8

Arrays

Key Concepts

- **Array and its need**
  - Declaring arrays
  - Memory allocation for arrays
  - Array initialization
  - Accessing elements of arrays
- **Array operations**
  - Traversal
  - Sorting
    - Selection sort
    - Bubble sort
  - Searching
    - Linear search
    - Binary search
- **Two dimensional (2D) arrays**
  - Declaring 2D arrays
  - Matrices as 2D arrays
- **Multi-dimensional arrays**

We used variables in programs to refer data. If the quantity of data is large, more variables are to be used. This will cause difficulty in accessing the required data. We learned the concept of data types in Chapter 6 and we used basic data types to declare variables and perform type conversion. In this chapter, a derived data type in C++, named 'array' is introduced. The word 'array' is not a data type name, rather it is a kind of data type derived from fundamental data types to handle large number of data easily. We will discuss the creation and initialization of arrays, and some operations like traversal, sorting, and searching. We will also discuss two dimensional arrays and their operations for processing matrices.

8.1 Array and its need

An **array** is a collection of elements of the same type placed in contiguous memory locations. Arrays are used to store a set of values of the same type under a single variable name. Each element in an array can be accessed using its position in the list, called index number or subscript.

Why do we need arrays? We will illustrate this with the help of an example. Let us consider a situation where we need to store the scores of 20 students in a class and has to find their class average. If we try to solve this problem by making use of variables, we will need 20 variables to store students' scores. Remembering

and managing these 20 variables is not an easy task and the program may become complex and difficult to understand.

```
int  a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t;
float avg;
cin>>a>>b>>c>>d>>e>>f>>g>>h>>i>>j>>k>>l>>m>>n>>o>>p>>q>>r>>s>>t;
avg  =  (a+b+c+d+e+f+g+h+i+j+k+l+m+n+o+p+q+r+s+t)/20.0;
```

As it is, this code is fine. However, if you want to modify it to deal with the scores of a large number of students, say 1000, you have a very long and repetitive task at hand. We have to find a way to reduce the complexity of this task.

The concept of array comes as a boon in such situations. As it is a collection of elements, memory locations are to be allocated. We know that declaration statement is needed for memory allocation. So, let us see how arrays are declared and used.

8.1.1 Declaring arrays

Just like the ordinary variable, the array is to be declared properly before it is used. The syntax for declaring an array in C++ is as follows.

```
data_type  array_name[size];
```

In the syntax, `data_type` is the type of data that the array variable can store, `array_name` is an identifier for naming the array and the `size` is a positive integer number that specifies the number of elements in the array. The following is an example:

```
int  num[10];
```

The above statement declares an array named `num` that can store 10 integer numbers. Each item in an array is called an `element` of the array. The elements in the array are stored sequentially as shown in Figure 8.1. The first element is stored in the first location; the second element is stored in the second location and so on.

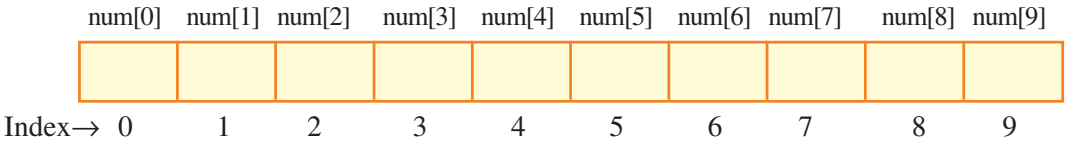


Fig. 8.1: Arrangement of elements in an array

Since the elements in the array are stored sequentially, any element can be accessed by giving the array's name and the element's position. This position is called the index or **subscript** value. In C++, the array index starts with zero. If an array is

declared as `int num[10];` then the possible index values are from 0 to 9. In this array, the first element can be referenced as `num[0]` and the last element as `num[9]`. The subscripted variable, `num[0]`, is read as "num of zero" or "num zero". It's a shortened way of saying "the num array subscripted by zero". So, the problem of referring to the scores of 1000 students can be resolved by the following statement:

```
int score[1000];
```

The array, named `score`, can store the scores of 1000 students. The score of the first student is referenced by `score[0]` and that of the last by `score[999]`.

8.1.2 Memory allocation for arrays

The amount of storage required to hold an array is directly related to its type and size. Figure 8.2 shows the memory allocation for the first five elements of array **num**, assuming 1000 as the address of the first element. Since `num` is an integer type array, size of each element is 4 bytes (in a system with 32 bit integer representation using GCC) and it will be represented in memory as shown in Figure 2.2.

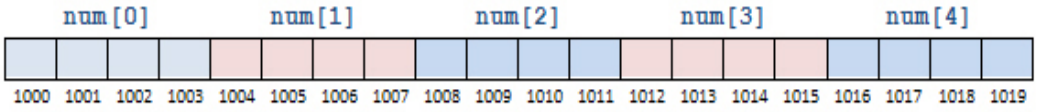


Fig. 8.2: Memory allocation for an integer array

For a single dimensional array, the total size allocated can be computed using the following formula:

total_bytes = `sizeof`(array_type) × size_of_array

For example, total bytes allocated for the array declared as `float a[10];` will be 4 × 10 = 40 bytes.

8.1.3 Array initialisation

Array elements can be initialised in their declaration statements in the same manner as in the case of variables, except that the values must be included in braces, as shown in the following examples:

```
int score[5] = {98, 87, 92, 79, 85};
char code[6] = {'s', 'a', 'm', 'p', 'l', 'e'};
float wgpa[7] = {9.60, 6.43, 8.50, 8.65, 5.89, 7.56, 8.22};
```

Initial values are stored in the order they are written, with the first value used to initialize element 0, the second value used to initialize element 1, and so on. In the first example, `score[0]` is initialized to 98, `score[1]` is initialized to 87, `score[2]` is initialized to 92, `score[3]` is initialized to 79, and `score[4]` is initialized to 85.

