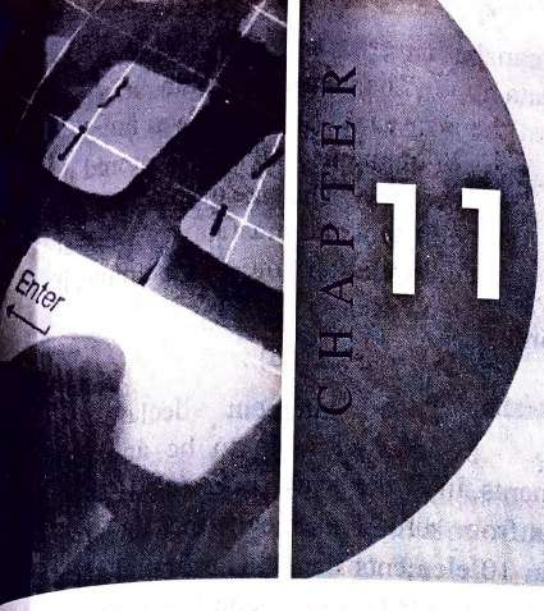


CHAPTER 11

Arrays



LEARNING OBJECTIVE

In the previous chapters, we have been storing data as simple variables. Although storing values in a simple variable is useful, we often need a variable that can store multiple values. In this chapter, we will read about arrays. An array is a user-defined data type which stores related information together. However, all the information stored in an array belongs to the same data type.

In this chapter we will learn how arrays are defined, declared, initialized, accessed, etc. This chapter will also discuss different operations that can be performed on array elements and different types of arrays such as two-dimensional arrays, multidimensional arrays, and sparse matrices.

11.1 INTRODUCTION

Take a situation in which we have 20 students in a class and we are asked to write a program that reads and prints the marks of all these 20 students. In this program, we will need 20 integer variables with different names, as shown in Figure 11.1.

Now to read values for these 20 different variables, we must have 20 read statements. Similarly, to print the value of these variables, we need 20 write statements. If it is just a matter of 20 variables, then it might be acceptable for the user to follow this approach. However, imagine if it will be possible to follow this approach if we had to read and print marks of the students

- in the entire course (say 100 students)
- in the entire college (say 500 students)
- in the entire university (say 10,000 students)



Figure 11.1 20 variables with different names

The answer is no, definitely not. To process a large amount of data, we need a data structure known as *array*. An array is a collection of similar data elements. These data elements have the same data type. The elements

Programming Tip: In an array of size *n*, the index ranges from 0 to *n* – 1.

of the array are stored in consecutive memory locations and are referenced by an index (also known as the subscript). Subscript indicates an ordinal number of the elements counted from the beginning of the array. Some examples where the concept of an array can be used include the following:

- List of temperatures recorded on every day of the month
- List of employees in a company
- List of students in a class
- List of products sold
- List of customers

11.2 DECLARATION OF ARRAYS

We have already seen that every variable must be declared before it is used. The same concept is true in case of array variables also. An array must be declared before being used. Declaring an array means specifying three things:

- Data type—what kind of values it can store, for example, `int, char, float, double`
- Name—to identify the array
- Size—the maximum number of values that the array can hold

Arrays are declared using the following syntax:

```
type name[size];
```

Here the type can be `int, float, double, char,` or any other valid data type. The number within brackets indicates the size of the array, i.e., the maximum number of elements that can be stored in the array. The size of the array is a constant and must have a value at compilation time. For example, if we write,

Programming Tip: To declare and define an array, you must specify its name, type, and size.

```
int marks[10];
```

This statement declares a `marks` variable to be an array containing 10 elements. In C, the array index (also known as subscript) starts from zero. This means that the array `marks` will contain 10 elements in all. The first element will be stored in `marks[0]`, the second element in `marks[1]`, and so on. Therefore, the last element, i.e., the 10th element will be stored in `marks[9]`. Note that 0, 1, 2, 3 written within square brackets are subscripts/indexes. In memory, the array will be stored as shown in Figure 11.2. Arrays of different data types and sizes and their memory representation is shown is Figure 11.3.

Points to Remember

- Note that C does not allow declaring an array whose number of elements is not known at the compile time. Therefore, the following array declarations are illegal in C.

```
int arr[];
int n, arr[n];
```

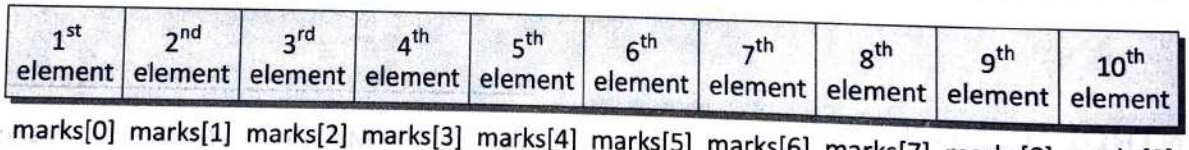


Figure 11.2 Memory representation of an array of 10 elements

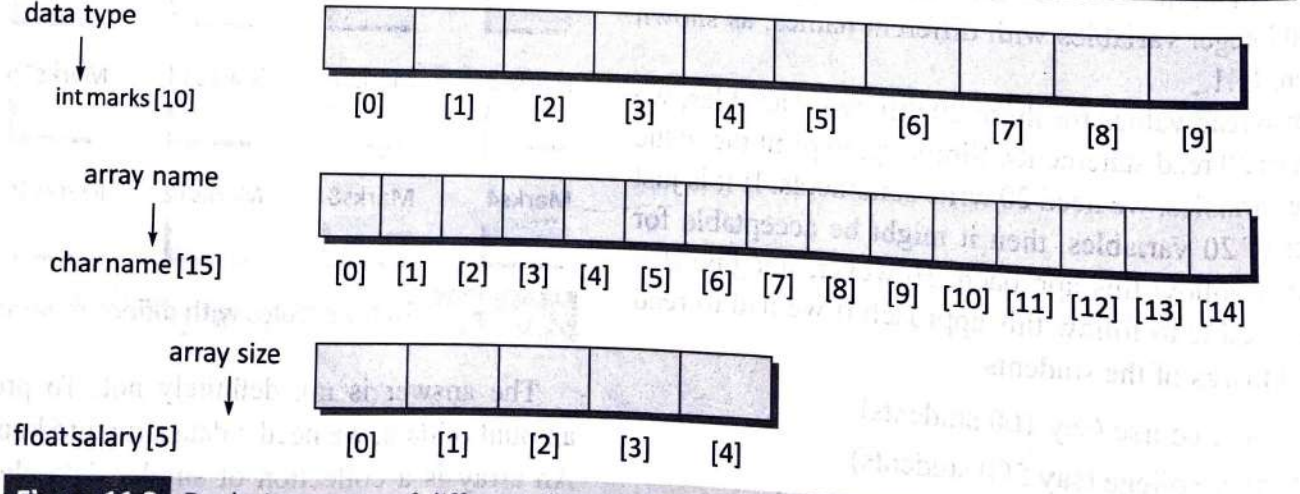


Figure 11.3 Declaring arrays of different data types and sizes

- Generally it is a good programming practice to define the size of an array as a symbolic constant as shown in the following code.

```
#include <stdio.h>
#define N 100
main()
{
   int arr[N];
   ..........
}
```

- The size of the array can be specified using an expression. However, the components of the expression must be available for evaluation of the expression when the program is compiled. Therefore, the following array declarations are valid in C language.

```
#include <stdio.h>
#define N 100
main()
{
   int i=10;
   int arr[N+10], my_arr[i-5*10];
   ...
}
```

Note: C array indices start from 0. So for an array with *N* elements, the index of the last element is *N* – 1.

- C never checks the validity of the array index— neither at compile time nor at run time. So even if you declare an array as

```
int arr[N];
```

The C compiler will not raise any error but the result of running such code is totally unpredictable. Even if you declare an array of 10 elements and later on by mistake try to access the 11th element, no error will be generated. However, the results will be unpredictable as the memory occupied by the (so-called) 11th element may be storing data of another object.

11.3 ACCESSING ELEMENTS OF THE ARRAY

For accessing an individual element of the array, the array subscript must be used. For example, to access the fourth element of the array, you must write `arr[3]`. The subscript/ index must be an integral value or an expression that evaluates to an integral value.

Programming Tip: To access all the elements of the array, you must use a loop. There is no single statement that can do the work.

Although storing the related data items in a single array enables the programmers to develop concise and efficient programs there is no single operation that can operate on all the elements of the array. To access all the elements of the array, you must use a loop. That is, we can access all the elements of the array by varying the value of the subscript into the array, but note that the subscript must be an integral value or an expression that evaluates to an integral value. As shown in Figure 11.2, the first element of the array `marks[10]` can be accessed by writing, `marks[0]`. Now to process all the elements of the array, we will use a loop as shown in Figure 11.4. Figure 11.5 shows the result of the code shown in Figure 11.4.

```
// Set each element of the array to -1
int i, marks[10];
for(i=0;i<10;i++)
      marks[i] = -1;
```

Figure 11.4 Code to initialize each element of the array to –1

The code accesses every individual element of the array and sets its value to –1. In the `for` loop, first the value of `marks[0]` is set to –1, then the value of the index `(i)` is incremented and the next value, i.e., `marks[1]` is set to –1. The procedure is continued until all the 10 elements of the array are set to –1.

