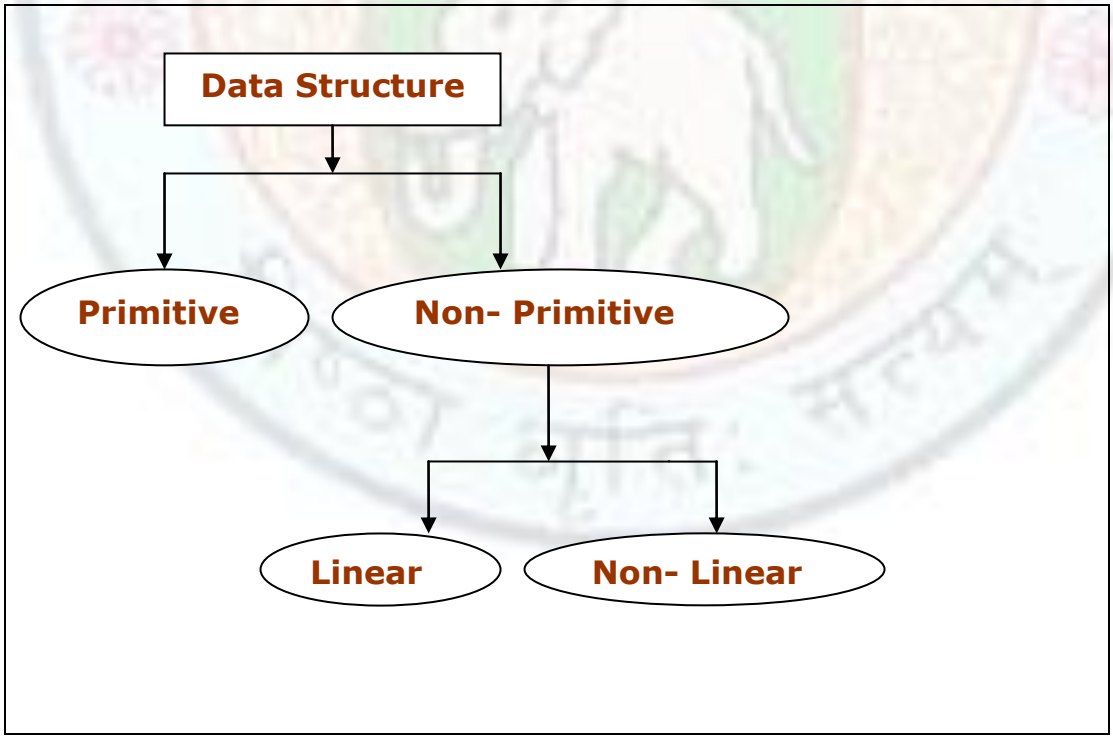
A[i] = array element
ML = memory location
(In C++, array index starts from 0, so dimension = 5 means that the size of the array is 5 staring from A[0] to A[4] as shown in above figure.)

Value addition: Common Misconception

Heading text: Data Structure

Body text:

- A **Primitive Data Structure** defines a set of primitive elements which don't involve any other elements as its subparts. For example, int, char.
- On the other hand, a **Non Primitive Data Structure** defines a set of derived elements. For example, an array which consist of a set of similar type of elements.
- **Linear Data Structure** defines a set of operations which do not create hierarchical structures among the elements of its data object. For example, array.
- **Non linear hierarchical** Data Structure defines such a set of operations which create a hierarchical structure among the elements of its data object. For example, trees, graphs etc.



Source: Self made

7.1.2 Declaration of One Dimensional Arrays

The syntax template of a one dimensional array declaration is:

Datatype ArrayName [Constant Integer expression];

In the syntax template, data type indicates the type of data stored in each component of the array. Constant Integer Expression indicates the size of the array declared. It must have a value greater than O. If the value is n, then the range of the index values is 0 to n-1.
For example,

```
int Number[3];
```

creates an array called Number that has a total of four elements, all of integer type.
ie, Number[0] specifies the first element of Number array
Number[1] specifies the second element of Number array
Number[2] specifies the third element of Number array
Number[3] specifies the fourth element of Number array

7.1.3 Initialization of One Dimensional Arrays

Assigning values to the array elements is called initialization of arrays. There are 2 methods of initialization

1. Array initialization after declaration.
2. Array initialization during declaration.

Array initialization after declaration:

Suppose we need to store following values in Number array declared above.

6	8	10	12

The code is as follows:

```
Number [0] = 6
Number [1] = 8
Number [2] = 10
Number [3] = 12
```

You can also use a for loop for the above initialization as follows:

```
int i, j;
j = 6;
for ( i=0, i<4, i++) // loop to traverse the array.
{
      Number[i] = j; // assigning values to array elements.
      j = j+2;         // increment j by 2
}                      // end of loop
```

Array initialization during declaration:

A variable can be initialized in its declaration.
For example.

```
int num = 25;
```

here the value 25 is called an initializer.

Similarly, an array can also be initialized in its declaration. A list of initial values for array elements can be specified. They are separated with commas and are enclosed within braces. For example,

```
int Number[3] = {6,8,10,12};
```

This code creates the Number Array as:

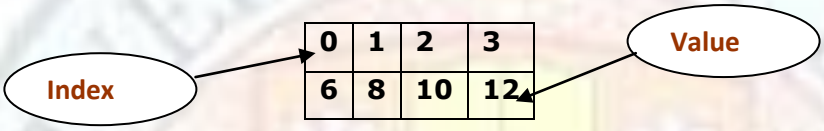


Figure 7.1 Array Initialization during declaration

Please Note:

1. There must be at least one initial value between braces.
2. If the number of initial values is less than the array size, then the remaining array elements will be initialized to zero.
3. In C++ the array size can be omitted when it is initialized during declaration.
   For example.
   `int Number[] = {6,8,10,12};`
   The compiler determines the size of Number array according to how many initial values are listed. Here the size of the Number array will be set to 4.

7.2 Printing Elements of a One Dimensional Array

The values of an array can be printed using cout statement in a for loop.
For example. in order to print the values of Number array you can write the following code:

```
for (i=0; i < 4 ; i++)
{
      cout<< "Number["<<i<<"] : "<<Number[i]<<"\n";
}
```

The output of above code will be:

```
Number[0] : 6;
Number[1] : 8;
Number[2] : 10;
Number[3] : 12;
```

7.3 More About One Dimensional Array

7.3.1 One Dimensional Character Array

Character array is a linear Data Structure which is used to store strings. For example.

char A[5];

stores maximum of 5 characters including NULL (\0) starting from 0^{th} location to 4^{th} location.

A string is always terminated at NULL character. Each location in an array A[] takes 1 byte of memory in C++. Suppose we need to store "AMIT" then the initialization is as follows:

char A[5] = {'A', 'M', 'I', 'T' '\0'};

The memory representation will be

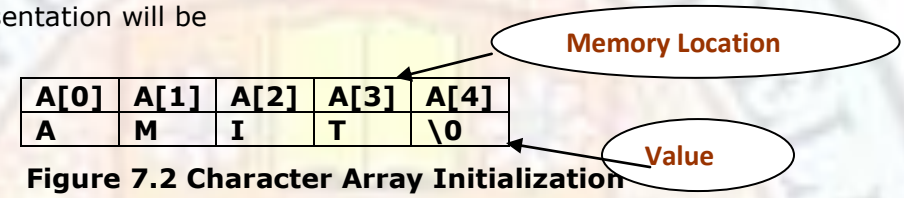
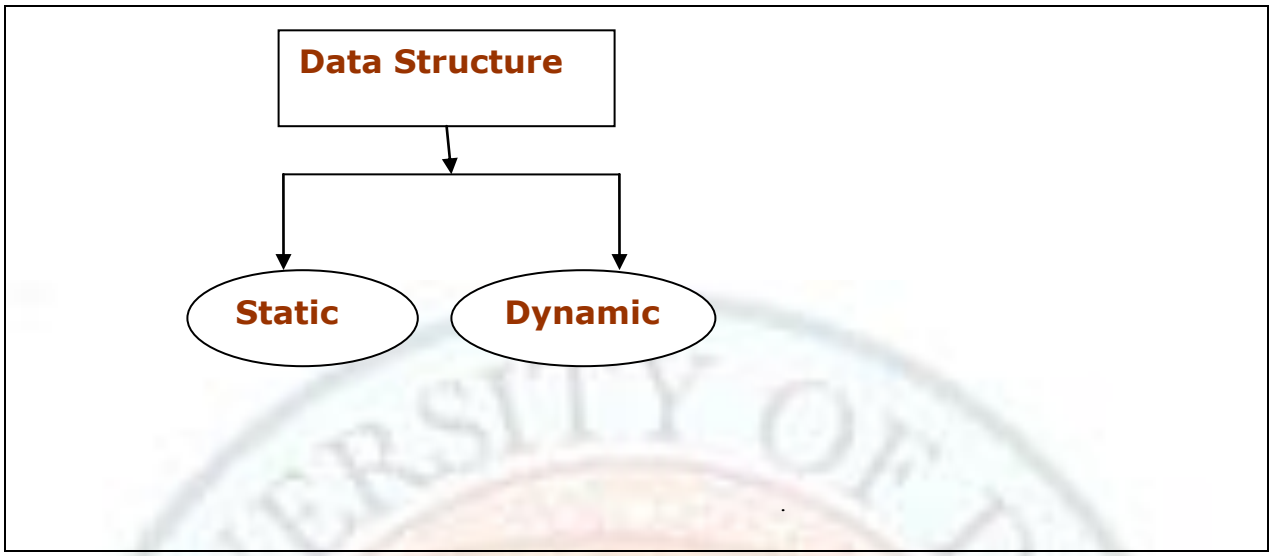


Figure 7.2 Character Array Initialization

It can also be initialized as: Char A[] ="AMIT";

Both the statements are equivalent & NULL is automatically inserted by the second type of initialization. Therefore, any string inserted in double quotes is treated to be a string terminated by a NULL character.

Value addition: Intresting Fact
Heading text: Dynamic Data Structures
Body text: An array can be static or dynamic in nature. • A Data Structure is referred to as **Static Data Structure** if it is created during compilation time. • A Data Structure is referred to as **Dynamic Data Structure** if it is created at run time.



7.3.2 One Dimensional Dynamic Arrays

The Number array that is declared is our previous example is a static array because it is allocated at compile time. The dimension must be a constant integer. You can also declare the Number array as a dynamic array as below. The new operator is used to create a dynamic array whose dimension can be a integer variable. These Dynamic Arrays are allocated at run time. The name of a dynamic array is a pointer to its element type. The syntax for declaring a dynamic array is:

```
Datatype* name of array = new datatype [n];
```

Where
Datatype is the element type,
n is the dimension of the array acquired at runtime.

Example 7.1:

Let us declare a dynamic array in order to store the age of user's children.

```
#include <iostream>
using namespace std;

int main()
{
        int n,i;
        cout << "How many children do you have ?";
        cin>>n;
        int* child  = new int[n]; //allocating memory for array child
        cout << "Please give me the age of your "<<n<<" children\n";
        for (i=0;i<n;i++) // accept values from user
        {
                cout<<"\t"<<i+1<<":";
                cin>>child[i];
        }
        cout<<"The age is\n";
```

```
        for (i=0;i<n;i++) //display array elements
        {
                cout<<"child["<<i+1<<"] = "<<child[i]<<"\n";
        }
        return 0;
}
```

Output:
How many children do you have? 4
Please give me the age of your 4 children;
1: 12
2: 15
3: 17
4: 19
The age is
child[1] = 12
child[2] = 15
child[3] = 17
child[4] = 19

Value addition: Warning

Heading text: Beware

Body text:

The C++ compiler will not flag an error if the array index goes out of bounds.
For ex.

```
int A[3]; //array declaration.
A[5] = 2; //index out of range.
```

This is perfectly legal, but will lead to unpredictable behavior.

7.3.3 Address Calculation of One Dimensional Arrays

Each array element possesses an address after memory is allocated to an array. Address of any element of the array can be calculated if we know the starting address (also called **base address**) and the datatype of array elements.

Let 'k' be an integer such that Lower Bound (LB) < k < Upper Bound (UB), and Base(A) denotes the address of first element of the array 'A' then the address of k^{th} element of this array can be calculated by the formula:

Address of A(k) = Base(A) +w(k)

Where
Base(A) = starting address/ address of first array element
w= no of memory location per array element i.e. size of datatype
for a character array 'w' = 2 where as for an integer array 'w' = 1.

Example 7.2:

Let us consider an integer array A[6]. If the base add of A is 100, then calculate the address of 5^{th} element of this array considering w = 1.

Solution:
We know,

Address of A(k) = Base(A) +w(k)

here

Base(A) = 100
w= 1
k = 4 (position 4 of the array holds A[5] element since array indexing starts from 0.)

so,

Address of A(4) = 100 + 1*4
= 104

Value addition: Common Misconception
Heading text: Need for Initialization
Body text: Before initialization of an array, the values stored in the elements are not zero or NULL in case of numeric and character arrays respectively. They are unknown and are referred to as garbage values. So initialization of array elements is necessary after declaration.