If the number of initial values is less than the size of the array, they will be stored in the elements starting from the first position and the remaining positions will be initialized with zero, in the case of numeric data types. For `char` type array, such positions will be initialised with `' '` (space bar) character. When an array is initialized with values, the size can be omitted. For example, the following declaration statement will reserve memory for five elements:

```
int num[] = {16, 12, 10, 14, 11};
```

8. 1.4 Accessing elements of arrays

The array elements can be used anywhere in a program as we do in the case of normal variables. We have seen that array is accessed element-wise. That is, only one element can be accessed at a time. The element is specified by the array name with the subscript. The following are some examples of using the elements of the score array:

```
score[0] = 95;
score[1] = score[0] - 11;
cin >> score[2];
score[3] = 79;
cout << score[2];
sum = score[0] + score[1] + score[2] + score[3] + score[4];
```

The subscript in brackets can be a variable, a constant or an expression that evaluates to an integer. In each case, the value of the expression must be within the valid subscript range of the array. An important advantage of using variable and integer expressions as subscripts is that, it allows sequencing through an array by using a loop. This makes statements more structured keeping away from the inappropriate usage as follows:

```
sum = score[0] + score[1] + score[2] + score[3] + score[4];
```

The subscript values in the above statement can be replaced by the control variable of `for` loop to access each element in the array sequentially. The following code segment illustrates this concept:

```
sum = 0;
for (i=0; i<5; i++)
sum = sum + score[i];
```

An array element can be assigned a value interactively by using an input statement, as shown below:

```
for(int i=0; i<5; i++)
    cin>>score[i];
```

When this loop is executed, the first value read is stored in the array element `score[0]`, the second in `score[1]` and the last in `score[4]`.

Program 8.1 shows how to read 5 numbers and display them in the reverse order. The program includes two `for` loops. The first one, allows the user to input array values. After five values have been entered, the second `for` loop is used to display the stored values from the last to the first.

Program 8.1: To input the scores of 5 students and display them in reverse order

```
#include <iostream>
using namespace std;
int main()
{
  int i, score[5];
  for(i=0; i<5; i++)  //  Reads  the  scores
  {
    cout<<"Enter a score: ";
    cin>>score[i];
  }
  for(i=4; i>=0; i--)  //  Prints  the  scores
    cout<<"score[" << i << "] is " << score[i]<<endl;
  return 0;
}
```

The following is a sample output of program 8.1:

```
Enter a score: 55
Enter a score: 80
Enter a score: 78
Enter a score: 75
Enter a score: 92
score[4] is 92
score[3] is 75
score[2] is 78
score[1] is 80
score[0] is 55
```

Let us do

1. Write array declarations for the following:
   (a) Scores of 100 students
   (b) English letters
   (c) A list of 10 years
   (d) A list of 30 real numbers
2. Write array initialization statements for the following:
   (a) An array of 10 scores: 89, 75, 82, 93, 78, 95, 81, 88, 77, and 82
   (b) A list of five amounts: 10.62, 13.98, 18.45, 12.68, and 14.76
   (c) A list of 100 interest rates, with the first six rates being 6.29, 6.95, 7.25, 7.35, 7.40 and 7.42.
   (d) An array of 10 marks with value 0.
   (e) An array with the letters of VIBGYOR.
   (f) An array with number of days in each month.
3. Write C++ code segment to input values into the array: `int ar[50];`
4. Write C++ code fragment to display the elements in the even positions of the array: `float val[100];`

8.2 Array operations

The operations performed on arrays include traversal, searching, insertion, deletion, sorting and merging. Different logics are applied to perform these operations. Let us discuss some of them.

8.2.1 Traversal

Basically traversal means accessing each element of the array at least once. We can use this operation to check the correctness of operations during operations like insertion, deletion etc. Displaying all the elements of an array is an example of traversal. If any operation is performed on all the elements in an array, it is a case of traversal. The following program shows how traversal is performed in an array.

Program 8.2: Traversal of an array

```
#include <iostream>
using namespace std;
int main()
{
int a[10], i;
cout<<"Enter the elements of the array :";
for(i=0; i<10; i++)
     cin >> a[i];
```

Reading array elements from the user

```
for(i=0; i<10; i++)
     a[i] = a[i] + 1;
cout<<"\nEntered elements of the array are...\n";
for(i=0; i<10; i++)
    cout<< a[i]<< "\t";
return 0;
}
```

A case of traversal

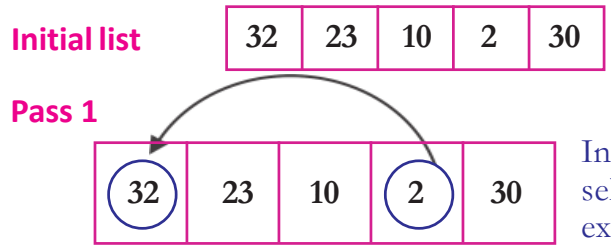
Another case of traversal

8.2.2 Sorting

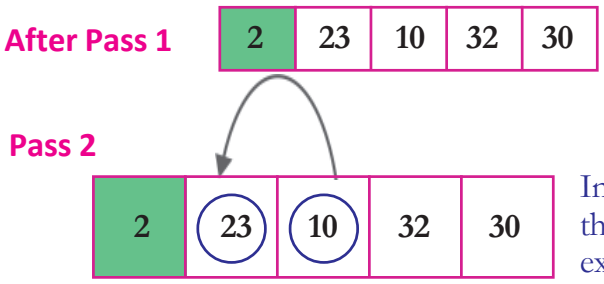
Sorting is the process of arranging the elements of the array in some logical order. This logical order may be ascending or descending in case of numeric values or dictionary order in case of strings. There are many algorithms to do this task efficiently. But at this stage, we will discuss two of the algorithms, known as "selection sort" and "bubble sort".