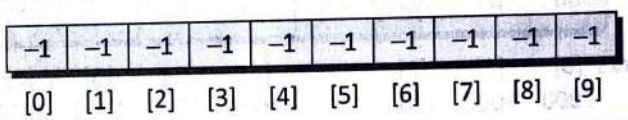


Figure 11.5 The array marks after executing the code given in Figure 11.4.

The name of the array is a symbolic reference for the address to the first byte of the array. Therefore, whenever we use the array name, we are actually referring to the first byte of that array. The index specifies an offset from the beginning of the array to the element being referenced.

Note: There is no single statement that can read, access, or print all the elements of the array. To do this, we have to do it using a for/while/do-while loop to execute the same statement with different index values.

11.3.1 Calculating the Address of Array Elements

You must be wondering how the C language knows where an individual element of the array is located in the memory?

The answer is that the array name is a symbolic reference for the address to the first byte of the array. When we use the array name, we are actually referring to the first byte of the array.

The subscript or the index represents the offset from the beginning of the array to the element being referenced. With just the array name and the index, C can calculate the address of any element in the array.

An array stores all its data elements in consecutive memory locations. So, storing just the base address, i.e, the address of the first element in the array is sufficient. The address of other data elements can simply be calculated using the base address. The formula for doing this calculation is,

```
Address of data element, A[k] = BA(A) + w(k
  - lower_bound)
```

Here, `A` is the array, `k` is the index of the element for which we have to calculate the address, `BA` is the base address of the array `A`, `w` is the word size of one element in memory (for example, size of `int` is 2), and `lower_bound` is the index of the first element in the array.

Example 11.1

Given an array `int marks[] = {99, 67, 78, 56, 88, 90, 34, 85}`. Calculate the address of `marks[4]` if base address = 1000.

Solution

99	67	78	56	88	90	34	85

marks	[0]	[1]	[2]	[3]	**[4]**	[5]	[6]	[7]
	1000	1002	1004	1006	**1008**	1010	1012	1014

We know that storing an integer value requires 2 bytes, therefore here, word size is 2 bytes.

```
marks[4] = 1000 + 2(4 - 0) = 1000 + 2(4)
         = 1008
```

Example 11.2 Given an array, `float avg[] = {99.0, 67.0, 78.0, 56.0, 88.0, 90.0, 34.0, 85.0}`. Calculate the address of `avg[4]` if base address = 1000.

Solution

99.0	67.0	78.0	56.0	88.0	90.0	34.0	85.0

avg	[0]	[1]	[2]	[3]	**[4]**	[5]	[6]	[7]
	1000	1004	1008	1012	**1016**	1020	1024	1028

We know that storing a floating point number requires 4 bytes, therefore, word size is 4 bytes.

```
avg[4] = 1000 + 4(4 - 0) = 1000 + 4(4) = 1016
```

Note: When we write `arr[i]`, the compiler interprets it as the contents of memory slot which is `i` slots away from the beginning of the array `arr`.

11.4 STORING VALUES IN ARRAYS

When we declare and define an array, we are just allocating space for the elements; no values are stored in the array. To store values in the array, there are three ways—first, to initialize the array elements; second, to input a value for every individual element; third, to assign values to the elements. This is shown in Figure 11.6.

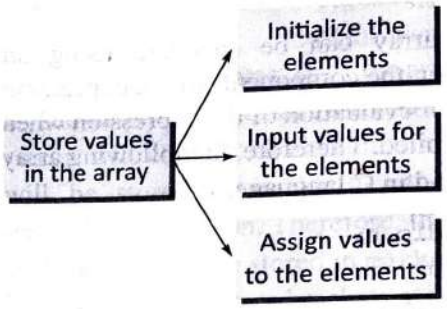


Figure 11.6 Storing values in an array

11.4.1 Initialization of Arrays

Elements of the array can also be initialized at the time of declaration as in the case of every other variable. When an array is initialized, we need to provide a value for every element in the array. Arrays are initialized by writing,

```
type array_name[size]={list of values};
```

The values are written with curly brackets and every value is separated by a comma. It is a compiler error to specify more number of values than the number of elements in the array. When we write,

```
int marks[5]={90, 82, 78, 95, 88};
```

an array with name `marks` is declared that has enough space to store `5` elements. The first element, i.e., `marks[0]` is assigned with the value `90`. Similarly, the second element of the array, i.e., `marks[1]` has been assigned `82`, and so on. This is shown in Figure 11.7.

Programming Tip: By default, the elements of the array are not initialized. They may contain some garbage value, so before using the array you must initialize the array or read some meaningful data into it.

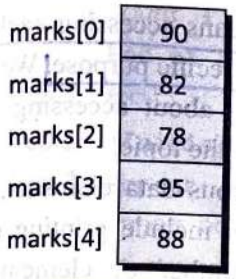


Figure 11.7 Initializing array marks[5]

While initializing the array at the time of declaration, the programmer may omit to mention the size of the array. For example,

```
int marks[]= {98, 97, 90};
```

This statement is absolutely legal. Here, the compiler will allocate enough space for all initialized elements. If the number of values provided is less than the number of elements in the array, the un-assigned elements are filled with zeros. Figure 11.8 illustrates initialization of arrays.

```
int marks[5] = {90, 45, 67, 85, 78};
```

90	45	67	85	78
[0]	[1]	[2]	[3]	[4]

```
int marks[5] = {90, 45};
```

90	45	0	0	0
[0]	[1]	[2]	[3]	[4]

Rest of the elements are filled with 0s

```
int marks[] = {90, 45, 72, 81, 63, 54};
```

90	45	72	81	63	54
[0]	[1]	[2]	[3]	[4]	[5]

```
int marks[5] = {0};
```

0	0	0	0	0	0
[0]	[1]	[2]	[3]	[4]	[5]

Figure 11.8 Initialization of array elements

Note If we have more initializers than the declared size of the array, then a compile time error will be generated. For example, the following statement will result in a compiler error.

```
int marks[3] = {1,2,3,4};
```

11.4.2 Inputting Values

An array can be filled by inputting values from the keyboard. In this method, a `while/do-while` or a `for` loop is executed to input the value for each element of the array. For example, look at the code shown in Figure 11.9.

In the code, we start with the index `i` at `0` and input the value for the first element of the array. Since the array can have `10` elements, we must input values for elements whose index varies from `0` to `9`. Therefore, in the `for` loop, we test for condition (`i<10`) which means the number of elements in the array.

```
// Input value of each element of the array
int i, marks[10];
for(i=0; i<10; i++)
      scanf("%d", &marks[i]);
```

Figure 11.9 Code for inputting each element of the array

11.4.3 Assigning Values

The third way to assign values to individual elements of the array is by using the assignment operator. Any value that evaluates to an appropriate data type as that of the array can be assigned to the individual array element. A simple assignment statement can be written as,

```
marks[3] = 100;
```

Here, `100` is assigned to the fourth element of the array which is specified as `marks[3]`.

We cannot assign one array to another array, even if the two arrays have the same type and size. To copy an array, you must copy the value of every element of the first array into the element of the second array. Figure 11.10 illustrates the code to copy an array.

```
// Copy an array at the individual element level
int i, arr1[10], arr2[10];
for(i=0;i<10;i++)
      arr2[i] = arr1[i];
```

Figure 11.10 Code to copy an array at the individual element level

In Figure 11.10, the code accesses each element of the first array and simultaneously assigns its value to the corresponding element of the second array. Finally, the index value `i` is incremented to access the next element in succession. Therefore, when this code is executed, `arr2[0] = arr1[0]`, `arr2[1] = arr1[1]`, `arr2[2] = arr1[2]`, and so on.

We can also use a loop to assign a pattern of values to the array elements. For example, if we want to fill an array with even integers starting (from `0`), then we will write the code as shown in Figure 11.11.

In the code, we assign to each element value equal to twice of its index, where index starts from zero. So after executing this code we will have, `arr[0] = 0`, `arr[1] = 2`, `arr[2] = 4`, and so on.

```
// Fill an array with even numbers
int i, arr[10];
for(i=0;i<10;i++)
      arr[i] = i*2;
```

Figure 11.11 Code for filling an array with even numbers

11.5 CALCULATING THE LENGTH OF THE ARRAY

Length of the array is given by the number of elements stored in it. The general formula to calculate the length of the array is,

```
Length = upper_bound - lower_bound + 1
```

where `upper_bound` is the index of the last element and `lower_bound` is the index of the first element in the array.

Usually, `lower_bound` is zero but this is not a compulsion as we can have an array whose index may start from any non-zero value.

Example 11.3 Let `Age` be an array of integers such that

`Age[0] = 2` `Age[1] = 5` `Age[2] = 3`
`Age[3] = 1` `Age[4] = 7`

Show the memory representation of the array and calculate its length.

Solution

Memory representation of the array `Age` is as given

2	5	3	1	7
Age[0]	Age[1]	Age[2]	Age[3]	Age[4]

`Length = upper_bound - lower_bound + 1`

Here, `lower_bound = 0, upper_bound = 4`

Therefore, `length = 4 – 0 + 1 = 5`.