7.4 Operations on One Dimensional Arrays

7.4.1 Basic Operations

Fundamental operations on an array are:

1. Traversing: It refers to accessing / processing each element of an array.
2. Insertion: It refers to the operation of adding new element to an array.
3. Deletion: It refers to removal of an element from an array.

Example 7.3:

Problem statement:

Program to insert an integer data item in a given array of elements in required position between 0 to n-1.

Logic:

The logic behind this is to copy the nth data element to n+1location, n-1data element to nth location, n-2 to n-1 location. This procedure is repeated till the position of insertion is reached. Then at this position, the new data is inserted.
For example, if we want to insert the new item in the second place then copying of data from 5^{th} pos to 6^{th} , 4^{th} to 5^{th} , 3^{rd} to 4^{th} & 2^{nd} to 3^{rd} will make the data to move to the right side. Thus, the process creates space for inserting new item. Now in the 2^{nd} pos, new item is placed. The whole process can be explained diagrammatically as follows:

Problem Statement:

Index 0	Index 1	Index 2	Index 3	Index 4
Position 1	Position 2	Position 3	Position 4	Position 5
29	38	79	56	43

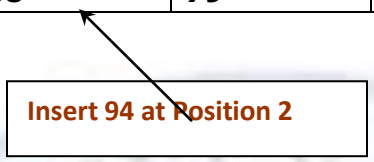


Figure 7.3 Insertion operation in a One dimensional array

Logic:
Step 1: Create an array with 6 positions instead of 5 positions.
Step 2: Initialize the array elements and insert 0 at last position.
Step 3: Move the element to the right side in order to get position 2 empty.
Step 4: Insert 94 at position 2.

Index 0	Index 1	Index 2	Index 3	Index 4	Index 5
Position 1	Position 2	Position 3	Position 4	Position 5	Position 6
29	38	79	56	43	0

Output:

Index 0	Index 1	Index 2	Index 3	Index 4	Index 5
Position 1	Position 2	Position 3	Position 4	Position 5	Position 6
29	94	38	79	56	43

Figure 7.4 Steps in Insertion of an element in a one dimensional array

7.4.2 Other Operations

Finding element with maximum value in an array

```
#include <iostream>
using namespace std;

int main()
{
  int a[7] ={44,77,33,66,55,88,22};  // array initialization
  int max = a[0]; //set first element as maximum element
 for(int i=1; i<7;i++)
 {
       if (a[i]>max)  //comparison
```

```
            max = a[i]; //assignment
  }
  cout<<" Element with maximum value in the array is "<<max;

  }
```

Reverse the array elements

```
#include <iostream>
using namespace std;

int main()
{
  int a[7] ={44,77,33,66,55,88,22}; //array initialization
  int temp;
  for(int i=6, j=0;j<3; i--,j++)
  {
        //swap first and last element.
        // second and second last element and so on till middle element is reached.
        temp=a[i];
        a[i]=a[j];
        a[j]=temp;
  }
  cout<<" The reversed array is : \n";
 for(int i=0;i<7;i++) //display array elements
  {
        cout<<a[i]<<"\n";
  }
}
```

7.5 Advantages and Disadvantages of One Dimensional Arrays

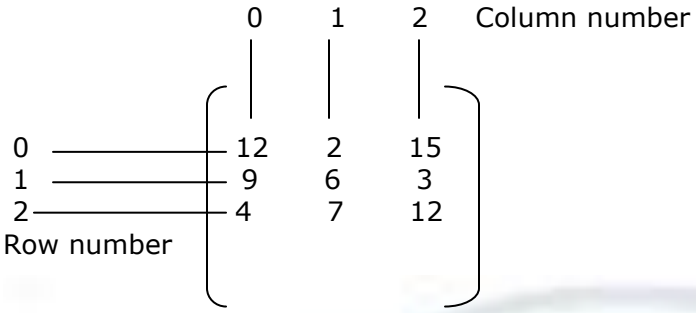
Advantages

1. Array locations are easily accessible using index.
2. Array is a useful data structure to store relatively permanent data which needs to be accessed contiguously.
3. Address calculation is very easy since the memory is allocated sequentially.
4. Operation like sorting, merging, traversing and searching are easy with arrays.

The major disadvantage is that it is relatively time consuming to insert and delete elements in an array because one needs to shift elements in order to create or remove space.

7.6 Two Dimensional Array

The **matrix** in mathematics is a two dimensional array written in the form of rows and columns. The element of the matrix is indicated by the row and column number in which it is located. The element of matrix is detected by writing a[i][j], where, i stands for row subscript, j stands for column subscript and 'a' is the name of the matrix.



The element at row 1 and column 2 is traced by supplying the values of the subscript as 1 and 2 respectively.

7.6.1 Declaration of Two Dimensional Arrays

The syntax template of a two dimensional array declaration is:

Datatype arrayName [constant Integer Expression 1][constant Integer Expression 2];

In the syntax template, data type discusses indicates the type of data of each component of the array, constant Integer Expression 1 indicates the number of rows of the array and constant Integer Expression 2 indicates the number of columns of the array. These must have a value greater than 0. If the value is n and m respectively, then the range of the index values for array rows is 0 to n-1 and for array columns is 0 to m-1. For example, consider an integer array as declared below

int a[4][3];

- By convention, the first subscript is understood to be for rows and the second for columns.
- It is a matrix of order 4*3 i.e. number of rows = 4 and number of columns = 3.
- This array can hold up to 4*3 = 12 integer values.
- Subscripts start with 0, so the subscripts range from 0 to 3 for the rows and 0 to 2 for columns.
- The following figure illustrates how the subscripts are specified for this array.

	0	**1**	**2**
0	**[0][0]**	**[0][1]**	**[0][2]**
1	**[1][0]**	**[1][1]**	**[1][2]**
2	**[2][0]**	**[2][1]**	**[2][2]**
3	**[3][0]**	**[3][1]**	**[3][2]**

Figure 7.5 Subscript Notation

- Following figure shows this matrix with sample assignment of values

10	**18**	**42**
13	**19**	**8**
63	**80**	**12**

16	13	9

Figure 7.6 Two dimensional array with sample data

Within main memory, storage locations are not arranged in the grid like pattern of fig 1.2. Instead, they are arranged in linear sequence beginning with location 0,1,2,3,4,……and so on. Fig 7.7 (a) and 7.7 (b) reflects two forms of data storage. Because of this, there must be manipulation behind the scene when a program requests an entry of a two dimensional array. This manipulation is called **transformation or mapping** of a two dimensional array elements to linear storage.

10	18	42	13	19	8	63	80	12	16	13	9

Figure 7.7 (a): linear storage of data from figure 7.6 (Row Major Form)

10	13	63	16	18	19	80	13	42	8	12	9

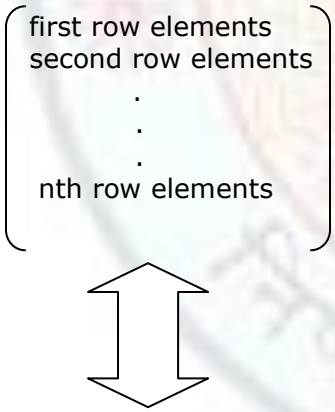
Figure 7.7 (b): another type of linear storage of data from figure 7.6 (Column Major Form)

7.6.2 Data Storage Forms in a Two Dimensional Array

Actually, the memory of the computer is essentially one dimensional. There are two ways of storing data in a two dimensional array.

- row major form
- column major form

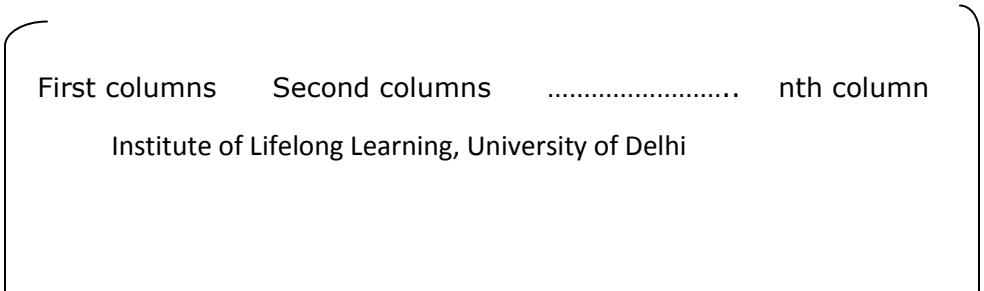
For a two dimensional array, storage in **row major form** implies that, elements of a row are stored contiguously. In a row major form the transformation or mapping of a two dimensional array elements to linear storage would appear as in figure 7.8 (a).



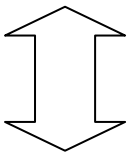
First row elements	Second row elements	……	……	nth row elements

Figure 7.8 (a): Mapping in row major form

Similarly, for a two dimensional array, storage in **column major form** implies that elements of a column are stored contiguously. Here, the transformation or mapping of a two dimensional array elements into linear storage would appear as in figure 7.8 (b)



elements elements elements



First column elements	Second column elements	……	…….	nth column elements

Figure 7.8 (b): Mapping in column major form

Value addition: Did You Know

Heading text: Two Dimensional Arrays….! Less Efficient

Body text:

All programmers should be aware that multidimensional arrays are inherently less efficient that one dimensional array because of the computations required by the transformations from row and column to linear address each time an entry of the array is accessed. Such a transformation is called a mapping function.

Based on these data storage patterns, there are two ways of accessing data from a two dimensional array.
In order to access the entry in the ith row and jth column of a two dimensional array stored in row major form, the following transformation is required.

[(Ncol * i) + j] where

Ncol represents the number of columns in the two dimensional array.

Example 7.4

A two dimensional array a[4][3] is stored in row major form. Then in order to access element a[2][1] from the array, following mapping is required. Clearly,
number of columns in array 'a' = Ncols = 3
i = 4
j = 3
So, to access element a[2][1] the transformation = (Ncol * i) + (j)
= (3*2) + (1)
= 6+1
= 7

The solution can be verified with the figure 7.9 (a)

10 20 30
40 50 60

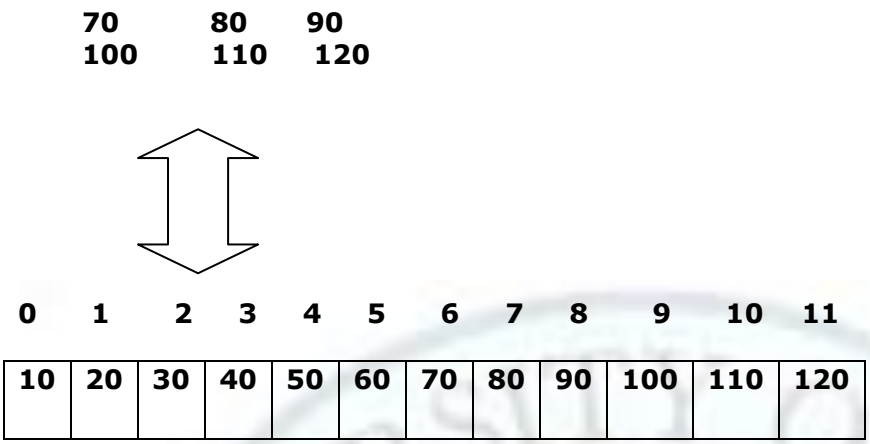


Figure 7.9 (a): Mapping in row major form

Similarly, to access the entry in the ith row and jth column of a two dimensional array stored in column major form, the following transformation is required.

[(Nrow * j) + i] where

Nrow represents number of rows in the two dimensional array.

Example 7.5

Two dimensional array a[4][3] is stored in column major form. Then in order to access element a[2][1] from the array, following transformation or mapping is required. Clearly, number of rows = Nrow =4,
i=2,
j=1
So, to access a[2][1] the transformation = (Nrow * j) + (i)
=(4*1) + (2)
= 6
The solution can be required be verified with the figure 7.9 (b)

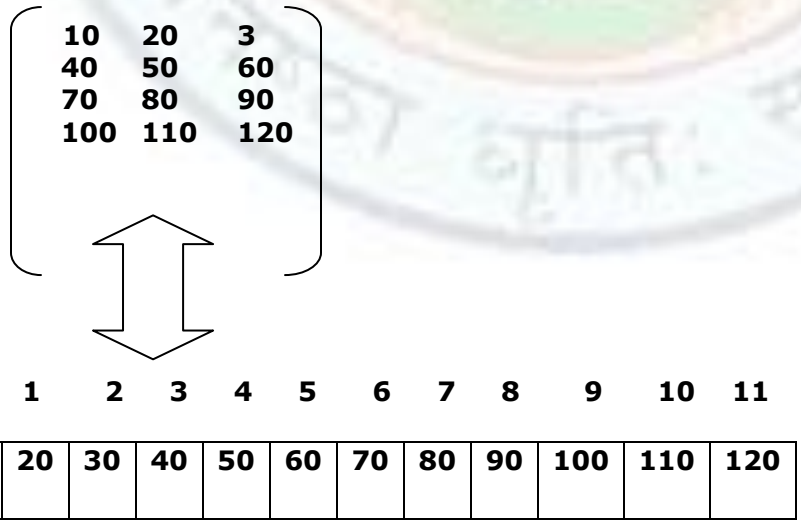


Figure 7.9 (b): Mapping in column major form

Value addition: Interesting Fact

Heading text: Row Major Form Vs. Column Major Form

Body text:

- Most high level computer languages implements 2 and higher dimensional arrays in row major form.
- Fortran in one of few high level computer languages that chooses to store a multidimensional array in column major form.