a. Selection sort

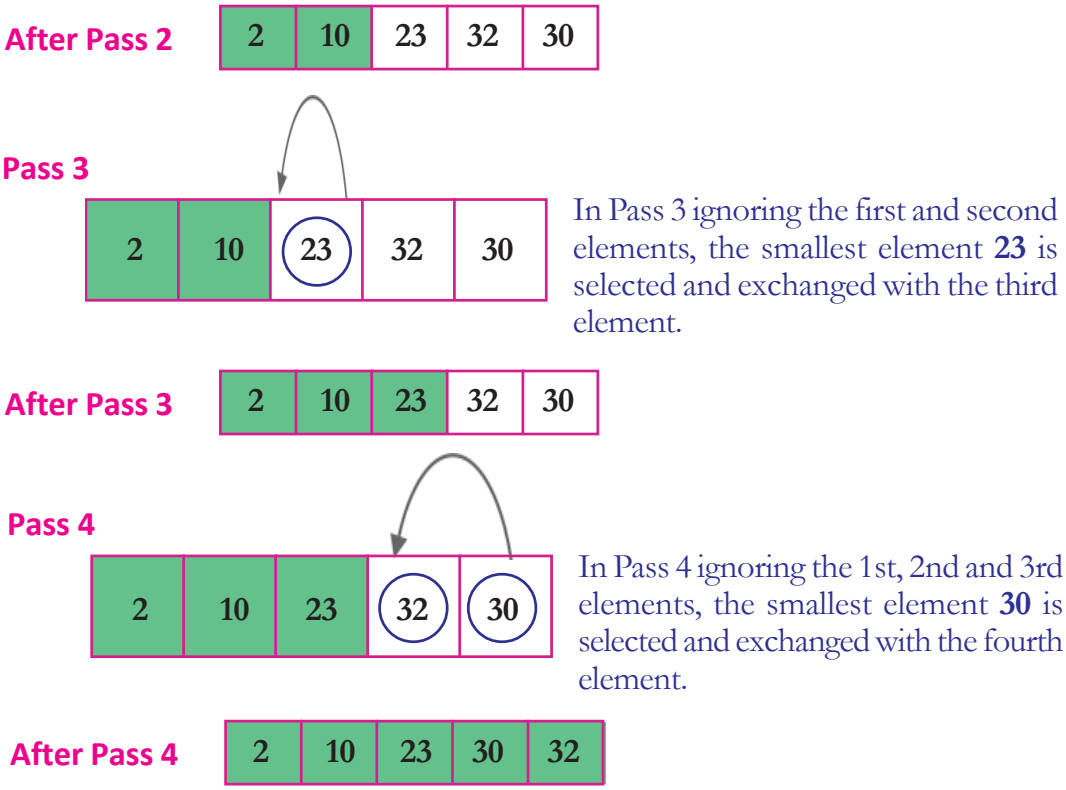
One of the simplest sorting techniques is the selection sort. To sort an array in ascending order, the selection sort algorithm starts by finding the minimum value in the array and moving it to the first position. At the same time, the element at the first position is shifted to the position of the smallest element. This step is then repeated for the second lowest value by moving it to the second position, and so on until the array is sorted. The process of finding the smallest element and exchanging it with the element at the respective position is known as a ***pass***. For 'n' number of elements there will be 'n – 1' passes. For example, take a look at the list of numbers given below.



In Pass 1, the smallest element **2** is selected from the list. It is then exchanged with the first element.



In Pass 2 excluding the first element, the smallest element **10** is selected and exchanged with the second element.



Though each pass is intended for an exchange, no exchange is made in a pass if the smallest value was already in the correct location. This situation is happened in the Pass 3.

Algorithm for selection sort

Step 1. Start

Step 2. Accept a value in N as the number of elements of the array

Step 3. Accept N elements into the array AR

Step 4. Repeat Steps 5 to 9, (N – 1) times

Step 5. Assume the first element in the list as the smallest and store it in MIN and its position in POS

Step 6. Repeat Step 7 until the last element of the list

Step 7. Compare the next element in the list with the value of MIN. If it is found smaller, store it in MIN and its position in POS

Step 8. If the first element in the list and the value in MIN are not the same, then swap the first element with the element at position POS

Step 9. Revise the list by excluding the first element in the current list

Step 10. Print the sorted array AR

Step 11. Stop

Program 8.3 uses **AR** as an integer array which can store maximum of 25 numbers, **N** as the number of elements to be sorted, **MIN** as the minimum value and **POS** as the position or index of the minimum value.

Program 8.3: Selection sort for arranging elements in ascending order

```
#include <iostream>
using namespace std;
int main()
{   int AR[25], N, I, J, MIN, POS;
    cout<<"How many elements? ";
    cin>>N;
    cout<<"Enter the array elements: ";
    for(I=0; I<N; I++)
    cin>>AR[I];
    for(I=0; I < N-1; i++)
    {
        MIN=AR[I];
        POS=I;
        for(J = I+1; J < N; J++)
            if(AR[J]<MIN)
            {
                MIN=AR[J];
                POS=J;
            }
        if(POS != I)
        {
            AR[POS]=AR[I];
            AR[I]=MIN;
        }
    }
    cout<<"Sorted array is: ";
    for(I=0; I<N; I++)
        cout<<AR[I]<<"\t";
    return 0;
}
```

A sample output of Program 8.3 is given below:

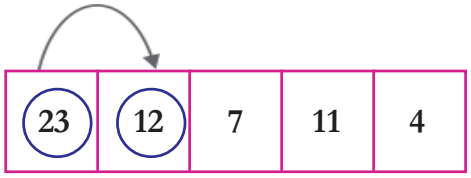
```
How many elements? 5
Enter the array elements: 12 3   6   1   8
Sorted array is: 1   3   6   8   12
```

b. Bubble sort

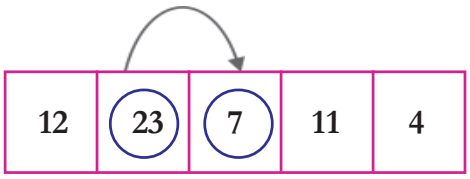
Bubble sort is a sorting algorithm that works by repeatedly stepping through lists that need to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order. This passing procedure is repeated until no swaps are required, indicating that the list is sorted. Bubble sort gets its name because larger element bubbles towards the top of the list. To see a specific example, examine the list of numbers.