11.6 OPERATIONS THAT CAN BE PERFORMED ON ARRAYS

There are a number of operations that can be preformed on arrays. These operations include:

- Traversal
- Insertion
- Search
- Deletion
- Merging
- Sorting

We will study all these operations in detail in this section.

11.6.1 Traversal

Traversing the array element means accessing each and every element of the array for a specific purpose. We have already seen this while reading about accessing array elements. This is just a re-visit of the topic.

If `A` is an array of homogeneous data elements, then traversing the data elements can include printing every element, counting the total number of elements, or performing any process on these elements. Since an array is a linear data structure (because all its elements forms a sequence), traversing its elements is very simple and straightforward. The algorithm for array traversal is given in Figure 11.12.

```
Step 1: [Initialization] Set I = lower_bound
Step 2: Repeat steps 3 to 4 while I <= upper_bound
Step 3: Apply Process to A[I]
Step 4: Set  I = I + 1
        [End of Loop]
Step 5: Exit
```

Figure 11.12 Algorithm for array traversal

In Step 1, we initialize index to the `lower_bound` of the array. In Step 2, a `while` loop is executed. Steps 3 and 4 form a part of the loop. Step 3 processes the individual array element as specified by the array name and index value. Step 4 increments the index value so that the next array element could be processed. The `while` loop in Step 2 is executed until all the elements in the array are processed. In other words, the `while` loop will be executed until `i` is less than or equal to the `upper bound` of the array.

Example 11.4 Assume that there is an array Marks[ ], such that the index of the array specifies the roll number of the student and the value of a particular element denotes the marks obtained by the student. For example, if it is given Marks[4] = 78, then the student whose roll number is 4 has obtained 78 marks in the examination. Now, write an algorithm to:

(a) Find the total number of students who have secured 80 or more marks.

(b) Print the roll number and marks of all the students who have got distinction.

Solution

(a) *Step 1:* `[Initialization] Set I = I + 1`

Step 2: `Repeat for I = lower_bound to upper_bound`

```
        If Marks[I] >= 80, then: Set Count =
        Count + 1
        End of Loop]
Step 3: Exit
```

(b)
```
Step 1: [Initialization] Set I = I + 1
Step 2: Repeat for I = lower_bound to
        upper_bound
        If Marks[I] >= 75, Write: I,
          Marks[I]
        [End of Loop]
Step 3: Exit
```

1. Write a program to read and display n numbers using an array.

```
#include <stdio.h>
#include <conio.h>
int main()
{
   int i=0, n, arr[20];
   clrscr();
   printf("\n Enter the number of elements:");
   scanf("%d", &n);
   for(i=0; i<n; i++)
   {
      printf("\n Arr[%d] = ", i);
      scanf("%d", &arr[i]);
   }
   printf("\n The array elements are ");
   for(i=0; i<n; i++)
      printf("Arr[%d] = %d\t", i, arr[i]);
   return 0;
}
```

Output

```
Enter the number of elements: 5
Arr[0] = 1
Arr[1] = 2
Arr[3] = 3
Arr[4] = 4
Arr[5] = 5
The array elements are
Arr[0] = 1    Arr[1] = 2    Arr[3] = 3
Arr[4] = 4    Arr[5] = 5
```

2. Write a program to read and display *n* random numbers using an array

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#define MAX 10
main()
{
   int arr[MAX], i, RandNo;
   for(i=0; i<MAX; i++)
   {
      /* Scale the random number in the range
         0 to MAX-1 */
      RandNo = rand() % MAX;
      // rand() is a pre-defined function
      arr[i] = RandNo;
   }
   printf("\n The contents of the array are:
          \n");
   for(i=0; i<MAX; i++)
      printf("\t %d", arr[i]);
   getch();
   return 0;
}
```

Output

```
The contents of the array are:
6   0   8   4   7   1   0   2   7   3
```

3. Write a program to find mean of *n* numbers using arrays.

```
#include <stdio.h>
#include <conio.h>
int main()
{
   int i, n, arr[20], sum = 0;
   float mean = 0.0;
   clrscr();
   printf("\n Enter the number of elements in
          the array: ");
   scanf("%d", &n);
   for(i=0; i<n; i++)
   {
      printf("\n Arr[%d] = ", i);
      scanf("%d",&arr[i]);
   }
   for(i=0; i<n; i++)
      sum += arr[i];
   mean = sum/n;
   printf("\n The sum of the array elements
          %d", sum);
   printf("\n The mean of the array element
          = %f", mean);
   return 0;
}
```

Output

```
Enter the number of elements in the array:
Arr[0] = 1
Arr[1] = 2
Arr[2] = 3
```

```
Arr[3] = 4
Arr[4] = 5
The sum of the array elements = 15
The mean of the array elements = 3.00
```

4. Write a program to find the largest of *n* numbers using arrays.

```
#include <stdio.h>
#include <conio.h>
int main()
{
   int i, n, arr[20], large = -1111;
   /* Note we have assigned large with a
   very small value so that any value in the
   array is assured to be greater than this
   value */
   clrscr();
   printf("\n Enter the number of elements in
           the array: ");
   scanf("%d", &n);
   for(i=0; i<n; i++)
      scanf("%d",&arr[i]);
   for(i=0; i<n; i++)
   {
      if(arr[i]>large)
         large = arr[i];
   }
   printf("\n The largest number is: %d",
           large);
   return 0;
}
```

Output

```
Enter the number of elements in the array: 5
1 2 3 4 5
The largest number is: 5
```

5. Write a program to print the position of the smallest of *n* numbers using arrays.

```
#include <stdio.h>
#include <conio.h>
int main()
{
   int i, n, arr[20], small = 1234, pos = -1;
   clrscr();
   printf("\n Enter the number of elements in
           the array: ");
   scanf("%d", &n);
   for(i=0; i<n; i++)
      scanf("%d",&arr[i]);
   for(i=0; i<n; i++)
   {
      if(arr[i] < small)
      {
      small = arr[i];
      pos = i;
      }
   }
   printf("\n The smallest number is : %d",
           small);
   printf("\n The position of the smallest
           number in the array is: %d", pos);
   return 0;
}
```

Output

```
Enter the number of elements: 5
1 2 3 4 5
The smallest number is: 1
The position of the smallest number in the
array is: 0
```

6. Write a program to interchange the largest and the smallest number in the array.

```
#include <stdio.h>
#include <conio.h>
int main()
{
   int i, n, arr[20], temp;
   int small = 9999, small_pos = 0;
   int large = -9999, large_pos = 0;
   clrscr();
   printf("\n Enter the number of elements in
           the array: ");
   scanf("%d", &n);
   for(i=0; i<n; i++)
   {
      printf("\n Enter the value of element
           %d: ",i);
      scanf("%d", &arr[i]);
   }
   for(i=0; i<n; i++)
   {
      if(arr[i] < small)
      {
         small = arr[i];
         small_pos = i;
      }
```

```
    if(arr[i] > large)
    {
       large = arr[i];
       large_pos = i;
    }
  }
  printf("\n The smallest of these numbers
         is : %d", small);
  printf("\n The position of the smallest
         number in the array is: %d",
         small_pos);
  printf("\n The largest of these numbers
         is: %d", large);
  printf("\n The position of the largest
         number in the array is: %d",
         large_pos);
  temp = arr[large_pos];
  arr[large_pos] = arr[small_pos];
  arr[small_pos] = temp;
  printf("\n The new array is: ");
  for(i=0; i<n; i++)
    printf(" \n arr[%d] = %d ", i, arr[i]);
  return 0;
}
```

Output

```
Enter the number of elements in the array: 5
1 2 3 4 5
Enter the value of elements 1: 1
Enter the value of elements 2: 2
Enter the value of elements 3: 3
Enter the value of elements 4: 4
Enter the value of elements 5: 5
The smallest of these numbers is : 1
The position of the smallest number in the
array is: 0
The largest of these numbers is: 5
The position of the largest number in the
array is: 4
The new array is:
5  4  3  2  1
```

7. Write a program to find the second biggest number using an array of n numbers.

```
#include <stdio.h>
#include <conio.h>
int main()
{
   int i, n, arr[20], large = -1111, second_
   large = -1234;
   clrscr();

   printf("\n Enter the number of elements
      in the array: ");
   scanf("%d", &n);
   for(i=0; i<n; i++)
   {
      printf("\n Enter the number: ");
      scanf("%d", &arr[i]);
   }
   for(i=0; i<n; i++)
   {
      if(arr[i]>large)
      large = arr[i];
   }
   for(i=0; i<n; i++)
   {
      if(arr[i] != large)
      {
         if(arr[i]>second_large)
         second_large = arr[i];
      }
   }
   printf("\n The numbers you entered are: ");
   for(i=0; i<n; i++)
      printf("%d ", arr[i]);
   printf("\n The largest of these numbers
   is: %d", large);
   printf("\n The second largest of these
   numbers is: %d", second_large);
   return 0;
}
```

Output

```
Enter the number of elements in the array: 5
Enter the number: 1
Enter the number: 2
Enter the number: 3
Enter the number: 4
Enter the number: 5
The numbers you entered are:
1  2  3  4  5
The largest of these numbers is: 5
The second largest of these numbers is: 4
```

8. Write a program to enter n number of digits. Form a number using these digits.