Address mapping

Each array element is given an address after allocating memory to that array. Address of any element of this array can be calculated if we know the starting address of the array and the data type of array elements. Starting address is also called **base address** of the array and is denoted by Base (a). The data type of array elements relate directly to number of memory locations per array element. It is denoted by w.

For row major form, the address of element a[k1][k2]= Base(a)+W[(Ncol*i)+ j]

For column major form, address of element a[k1][k2]= Base(a)+W[(Nrow*j)+ i]

Value addition: Activity

Heading text: Address Mapping

Body text:

In the example 1.1 and example 1.2, if the Base Address of 'a' is 52722 and each element requires two bytes of memory, then address mapping is as follows

Address mapping in Row major form for element a[2][1] is :-

Address of a[2][1]=Base (a) + W[(Ncol*i)+ j]

=52722 +2[7]

=52736

Address mapping in Column major form for element a[2][1] is :-

Address of a[2][1]=Base (a) + W[(Nrow*j)+ i]

=52722 +2[6]

=52734

7.6.3 Initialization of a Two Dimensional Array

There are two methods of initialization:

Array initialization after declaration:-

Suppose we need to store following values in Number [3][4] array

$$\begin{pmatrix} 3 & 6 & 9 & 12 \\ 15 & 18 & 21 & 2 \end{pmatrix}$$

27 30 33 36

Code for this will be as follows

```
Number [0][0] = 3;
Number [0][1] = 6;
Number [0][2] = 9;
Number [0][3] = 12;
Number [1][0] = 15;
Number [1][1] = 18;
Number [1][2] = 21;
Number [1][3] = 24;
Number [2][0] = 27;
Number [2][1] = 30;
Number [2][2] = 33;
Number [2][3] = 36;
```

One can also make use of two for loops for above initialization as follows:

```
int i,j,x;
x = 3;
for (i=0;i<3;i++)  \\ row index
  for(j=0;j<4;j++)  \\ column index
    {
       Number[i][j]=x;  \\ value assignments
       x =x+3;
     }
```

Array Initialization during declaration :-

A list of initial values for array elements can be specified during declaration. They are separated with commas and also are enclosed within braces. The elements in each pair of curly braces are elements of a row and are in sequence of columns. For Example,

```
int Number[3][4]= { {3,6,9,12},
                    {15,18,21,24},
                    {27,30,33,36}
                  };
```

7.6.4 Printing Elements of a Two Dimensional Array

The values of a two dimensional array can be printed using "cout" statement in nested for loops. The usual format is:

```
for each row
   for each column
     do   something to array [row][column]
```

Figure 7.10 Usual format of a two Dimensional Array

For Example, in order to print the values of Number array following code is required:-

```
int number[3][4]={{3,6,9,12,15,18,21,24,27,30,36}};
for(i=0;i<3;i++)
```

```
for(j=0;j<3;j++)
   cout<<number[i][j]<<\t;
cout<<endl;      \\ new line at end of each row
```

the output of above code will be:

3	6	9	12
15	18	21	24
27	30	33	36

7.7 More About Two Dimensional Arrays

7.7.1 Two Dimensional Character Array

Two Dimensional character array:-

The two dimensional character array can be declared & initialized as follows

char c[3][6]={"Suresh" ,"Ramesh", "Anita" };

This indicates that c is a two dimensional character array having memory of 6 characters each and 3 string are supplied where

C[0] string is Suresh
C[1] string is Ramesh
C[2] string is Anita

7.7.2 Two Dimensional Dynamic Array

1. Dynamic allocation of arrays of more than one dimension is not so easily done, because dynamic allocation of an n-dimensional array actually requires dynamic allocation of n-1 dimensional arrays. To allocate a 2-dimensional array you must first allocate memory sufficient to hold all the elements of the array (the data), then allocate memory for pointers to each row of the array. One such method for dynamic allocation is presented here. First, we allocate memory for an array which contains a set of pointers. Next, we allocate memory for each array which is pointed by pointers. For example,

```
int **dArray = 0;
dArray = new int *[Rows]; //memory allocated for elements of rows.
for( int i=0; i<Rows; i++)
        dArray[i]=new int[Columns];
        // memory allocated for elements of each column.
```

2. Pointer is an integer variable that holds the memory address of a variable.

Value addition: Common terms
Heading text: Various forms of Arrays
Body text: • **Lower Triangular Array** 'a' is an nxn array in which a[i][j]==0 if i<j i.e,. all the elements above the diagonal are zero. • **A strictly lower triangular array** a is an n by n array in which a[i][j]==0 if i<=j. • **A tri diagonal matrix** is a square matrix is which all entries are 0 except possibly those on the main diagonal and the diagonal above and below it. i.e., T is a tri-diagonal matrix, then T[i][j]==0 if absolute value of i-j is greater than 1.

- **Band Matrices** are the matrices in which the non zero entries tend to cluster around the middle of each row. For Square matrices, this is equivalent to saying that the non-zero values tend to cluster around the diagonal.

7.8 Operations on Two Dimensional Arrays

7.8.1 Sum of Matrix Elements

Problem Statement:
Calculate the sum of all the elements of matrix
1. Declare a matrix 'a'
2. Initialize it
3. Declare an integer variable 'sum'.
4. Initialize sum=0.
5. Repeat
   for each row
      for each column
        sum = sum +a[row][column]
6. Display sum of all the elements of this matrix.

Program

```
#include<iostream.h>
 #include<conio.h>
void main()
{
clrscr();
int a[20][20],sum;
int i,j,m,n;
cout<<"Enter the order mxn of the matrix \n";
cin>>n>>m;
cout<<"enter the elements of the matrix m and n"<<" m is"<<m<<" n is"<<n;
for(i=1;i<=m;i++)
{
for(j=1;j<=n;++j)
cin>>a[i][j];
}
sum=0;
for(i=1;i<=n;++i)
 {
   for(j=1;j<=n;++j)
   sum=sum+a[i][j];
}
cout<<"sum of element is " <<sum;
getch();
}
```

Output

```
Enter the order mxn of the matrix
2
2
enter the elements of the matrix m and n m is2 n is2
```

```
1
2
2
1
sum of element is 6
```

7.8.2 Addition of Two Matrices

Problem Statement

Consider two matrices of order n*m. Compute the matrix addition and place the result in third matrix.

Steps

1. Declare three matrices named a, b and c. The order of these matrices should be same. i.e., number of rows and number of columns for these matrices should be same.
2. Initialize matrix A and matrix B.
3. Repeat
   for each row
      for each column
         c[row][column]=a [row][column]+b[row][column]
   This nested for loop adds corresponding elements of matrix 'a' to corresponding element of matrix 'b'.
4. Display the contents of matrix c i.e.,
   Repeat
   for each row
      for each column
         display c[row][column]

Program

```
#include<iostream.h>
#include<conio.h>
void main()
{
clrscr();
int a[20][20], b[20][20], c[20][20];
int i,j,m,n,q;
cout<<"input row and column of a matrix A and a matrix B \n";
cin>>m>>n;
cout<<"enter the elements of matrix A \n";
for(i=0;i<m;++i)
{
for(j=0;j<n;++j)
cin>>a[i][j];
}
cout<<"enter the elements of matrix A \n";
for(i=0;i<m;++i)
{
for(j=0;j<n;++j)
cin>>b[i][j];
}
for(i=0;i<m;++i)
{
for(j=0;j<n;++j)
```

```
c[i][j]=a[i][j]+b[i][j];
}
cout<<"\n The sum of two matrix is:\n";
for(i=0;i<m;++i)
{
cout<<"\n";
for(j=0;j<n;++j)
cout<<" "<<c[i][j];
}
getch();
}
```

Output

```
Input row and column of a matrix A and a matrix B
2
2
enter the elements of matrix A

2
2
2
2
enter the elements of matrix A
4
4
4
4

 The sum of two matrix is:

 6 6
 6 6
```

7.8.3 Transpose of a Matrix

Problem Statement:

Calculate the transpose of a two dimensional square matrix. The **transpose** of a matrix of the order of 2x3, is a matrix is of the order 3x2. It is obtained by converting the rows into columns and column into rows respectively.

For Example

$$\begin{bmatrix} 3 & 2 \\ 5 & 6 \\ 2 & 9 \end{bmatrix}_{3x2} \xrightarrow{\text{transpose}} \begin{bmatrix} 3 & 5 & 2 \\ 2 & 6 & 9 \end{bmatrix}_{2x3}$$

Steps

1. Declare a matrix 'A' of some order
2. Also declare another matrix 'B' of the reverse order as that of 'A'.
3. Initialize matrix A
4. Repeat

```
for each row
   for each column
      B[column][row]= A[row][column]
```

5. Display values of matrix B

```
Repeat
for each row
   for each column
     display B[row][column]
```

Program

```
#include<iostream.h>
#include<conio.h>
void main()
{
clrscr();
int a[20][20],b[20][20];
int i,j,m,n;
cout<<"input row and column of a matrix A  \n";
cin>>m>>n;
cout<<"enter the elements of matrix A \n";
for(i=0;i<m;++i)
{
for(j=0;j<n;++j)
cin>>a[i][j];
}
for(i=0;i<n;++i)
{
for(j=0;j<m;++j)
b[i][j]=a[j][i];
}
cout<<"\n before transpose of matrix:\n";
for(i=0;i<n;++i)
{
cout<<"\n";
for(j=0;j<m;++j)
cout<<" "<<a[i][j];
}
cout<<"\n Transpose of matrix is:\n";
for(i=0;i<n;++i)
{
for(j=0;j<m;++j)
cout<<" "<<b[i][j];
cout<<"\n";
}
getch();
}
```

Output

```
input row and column of a matrix A
2
2
enter the elements of matrix A
12
```

```
13
14
15

 before transpose of matrix:

 12 13
 14 15
 Transpose of matrix is:
 12 14
 13 15
```

7.8.4 Trace of a Matrix

The **trace** of an n*n square matrix A is defined to be the sum of the elements on the main diagonal of A (the diagonal from the upper left to the lower right) i.e.,
trace (A)=a11+a22+a33…….+ann
Trace is defined only for square matrices.

Following is the code to calculate trace of a matrix.

```
int trace=0;
for (int i=0;i<rows;i++)
        trace=trace+matrix[i][i];
cout<<trace;
```

7.9 Sparse Matrix

Sometimes, the total no of entries in a multidimensional array quite easily exceed the main memory limitations. For E.g. an array to store the coefficients for a 50*50 system of equations would require 2500 memory locations. Yet, in actual applications, it quite often turns out that very high percentage of these memory locations are filled with zeros. Such an array with a low percentage of non zero entries is said to be **sparse.**
There is no precise definition of when a matrix is sparse and when it is not, but it is a concept which can be recognized intuitively. Storing more number of zeros is a waste of memory because operations like insertion, deletion on multi dimensional arrays are time consuming. As computer scientists, we are interested in studying ways to represent such multi dimensional sparse arrays in a way that the operations on them can be carried out efficiently. The data structure problem of a sparse array is to develop techniques that accurately represent the data it contains without wasting the memory required to store an exceedingly large number of zeros. A matrix having more zero entries such that only non-zero entries of the matrix may be saved using any alternate form in lesser space is considered as **sparse matrix**.

7.9.1 Representation of a Sparse Matrix

The non-zero elements of a normal matrix is usually represented in a n*3 sparse matrix where n is the number of non-zero elements of the normal matrix. Each row of the sparse array is a list of 3-tuples of the form (i, j, value) where i,j is the row, column of the non-zero element of normal matrix and value is that non-zero element itself. The first row of sparse matrix is the header row specifying following information:

Location (0,0) stores the row size of the original matrix.
Location (0,1) stores the column size of the original matrix.
Location (0,2) stores the number of non zero elements of the original matrix.

The sparse matrix is always constituted of 3 columns and many rows depending on non-zero entries in the normal matrix.