Initial list

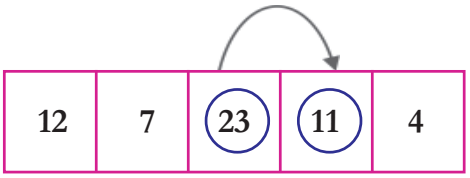
Pass 1



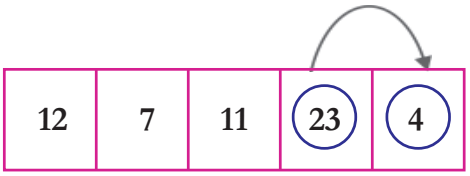
The first comparison results in exchanging the first two elements, 23 and 12.



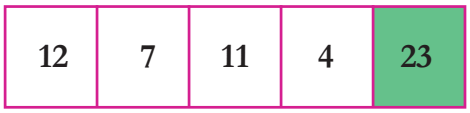
The second comparison results in exchanging the second and third elements 23 and 7 in the revised list.



The third comparison results in exchanging the third and fourth elements 23 and 11 in the revised list.

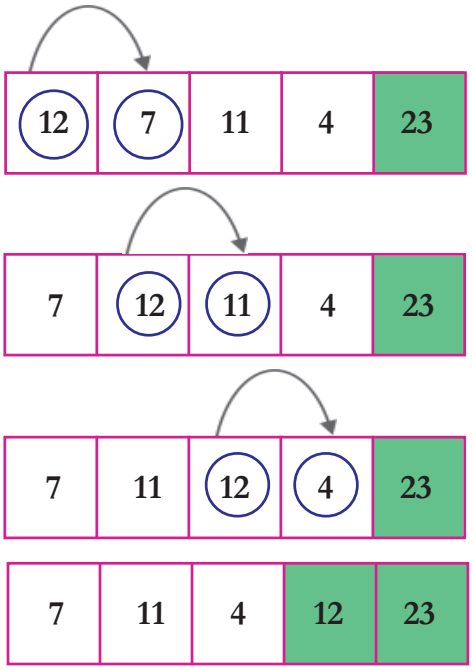


The fourth comparison results in exchanging the fourth and fifth elements 23 and 4 in the revised list.



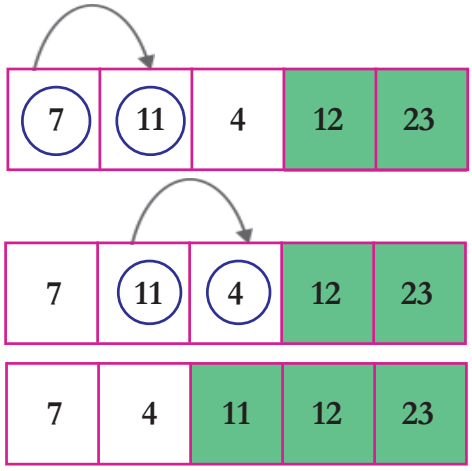
After the end of the first pass, the largest element 23 is bubbled to the last position of the list.

Pass 2



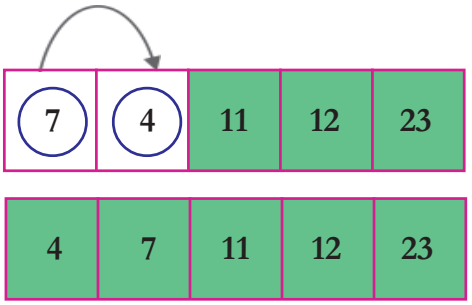
In the second pass, we consider only the four elements of the list, excluding 23. The same process is continued as in Pass 1 and as a result of it 12 is bubbled to fourth position, which is the second largest element in the list.

Pass 3



In the third pass, we consider only the three elements of the list, excluding 23 and12. The same process is continued as in the above pass and as a result of it '11' is bubbled to the 3rd position.

Pass 4



In the last pass we consider only the two elements of the list, excluding 23, 12 and 11. The same process is continued as in the above pass and as a result of it 7 is bubbled to 2nd position and eventually 4 is placed in the first position.

Now we get the sorted list of elements. In bubble sort, to sort a list of 'N' elements we require (N-1) passes. In each pass the size of the revised list will be reduced by one.

Algorithm for bubble sort

Step 1. Start
Step 2. Accept a value in N as the number of elements of the array
Step 3. Accept N elements into the array AR
Step 4. Repeat Steps 5 to 7, (N - 1) times
Step 5. Repeat Step 6 until the second last element of the list
Step 6. Starting from the first position, compare two adjacent elements in the list. If they are not in proper order, swap the elements.
Step 7. Revise the list by excluding the last element in the current list.
Step 8. Print the sorted array AR
Step 9. Stop

Program 8.4 uses **AR** as an integer array to store maximum of 25 numbers, **N** as the number of elements to be sorted.

Program 8.4: Bubble sort for arranging elements in ascending order

```
#include <iostream>
using namespace std;
int main()
{   int AR[25],N;
    int I, J, TEMP;
    cout<<"How many elements? ";
    cin>>N;
    cout<<"Enter the array elements: ";
    for(I=0; I<N; I++)
        cin>>AR[I];
    for(I=1; I<N; I++)
        for(J=0; J<N-I; J++)
            if(AR[J] > AR[J+1])
            {
                TEMP = AR[J];
                AR[J] = AR[J+1];
                AR[J+1] = TEMP;
            }
    cout<<"Sorted array is: ";
    for(I=0; I<N; I++)
        cout<<AR[I]<<"\t";
}
```

The following is a sample output of Program 8.4.

```
How many elements? 5
Enter the array elements: 23   10   -3   7    11
Sorted array is: -3    7    10    11    23
```

8.2.3 Searching

Searching is the process of finding the location of the given element in the array. The search is said to be successful if the given element is found, that is the element exists in the array; otherwise unsuccessful. There are basically two approaches to search operation: linear search and binary Search.

The algorithm that one chooses generally depends on organization of the array elements. If the elements are in random order, the linear search technique is used, and if the array elements are sorted, it is preferable to use the binary search technique. These two search techniques are described below:

a. Linear search

Linear search or sequential search is a method for finding a particular value in a list. Linear search consists of checking each element in the list, one at a time in sequence starting from the first element, until the desired one is found or the end of the list is reached.

Assume that the element '45' is to be searched from a sequence of elements 50, 18, 48, 35, 45, 26, 12. Linear search starts from the first element 50, comparing each element until it reaches the 5th position where it finds 45 as shown in Figure 8.3.