```
#include <stdio.h>
#include <conio.h>
#include <math.h>
int main()
{
   int number=0, digit[10], numofdigits, i;
   clrscr();

   printf("\n Enter the number of digits: ");
```

```
    scanf("%d", &numofdigits);

    for(i=0; i<numofdigits; i++)
    {
       printf("\n Enter digit %d: ", i);
       scanf("%d", &digit[i]);
    }
    i=0;
    while(i<numofdigits)
    {
       number = number + digit[i] * pow(10, i);
       i++;
    }
    printf("\n The number is: %d", number);
    return 0;
}
```

Output

```
Enter the number of digits: 3
Enter digit 0: 3
Enter digit 1: 4
Enter digit 2: 5
The number is: 543
```

9. Write a program to find whether the array of integers contain a duplicate number.

```
#include <stdio.h>
#include <conio.h>
int main()
{
   int array1[10], i, n, j, flag=0;
   clrscr();
   printf("\n Enter the size of the array:");
   scanf("%d", &n);
   for(i=0; i<n; i++)
   {
      scanf("%d", &array1[i]);
   }
   for(i=0; i<n; i++)
   {
      for(j=i+1; j<n; j++)
      {
      if(array1[i] == array1[j] && i!=j)
      {
         flag = 1;
         printf("\n Duplicate number %d found
            at location %d and %d", array1[i],
            i, j);
       }
   }
   }
   if(flag == 0)
     printf("\n No duplicate");
     return 0;
}
```

Output

```
Enter the size of the array: 5
1 2 3 4 5
No duplicate
```

10. Write a program to read marks of 10 students in the range of 0–100. Then make 10 groups: 0–10, 10–20, 20–30, etc. Count the number of values that falls in each group and display the result.

```
#include <stdio.h>
#include <conio.h>
main()
{
   int marks[50], i;
   int group[10]={0};
   printf("\n Enter the marks of 10 students
   : \n");
   for(i=0; i<10; i++)
   {
      printf("\n MARKS[%d] = ", i);
      scanf("%d", &marks[i]);
      ++group[(int)(marks[i])/10];
   }
   printf("\n\n **************************");
   printf("\n GROUP \t\t FREQUENCY");
   for(i=0; i<10; i++)
     printf("\n %d \t\t %d", i, group[i]);
   getch();
   return 0;
}
```

Output

```
Enter the marks of 10 students:
MARKS[0] = 95
MARKS[1] = 88
MARKS[2] = 67
MARKS[3] = 78
MARKS[4] = 81
MARKS[5] = 98
MARKS[6] = 55
MARKS[7] = 45
MARKS[8] = 72
MARKS[9] = 90
**************************
GROUP       FREQUENCY
0             0
1             0
2             0
3             0
4             1
5             1
6             1
```

```
7          2
8          2
9          3
```

11. Modify the program in the previous question to display the frequency histograms of each group.

```
#include <stdio.h>
main()
{
   int marks[10], i;
   int group[10]={0};
   printf("\n Enter the marks of 10 students
   : \n");
   for(i=0; i<10; i++)
   {
      printf("\n MARKS[%d] = ", i);
      scanf("%d", &marks[i]);
      ++group[(int)(marks[i])/10];
   }
   printf("\n\n *************************");
   printf("\n GROUP \t\t FRQUENCY");
   for(i=0; i<10; i++)
      printf("\n %d \t\t %d", i, group[i]);
         i=0;
   printf("\n\n FREQUENCY HISTOGRAM");
   for(index=0; index<10; index++)
   {
      printf("\n GROUP %d | ",index);
      for(i=0; i<group[index]; i++)
      printf(" *");
   }
   getch();
   return 0;
}
```

Output

```
Enter the marks of 10 students:
MARKS[0] = 95
MARKS[1] = 88
MARKS[2] = 67
MARKS[3] = 78
MARKS[4] = 81
MARKS[5] = 98
MARKS[6] = 55
MARKS[7] = 45
MARKS[8] = 72
MARKS[9] = 90
*************************
GROUP      FREQUENCY
0          0
1          0
2          0
3          0
4          1
5          1
6          1
7          2
8          2
9          3
FREQUENCY HISTOGRAM
GROUP 0 |
GROUP 1 |
GROUP 2 |
GROUP 3 |
GROUP 4 |
GROUP 5 | *
GROUP 6 | *
GROUP 7 | * *
GROUP 8 | * *
GROUP 9 | * * *
```

12. Write a program to read a sorted list of floating point values and then calculate and display the median of the values.

```
#include <stdio.h>
#include <conio.h>
main()
{
   int i, j, n;
   float median, values[10];
   printf("\n Enter the size of the array: ");
   scanf("%d", &n);
   printf("\n Enter the values: ");
   for(i=0; i<n; i++)
      scanf("%f", &values[i]);
   if(n%2 == 0)
      median = (values[n/2]+values[n/2+1])/2.0;
   else
      median = values[n/2 + 1];
   printf("\n MEDIAN = %f", median);
   getch();
   return 0;
}
```

Output

```
Enter the size of the array: 5
Enter the values:
12 34 56 78 89
MEDIAN = 56.00
```

11.6.2 Insertion

Inserting an element in the array means adding a new data element in an already existing array. If the element has to be inserted at the end of the existing array, then the task of insertion is quite simple. We just have to add 1 to the upper_bound and assign the value. Here, we assume that the memory space allocated for the array is still available. For example, if an array is declared to contain

10 elements, but currently it is having only 8 elements, then obviously there is space to accommodate two more elements. However, if it already has 10 elements, then we will not be able to add another element to it.

Figure 11.13 shows an algorithm to insert a new element to the end of the array.

```
Step 1: Set upper_bound = upper_bound + 1
Step 2: Set A[upper_bound] = VAL
Step 3: EXIT
```

Figure 11.13 Algorithm to append a new element to an existing array

In Step 1 of the array, we increment the value of the upper_bound. In Step 2, the new value is stored at the position pointed by upper_bound.

For example, if we have an array that is declared as

```
int marks[60];
```

The array is declared to store marks of all the students in the class. Now suppose that there are 54 students and a new student comes and is asked to given the same test. The marks of this new student would be stored in marks[55]. Assuming that the student secured 68 marks, we will assign the value as,

```
marks[55] = 68;
```

However, if we have to insert an element in the middle of the array, then this is not a trivial task. On an average, we might have to move as much as half of the elements from its position in order to accommodate space for the new element.

For example, consider an array whose elements are arranged in ascending order. Now, if a new element has to be added, it will have to be added probably somewhere in the middle of the array. To do this, what we will have to do is, first find the location where the new element will be inserted and then move all the elements (that have a value greater than that of the new element) one space to the right so that space can be created to store the new value.

Example 11.5 Data[] is an array that is declared as int Data[20]; and contains the following values:

```
Data[] = {12, 23, 34, 45, 56, 67, 78, 89, 90, 100};
```

(a) Calculate the length of the array.

(b) Write the upper and lower bounds.

(c) Give the memory representation of the array.

(d) If a new data element with value 75 has to be inserted, find its position.

(e) Insert the new data element and then give the memory representation of the array.

(f) length of the array = number of elements

Therefore, length of the array = 10

(g) By default, lower_bound = 0, (it can be set to any value) and upper_bound = 9

12	23	34	45	56	67	78	89	90	100
Data [0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

(h) Since the elements of the array are stored in ascending order, the new data element will be stored after 67, i.e., at the 6th location. So, all the array elements from the 6th position will be moved one space towards the right to accommodate the new value.

12	23	34	45	56	67	75	78	89	90	100
Data [0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]

Algorithm to Insert an Element in the Middle of the Array

The algorithm INSERT will be declared as INSERT(A, N, POS, VAL). The arguments are

(a) A, the array in which the element has to be inserted

(b) N, the number of elements in the array

(c) POS, the position at which the element has to be inserted, and

(d) VAL, the value that has to be inserted.

In the algorithm given in Figure 11.14, in Step 1, we first initialize I with the total number of elements in the array.