For e.g. consider a normal matrix A of size 5*4 as follows:

$$\begin{pmatrix} 0 & 2 & 0 & 0 \\ 0 & 8 & 0 & 9 \\ 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 7 & 0 & 0 \end{pmatrix}$$

Here, total number of elements is 20, out of which only 5 elements are non-zero entries. So, one can save 15 memory locations if this normal matrix is saved as a sparse matrix Sparse (A) of size 3*(5+1) as follows. Note that, first row of the sparse matrix gives information about normal matrix. i.e.

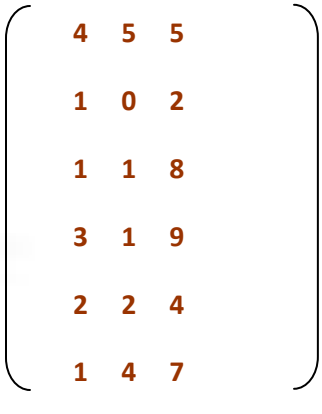
Location (0,0) stores the row size of original matrix. (5)
Location (0,1) stores the column size of original matrix. (4)
Location (0,2) stores the number of non-zero elements of original matrix. (5)

$$\begin{pmatrix} 5 & 4 & 5 \\ 0 & 1 & 2 \\ 1 & 1 & 8 \\ 1 & 3 & 9 \\ 2 & 2 & 4 \\ 4 & 1 & 7 \end{pmatrix}$$

7.9.2 General Operations on a Sparse Matrix

Transpose of a Sparse Matrix:

The transpose of a sparse matrix is obtained by reversing its row and column indices i.e. interchanging the first column and second column of the sparse array. Consider the sparse matrix Sparse (A) obtained in above example. Its corresponding transpose T(A) is :

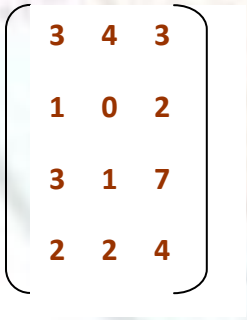


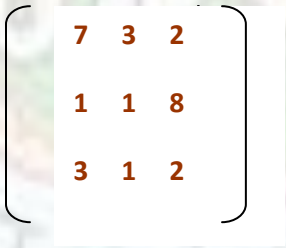
Addition of Sparse Matrices:

The addition of 2 sparse matrices is quite different from addition of normal matrices. It is slightly complicated because one needs to check the presence of corresponding elements. I.e. elements are added if they have same row-column entries otherwise the elements are simply copied in the resultant array.

Consider the addition of two sparse matrices A and B as follows:

Sparse (A) + Sparse (B) = Sparse (Sum)





Logic:
From the above two Sparse matrices, first rows are simply copied as they give information about original matrices. Row 2 of Sparse (A) and row 1 of Sparse (B) have same row-column dimensions [i.e. row 3 and column 1 of original matrices], thus data 7 gets added to data 2 to get 9 in Sparse (Sum) matrix. The other elements where there is no match are copied from both matrices to Sparse (Sum).

Value addition: Intellectual Text

Heading text: Application of Sparse Matrix in Real World

Body text:

Sparse matrices have many uses in almost all the areas of the natural sciences. One such application is in the equilibrium conditions across electrical network. Kirchoff's laws of Electrical Equilibrium state:

- The sum of incoming currents at each node is equal to the sum of outgoing currents.
- Around every closed loop, the sum of voltage is 0.

These 2 laws can be completely manifested by the sparse matrix. The row in the matrix represents nodes. The columns indicate directed electrical paths. -1 is used to indicate a node where a path ends, 1 to indicate a node where a path begins, and 0 to indicate that the particular path does not meet the corresponding node.

7.10 Multi Dimensional Arrays

Arrays can have more than one dimension .These arrays are called **multidimensional arrays**. They can be regarded as array of arrays. They are very similar to standard arrays with the exception that they have multiple sets of square brackets after the arrays identifiers.

7.10.1 Declaration of Multi Dimensional Arrays

If two square brackets are used while expressing an array, it is said to be a two dimensional array. For example the declaration

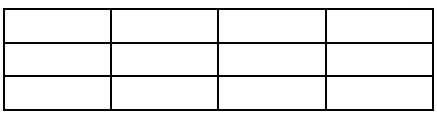
```
int a[10][20];
```

declares a two dimensional array 'a' which can store 10x20=200 integer type elements.

Similarly if three square brackets are used while expressing an array, it is said to be a three dimensional array. For example, the declaration

```
float records[2][3][4];
```

declares a three dimensional array records which can store 24 elements of type float.
This three dimensional array can be regarded as array of arrays that is it can be viewed as single dimensional array records[2] of a 2 two dimensional arrays records[3][4] as shown in figure 7.11(a) and 7.11 (b). Note that, only two occurrences of two dimensional array records[3][4] is required because the single dimensional array records[2] contains only two elements at position 0 and 1. (array index starts at zero).



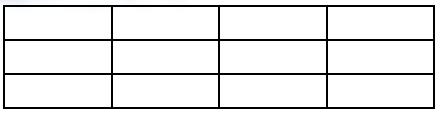


Figure 7.11(a) – first occurrence of two dimensional array records[3][4].
Figure 7.11(b) – second occurrence of two dimensional array records[3][4].

7.10.2 Initialization of Multi Dimensional Arrays

This three dimensional array can be initialized during the declaration by following lines of code.

```
float records[2][3][4]={
                        {19, 99, 8, 10},
                        {23, 44, 57, 61},
                        {71, 81, 32, 15},
                        {64, 90, 36, 12},
                        {61, 84, 18, 96},
                        {17, 8, 12, 34}
                       };
```

The values are stored sequentially first in the array records[3][4] at position 0 and then in the array records[3][4] at position 1. This is depicted in figure 7.12(a) and 7.12(b).

19	99	8	10
23	44	57	61
71	81	32	15

7.12(a)

64	90	36	12
61	84	18	96
17	8	12	34

7.12(b)

Figure 7.12(a) position '0' of array records[3][4]
Figure 7.12(b) position '1' of array records[3][4]

In order to access any element of the records array, we need to specify all three subscripts as follows.

```
cout<<records[1][2][3];
```

This statement will print '34' i.e. the last element of the array. Here first subscript [1] refers to the selection of two dimensional array at position 1. Second and third subscripts refer to the particular row and column of the selected two dimensional array.

7.10.3 Printing Elements of Multi Dimensional Array

The values of an array can be printed using "cout" statement in a nested for loop. For example, in order to print the values of records array following code is required .

```
int i,j,k;
for(i=0;i<2;i++)   // 2 dimensional array subscript
 for (j=0;j<3;j++)     //  row subscript
   for(k=0;k<4;k++)     // column subscript
       cout<<records[i][j][k];
```

Value addition: Interesting Fact

Heading text: Usage of Multi Dimensional Arrays

Body text:

A three dimensional array is useful when a value is determined by three inputs. For example, an array of temperatures might be indexed by latitude, longitude and attitude.

7.10.4 Data Storage Forms of Multi Dimensional Arrays

The memory of a computer is essentially one dimensional. So, one needs to convert the three dimensional array form to single dimensional form in order to store the data. There are two forms by which three dimensional data can be store in a linear sequence of memory.

- **Left subscript major form** : grouping together all the data associated with a given value of the leftmost subscript as in figure 7.13(a)

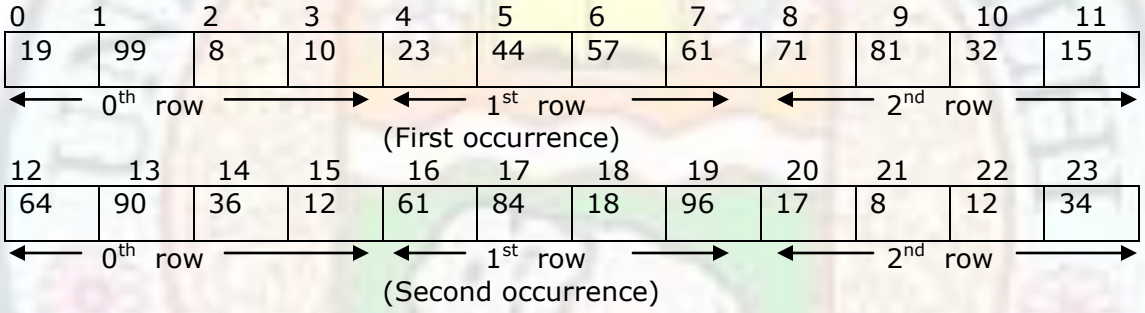


Fig 7.13(a) left subscript major form for the data of figure 7.12 (Row Major Form)

- **Right subscript major form** : grouping together all the data associated with a given value of the right most subscript as in figure 7.13(b)

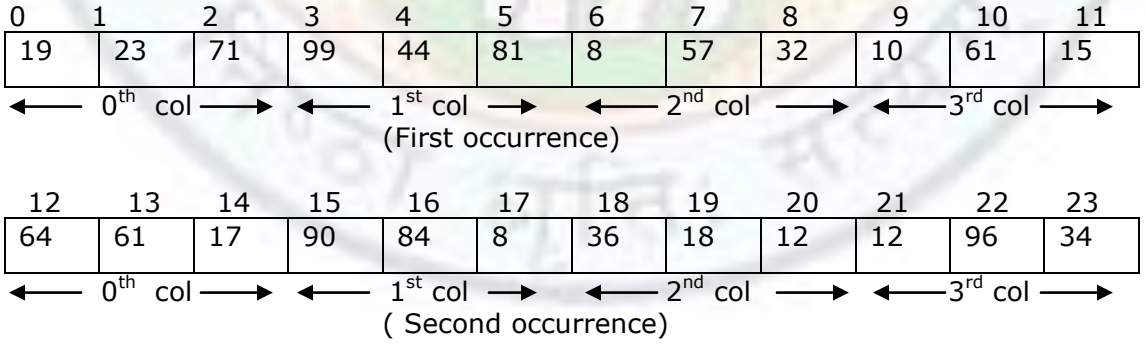


Fig 7.13(b) right subscript major form for the data of figure 7.12
(Column Major Form)

Value addition: Related Terms
Heading text: Definitions
Body text: • **Associative array**: An abstract data structure model that generalizes arrays to arbitrary indices. • **Parallel array**: Array of records, with each field stored as a separate array. • **Array Processor**: a computer to process arrays of data. • **Array slicing**: the extraction of sub arrays of an array.

7.10.5 More About Multi Dimensional Arrays

If we interpret the indices to be n-dimensional (i1,i2,…….i3) then it is called an n dimensional array. For example, if we declare a multi dimensional array as

```
int students[2][3][9][7];
```

then four pairs of square brackets indicate that it is a four dimensional array consisting of 2*3*9*7 = 378 elements of integer type. Such higher dimensional arrays rarely used.
In order to calculate total number of elements in a multidimensional array, one should be aware of lower and upper bounds of a dimension. The smallest element of an arrays index representing a dimension of the array is said to be the **lower bound** of that dimension. The largest element of an array's index representing a dimension of the array is said to be the **upper bound** of that dimension.

In the n-dimensional array , if the array declared as A[l1….u1,l2…..u2,…….,ln…….un] where li and ui are the lower bound and upper bound of the ith dimension then

- Total number of elements = (u1-l1)(u2-l2)…….(un-ln).
  For example, in the above declaration,
  Total number of elements = (2-0)(3-0)(9-0)(7-0)
  = 2*3*9*7
  = 378
- General formula for Row Major Form for A[s1,s2,s3…..sn] = Base Address of array + ((s1*u2*u3*…un)+(s2*u3*u4*……un)…+(sn-1*un)+sn)*size of array

- General formula for Column Major Form for A[s1,s2,..sn] = Base Address of array + ((sn*u1*u2*…un-1)+(sn-1*u1*u2*……un-2)…+(s2*u1)+s1)*size of array

7.11 Basic Concept of Searching

Searching refers to the operation of finding the location of the given data item in a collection of items. The search is said to be successful or unsuccessful depending on whether the element that is to be searched is found or not. The problem of searching may be linked to situation that we have all faced in manual record keeping system. For example, let us consider a telephone directory which contains names address and telephone numbers of a large number of people. Given a name, the task here can be to find out the telephone number and address of that particular person.

The telephone directory in the above example can be considered as a file or a table containing a large collection of records. Each records has one or more fields like name, telephone no, address. One of these fields is used to distinguish records of a file. This field is known as a key. While searching, we are asked to find a record that contains other information associated with a given key. In the above example, (searching the telephone number and address associated with a name), the key is name.