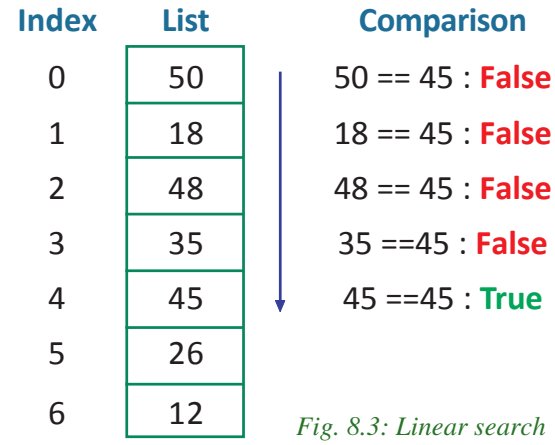


Fig. 8.3: Linear search

Algorithm for Linear Search

Step 1. Start

Step 2. Accept a value in N as the number of elements of the array

Step 3. Accept N elements into the array AR

Step 4. Accept the value to be searched in the variable ITEM

Step 5. Set LOC = -1

Step 6. Starting from the first position, repeat Step 7 until the last element

Step 7. Check whether the value in ITEM is found in the current position. If found then store the position in LOC and Go to Step 8, else move to the next position.

Step 8. If the value of LOC is less than 0 then display "Not Found", else display the value of LOC + 1 as the position of the search value.

Step 9. Stop

Program 8.5 uses **AR** as an integer array to store maximum of 25 numbers, **N** as the number of elements in the array, **ITEM** as the element to be searched and **LOC** as the position or index of the search element.

Program 8.5: Linear search to find an item in the array

```
#include <iostream>
using namespace std;
int main()
{
    int AR[25], N;
    int I, ITEM, LOC=-1;
    cout<<"How many elements? ";
    cin>>N;
    cout<<"Enter the array elements: ";
    for(I=0; I<n; I++)
       cin>>AR[I];
   cout<<"Enter the item you are searching for: ";
    cin>>ITEM;
    for(I=0; I<N; I++)
        if(AR[I] == ITEM)
        {
            LOC=I;
            break;
        }
    if(LOC!=-1)
        cout<<"The item is found at position "<<LOC+1;
    else
        cout<<"The item is not found in the array";
    return 0;
}
```

A sample output of program 8.5 is given below:

```
How many Elements?  7
Enter the array elements: 12 18 26 35 45 48 50
Enter the item you are searching for: 35
The item is found at position 4
```

The following output shows the other side:

```
How many Elements? 7
Enter the array elements: 12 18 26 35 45 48 50
Enter the item you are searching for: 25
The item is not found in the array
```

As noted previously, an advantage of the linear search method is that the list need not be in sorted order to perform the search operation. If the search item is towards the front of the list, only a few comparisons are enough. The worst case occurs when the search item is at the end of the list. For example, for a list of 10,000 elements, maximum number of comparisons needed is 10,000.

b. Binary search

The linear search algorithm which we have seen is the most simple and convenient for small arrays. But when the array is large and requires many repeated searches, it makes good sense to have a more efficient algorithm. If the array contains a sorted list then we can use a more efficient search algorithm called Binary Search which can reduce the search time.

For example, suppose you want to find the meaning of the term 'modem' in a dictionary. Obviously, we don't search page by page. We open the dictionary in the middle (roughly) to determine which half contains the term being sought. Then for subsequent search one half is discarded and we search in the other half. This process is continued till we locate the required term or it results in unsuccessful search. The second case concludes that the term is not found in the dictionary. This search method is possible in a dictionary because the words are in sorted order.

Binary search is an algorithm which uses minimum number of searches for locating the position of an element in a sorted list, by checking the middle, eliminating half of the list from consideration, and then performing the search on the remaining half. If the middle element is equal to the searched value, then the position has been found; otherwise the upper half or lower half is chosen for search, based on whether the element is greater than or less than the middle element.

Algorithm for binary search

Step 1. Start

Step 2. Accept a value in MAX as the number of elements of the array

Step 3. Accept MAX elements into the array LIST

Step 4. Accept the value to be searched in the variable ITEM

Step 5. Store the position of the first element of the list in FIRST and that of the last in LAST

Step 6. Repeat Steps 7 to 11 While (FIRST <= LAST)

Step 7. Find the middle position using the formula (FIRST + LAST)/2 and store it in MIDDLE

Step 8. Compare the search value in ITEM with the element at the MIDDLE of the list

Step 9. If the MIDDLE element contains the search value in ITEM then stop search, display the position and go to Step 12.

Step 10. If the search value is smaller than the MIDDLE element

Then set LAST = MIDDLE - 1

Step 11. If the search value is larger than the MIDDLE element

Then set FIRST = MIDDLE + 1

Step 12. Stop

In Program 8.6, **LIST** is used as an integer array to store maximum of 25 numbers, **MAX** as the number of elements in the array, **ITEM** as the element to be searched and **LOC** as the position number or index of the search element. **FIRST**, **LAST** and **MIDDLE** are used to the refer the first, last and middle positions respectively of the list under consideration.

Program 8.6: Binary search to find an item in the sorted array

```
#include <iostream>
using namespace std;
int main()
{   int LIST[25],MAX;
    int FIRST, LAST, MIDDLE, I, ITEM, LOC=-1;
    cout<<"How many elements? ";
    cin>>MAX;
    cout<<"Enter array elements in ascending order: ";
    for(I=0; I<MAX; I++)
        cin>>LIST[I];
```

```
    cout<<"Enter the item to be searched: ";
    cin>>ITEM;
    FIRST=0;
    LAST=MAX-1;
    while(FIRST<=LAST)
    {
         MIDDLE=(FIRST+LAST)/2;
         if(ITEM == LIST[MIDDLE])
         {
              LOC = MIDDLE;
              break;
         }
         if(ITEM < LIST[MIDDLE])
              LAST = MIDDLE-1;
         else
              FIRST = MIDDLE+1;
    }
    if(LOC != -1)
         cout<<"The item is found at position "<<LOC+1;
    else
         cout<<"The item is not found in the array";
    return 0;
}
```

The following is a sample output of Program 8.6

```
How many elements? 7
Enter array elements in ascending order: 21 28 33 35 45 58 61
Enter the item to be searched: 35
The item is found at position 4
```

Let us consider the following sorted array with 7 elements to illustrate the working of the binary search technique. Assume that the element to be searched is 45.