```
Step 1: [INITIALIZATION] SET I = N
Step 2: Repeat Steps 3 and 4 while I >= POS
Step 3:                SET A[I + 1] = A[I]
Step 4:                SET I = I - 1
        [End of Loop]
Step 5: SET N = N + 1
Step 6: SET A[POS] = VAL
Step 7: EXIT
```

Figure 11.14 Algorithm to insert a new element at a specified position

In Step 2, a while loop is executed which will move all the elements that have index greater than POS one space towards right to create space for the new element. In Step 5, we increment the total number of elements in the array by 1 and finally in Step 6, the new value is inserted at the desired position.

Now, let us visualize this algorithm by taking an example. Initial Data[] is given as shown in Figure 11.15.

Calling INSERT(Data, 6, 3, 100) will lead to the following processing in the array:

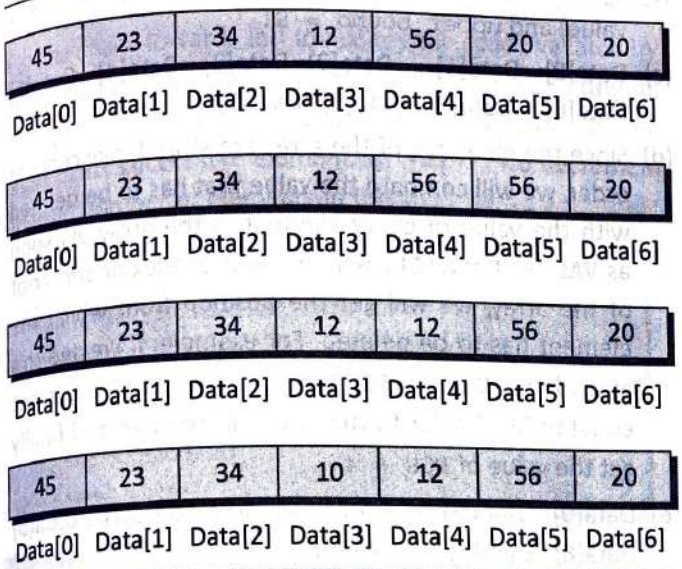


Figure 11.15 Inserting a new value in an existing array

13. Write a program to insert a number at a given location in an array.

```
#include <stdio.h>
#include <conio.h>
int main()
{
   int i, n, num, pos, arr[10];
   clrscr();
   printf("\n Enter the number of elements
   in the array: ");
   scanf("%d", &n);
   for(i=0; i<n; i++)
   {
      printf("\n Enter the values:");
      scanf("%d", &arr[i]);
   }
   printf("\n Enter the number to be
   inserted: ");
   scanf("%d", &num);
   printf("\n Enter the position at which
   the number has to be added: ");
   scanf("%d", &pos);
   for(i=n; i>=pos; i--)
      arr[i+1] = arr[i];
   arr[pos] = num;
   n++;
   printf("\n The array after insertion of
   %d is: ", num);
   for(i=0; i<n+1; i++)
      printf("\n Arr[%d] = %d", i, arr[i]);
   getch();
   return 0;
}
```

Output

```
Enter the number of elements in the array: 5
Enter the values:
1 2 3 4 5
Enter the number to be inserted: 7
Enter the position at which the number has
to be added: 3
The array after insertion of 7 is:
1 2 3 7 4 5
```

14. Write a program to insert a number in an array that is already sorted in ascending order.

```
#include <stdio.h>
#include <conio.h>
int main()
{
   int i, n, j, num, arr[10];
   clrscr();
   printf("\n Enter the number of elements
   in the array: ");
   scanf("%d", &n);
   printf("\n Enter the array elements:");
   for(i=0; i<n; i++)
      scanf("%d", &arr[i]);
   printf("\n Enter the number to be inserted: ");
   scanf("%d", &num);
   for(i=0; i<n; i++)
   {
      if(arr[i] > num)
      {
         for(j=n-1; j>=i; j--)
            arr[j+1] = arr[j];
         arr[i] = num;
         break;
      }
   }
   n++;
   printf("\n The array after insertion of
   %d is: ", num);
   for(i=0; i<n+1; i++)
      printf("\t %d", arr[i]);
   getch();
   return 0;
}
```

Output

```
Enter the number of elements in the array: 5
Enter the array elements:
1 2 3 4 5
Enter the number to be inserted: 6
The array after insertion of 6 is:
1 2 3 4 5 6
```

11.6.3 Deletion

Deleting an element from the array means removing a data element from an already existing array. If the element has to be deleted from the end of the existing array, then the task of deletion is quite simple. We just have to subtract 1 from the upper bound. Figure 11.16 shows an algorithm to delete an element from the end of the array.

```
Step 1: Set upper_bound = upper_bound - 1
Step 2: EXIT
```

Figure 11.16 Algorithm to delete the last element of an array

For example, if we have an array that is declared as

```
int marks[];
```

The array is declared to store marks of all the students in the class. Now suppose there are 54 students and the student with roll number 54 leaves the course. The marks of this student was therefore stored in `marks[54]`. We just have to decrement the upper bound. Subtracting `1` from the upper bound will indicate that there are `53` valid data in the array.

However, if we have to delete the element from the middle of the array, then this task is not trivial. On an average, we might have to move as much as half of the elements from its position in order to occupy the space of the deleted element.

For example, consider an array whose elements are arranged in ascending order. Now, if an element has to be deleted probably from somewhere middle in the array. To do this, what we will have to do is, first find the location from where the element has to be deleted and then move all the elements (that have a value greater than that of the element) one space towards the left so that space vacated by the deleted element be occupied by rest of the elements.

Example 11.6 `Data[]` is an array that is declared as `int Data[10];` and contains the following values:

```
Data[] = {12, 23, 34, 45, 56, 67, 78, 89, 90, 100};
```

(a) Calculate the length of the array.

(b) Write the upper and lower bounds.

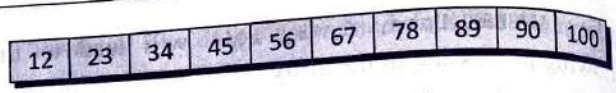
(c) Give the memory representation of the array.

(d) If a data element with value 56 has to be deleted, find its position.

(e) Delete the data element and hence give the memory representation of the array.

Solution

(a) Length of the array = Number of elements
Therefore, length of the array = 10.



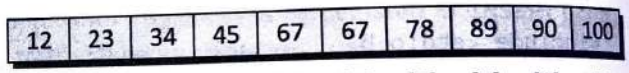
(b) By default, `lower_bound = 0`, (it can be set to any value) and `upper_bound = 9`.

(c) Data[0] Data[1] Data[2] Data[3] Data[4] Data[5] Data[6] Data[7] Data[8] Data[9]

(d) Since the elements of the array are stored in ascending order, we will compare the value that has to be deleted with the value of every element in the array. As soon as `VAL = Data[I]`, where `I` is the index or subscript of the array, we will get the position from which the element has to be deleted. For example, if we see this array, here `VAL = 56. Data[0] = 12` which is not equal to 56. Similar to this, we will compare and finally get the value of `POS = 4`.

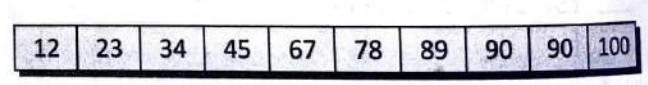
(e) Data[0] Data[1] Data[2] Data[3] Data[4] Data[5] Data[6] Data[7] Data[8] Data[9]















Algorithm to Delete an Element from the Middle of the Array

The algorithm `DELETE` will be declared as `DELETE(A, N, POS)`. The arguments are as follows:

(a) `A`, the array from which the element has to be deleted

(b) `N`, the number of elements in the array

(c) `POS`, the position from which the element has to be deleted

Figure 11.17 shows the algorithm in which we first initialize I with the position from which the element has to be deleted. In Step 2, a while loop is executed which will move all the elements that have index greater than POS one space towards left to occupy the space vacated by the deleted element. When we say that we are deleting an element, we are actually overwriting the element with the value of its successive element. In Step 5, we decrement the total number of elements in the array by 1.

```
Step 1: [INITIALIZATION] SET I = POS
Step 2: Repeat Steps 3 and 4 while I <= N - 1
Step 3: SET A[I] = A[I + 1]
Step 4: SET I = I + 1
        [End of Loop]
Step 5: SET N = N - 1
Step 6: EXIT
```

Figure 11.17 Algorithm to delete an element from the middle of the array

Now, let us visualize this algorithm by taking an example and having a look at Figure 11.17. Initial Data[] is given as shown in Figure 11.18. Calling DELETE (Data, 6, 2) will lead to the following processing in the array:

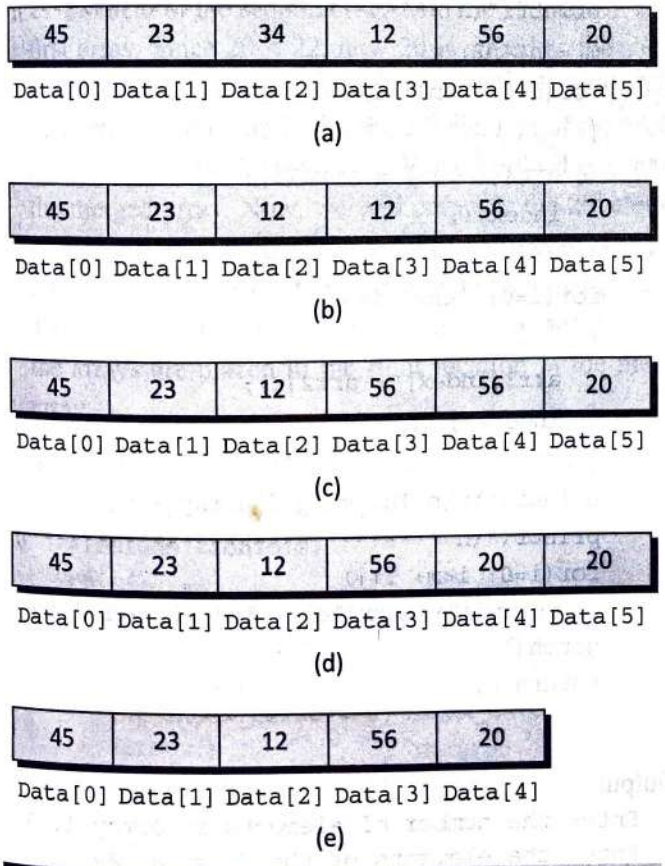


Figure 11.18 Deleting elements from the array

15. Write a program to delete a number from a given location in an array.