Value addition: Intellectual Context

Heading text: Types of Keys

Body text:

- A key may be contained within the record, as we have seen in the example of telephone directory, at the specific offset from the start of the record. Such a key called an **internal key or an embedded key.**
- There may be separate table of keys that includes pointer to the records, such keys are called **external keys.**
- There is at least one set of keys for every file, which is unique & it determines a record uniquely. Such key is called the **primary key.** For example, the telephone number is a primary key for a telephone directory because we cannot have two records with the same telephone number.
- Keys are not always unique. For example, if we are using name as key in telephone directory, there may be one more persons with same name. Such key is called **secondary key.**

7.11.1 Introductory Consideration

A table or file may be organized as an array of records such as table 7.2

Relative Position	Key	Data Name	Address
1.	25888783	Sunil Mehta	26/77, West Patel Nagar, New Delhi
2.	25885210	Sunita Narang	4-R, Tara Appartment, Bombay
3.	25745102	Seema Gupta	wz-18, New Rohtak Road, Cheenai
.			
.			
.			
n	9810714492	Sourabh Ahuja	4-H, Vikrant Appartment, Jannat

Table 7.2 General format of data for search operation

Each key and data pair constitutes the fields within a record of our table. Our search operation generally uses keys of integer data type. The array of fig 1.1 could exist either in main memory or on a permanent storage medium such as a magnetic disk. This distinction will not affect the logic of search algorithms. However, the distinction is often important because the efficiency of an algorithm may be influenced by the type of storage media to which is applied.

- Actually, if there are many records, it will, then, be necessary to store the records on secondary storage. This kind of searching where the file/array is kept in secondary storage is called **external searching**.

- Whereas searching where the file/array of records is kept in the main memory is called **internal searching**.

7.11.2 Search Algorithm

The process of searching an item can be explained by an algorithm given below. In a given data base, we match every item to the wanted information 'a'. If it is found then location of item 'a' is returned. If none of the items in the data base matches the wanted item 'a' then the search remains unsuccessful. Thus, a searching algorithm accepts two arguments.

1. The wanted item 'a'.
2. The data base to be searched. Usually the data base is an array of records.

This algorithm returns one of the following:

1. The wanted item's location if the search is successful.
2. Returns nothing if the search is unsuccessful.

```
Search(wanted item 'a', database)

For each item in database

        If the item matches the wanted information 'a'  then

                exit with this item's location in the database
```

Figure 7.14 Search Algorithm

There are many search techniques available. In this unit we will discuss only two basic searching techniques.

1. Linear Search.
2. Binary Search.

Actually, an array which stores large numbers of items can be ordered or unordered. If it is unordered then Linear Search in preferred otherwise Binary Search is preferred.

Value addition: Fact
Heading text: Variant of Search Algorithm
Body text: An algorithm which adds a new record with the target as its key, when a search is unsuccessful is called search & insert algorithm.

7.12 Linear Search

7.12.1 Concept of Linear Search

Often an array of elements has not been sorted in any particular order. When this is the case and we wish to determine the location of a particular value in an array, we must use a special case called the **linear search** to solve this problem. With a linear search technique, you look at every element in the array, one-at-time beginning at the first. When the value is found, quit immediately. If you get through the entire array without quitting, then the search is unsuccessful i.e. the value is not in the array. It is one of the simplest search

techniques and is also called **sequential search**. For each time in the array, starting from 0^{th} location we look through all the items in the array. If the array item is same as item-to-be-found then we return location of this array item. Otherwise, we have gone through whole array without getting success and we return -1 indicating unsuccessful search.

7.12.2 Pseudo code of Linear Search

```
int LinearSearch (int a[], int aSize, int toFind)

{

        // look through all items, starting from 0th location .

        for(int i=0; i<aSize; i++)

                if(a[i]==toFind)        //comparison

                        return i;

// you have gone through the whole array without success

return -1;

}
```

Figure 7.15 Pseudo code of Linear Search

The function LinearSearch() takes 3 arguments
1) Integer array 'a'.
2) Array size 'aSize' indicating number of elements in it.
3) The item to be matched 'toFind'.
It returns the location of the searched value in the array.
If the toFind element is not in the array, then it returns -1.

The above mentioned pseudo code can be explained through an example,
Suppose, we have an unsorted array 'a' with following 6 values.

Location	0	1	2	3	4	5
Data	800	100	400	500	200	600

Suppose, the element to be searched is 500, then 500 is compared with all elements starting with element 800 at 0th location. The search process ends either when 500 is found or all the array elements are checked and unsuccessful search results.
In this example, four comparisons are made and location 3 is returned. Step by step comparisons are shown as below.

i is set to 0
1. Comparison with element at 0^{th} loc.
   800!=500,
   i=i+1=1
2. Comparison with element at 1^{st} location.

100!=500,
i=i+1=2
3. Comparison with element at 2nd location.
400!=500,
i=i+1=3
4. Comparison with element at 3rd location.
500=500
return index '3' ,and stop.

If element to be searched is 900 then following 6 comparisons are made and -1 is retuned indicating an unsuccessful search.

i set to 0,
1. Comparison with element at 0th location.
800 != 900
i=i+1=1.
2. Comparison with element at 1st location.
100!=900
i=i+1=2.
3. Comparison with element at 2nd location.
400!=900
i=i+1=3.
4. Comparison with element at 3rd location.
500!=900
i=i+1=4.
5. Comparison with element at 4th location.
200!=900
i=i+1=5.
6. Comparison with element at 5th location.
600!=900
i=i+1=6.
Return -1 (Array 'a' is traversed completely and the element 900 is not found)

The Complete code for linear search is as follows

```
#include<iostream.h>
int LinearSearch (int[], int, int); //prototype
int main()
{
const int aSize=10;
int a[aSize]={5, 10, 22, 32, 45, 67, 73, 98, 99, 101};
int toFind, location;
cout<<"enter the item you are searching for ";
cin>>toFind;
location=LinearSearch (a, aSize, toFind); //function call.
If (location!=-1)       //successful search
        Cout<<"the item was found at location"<<location<<endl;
else                    //unsuccessful search
        cout<<"the item was not found in the array" <<endl;
return 0;
}
```

```
int LinearSearch(int a[], int aSize, int toFind)
{
  //this function returns the location of tofind element in the array. -1 is the returned
  //if the value is not found.
  int i;
  for(i=0; i<aSize; i++)        //loop for traversing array elements.
 {
       if(a[i]==tofind)          //comparison
              return i;          //successful search
  }
 return -1;                      //unsuccessful search
}
```

Value addition: Did You Know
Heading text: Return value of linear search
Body text: There are 2 common forms of code for linear search. The difference is what is returned as the result of search. 1. Sometimes the location of the data is returned. 2. Sometimes the data itself is returned.

7.12.3 Analysis of Linear Search

In real word problems, one need to search through a large list to find a particular item for example, given an account number, the manager would like to find out the balance in this account. So, for such real world problems, time need to traverse through the array elements in order to find the item matched or not is of utmost importance. For simple analysis of algorithms, we look for a dominant operation and account the number of time that dominant operation has been performed. In case of searching, the dominant operation is the comparison. Let us examine how long it will take to find an item using linear search we are interested in. Following three cases are considered:

1. **Best case**: If the element to be found is at first location of the array. Here, number of comparison would be 1.
2. **Worst case**: If the element to be found is either present at last location of the array or the element is not present in the array. Here number of comparison would be n where 'n' is the number of elements in the array.
3. **Average case**: If the element is not present in the first position or at the last position of the array, then searching this element requires average number of comparisons. If the size of array is 'n' then

Average Case Time is =n*(n+1)/2

7.12.4 Advantages and Disadvantages of Linear Search

Advantages of Linear Search:-

1. It is usually very simple to implement.
2. It is practical to use when the array has few elements.

Disadvantages of Linear Search:-

1. It is slower if the data set is very big.
2. It is not very efficient.

7.12.5 Ways to improve efficiency of Linear Search

Ways to improve efficiency of linear search:-

1. Using Sorted Array :-

   The average performance of linear search can be improved by using it on sorted data element. In this case, the unsuccessful search can be terminated immediately if the unmatched array element is greater then the element to be found. This avoids us examining the entire array elements thus avoiding worst case.

2. Using Transposition Method :-

   The performance of linear search can be improved if the desired value is more likely to be near the beginning of the array than to its end. Therefore if some values are much more likely to be searched than others, it is desirable to place the at the beginning of the array. One method which causes elements to move towards the front of the array gradually is the transposition method in which a successfully retrieved element is swapped with the element that precedes it. Over much retrieval, the most frequently retrieved elements will tend to be grouped at the front of the array.

3. Using A Sentinel Method:-

   Linear search in an array is usually programmed by stepping up an index variable in the 'for' loop until it reaches the last index. This normally requires two comparisons instruction for each array element.
   a) One to check weather the element has desired value.
   b) Second to check weather the index has reached the end of the array.
   We can reduce the work of second comparison by inserting the desired item itself as a sentinel value at the far end of array, as in the following pseudo code:

```
set a[aSize+1] to toFind.
set i=0;
repeat this loop:
if(a[i]==toFind)
    exit the loop;
set i to i+1;
return i;
```

   With this strategy, it is not necessary to check the value of i against the array size aSize. If toFind is present in the array 'a' then the function will return array index 'i' which will always be between 0 and aSize, then the loop terminates. If toFind is not present in the array 'a' then the function will return array index 'i' which will always be aSize+1. Thus the loop will terminate automatically when index i reaches aSize+1. However, this method is possible only if the array slot a[aSize=1] exist and is not being otherwise used.

7.13 Binary Search

7.13.1 Concept of Binary Search

Actually, linear search is highly inefficient for large collection of elements because in worst case, we will have to make N comparisons where N is very large say 500. It is just like searching for a given name in a large telephone directory by reading one name at a time starting at the front of directory. This is time consuming. So, we can think of another way of searching for a given name in this large telephone directory. When we open the book (usually in the middle), we look at the name on top of the page, and decide if this name is bigger or smaller than the one we are looking for. If the name we are looking for is bigger, then we continue searching ahead otherwise search proceeds backwards. This is the basic technique behind **Binary Search**. This search technique is very fast and efficient for large arrays. But it requires the array elements be sorted.

Logic

In **Binary Search**, to search an element we compare it with the element present at the middle i.e.

```
mid = (low + high)/2
```

location of the array where low= start index of array, high=last index of array. If it is equal then the search is said to be successful, otherwise, the array is divided into two parts one starting from 0 to (mid-1) and the second partition from (mid+1) to aSize-1. As a result, first partition contains element less than the middle element and second partition contains element greater than the middle element. The searching is now continued in either of two partitions depending upon weather the element is greater or smaller than the middle element. If the element is smaller then the middle element then the searching will be continued in the first partition otherwise in second partition. This process continues until the desired element is found or the size of the partition becomes less than 1 i.e. unsuccessful search.

There are two versions of Binary Search.

- Recursive version
- Non-Recursive version

7.13.2 Recursive Version of Binary Search

If the item being searched for is not equal to the middle element of array, we must search the sub array using the same method. Thus the search method is defined in terms of itself with the smaller array as input.

The pseudo code is as follows:

```
BinarySearchRecursive(a,low,high,target)
//search for target in a[low] to a[high]
//examine the middle element in the list mid=(low+high)/2
//if the middle element contains the target key then stop searching.

 if(target==a[mid])
         return(mid)

//if the middle element is larger than the target, search the first.
 if(target<a[mid])
                 BinarySearchRecursive(a,low,mid-1,target)
 else
                 BinarySearchRecursive(a,mid+1,high,target)
```

Figure 7.16 Pseudo code for Recursive Binary search

Here, we use two indices, low and high to enclose the part of the array in which we are looking for the target key. Initially, the algorithm may be called with low equal to 0 and high equal to aSize-1. This algorithm works fine, if the target key is there in the array but what will happen if the desired key is not there in the list? We know that the target key is not there when low>high. So, we can include following if block at the beginning of the above pseudo code.

```
if(low>high)
 return -1
```

Therefore, if the binary search algorithm returns -1, we know that the search is unsuccessful. For successful search, it returns the index of the key which matches the target key. Thus, complete recursive function for binary search is as follows.

```
BinarySearchRecursive(a,low,high,target)

{

    if(low>high)

        return -1;

        mid =(low+ high)/2;

    if(target==a[mid])

        return mid;

    if(target<a[mid])

         BinarySearchRecursive(a,low,mid-1,target)

    else

         BinarySearchRecursive(a,mid+1,high,target)

}
```

Figure 7.17 Complete version of Recursive Binary Search

Let us apply this algorithm to an example.
Suppose array 'a' contains elements as 19,21,27,30,35,40,43 and that we wish to search for 27. Then, the steps are as follows.

Initialize low=0 and high=6.

Low			mid			high
0	1	2	3	4	5	6
19	21	27	30	35	40	43

Is low>high? No
Mid= (0+6)/2
Is 27==a[3]? No
27<a[3] Yes, repeat the steps with
low=0 and high=mid-1=2

Low	mid	high				
0	1	2	3	4	5	6
19	21	27	30	35	40	43

Is low>high? No
Mid= (0+2)/2=1
Is 27== a[1]? No
27<a[1] No, repeat the steps with
low=mid+1=2 and high=2

		low,high				
0	1	2	3	4	5	6
19	21	27	30	35	40	43

Is low>high? No
Mid=(2+2)/2=2
Is 27==a[2] ? Yes
return(2).