0	1	2	3	4	5	6
21	28	33	35	45	58	61

FIRST = 0

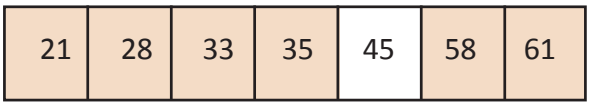
LAST = 6

As FIRST<=LAST, let's start iteration

0	1	2	3	4	5	6
21	28	33	35	45	58	61

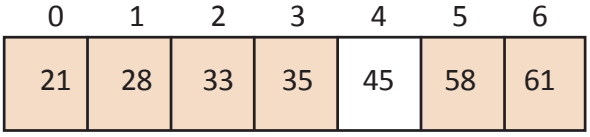
MIDDLE = (**FIRST**+**LAST**)/2 = (0+6)/2 = 3
Here **LIST[3]** is not equal to **45** and **LIST[3]** is less than search element therefore, we take
FIRST = **MIDDLE** + 1 = 3 + 1 = 4, **LAST** = 6

As FIRST<=LAST, we start next iteration.



MIDDLE = (**FIRST**+**LAST**)/2 = (4+6)/2= 5
Here **LIST[5]** is not equal to **45** and **LIST[5]** is greater than the search element therefore, we take **FIRST** = 4, **LAST** = **MIDDLE** - 1=5 -1 = 4,

As FIRST<=LAST, we start next iteration



MIDDLE = (**FIRST**+**LAST**)/2 = (4+4)/2 = 4
Here **LIST[4]** is equal to **45** and the **search terminates successfully**.

In Binary search, an array of 100,00,00,000 (hundred crores) elements requires a maximum of only 30 comparisons to search an element. If the number of elements in the array is doubled, only one more comparison is needed.

Table 8.1 shows how linear search method differs from binary search:

Linear search method	Binary search method
• The elements need not be in any order	• The elements should be in sorted order
• Takes more time for the process	• Takes very less time for the process
• May need to visit all the elements	• All the elements are never visited
• Suitable when the array is small	• Suitable when the array is large

Table 8.1: Comparison of linear and binary search methods

8. 3 Two dimensional (2D) arrays

Suppose we have to store marks of 50 students in six different subjects. Here we can use six single dimensional arrays with 50 elements each. But managing these data with this arrangement is not an easy task. In this situation we can use an array of arrays or two dimensional arrays.

A two dimensional array is an array in which each element itself is an array. For instance, an array AR[m][n] is a 2D array, which contains **m** single dimensional arrays, each of which has **n** elements. Otherwise we can say that AR[m][n] is a table containing **m** rows and **n** columns.

8.3.1 Declaring 2D arrays

The general form of a two dimensional array declaration in C+ + is as follows :

```
data_type  array_name[rows][columns];
```

where `data_type` is any valid data type of C++ and elements of this 2D array will be of this type. The `rows` refers to the number of rows in the array and `columns`

refers to the number of columns in the array. The indices (subscripts) of rows and columns, start at 0 and ends at (`rows-1`) and (`columns-1`) respectively. The following declaration declares an array named `marks` of size 5 × 4 (5 rows and 4 columns) of type integer.

```
int marks[5][4];
```

The elements of this array are referred to as `marks[0][0]`, `marks[0][1]`, `marks[0][2]`,`marks[0][3]`,`marks[1][0]`,`marks[1][1]`,..., `marks[4][3]` as shown in Figure 8.4.

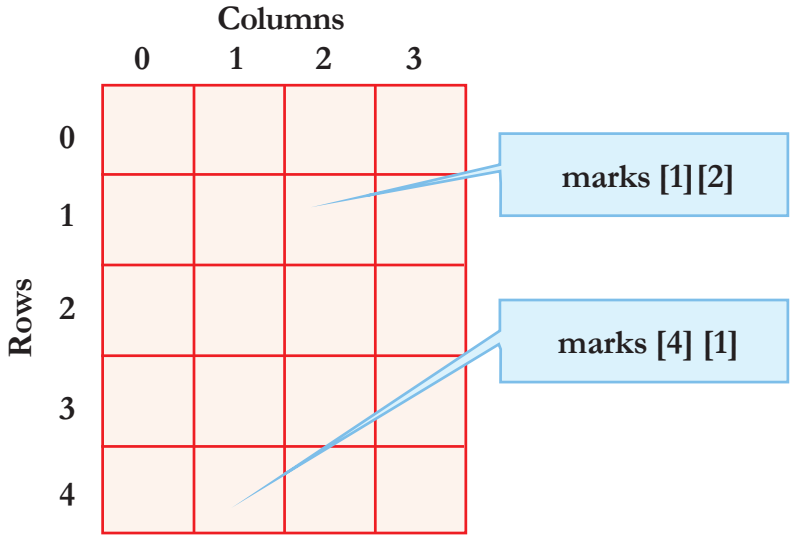


Fig. 8.4:Structuire of a 2D array

The amount of storage required to hold a two dimensional array depends upon its base type, number of rows and number of columns. The formula to calculate total number of bytes required for a 2D array is as follows:

total_bytes = `sizeof`(base type) × number of rows × number of columns

For instance, the above declared array `marks[5][4]` requires 4×5×4=80 bytes.

8.3.2 Matrices as 2D arrays

Matrix is a useful concept in mathematics. We know that a matrix is a set of $m \times n$ numbers arranged in the form of a table with m rows and n columns. Matrices can be represented through 2D arrays. The following program illustrates some operations on matrices. To process a 2D array, you need to use nested loops. One loop processes the rows and other the columns. Normally outer loop is for rows and inner loop is for columns. Program 8.7 creates a matrix `mat` with `m` rows and `n` columns.