```
#include <stdio.h>
#include <conio.h>
int main()
{
   int i, n, pos, arr[10];
   clrscr();

   printf("\n Enter the size of the array:");
   scanf("%d", &n);
   printf("\n Enter the elements of the
   array : ");
   for(i=0; i<n; i++)
      scanf("%d", &arr[i]);
   printf("\n Enter the position from which
   the number has to be deleted: ");
   scanf("%d", &pos);

   for(i=pos; i<n; i++)
      arr[i] = arr [i+1];
   n--;
   printf("\n The array after deletion is:");
   for(i=0; i<n; i++)
      printf("\n Arr[%d] = %d", i, arr[i]);
   getch();
   return 0;
}
```

Output

```
Enter the size of the array: 5
Enter the elements of the array:
1 2 3 4 5
Enter the position from which the number has
to be deleted: 3
The array after deletion is:
Arr[0] = 1
Arr[1] = 2
Arr[2] = 3
Arr[3] = 5
```

16. Write a program to delete a number from an array that is already sorted in ascending order.

```
#include <stdio.h>
#include <conio.h>
int main()
{
   int i, n, j, num, arr[10];
   clrscr();
   printf("\n Enter the number of elements
   in the array: ");
   scanf("%d", &n);
```

```
    for(i=0; i<n; i++)
       scanf("%d", &arr[i]);
    printf("\n Enter the number to be
    deleted: ");
    scanf("%d", &num);

    for(i=0; i<n; i++)
    {
       if(arr[i] == num)
       {
          for(j=i; j<n; j++)
          arr[j] = arr[j+1];
       }
    }
    printf("\n The array after deletion is:
    ");
    for(i=0; i<n-1; i++)
       printf("\t%d", arr[i]);
    getch();
    return 0;
}
```

Output

```
Enter the number of elements in the array:
5
1 2 3 4 5
Enter the number to be deleted: 3
The array after deletion is: 1 2 4 5
```

11.6.4 Merging

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

If the arrays are unsorted, then merging the arrays is very simple as one just needs to copy the contents of one array into another. However, merging is not a trivial task when the two arrays are sorted and the merged array also needs to be sorted. Let us first discuss the merge operation on unsorted arrays. This operation is shown in Figure 11.19.

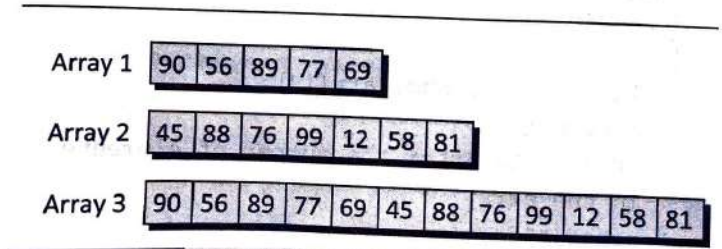


Figure 11.19 Merging of two unsorted arrays

17. Write a program to merge two unsorted arrays.

```
#include <stdio.h>
#include <conio.h>
main()
{
   int arr1[10], arr2[10], arr3[20];
   int i, n1, n2, m, index=0;
   clrscr();
   printf("\n Enter the number of elements
   in array 1: ");
   scanf("%d", &n1);
   printf("\n\n Enter the elements of the
   first array");
   for(i=0; i<n1; i++)
      scanf("%d", &arr1[i]);
   printf("\n ****************************");

   printf("\n Enter the number of elements
   in array 2: ");
   scanf("%d", &n2);
   printf("\n\n Enter the Elements of the
   second array");
   for(i=0; i<n2; i++)
      scanf("%d", &arr2[i]);
   printf("\n ****************************");
   m = n1+n2;
   for(i=0; i<n1; i++)
   {
      arr3[index] = arr1[i];
      index++;
   }
   for(i=0; i<n2; i++)
   {
      arr3[index] = arr2[i];
      index++;
   }
   printf("\n\n The merged array is");
   printf("\n ****************************");
   for(i=0; i<m; i++)
      printf("\t Arr[%d] = %d", i, arr3[i]);
   getch();
   return 0;
}
```

Output

```
Enter the number of elements in array 1: 3
Enter the elements of the first array
10 20 30
****************************
Enter the number of elements in array 2: 3
```

```
Enter the elements of the second array
15 25 35
*****************************
The merged array is
*****************************
Arr[0]= 10    Arr[1] = 20    Arr[3] = 30
Arr[4] = 15   Arr[5] = 25    Arr[6] = 35
```

If we have two sorted arrays and the resultant merged array also needs to be a sorted one, then the task of merging the arrays becomes a little difficult. The task of merging can be explained using Figure 11.20.

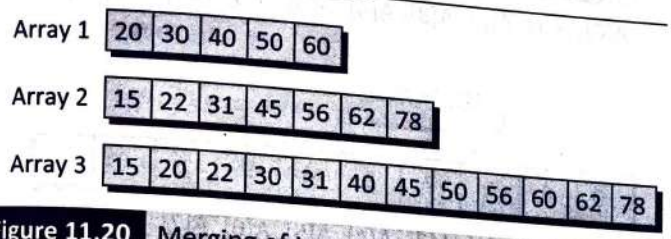


Figure 11.20 Merging of two sorted arrays

The figure shows how the merged array is formed using two sorted arrays. Here, we first compare the 1st element of array 1 with the 1st element of array 2, put the smaller element in the merged array. Since 20 > 15, we put 15 as the first element in the merged array. We then compare the 2nd element of the second array with the 1st element of the first array. Since 20 < 22, now 20 is stored as the second element of the merged array. Next, 2nd element of the first array is compared with the 2nd element of the second array. Since 30 > 22, we store 22 as the third element of the merged array. Now, we will compare the 2nd element of the first array with 3rd element of the second array. As 30 < 31, we store 30 as the 4th element of the merged array. This procedure will be repeated until elements of both the arrays are placed in the right location in the merged array.

18. Write a program to merge two sorted arrays.

```
#include <stdio.h>
#include <conio.h>
main()
{
   int arr1[10], arr2[10], arr3[20];
   int i, n1, n2, m, index=0;
   int index_first = 0, index_second = 0;
   clrscr();
   printf("\n Enter the number of elements
   in array1: ");
   scanf("%d", &n1);
   printf("\n\n Enter the elements of the
   first array");
   printf("\n ****************************");
   for(i=0; i<n1; i++)
      scanf("%d", &arr1[i]);
   printf("\n Enter the number of elements
   in array2 : ");
   scanf("%d", &n2);
   printf("\n\n Enter the elements of the
   second array");
   printf("\n ****************************");
   for(i=0; i<n2; i++)
      scanf("%d", &arr2[i]);
   m = n1+n2;
   while(index_first < n1 && index_second < n2)
   {
      if(arr1[index_first]<arr2[index_second])
      {
         arr3[index] = arr1[index_first];
         index_first++;
      }
      else
      {
         arr3[index] = arr2[index_second];
         index_second++;
      }
      index++;
   }

   /* if elements of the first array is over
   and the second array has some elements */
   if(index_first == n1)
   {
      while(index_second<n2)
      {
         arr3[index] = arr2[index_second];
         index_second++;
         index++;
      }
   }
   /* if elements of the second array is over
   and the first array has some elements */
   else if(index_second == n2)
   {
      while(index_first<n1)
      {
         arr3[index] = arr1[index_first];
         index_first++;
         index++;
      }
   }
   printf("\n\n The contents of the merged
   array are");
   printf("\n ****************************");
   for(i=0; i<m; i++)
      printf("\n Arr[%d] = %d", i, arr3[i]);
   getch();
   return 0;
}
```

Output

```
Enter the number of elements in array1: 3
Enter the Elements of the first array
```

```
****************************
10 20 30
Enter the number of elements in array2: 3
Enter the Elements of the second array
****************************
15 25 35
The contents of the merged array is
****************************
Arr[0]= 10    Arr[1] = 15    Arr[3] = 20
Arr[4] = 25   Arr[5] = 30    Arr[6] = 35
```

11.6.5 Searching Array Elements

Searching means to find whether a particular value is present in the array or not. If the value is present in the array then searching is said to be successful and the searching process gives the location of that value in the array. Otherwise, if the value is not present in the array, the searching process displays the appropriate message and in this case searching is said to be unsuccessful.

There are two popular methods for searching the array elements. One is *linear search* and the second is *binary search*. The algorithm that should be used depends entirely on how the values are organized in the array. For example, if the elements of the array are arranged in ascending order, then binary search should be used as it is more efficient for sorted list in terms of complexity. We will discuss these two methods in detail in this section.

Linear Search

Linear search, also known as sequential search, is a very simple, method used for searching an array for a particular value. It works by comparing every element of the array one by one in sequence until a match is found. Linear search is mostly used to search an unordered list of elements (array in which data elements are not sorted). For example, if an array `A[]` is declared and initialized as,