The answer is returned to the previous call, since there are no steps to be executed after this in the algorithm, the answer is returned to the main caller.

Let us examine how the algorithm searches for an item which is not there in the array. Assume that we are trying to search for 20 in the array a, as in the previous example.

Low			mid			high
0	1	2	3	4	5	6
19	21	27	30	35	40	43

Is low>high ? No
Mid= (0+6)/2=3
Is 20==a[3] ?No
20<a[3] Yes , repeat the steps with
low=0 and high=mid-1=2.

Low		mid		high		
0	1	2	3	4	5	6
19	21	27	30	35	40	43

Is low>high ?No
Mid = (0+2)/2=1
Is 20==a[1] ?No.
Is 20<a[1] ?Yes , repeat the steps with
low=0 and high=mid-1=1.

Low,mid		high				
0	1	2	3	4	5	6
19	21	27	30	35	40	43

Is low>high ? No
Mid=(0+1)/2=0
Is 20==a[0] ?No
Is 20<a[0] No , repeat the steps with
low=mid+1=1 and high=0

high		low				
0	1	2	3	4	5	6
19	21	27	30	35	40	43

Is low>high ? Yes
return -1.

The algorithm returns -1, which indicates that the item 20 does not exist in the array.

We have defined binary search algorithm recursively. However, the overhead associated with recursion may make it inappropriate for use in practical situations where efficiency is a prime consideration. We therefore, present another version of the binary search algorithm which is non recursive.

```
BinarySearchNonRecursive(a,n,target)
{
   low=0;
   high=n-1;
   while(low<=high)
    {
       Mid=(low+high)/2;
       if (target==a[mid])
          return(mid);
       if(target<a[mid])
          high=mid-1;
       else
          low=mid+1;
     }
      return -1;
```

Figure 7.18 pseudo code for Non recursive binary search

7.13.3 Analysis of Binary Search

Best Case: If the element to be searched is there in the middle of the array then function requires only 1 comparison to yield best case.

Average Case/Worst Case: If the element is present in the array but not a middle position then average time is equal to log n to the base 2.

Value addition: Frequently Asked Question

Heading text: Linear Search Vs. Binary Search

Body text:

Why use a binary search when a linear search is easier to code? The answer lies in the efficiency of the search .With a linear search , if you double the size of an array, you could potentially twice as much time to find an item .But with a Binary Search, doubling the size of the array , merely adds one more item to look at. Each time you look with a Binary Search, you eliminate half of the remaining array items as possible matches. For example,
If total array elements are n=500 then
Worst case for linear search = n = 500
Worst case for binary search= log (500 to the base 2) = 2.69897

If total array elements are n =1000 then
Worst case for linear search = n = 1000
Worst case for binary search= log (1000 to the base 2) = 3

7.13.4 Advantages and Disadvantages of Binary Search

Advantages of Binary Search

1) It is easy to implement.
2) It works fast and is efficient.

Disadvantages of Binary Search

1) It is not guaranteed to be faster for searching small arrays as compared to sequential search.
2) It is suitable only if few insertions are to be made to the array as these operations requires movement of many items.
3) It is suitable only for arrays, as it is not possible to find the midpoint of a link list.

Value addition: Did you know
Heading text: Search Algorithm depends on Data Structure
Body text: Basically, the searching depends on the data structure used. An array stored with data items can be searched for the data using basic searching techniques, linear search and binary search. Link list can be searched using linear search, trees are searched using binary search tree, graph requires the separate techniques such as breath first search and depth first search.

7.14 Basic Concept of Sorting

7.14.1 Definition of Sorting

Sorting is an arrangement of data items in a sequential order according to an ordering criterion. For Example, if we work with a student file, we might be interested in sorting the student names in alphabetical order. When it comes to the ranks obtained by the students then we are interested in arranging the records based on ascending order of the ranks.

7.14.2 Need for Sorting

1. Efficient sorting is important to optimizing the use of other algorithms (such as search algorithms) that require sorted listed to work correctly. For example binary search works on sorted array elements only.
2. It is also often useful for organizing data and for producing human readable output.
3. The operation of sorting is most often performed in business data processing applications and scientific applications. It is important for presentation of the data extracted from database. Most people prefer to get reports sorted into some relevant order before wading through pages of data.

7.15 Sorting Algorithm

7.15.1 Objective of a Sorting Algorithm

1. Minimize exchanges of large amount of data movement. When data items are large and the number of data items is excessive, then swapping of data items takes large amount of processing time.
2. Move data from secondary storage to main memory in large blocks, because the larger the data block to be moved, the more efficient is the corresponding

algorithm. This is a key part of external sorting.
3. If possible, retain all the data in main memory. In this case, random access into an array can be effectively used. This is a key part of internal sorting.

Output

1. The output is in ascending or decreasing order.
2. The output is a permutation, or reordering, of the input.

7.15.2 Categories of Sorting Algorithms

There are two basic categories of sorting methods:

1. **Internal sorting:** Internal sorting methods are applied when the entire collection of data to be sorted is small enough that the sorting can take place within main memory. The time required to read or write is not considered to be significant in evaluating the performance of internal sorting methods.

2. **External sorting:** External sorting method are applied to larger collection of data which reside on secondary storage devices. The read and write access time are major concern in determining sort performances of external sorting methods.

In the world of sorting, there are plenty of techniques available for internal and external sorting. Following are some well known internal sorting techniques. Out of these, in this chapter we will discuss Bubble Sort, Selection Sort and Insertion Sort.

Bubble Sort	Merge Sort
Selection Sort	Radix Sort
Insertion Sort	Shell Sort
Quick Sort	Heap Sort

Table 7.3 Various Sort Methods

Other Categories:

Sorting Algorithm used in computer science are often classified by:

1. **Computational complexity** (worst, average and best behavior) of element comparisons in terms of the size of the list.
2. **Memory usage** (and use of other computer resources). In particular, some sorting algorithms are **in place**. This means that they need no memory beyond the items being sorted and they don't need to create auxiliary locations for data to be temporally stored, as in other sorting algorithms.
3. **Stability: Stable sorting** algorithms maintain the relative order of elements with equal values. If all elements are different then this distinction is not necessary. But if there are similar elements, then a sorting algorithm is stable if whenever there are two elements (say R and S) with the same value, and R appear before S in the original array, then R will always appear before S in the sorted array. **Unstable sorting** algorithms may change the relative order of the elements with equal value, but stable sorting algorithms never do so. Assume the following pair of numbers to be sorted by their first component.

   (4,2), (3,7), (3,1), (5,6)

   In this case, two different results are possible, one which maintains the relative order of elements with equal values and one which does not:

   (3, 7) (3, 1) (4, 2) (5, 6)

(Ordered of similar elements maintained so it is a stable sort)

(3, 1) (3, 7) (4, 2) (5, 6)
(Ordered of similar elements is not maintained so it is an unstable sort)

Value addition: Intellectual Fact
Heading text: Stable Sorting Vs. Unstable Sorting
Body text: • Generally, stable sorting is a harder problem, because a stable sorting algorithm is rearranging input according to unique permutation of input that will produce the stable sorted output; where as an unstable sorting algorithm only has to find one among many permutations (if there are equal value elements) that produces a unstable permutation of input. It is then expected that stable sorting algorithms should use more time and/or space than unstable sorting algorithms. • Few of the well known sorting algorithms are stable such as Bubble sort, Selection sort. Unstable sorting algorithms include Insertion sort.

7.15.3 Analysis of Sorting Algorithms

There are basically three main considerations that should affect a programmer's decision to choose from a variety of sorting methods:

1. Programming time
2. Execution time of the program
3. Memory or auxiliary space needed for the program environment.

First, evaluate carefully the need for sorting data. It is pointless to write a complex program to sort a short file. If number of records in the file is no more than 100, and each record no longer than 80 bytes, any sorting procedure probably would be adequate. However, when both the file size and the record size are large, minimize processing time by choosing an efficient sorting routine. Quite often, advance knowledge about the structure of the file and the data in it can help you make a proper choice. However, if no such advance knowledge about the data is available, then you may need to assume the worst case scenario for the sort. Unfortunately, there is no such thing as a best sorting method fitting all applications.

The efficiency of a sorting algorithm is measured by the run time used for execution of the algorithm. When the file is large, choice of a proper sorting method resulting in the least possible run time is crucial. The run time is a function of the number 'n' of items to be sorted. Each of the sorting algorithms is made up of several operations, such as comparisons between data items, interchanges depending on the result of comparison and assignments. Generally efficiency complexity measures only the number of comparisons, since the number of other operations is at most a constant factor of the number of comparisons. There are three cases to be considered for analyzing the sorting methods.

- **Best case**: In this case, the input array is assumed to be in desired sorted order.
- **Average case**: In this case, the data items are dispersed all over the file & there is no order among the elements of the array.
- **Worst case**: In this case, the input file is assumed to be in reverse order.

The various sort algorithms considered in this chapter have been compared for their relative efficiencies. When 'n', the number of items to be sorted, becomes large then efficiency is

usually proportional to some function of n. This function is denoted by f(n). We use the notation O(f(n)) to mean proportional to f(n).

Value addition: Did You Know

Heading text: Simple and Complex Sorting Algorithms

Body text:

Sorting algorithms can be characterized in the following two ways:

- Simple algorithms which require order of n^2 written as $O(n^2)$ comparison to sort n items.
- Complex algorithms that require the O(log(n)) comparisons to sort n items.

The difference lies in the fact that the first method moves data only over small distances in the process of sorting, whereas the second method moves data over large distances, so that items settle into the proper order sooner, thus resulting in fewer comparisons.

7.16 Bubble Sort

7.16.1 Concept of Bubble Sort

The **Bubble sort** is the probably the simplest of the sorting algorithms. Its name comes from the idea that the largest element 'bubbles up' to the top (the high end) of the array like the bubbles in a carbonated beverage or smallest data bubbles down. Unfortunately, it is the slowest sort algorithm. The bubble sort works by comparing each item in the list with the item next to it, and swapping them if required. The algorithm repeats this process until it makes a pass all the way through the list without swapping any items (in other words, all items are in the correct order). This causes larger values to 'bubble" to the end of the array while smaller values "sink" towards the beginning of the array.

Logic

Let us write the function BubbleSort () to implement the bubble sort. It takes two arguments:

- A one dimensional array named numbers [ ] to store the elements
- n which specifies the total number of integer data items that are to be sorted.

Initially array numbers [] contains the randomly stored data and sorting should be applied to the same array. At the end of sorting, array numbers [] is stored with sorted data items. When this approach is used, at most n-l passes (iterations) are required. During the first pass, the elements at numbers[0] and numbers[1] are compared. If they are not in order then they are interchanged. This procedure is repeated for next two elements of the numbers[] array i.e. the elements at numbers[1] and numbers[2] are compared, then elements at numbers[2] and numbers[3] are compared and so on. This process will make larger data items to bubble up or move to the end of the array. After the 1^{st} pass, the element with the largest value will be placed in the n-1th position. On each successive pass, the element with the next largest data item will be placed in the n-2th position, n-3th position ... 1^{st} position and 0^{th} position respectively, there by sorting the array.

7.16.2 Pseudo code of Bubble Sort

```
void BubbleSort ( int numbers[], int n)
{
  int i, j, temp n1;
  n1=n-1;
   for (i= 0; i<n-1; i++)
     for(j=0; j <n1-i; j++)
       if (numbers[j] > numbers [j + 1]) //swapping
         {
              temp = numbers[j];
              numbers[j]=numbers[j+1];
              numbers[j] = temp;
         }
   }
```

Figure 7.19 Pseudo code of Bubble sort

Let us consider array numbers[] = { 77,44,99,66,33,55,88,22,44 }
This sort goes through the iterations as shown in table 1.1. After first iteration, the element with largest value 99 is placed at position 8th position i.e. nth position. After second iteration, the next largest element in the array 88 is placed at 7th position i.e. (n-1)th position and so on.

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]
77	44	99	66	33	55	88	22	44
44	77							
		66	99					
			33	99				
				55	99			
					88	99		
						22	99	
							44	99
44	**77**	**66**	**33**	**55**	**88**	**22**	**44**	**99**
	66	77						
		33	77					
			55	77				
					22	88		
						44	88	
44	**66**	**33**	**55**	**77**	**22**	**44**	**88**	**99**
	33	66						
		55	66					
				22	77			
					44	77		
44	**33**	**55**	**66**	**22**	**44**	**77**	**88**	**99**
33	44							
			22	66				
				44	66			
33	**44**	**55**	**22**	**44**	**66**	**77**	**88**	**99**
		22	55					
			44	55				
33	**44**	**22**	**44**	**55**	**66**	**77**	**88**	**99**
	22	44						
33	**22**	**44**	**44**	**55**	**66**	**77**	**88**	**99**

22	33							
22	**33**	**44**	**44**	**55**	**66**	**77**	**88**	**99**

Table 7.4 Iterations of a bubble sort

7.16.3 Analysis of Bubble Sort

Generally for n number of elements there will be n-1 iterations. The bubble sort is generally considered to be the most inefficient sorting algorithm in common usage.