Program 8.7: To create a matrix with m rows and n columns

```
#include <iostream>
using namespace std;
int main()
{   int m, n, row, col, mat[10][10];
    cout<< "Enter the order of matrix: ";
    cin>> m >> n;
    cout<<"Enter the elements of matrix\n";
    for (row=0; row<m; row++)
        for (col=0; col<n; col++)
            cin>>mat[row][col];
    cout<<"The given matrix is:";
    for (row=0; row<m; row++)
    {
        cout<<endl;
        for (col=0; col<n; col++)
            cout<<mat[row][col]<<"\t";
    }
    return 0;
}
```

Creation of the matrix

Display the elements in matrix format

A sample output of Program 8.7 is given below:

```
Enter the order of matrix: 3  4
Enter the elements of matrix
1  2  3  4  2  3  4  5  3
The given matrix is:
1   2   3   4
2   3   4   5
3   4   5   6
```

Note that the elements of the matrix are entered sequentially but the output is given with the specified matrix format.

Let us see a program that accepts the order and elements of two matrices and displays their sum. Two matrices can be added only if their order is the same. The elements of the sum matrix are obtained by adding the corresponding elements of the operand matrices. If A and B are two operand matrices, each element in the sum matrix C will be of the form C[i][j] = A[i][j] + B[i][j], where i indicates the row position and j the column position.

Program 8.8: To find the sum of two matrices if conformable

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main()
{   int m1, n1, m2, n2, row, col;
    int A[10][10], B[10][10], C[10][10];
    cout<<"Enter the order of first matrix: ";
    cin>>m1>>n1;
    cout<<"Enter the order of second matrix: ";
    cin>>m2>>n2;
    if(m1!=m2 || n1!=n2)
    {
        cout<<"Addition is not possible";
        exit(0);
    }
    cout<<"Enter the elements of first matrix\n";
    for (row=0; row<m1; row++)
        for (col=0; col<n1; col++)
            cin>>A[row][col];
    cout<<"Enter the elements of second matrix\n";
    for (row=0; row<m2; row++)
        for (col=0; col<n2; col++)
            cin>>B[row][col];
    for (row=0; row<m1; row++)
        for (col=0; col<n1; col++)
            C[row][col] = A[row][col] + B[row][col];
    cout<<"Sum of the matrices:\n";
    for(row=0; row<m1; row++)
    {
        cout<<endl;
        for (col=0; col<n1; col++)
            cout<<C[row][col]<<"\t";
    }
}
```

To use exit() function

Program terminates

Creation of first matrix

Creation of second matrix

Matrix addition process. Instead of m1 and n1, we can use m2 and n2.

A sample output of Program 8.8 is given below:

```
Enter the order of first matrix: 3   4
Enter the order of second matrix: 3   4
Enter the elements of first matrix
2    5    -3    7
5    12   4     9
-3   0    6     -5
```

Here the elements are entered in matrix form. But it is not essential

```
Enter the elements of second matrix
1    4    3    5
4    -5   7    13
3    -4   7    9
Sum of the matrices:
3    9    0    12
9    7    11   22
0    -4   13   4
```

The subtraction operation on matrices can be performed in the same fashion as in Program 8.8 except that the formula is C[i][j] = A[i][j] – B[i][j].

Now, let us write a program to find the sum of the diagonal elements of a square matrix. A matrix is said to be a square matrix, if the number of rows and columns are the same. Though there are two diagonals for a square, here we mean the elements `mat[0][0]`, `mat[1][1]`, `mat[2][2]`, ..., `mat[n-1][n-1]`, where **`mat`** is the 2D array. These diagonal elements are called leading or major diagonal elements. Program 8.9 can be used to find the sum of the major diagonal elements.

Program 8.9: To find the sum of major diagonal elements of a matrix

```
#include <iostream>
using namespace std;
int main()
{   int mat[10][10], n, i, j, s=0;
    cout<<"Enter the rows/columns of square matrix: ";
    cin>>n;
    cout<<"Enter the elements\n";
    for(i=0; i<n; i++)
        for(j=0; j<n; j++)
            cin>>mat[i][j];
    cout<<"Major diagonal elements are\n";
    for(i=0; i<n; i++)
    {
        cout<<mat[i][i]<<"\t";
        s = s + mat[i][i];
    }
    cout<<"\nSum of major diagonal elements is: ";
    cout<<s;
    return 0;
}
```

Accesses only the diagonal elements to find the sum

When Program 8.9 is executed the following output is obtained:

```
Enter the rows/columns of square matrix: 3
Enter the elements
3    5    -2
7    4    0
2    8    -1
Major diagonal elements are
3    4    -1
Sum of major diagonal elements is: 6
```

Each matrix has a transpose. It is obtained by converting row elements into column elements or vice versa. Program 8.10 shows this process.

Program 8.10: To find the transpose of a matrix

```
#include <iostream>
using namespace std;
int main()
{   int ar[10][10], m, n, row, col;
    cout<<"Enter the order of matrix: ";
    cin>>m>>n;
    cout<<"Enter the elements\n";
    for(row=0; row<m; row++)
        for(col=0; col<n; col++)
            cin>>ar[row][col];
    cout<<"Original matrix is\n";
    for(row=0; row<m; row++)
    {
        cout<<"\n";
        for(col=0; col<n; col++)
            cout<<ar[row][col]<<"\t";
    }
    cout<<"\nTranspose of the entered matrix is\n";
    for(row=0; row<n; row++)
    {
        cout<<"\n";
        for(col=0; col<m; col++)
            cout<<ar[col][row]<<"\t";
    }
    return 0;
}
```

Note that the positions of row size and column size are changed in loops

Subscripts also changed their positions

A sample output of Program 8.10 is given below:

```
Enter the order of matrix: 4    3
Enter the elements
3     5     -1
2     12    0
6     8     4
7     -5    6
Original matrix is
3     5     -1
2     12    0
6     8     4
7     -5    6
Transpose of the entered matrix is
3     2     6     7
5     12    8     -5
-1    0     4     6
```

These elements can be entered in a single line

When data is arranged in tabular form, in some situations, we may need sum of elements of each row as well as each column. Program 8.11 helps the computer to perform this task.

Program 8.11: To find the row sum and column sum of a matrix