```
int A[] = {10, 8, 2, 7, 3, 4, 9, 1, 6, 5};
```

and `VAL = 7`, then searching means to find out whether the value '7' is present in the array or not. If yes, then the search is successful and it returns the position of occurrence of `VAL`. Here, the `POS = 3` (index starting from 0). Figure 11.21 illustrates this concept.

Thus, we see that linear search executes in O(n) time where n is the number of elements in the array. Obviously, the best case of linear search is when `VAL` is equal to the first element of the array. In this case, only one comparison will be made.

Likewise, the worst case will happen when either `VAL` is not present in the array or it is equal to the last element of the array. In both the cases, n comparisons will have to be made. However, the performance of the linear search algorithm (Figure 11.22) can be improved by using a sorted array.

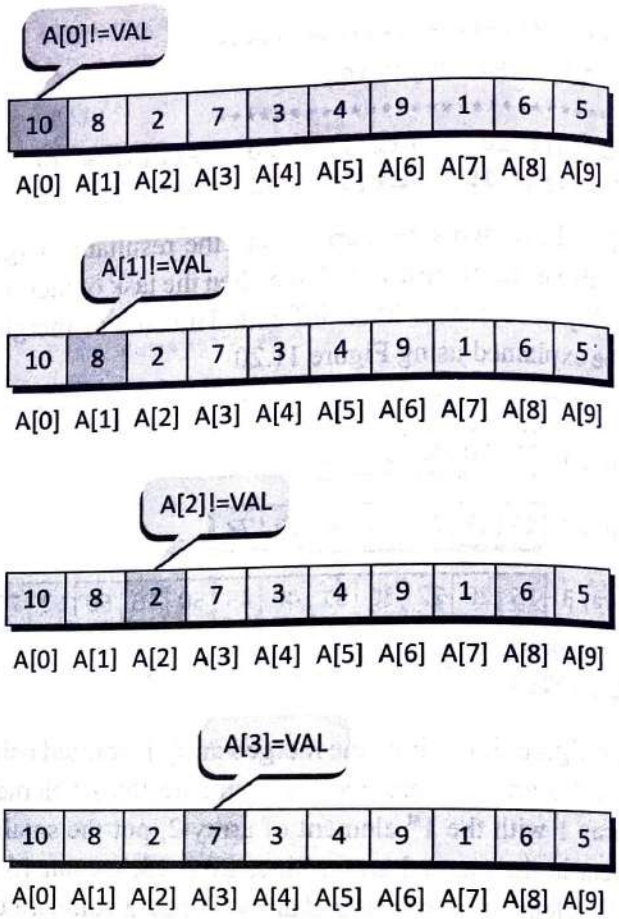


Figure 11.21 Linear search

```
LINEAR_SEARCH(A, N, VAL, POS)

Step 1: [INITIALIZE] SET POS = -1
Step 2: [INITIALIZE] SET I = 0
Step 3:     Repeat Step 4 while I<N
Step 4:             IF A[I] = VAL, then
                        SET POS = I
                        PRINT POS
                        Go to Step 6
                [END OF IF]
        [END OF while LOOP]
Step 5: PRINT "Value not present in the array"
Step 6: EXIT
```

Figure 11.22 Algorithm for linear search

In Steps 1 and 2 of the algorithm, we initialize the value of `POS` and `I`. In Step 3, a `while` loop is executed that would be executed until `I` is less than `N` (total number of elements in the array). In Step 4, a check is made to see if a match is found between the current array element and the `VAL`. If a match is found, then the position of the array element is printed else the value of `I` is incremented to match the next element with `VAL`. However, if all the array elements have been compared with `VAL`, and no match is found then it means that the `VAL` is not present in the array.

19. Write a program to implement linear search.

```
#include <stdio.h>
#include <conio.h>
main()
{
   int arr[10], num, i, n, found = 0, pos = -1;
   clrscr();
   printf("\n Enter the number of elements
   in the array : ");
   scanf("%d", &n);
   printf("\n Enter the elements:");
   for(i=0; i<n; i++)
      scanf("%d", &arr[i]);
   printf("\n Enter the number that has to
   be searched: ");
   scanf("%d", &num);
   for(i=0; i<n; i++)
   {
      if(arr[i] == num)
      {
         found = 1;
         pos = i;
         printf("\n %d is found in the array
            at position = %d", num, i);
         break;
      }
   }
   if (found == 0)
      printf("\n %d does not exist in the
         array", num);
   getch();
   return 0;
}
```

Output

```
Enter the number of elements in the array:
5
Enter the elements: 1 2 3 4 5
Enter the number that has to be searched: 7
7 does not exist in the array
```

Binary Search

We have seen that the linear search algorithm is very slow. If we have an array with 1 million entries, then to search a value from that array, we would need to make 1 million comparisons in the worst case. However, if the array is sorted, we have a better and efficient alternative, known as binary search.

Binary search is a searching algorithm that works efficiently with a *sorted list*. The algorithm finds out the position of a particular element in the array. The mechanism of binary search can be better understood by using the analogy of a telephone directory. When we are searching for a particular name in the directory, we will first open the directory from the middle and then decide whether to look for the name in the first part of the directory or in the second part of the directory. Again we will open some page in the middle and the whole process will be repeated until we finally find the name.

Take another analogy. How do we find words in a dictionary? We first open the dictionary somewhere in the middle. Then we compare the first word on that page with the desired word whose meaning has to be found. If the word comes before the word that appear on the page we will look in the first half of the dictionary else we will look in the second half. Again, we will open up some page in the first half of the dictionary pages. Compare the first word on that page with the desired word and repeat the same procedure until we finally get the word. The same mechanism is applied in binary search.

Now let us consider how this mechanism will be applied to search for a value in a sorted array. Given an array that is declared and initialized as,

```
int A[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

and VAL = 9, the algorithm (Figure 11.23) will proceed in the following manner.

```
BEG = 0, END = 10, MID = (0 + 10)/2 = 5
```

Now, VAL = 9 and A[MID] = A[5] = 5

A[5] is less than VAL, therefore, we will now search for the value in the later half of the array. So, we change the values of BEG and MID.

Now, BEG = MID + 1 = 6, END = 10, MID = (6 + 10)/2 = 16/2 = 8

Now, VAL = 9 and A[MID] = A[8] = 8

A[8] is less than VAL, therefore, we will now search for the value in the later half of the array. So, again we change the values of BEG and MID.

Now, BEG = MID + 1 = 9, END = 10, MID = (9 + 10)/2 = 9

Now VAL = 9 and A[MID] = 9.

Now VAL = 9 and A[MID] = 9.

In this algorithm we see that BEG and END are the beginning and ending positions of the segment that we are looking to search for the element. MID is calculated as (BEG + END)/2. Initially, BEG = lower_bound and END = upper_bound. The algorithm will terminate when A[MID] = VAL. When the algorithm ends, we will set POS = MID. POS is the position at which the value is present in the array.

However, if VAL is not equal to A[MID], then the values of BEG, END, and MID will be changed depending on whether VAL is smaller or greater than A[MID].

(a) If VAL < A[MID], then VAL will be present in the left segment of the array. So, the value of END will be changed as, END = MID - 1

```
BINARY_SEARCH(A, lower_bound, upper_bound, VAL, POS)

Step 1: [INITIALIZE] SET BEG = lower_bound, END = upper_bound, POS = -1
Step 2: Repeat Step 3 and Step 4 while BEG <= END
Step 3:       SET MID = (BEG + END)/2
Step 4:       IF A[MID] = VAL, then
                  POS = MID
                  PRINT POS
                  Go to Step 6
              IF A[MID] > VAL then;
                  SET END = MID - 1
                  ELSE
                  SET BEG = MID + 1
              [END OF IF]
        [END OF while LOOP]
Step 5: IF POS = -1, then
              PRINTF "VAL IS NOT PRESENT IN THE ARRAY"
        [END OF IF]
Step 6: EXIT
```

Figure 11.23 Algorithm for binary search

(b) If VAL > A[MID], then VAL will be present in the right segment of the array. So, the value of BEG will be changed as, BEG = MID + 1

Finally, if that VAL is not present in the array, then eventually, END will be less than BEG. When this happens, the algorithm should terminate as it will indicate that the element is not present in the array and the search will be unsuccessful. Let us consider another example.

```
int A[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
  10}; and VAL = 2
Step 1: BEG = 0, END = 10, MID = 5
  A[MID] > VAL
Step 2: BEG = 0, END = MID - 1 = 4, MID = 2
  A[MID] = VAL
```

Figure 11.23 shows an algorithm for binary search.

In Step 1, we initialize the value of variables—BEG, END and POS. In Step 2, a while loop is executed until BEG is less than or equal to END. In Step 3, value of MID is calculated. In Step 4, we check if the value of MID is equal to VAL (item to be searched in the array). If a match is found then value of the POS is printed and the algorithm exits. However, if a match is not found then if the value of A[MID] is greater than VAL, then the value of END is modified otherwise if A[MID] is less than VAL, then value of BEG is altered. In Step 5, if the value of POS = -1, then it means VAL is not present in the array and an appropriate message is printed on the screen before the algorithm exits.