Best-case: Under best case condition if the array is already sorted, the bubble sort requires the outer loop to be executed for n-l time and the inner loop will be there for n-l, n-2, n-3 .... 1 times for each iteration but never enters the true block of if condition. Hence the time complexity is $0(n^2)$.

Average case: In average case there will be approximately n(n-l)/2 comparisons in the inner loop. Hence,

$$f(n)= \frac{n-l}{2} + \frac{n-2}{2} + ... \frac{2}{2} + \frac{1}{2} = \frac{n(n-l)}{4} = 0(n^2)$$

Worst case: When the contents of an array are stored in reverse order then insertion sort requires n^2 time called as worst case time. In this arrangement there are n-l comparisons in the first iteration, which places the largest element in the last position and there are n-2 comparisons in the second iteration, which places the second largest in the next to last position. Hence

$$f(n) = n - 1 + n - 2 + ... + 2 + 1 = \frac{n(n-l)}{2} = O(n^2)$$

Thus, irrespective of best, average and worst case, complexity is $O(n^2)$.

7.16.4 Advantages and Disadvantages of Bubble Sort

Advantages:

- It is simple of all sorting algorithms and easy to implement.
- An in place sort i.e., space complexity is O (1).

Disadvantages:

- It is inefficient because it takes large number of comparisons resulting in $O(n^2)$ complexity. For example, if we have 100 elements, then total number of comparisons will be 10000.

Did u know
Program for bubble sort with the output

```
#include<iostream.h>
#include<conio.h>
void main()
{
int i,j,n,numbers[10],temp;
clrscr();
cout<<"\n\n\nHow many numbers :";
cin>>n;
cout<<"\nEnter the numbers :\n";
for(i=0;i<n;i++)
cin>>numbers[i];
for(i=0;i<n;i++)
```

```
{
for(j=0;j<i;j++)
{
if (numbers[j]>numbers[j+1])
{
temp=numbers[j];
numbers[j]=numbers[j+ 1];
numbers[j+1]=temp;
}
}
}
cout<<"\nAscending order ... \n";
for(i=0;i<n;i++)
{
cout<<numbers[i]<<endl;
}
getch();
```

OUTPUT
How many numbers :4

Enter the numbers :
12
11
13
14
Ascending order ...
11
12
13
14

7.17 Selection Sort

7.17.1 Concept of Selection Sort

The **Selection Sort** works like the Bubble Sort: on each iteration of the main loop, the next largest element is moved into its correct position. But instead of "bubbling" these elements to the end of the array, Selection Sort finds the element to be moved (i.e. largest element) without moving any other element of the array. Then it just puts this element into place with a single swap. Thus, it is more efficient than the Bubble Sort. This implementation discussed below puts the next smallest element (instead of the next largest element) in place on each iteration of the main loop. Both implementations work equally well.

Logic

We perform a search through the array starting from the first element at 0^{th} position, in order to locate the element with the smallest value. When this element is found, we interchange it with the element at 0^{th} position in the array. As a result of this interchange, the smallest element is placed at 0^{th} position of the array. In the second iteration, we locate the second smallest element traversing the array from 1^{st} position onwards. We then interchange this element with the element at 1^{st} position of the array. We continue this

process of searching for the element with the smallest value in the unsorted portion of the array and placing it in it's proper position until we have placed all the elements in the proper order.

7.17.2 Pseudo code of Selection Sort

```
void SelectionSort(int a[], int n )
{
  int i,j,temp, smallestElementIndex;
  for(i=0; i<n; i++)
  {
        smallestElementIndex = i;
        for(j=i+1; j<n; j++)
   {
      if(a[smallestElementIndex] > a[j])
          smallestElementIndex = j;
     }
     if(smallestElementIndex ! = i)
      {
         temp =a[i];
         a[i] = a[smallestElementIndex];
         a[smallestElementIndex] =temp;
      }
  }
 }
```

Figure 7.20 Pseudo code of Selection Sort

Let us consider array numbers[] = { 77,44,99,66,33,55,88,22,44 }
This sort goes through the iterations as shown in table 1.2. After first iteration, the element with smallest value 22 is placed at position 0^{th} position. After second iteration, the next smallest element in the array 33 is placed at 1^{st} position and so on.

A[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]
77	44	99	66	33	55	88	22	44
22							77	
22	**44**	**99**	**66**	**33**	**55**	**88**	**77**	**44**
	33			44				
22	**33**	**99**	**66**	**44**	**55**	**88**	**77**	**44**
		44		99				
22	**33**	**44**	**66**	**99**	**55**	**88**	**77**	**44**
			44					66
22	**33**	**44**	**44**	**99**	**55**	**88**	**77**	**66**
				55	99			
22	**33**	**44**	**44**	**55**	**99**	**88**	**77**	**66**
					66			99
22	**33**	**44**	**44**	**55**	**66**	**88**	**77**	**99**
						77	88	
22	**33**	**44**	**44**	**55**	**66**	**77**	**88**	**99**

Table 7.5 Iterations of Selection sort

7.17.3 Analysis of Selection Sort

Best case, average case and worst case is $O(n^2)$.

7.17.4 Advantages and Disadvantages of Selection Sort

Advantages:
- It is simple and easy to implement.

Disadvantages:
- It is inefficient for large arrays.

Value addition: Interesting Fact
Heading text: Selection Sort Vs. Bubble Sort
Body text: • The number of comparisons is the same as that of Bubble Sort namely n(n-1)/2. • The number of swaps depends on data. But is no more than n-1 times. • Since swaps are typically more expensive than compares, Selection sort is generally better algorithm than bubble sort.

7.18 Insertion Sort

7.18.1 Concept of Insertion Sort

The **Insertion sort** is named because on each iteration of it's main loop, it inserts the next element in its corresponding position relative to the sub array that has already been processed. This is the common method people use to sort playing cards dealt to them in card games.

Logic

Suppose an array 'a' with 'n' elements is in memory. The insertion sort algorithm scans array 'a' from a[0] to a[n-1], inserting each element a[k] into it's proper position in the previously sorted sub array a[0], a[1],…..a[k-1]. i.e.

Pass I : a[0] by itself is trivially sorted.
Pass II : a[1] is inserted either before or after a[0] so that a[0], a[1] is sorted.
Pass III: a[2] is inserted into it's proper place in a[0] and a[1] i.e. before a[0] or between a[o] and a[1] or after a[1]. So that a[0],a[1],a[2] is sorted.
.
.
.
Pass n: a[n-1] is inserted into it's proper place in a[0], a[1], a[2],….a[n-2] so that a[0], a[1], a[2],…..a[n-1] is sorted.

There remains only the problem of deciding how to insert a[k] in it's proper place in the sorted sub array a[0], a[1],…a[k-1]. This can be accomplished by comparing a[k] with a[k-1], comparing a[k] with a[k-2] and so on, until first meeting an element a[j] such that a[j]<=a[k]. Then each of the elements a[k-1], a[k-2]…a[j+1] is moved forward one

position, and a[k] is inserted in the j+1th position in the array.

7.18.2 Pseudo code of Insertion Sort

```
void InsertionSort(int a[], int n )
{
 int i,j,temp;
 for(i=1; i<n; i++)
 {
      temp = a[i];
      for(j=i-1; j>0 && temp<a[j]; j--)
  {
      a[j+1]=a[j];
  }
  a[j+1] = temp;
 }
}
```

Figure 7.21 Pseudo code of Insertion Sort

Let us consider array numbers[] = { 77,44,99,66,33,55,88,22,44 }
This sort goes through the iterations as shown in table 1.3. After first iteration, the sub array is sorted. After 1st pass, a[0], a[1] sub array is sorted. After 2nd pass, a[0], a[1], a[2] sub array is sorted and so on.

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]
77	44	99	66	33	55	88	22	44
44	77							
44	**77**	**99**	**66**	**33**	**55**	**88**	**22**	**44**
	66	77	99					
44	**66**	**77**	**99**	**33**	**55**	**88**	**22**	**44**
33	44	66	77	99				
33	**44**	**66**	**77**	**99**	**55**	**88**	**22**	**44**
		55	66	77	99			
33	**44**	**55**	**66**	**77**	**99**	**88**	**22**	**44**
					88	99		
33	**44**	**55**	**66**	**77**	**88**	**99**	**22**	**44**
22	33	44	55	66	77	88	99	
22	**33**	**44**	**55**	**66**	**77**	**88**	**99**	**44**
			44	55	66	77	88	99
22	**33**	**44**	**44**	**55**	**66**	**77**	**88**	**99**

Table 7.6 Iterations of Insertion sort

7.18.3 Analysis of Insertion Sort

Best case is O(n) whereas average case and worst case is $O(n^2)$.

Value addition: Did You Know
Heading text: Improvement on time taken by Insertion Sort
Body text: • Time taken by insertion sort can be saved by performing a binary search, rather than a linear search to find the location in which to insert a[k] in the sub array a[0], a[1],….a[k-1]. This requires log k comparisons rather than k-1 comparisons. However, one still need to move (k-1)/2 elements forward. Thus, the order of complexity is not changed. • This sort is not very useful by itself but it forms the basis of shell sort.

7.18.4 Advantages and Disadvantages of Insertion Sort

Advantages:

- It is easy to implement with linear search.
- An in place sort i.e., space complexity is O (1).

Disadvantages:

- Lot of data movement exist in insertion sort.

7.19 Arrays and Functions

7.19.1 Concept of Arrays with regard to Functions

Let us consider a situation of a company in which general manager organizes the working of the company. Apart from general manager the company also compromises production officer, account officer, personal officer & store officer. In order to operate the company smoothly, the general manager delegates various tasks to these officers. For example, the work of production officer is to manage the production, the account officer maintains accounts, and personal officer maintains the records of employee whereas the store officer maintains the inventory in the store. In the world of programming, this situation is similar to a main function calling sub-function to perform various tasks i.e. here general manager acts as a main() and rest of the managers are calls to sub-functions in main() program.

Now, suppose this company has three branches. Each branch has its own production officer, personal officer, accounts officer & store officer. The general manager sitting in main office, delegate similar type of work to personal officers of all three branches. Of course, these personal officers will maintain records of those employees who work in their branch. The situation can be compared to the one in which the main() becomes the general manager & plays a role of calling a sub-function with an array of three elements. All the three array elements (i.e. three personal officers) perform similar type of task as assigned by main() program.

Thus, in the world of modular programming often one need to pass array as an argument of a function. Normally, in a function program the input variables are supplied through arguments. These input variables are supplied from the main program to the function

program, when a call is given to this function program. If these input variables are of the same data type then they can be passed as an array.

```
sub function( array as a parameter )
{
.
.
//all elements of the array
//performs the task assigned by main().
.
.
}
main()
{
.
.
//call a sub function with array as an argument
.
.
}
```

Figure 7.22 General Format of array as a parameter of a function

A simple example can be consider as:

```
void func(int a[53])      //function definition
{
        a[0]=10;
        a[1]=20;
        a[2]=30;
}
main()
{
        int a[3];
        func(a);  //function call
        for(i=0;i<3;i++)
                cout<<a[i]<<endl;
}
```

Output is as follows:
10
20
30

This program defines an array 'a' of three elements. The array 'a' is then passed to the function func(). The declaration

"void func(int a[3])"

defines the function name as "func" and declares that nothing will be passed back as the result. It accepts a data type called 'a' which is declared as an array of 3 integer elements. Actually, an array name is generally equivalent to a pointer to the first element, and arrays are passed to functions as pointers. A **pointer** is a variable that holds the address of memory location of another normal variable. Infact, the statement

"ptr = &var"

means that a pointer variable is initialized with the address of the variable var. So, when elements in the array are modified in the function, they are also modified in the calling procedure.

Value addition: Interesting Fact

Heading text: Relation between Arrays and Pointers

Body text:

int a[5];

makes the variable 'a' a pointer to first cell of a block of memory big enough to store 5 integers. Space for those 5 integers is reserved as shown in fig 1.1, so it doesn't get clobbered.

Memory location	0	1	2	3	4	5	6	7	8
Element	A[0]	A[1]	A[2]	A[3]	A[4]	….	….	….	….

--------- Reserved ----------

Therefore, a[0]=*(a+0), a[1]=*(a+1) ……,a[n]=*(a+n)
Where, a is the address of first element.