```
#include <iostream>
using namespace std;
int main()
{
int ar[10][10], rsum[10]={0}, csum[10]={0};
int m, n, row, col;
cout<<"Enter the number of rows & columns in the array: ";
cin>>m>>n;
cout<<"Enter the elements\n";
for(row=0; row<m; row++)
    for(col=0; col<n; col++)
        cin>>ar[row][col];
for(row=0; row<m; row++)
    for(col=0; col<n; col++)
    {
        rsum[row] += ar[row][col];
        csum[col] += ar[row][col];
    }
cout<<"Row sum of the 2D array is\n";
```

Row elements and column elements are added separately and each sum is stored in respective locations of the arrays concerned

```
for(row=0; row<m; row++)
    cout<<rsum[row]<<"\t";
cout<<"\nColumn sum of the 2D array is\n";
for(col=0; col<n; col++)
    cout<<csum[col]<<"\t";
return 0;
}
```

A sample output of Program 8.11 is given below:

```
Enter the number of rows & columns in the array: 3   4
Enter the elements
3    12    5     0
4    -6    2     1
5    7     -6    2
Row sum of the 2D array is
20   1     8
Column sum of 2D array is
12   13    1     3
```

8.4 Multi-dimensional arrays

Each element of a 2D array may be another array. Such an array is called 3D (Three Dimensional) array. Its declaration is as follows:

```
data_type  array_name[size_1][size_2][size_3];
```

The elements of a 3D array are accessed using three subscripts. If ar[10][5][3] is declared as a 3D array in C++, the first element is referenced by ar[0][0][0] and the last element by ar[9][4][2]. This array can contain 150 (10 × 5 × 3) elements. Similarly more sizes can be specified while declaring arrays so that multi-dimensional arrays are formed.

Let us sum up

Array is a collection of elements placed in contiguous memory locations identified by a common name. Each element in an array is referenced by specifying its subscript with the array name. Array elements are accessed easily with the help of `for` loop. Operations like traversing, sorting and searching are performed on arrays. Bubble sort and selection sort methods are used to sort the elements. Linear search and binary search techniques are applied to search an element in an array. Two dimensional arrays are used to solve matrix related problems. We need two subscripts to refer to an element of 2D array. Besides 2D arrays, it is possible to create multidimensional arrays in C++.

Learning outcomes

After the completion of this chapter learner will be able to

- identify scenarios where an array can be used.
- declare and initialize single dimensional and 2D arrays.
- develop logic to perform various operations on arrays like sorting and searching.
- solve matrix related problems with the help of 2D arrays.

Lab activity

1. Write a C++ program to input the amount of sales for 12 months into an array named `SalesAmt`. After all the input, find the total and average amount of sales.
2. Write a C++ program to create an array of N numbers, find the average and display those numbers greater than the average.
3. Write a C++ program that specifies three one-dimensional arrays named price, quantity and amount. Each array should be capable of holding 10 elements. Use a `for` loop to input values to the arrays price and quantity. The entries in the array amount should be the product of the corresponding values in the arrays price and quantity (i.e. amount[i] = price[i] ×quantity[i]). After all the data has been entered, display the following output, with the corresponding value under each column heading as follows:

Price	Quantity	Amount
_____	_____	_____
_____	_____	_____

4. Write a C++ program to input 10 integer numbers into an array and determine the maximum and minimum values among them.
5. Write a C++ program which reads a square matrix of order **n** and prints the upper triangular elements. For example, if the matrix is

```
2   3   1
7   1   5
2   5   1
```

The output should be

```
2   3   1
    1   5
        1
```

6. Write a program which reads a square matrix of order **n** and prints the lower triangular elements. For instance, if the matrix is

```
2  3  1
7  1  5
2  5  7
```

The output will be

```
2
7  1
2  5  7
```

7. Write a C++program to find the sum of leading diagonal elements of a matrix.
8. Write a C++ program to find the sum of off diagonal elements of a matrix.
9. Write a program to print the Pascal's triangle as shown below:

```
1
1  2  1
1  3  3  1
1  4  6  4  1
```

Sample questions

Very short answer type

1. All the elements in an array must be _____ data type.
2. The elements of an array with ten elements are numbered from ____ to ____.
3. An array element is accessed using _____.
4. If AR is an array, which element will be referenced using `AR[7]`?
5. Consider the array declaration `int a[3]={2,3,4};` What is the value of `a[1]`?
6. Consider the array declaration `int a[ ]={1,2,4};` What is the value of `a[1]`?
7. Consider the array declaration `int a[5]={1,2,4};` What is the value of `a[4]`?
8. Write array initialization statements for an array of 6 scores: 89, 75, 82, 93, 78, 95.
9. Printing all the elements of an array is an example for _____ operation.
10. How many bytes are required to store the array `int a[2][3];`?
11. In an array of m elements, Binary search required maximum of n search for finding an element. How many search required if the number of elements is doubled?

12. Write the statement to initialize an array of 10 marks with value 0.
13. State True or false: The compiler will complain if you try to access array element 16 in a ten-element array.

Short answer type

1. Define an Array.
2. What does the declaration `int studlist[1000];` mean?
3. How is memory allocated for a single dimensional array?
4. Write C++ statements to accept an array of 10 elements and display the count of even and odd numbers in it.
5. Write the initialization statement for an array num with values 2, 3, 4, 5.
6. What is meant by traversal?
7. Define sorting.
8. What is searching?
9. What is bubble sort?
10. What is binary search?
11. Define a 2D array.
12. How is memory allocated for a 2D array?

Long answer type

1. An array AR contains the elements 25, 81, 36, 15, 45, 58, 70. Illustrate the working of binary search technique for searching an element 45.
2. Write C++ statements to accept two single dimensional array of equal length and find the difference between corresponding elements.
3. Illustrate the working of bubble sort method for sorting the elements 32, 25, 44, 16, 37, 12.
4. If 24, 45, 98, 56, 76, 24, 15 are the elements of an array, illustrate the working of selection sort for sorting.
5. Write a program to find the difference between two matrices.
6. Write a program to find the sum and average of elements in a 2D array.
7. Write a program to find the largest element in a 2D array.