We could have declared 'a' in the almost equivalent way as below:

int* a;

Note that in principle we still can set and access the elements in the array in the same manner. For example, in order to access second element of the array 'a', we would use following statement.

a[1]=2;

The difference is that we haven't reserved any space. This is a source of errors. If you declare strings or array in this manner, and fail to allocate space, your program won't work. You need to allocate space (as described in more detail below) with something like:

a=new int[5];

7.19.2 Array as an argument of a Function

- **Call By Reference**:

If an array variable becomes argument of a function program then the whole array is passed to a function program. This means if an array contains 10 elements then all the elements are passed to function program by making array to be argument of function program. This process of passing an array to a function program when a call is given is called call by reference. The advantage of this process is to pass the multiple values to the function program.
For example,

```
void sum_array(int a[], int n)
{
    int i;
    int sum=0;
    for(i=0;i<5;i++)
      {
            sum=sum+a[i];
      }
    cout<<sum;
    }

    main()
    {
            int a[5]={1,2,3,4,5};
            sum_array(a,5);
    }
```

Output:
15

In this function program, Call by Reference method is used in which the complete array 'a' has been made as an argument of function program along with size of the array. The function program receives the array and the sum of all the element of array is calculated by function program. Array as a whole can only be passed by reference in C++.

- **Call By Value**:

It is possible to send a single element of the array as an argument to a function program. The modification on this value is possible. This method is known as Call by Value.

For example,

```
int sum=0;
Void sum_array(int b)
{
    sum=sum+b;
}
```

```
main()
{
    int a[5]={1,2,3,4,5}
    int I;
    for(i=0;i<4;i++)
    {
            Sum_array(a[i]);
    }
    cout<<sum;
}
```

Output:
15

In this program the array is declared in the main program and single element of array is passed to the function program and the element of array is captured by a variable of the function program. Therefore, the sum of all the elements is calculated by passing the individual elements to the function program which is called 5 times in the main function. This situation is call by value. The sum of all elements is calculated by using global variable 'sum'. The variables which are declared before the function program or the main() program are global and exist globally for all the functions inside the program.

7.19.3 Returning an Array from a Function

A function can return only a single variable to the main() program. If we want a function to return multiple values such as elements of an array, then it becomes necessary to declare array as a pointer. Following examples illustrates the importance of declaring arrays as pointers.

```
void test(int c[])
{
  c[0] =1;
}

void main()
{
   int a[5];
   test(a);
   cout << a[0];
}
```

results in `1' being written out.
The test function may be written entirely equivalently as:

```
void test(int* c)
{
  c[0] =1;
}
```

If we want a function to return an array, the pointer version is essential. The following illustrates this:

```
int* test()
{
  int* a;
  a = new int[2];
  a[0]=1; a[1]=2;
  return a;
}

void main()
{
   int* b;
   b = test();
   cout << b[0] << b[1];
}
```

Note that space for the array (being created) is explicitly allocated within the function using new.

Summary

- An array is a collection of variables of same type such that array elements are accessed by an index and it's elements are stored in consecutive memory location. In general, data of same type which is not expected to change is preferably stored in arrays.
- An array of a one dimensional array is called a two dimensional array. It is represented as a matrix where elements are arranged in the form of rows and columns.
- Some common operations on it are matrix addition, transpose and trace.
- The operations which require less movement of data elements in an array are easier to perform. It has been found that, operations such as insertion and deletion are not as common as searching and sorting on arrays.
- In Linear search, you look at every element in the array, one-at-time beginning at the first element. When the value is found, quit immediately. If you get through the entire array without quitting, then the search is unsuccessful i.e. the value is not in the array.
- In Binary Search, to search an element we compare it with the element present at the middle location of the array. If it is equal then the search is said to be successful, otherwise, the array is divided into two partitions with first partition containing elements with the value less than the middle element and second partition containing elements with the value greater than the middle element. The searching is now continued in either of two partitions depending upon weather the element is greater or smaller than the middle element. This process continues until the desired element is found.
- Linear Search can be used on unsorted arrays whereas Binary Search requires the array to be sorted.
- Linear search is not very efficient because if the item to be found is at the end of the array, then all previous items must be compared with the item to be found. Whereas, Binary search divides the search space into two sub partitions on each unsuccessful search.
- Bubble sort, the simplest sort algorithm, bubbles up the largest element to the end of the array. Unfortunately, it is the slowest sort algorithm.
- Selection Sort finds the smallest element to be moved without moving any other element of the array and place it in it's correct position with a single swap. Thus, it is more efficient than the Bubble Sort.
- Insertion sort, on each iteration, inserts the next element in its corresponding position relative to the sub array that has already been processed.
- When arrays are passed as arguments in a function call, the modifications made to the array elements in the called function are reflected in the main() program. This is due to the fact that array name is generally a pointer to the first element of the array.
- If we want a function to return an array, it is essential to declare array as pointer.
- Array of arrays are called multi dimensional arrays. Usually, arrays with dimension higher than three are rarely used.

Exercise

7.1 Accept the number of children and their names from the user. Create a dynamic array to store the data and display it using a "for" loop.

7.2 Write a program to delete an integer data item from an array of elements in required position between 0 to n-1.

7.3 Write a program to sum the corresponding elements of two arrays into third array.

7.4 Write a program to concatenate 2 string arrays.

7.5 Write a program to check if the array contains a palindrome.

7.6 Write a program to declare a two dimensional array A[3][4]. Swap the elements of column no. 2 and 3 of this array.

7.7 Write a program to implements multiplication of 2 dimensional arrays.

7.8 Write a program to find largest number in a 2 dimensional array.

7.9 Write a program to implement addition of 2 spare matrices.

7.10 Write a program to represent a given matrix as sparse and find its transpose.

7.11 Suppose that a 2 dimensional array declared as "char a[5][6]", is internally stored in contiguous memory locations "100" in a column major form. What will be the memory address for element a[3][4].

7.12 Write a program to add to sparse matrices.

7.13 Write a program to represent a given matrix as sparse and find it's transpose.

7.14 Write a program to implement linear search such that search returns the data along with its location?

7.15 Write a program to implement linear search using a sentinel.

7.16 Write a program to implement recursive version of Binary Search.

7.17 Write a program to implement non-recursive version of Binary Search.

7.18 Write a program to implement Selection sort

7.19 Write a program to implement insertion sort using
- linear search
- binary search

Glosssary

Array: An array is a collection of same type of data items which are stored in consecutive memory locations under a common name.

Array Processor: a computer to process arrays of data.

Array slicing: the extraction of sub arrays of an array.

Associative array: An abstract data structure model that generalizes arrays to arbitrary indices.

Average Case of binary search If the element is present in the array but not a middle position then average time is equal to log n to the base 2.

Average case of linear search: If the element is not present in the first position or at the last position of the array, then searching this element requires average number of comparisons.

Average case of sorting: In this case, the data items are dispersed all over the file & there is no order among the elements of the array.

Band Matrices: They are the matrices in which the non zero entries tend to cluster around the middle of each row. For Square matrices, this is equivalent to saying that the non-zero values tend to cluster around the diagonal.

Base address: Starting address of the array is called base address.

Best Case of binary search: If the element to be searched is there in the middle of the array then function requires only 1 comparison to yield best case.

Best case of linear search: If the element to be found is at first location of the array. Here, number of comparison will be 1.

Best case of sorting: In this case, the input array is assumed to be in desired sorted order.

Binary Search: In Binary Search, to search an element we compare it with the element present at the middle location of the array. If it is equal then the search is said to be successful, otherwise, the array is divided into two partitions with first partition containing elements with the value less than the middle element and second partition containing elements with the value greater than the middle element. The searching is now continued in either of two partitions depending upon weather the element is greater or smaller than the middle element. This process continues until the desired element is found.

Bubble sort: the simplest sort algorithm, bubbles up the largest element to the end of the array. Unfortunately, it is the slowest sort algorithm.

Character array: It is a linear Data Structure which is used to store strings.

Column major form: For a two dimensional array, storage in column major form implies that elements of a column are stored contiguously.

Dimension: The number of elements in the array is called its dimension or size or length.

Dynamic Data Structure: A Data Structure is referred to as Dynamic Data Structure if it is created at run time.

External key: There may be separate table of keys that includes pointer to the records; such keys are called external keys.

External searching: Actually, if there are many records, it will, then, be necessary to store the records on secondary storage. This kind of searching where the file/array is kept in secondary storage is called external searching.

External sorting: External sorting methods are applied to larger collection of data which reside on secondary storage devices. The read and write access time are major concern in determining sort performances of external sorting methods.

In place sort: This means that they need no memory beyond the items being sorted and they don't need to create auxiliary locations for data to be temporally stored, as in other sorting algorithms.

Insertion sort: It, on each iteration, inserts the next element in its corresponding position relative to the sub array that has already been processed.

Internal key: A key may be contained within the record, as we have seen in the example of telephone directory, at the specific offset from the start of the record. Such a key called an internal key or an embedded key.

Internal searching: Whereas searching where the file/array of records is kept in the main memory is called internal searching.

Internal sorting: Internal sorting methods are applied when the entire collection of data to be sorted is small enough that the sorting can take place within main memory. The time required to read or write is not considered to be significant in evaluating the performance of internal sorting methods.

Linear Data Structure: It defines a set of operations which do not create hierarchical structures among the elements of its data object. For example, an array.

Linear search: In Linear search, you look at every element in the array, one-at-time beginning at the first element. When the value is found, quit immediately. If you get through the entire array without quitting, then the search is unsuccessful i.e. the value is not in the array.

Lower bound of an array dimension: The smallest element of an arrays index representing a dimension of the array is said to be the lower bound of that dimension.

Lower Triangular Array: It is an nxn array 'a' in which a[i][j]==0 if i<j i.e,. all the elements above the diagonal are zero.

Matrix: The matrix in mathematics is a two dimensional array written in the form of rows and columns.

Multidimensional arrays: Arrays can have more than one dimension .These arrays are called multidimensional arrays.

Non linear hierarchical Data Structure: It defines such a set of operations which create a hierarchical structure among the elements of its data object. For example, trees, graphs etc.

Non Primitive Data Structure: It defines a set of derived elements. For example, an array which consist of a set of similar type of elements.

Parallel array: Array of records, with each field stored as a separate array.

Pointer: A pointer is a variable that holds the address of memory location of another normal variable.

Primary key: There is at least one set of keys for every file, which is unique & it determines a record uniquely. Such key is called the primary key. For example, the telephone number is a primary key for a telephone directory because we cannot have two records with the same telephone number.

Primitive Data Structure: It defines a set of primitive elements which don't involve any other elements as its subparts. For example, int, char.

Row major form: For a two dimensional array, storage in row major form implies that, elements of a row are stored contiguously.

Searching: It refers to the operation of finding the location of the given data item in a collection of items.

Secondary key: Keys are not always unique. For example, if we are using name as key in telephone directory, there may be one more persons with same name. Such key is called secondary key.

Selection Sort: It finds the smallest element to be moved without moving any other element of the array and place it in it's correct position with a single swap. Thus, it is more efficient than the Bubble Sort.

Sorting: It is an arrangement of data items in a sequential order according to an ordering criterion.

Sparse matrix: A matrix having more zero entries such that only non-zero entries of the matrix may be saved using any alternate form in lesser space is considered as sparse matrix.

Stable sorting: Those sorting methods which maintains the relative order of elements with equal values.
Static Data Structure: A Data Structure is referred to as Static Data Structure if it is created during compilation time.

Strictly lower triangular array: It is an nxn array 'a' in which a[i][j]==0 if i<j.

Trace: The **trace** of an n*n square matrix A is defined to be the sum of the elements on the main diagonal of A

Transpose: The **transpose** of a matrix of the order of 2x3, is a matrix is of the order 3x2. It is obtained by converting the rows into columns and column into rows respectively.

Tri diagonal matrix: It is a square matrix is which all entries are 0 except possibly those on the main diagonal and below it. i.e., T is a tri-diagonal matrix, then T[i][j]==0 if absolute value of i-j is greater than 1.

Upper bound of an array dimension: The largest element of an array's index representing a dimension of the array is said to be the upper bound of that dimension.

Unstable sorting: Those sorting methods which may change the relative order of the elements with equal value

Worst Case of binary search: If the element is present in the array but not a middle position then average time is equal to log n to the base 2.

Worst case of linear search: If the element to be found is either present at last location of the array or the element is not present in the array. Here number of comparison will be n where 'n' is the number of elements in the array.

Worst case of sorting: In this case, the input file is assumed to be in reverse order.

References

Work Cited

1. R.G. Dromey, How to solve it by Computer, Pearson Education
2. B. A. Forouzan and R. F. Gilberg, Computer Science, A Structured Approach using C++, Cengage Learning, 2004.

Suggested Reading

1. S.S. Khandara, Programming in C, C++, S.Chand
2. John R. Hubbard, Data Structure with C++, Tata Mc Graw Hill

Web links

1. Wekepedia.com
2. Google.com