The complexity of the binary search algorithm can be expressed as f(n), where n is the number of elements in the array. The complexity of the algorithm is calculated depending on the number of comparisons that are made. In binary search algorithm, we see that with each comparison, the size of the segment where search has to be made is reduced to half. Thus, we can say that, in order to locate a particular VAL in the array, the total number of comparisons that will be made is given by,

$$2^{f(n)} > n \text{ or } f(n) = \log_2 n$$

20. Write a program to implement binary search.

```
#include <stdio.h>
#include <conio.h>
main()
{
   int arr[10], num, i, n, pos = -1, beg, end,
   mid, found = 0;
   clrscr();
   printf("\n Enter the number of elements
   in the array: ");
   scanf("%d", &n);
   printf("\n Enter the elements: ");
   for(i=0; i<n; i++)
   {
      scanf("%d", &arr[i]);
   }
   printf("\n Enter the number that has to
   be searched: ");
   scanf("%d", &num);

   beg = 0, end = n-1;
   while(beg ≤ end)
   {
      mid = (beg + end)/2;
```

SORTING OF ARRAYS

Sorting Arrays: Sorting means arranging the elements of an array so that they are placed in some relevant order which may be either ascending or descending. Some of the sorting algorithms are discussed below:

a. Bubble Sort: It is a simple algorithm that sorts the array elements by repeatedly moving the largest element to the highest index position of the array (for ascending order). Here, consecutive adjacent pairs of elements are compared with each other. If the element at the lower index is greater than the element at the higher index, the two elements are interchanged so that the element is placed before the bigger one. At the end of each pass, the largest element in the array will be placed in its correct position. This process will continue till the list becomes sorted. The algorithm for bubble sort is as follows:

```
BUBBLE_SORT (A, N)
Step 1: [INITIALIZE] Set I=0
Step 2: Repeat Step 3 to 7 while I < N – 1
Step 3:         Set J=0
Step 4:         Repeat Step 5 and 6 while J < N – 1 – I
Step 5:                 If A[J] > A[J + 1]
                                SWAP A[J] and A[J + 1]
                        [END OF IF]
Step 6:                 J = J + 1
                [END OF INNER LOOP]
Step 7:         I=I+1
        [END OF OUTER LOOP]
Step 8: EXIT
```



prabhatdash Organised as per topic

Code | Blame | 18 lines (18 loc) · 437 Bytes | Code 55% faster with GitHub Copilot

```c
#include<stdio.h>
int main(){
    int arr[]={10,77,1,5,100,20,6,99};
    int size=sizeof(arr)/sizeof(int);
    int temp;
    for(int i=0;i<size-1;i++){
        for(int j=0;j<size-i-1;j++){
             if(arr[j] > arr[j+1]){
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
    for(int i=0;i<size;i++){
        printf("%d\n",arr[i]);
    }
}
```

```
Arr[0] = 35  Arr[1] = 25  Arr[2] = 15
Arr[3] = 30  Arr[4] = 20  Arr[5] = 10
```

24. Write a program to interchange the biggest and the smallest number in the array.

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

void read_array(int my_array[], int);
void display_array(int my_array[], int);
void interchange(int arr[], int);
int find_biggest_pos(int my_array[10], int n);
int find_smallest_pos(int my_array[10], int n);

int main()
{
   int arr[10], n;
   clrscr();

   printf("\n Enter the size of the array:");
   scanf("%d", &n);
   read_array(arr, n);
   display_array(arr, n);
   interchange(arr, n);
   display_array(arr, n);
   getch();
   return 0;
}
void read_array(int my_array[10], int n)
{
   int i;
   for(i=0; i<n; i++)
      scanf("%d", &my_array[i]);
}
void display_array(int my_array[10], int n)
{
   int i;
      printf("n");
   for(i=0; i<n; i++)
      printf("\t Arr[%d] = %d", i, my_
   array[i]);
}
void interchange(int my_array[10], int n)
{
   int temp, big_pos, small_pos;
   big_pos = find_biggest_pos(my_arr, n);
   small_pos = find_smallest_pos(my_arr,n);
   temp = my_array[big_pos];
   my_array[big_pos] = my_array[small_pos];
   my_array[small_pos] = temp;
}
int find_biggest_pos(int my_array[10], int n)
{
   int i, large = -123456, pos=-1;
   for(i=0; i<n; i++)
   {
      if (my_array[i] > large)
      {
         large = my_array[i];
         pos=i;
      }
   }
   return pos;
}
int find_smallest_pos(int my_array[10], int n)
{
   int i, small = 123456, pos=-1;
   for(i=0; i<n; i++)
   {
      if (my_array[i] < small)
      {
         small = my_array[i];
         pos=i;
      }
   }
   return pos;
}
```

Output

```
Enter the size of the array: 5
1 2 3 4 5
Arr[0] = 1    Arr[1] = 2    Arr[2] = 3
Arr[3] = 4    Arr[4] = 5
Arr[0] = 5    Arr[1] = 2    Arr[2] = 3
Arr[3] = 4    Arr[4] = 1
```

Note: If a function receives an array that does not change it, then the array should be received as a constant array. This would ensure that the contents of the array are not accidentally changed. To declare an array as constant, prefix its type with the const keyword, as follows:

```
int sum(const int arr[ ], int n);
```

11.8 TWO-DIMENSIONAL ARRAYS

Till now we have read only about one-dimensional arrays (1D). A 1D array is organized linearly and only in one direction. However, at times, we need to store data in the form of matrices or tables. Here the concept of one-dimension array is extended to incorporate two-dimensional (2D) data structures. A 2D array is specified using two subscripts where one subscript denotes row and the other denotes column. C considers the 2D array as an array of 1D arrays. Figure 11.28 shows a 2D array which can be viewed as an array of arrays.

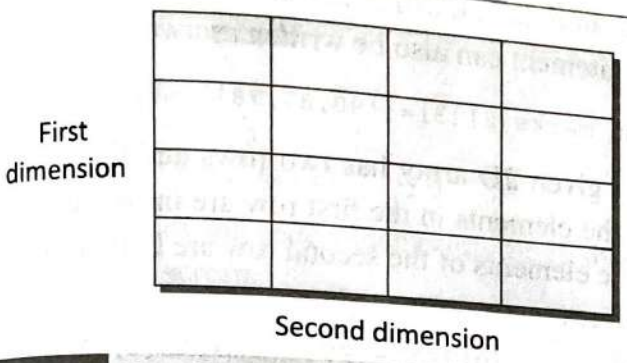


Figure 11.28 Two-dimensional array

11.8.1 Declaration of Two-dimensional Arrays

Similar to one-dimensional arrays, two-dimensional arrays must be declared before being used. The declaration statement tells the compiler the name of the array, the data type of each element in the array, and the size of each dimension. A two-dimensional array is declared as:

```
data_type array_name[row_size][column_size];
```

Therefore, a two-dimensional $m \times n$ array is an array that contains $m \times n$ data elements and each element is accessed using two subscripts, i and j where $i <= m$ and $j <= n$.

For example, if we want to store the marks obtained by 3 students in 5 different subjects, then we can declare a two-dimensional array as

```
int marks[3][5]
```

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



Each dimension of the 2D array is indexed from zero to its maximum size minus one. The first index selects the row and the second selects the column.

The pictorial form of a 2D array is given in Figure 11.29.

Hence, we see that a 2D array is treated as a collection of 1D arrays. The elements of the 2D array comprises 1D array (the rows). To understand this, we can also see the representation of a 2D array as shown in Figure 11.30.

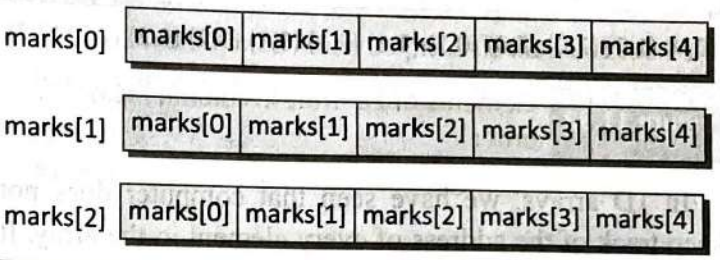


Figure 11.30 Representation of 2D array marks[3][5] as individual 1D arrays

Although Figure 11.30 showns a rectangular picture of a 2D array, these elements will be actually stored sequentially in memory. Since computer memory is basically one-dimensional, a multidimensional array cannot be stored in memory as a grid.

Basically, there are two ways of storing a two-dimensional array in memory. The first way is *row major order* and the second is *column major order*. Let us see how the elements of a 2D array are stored in a row major order. Here, the elements of the first row are stored before the elements of the second and third row, i.e., the elements of the array are stored row by row where n elements of the first row will occupy the first nth locations. This is illustrated in Figure 11.31.

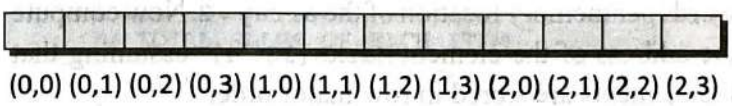


Figure 11.31 Elements of 2D array in row major order

When we store the elements in a column major order, the elements of the first column are stored before the elements

Row/Columns	Column 0	Column 1	Column 2	Column 3	Column 4
Row 0	`marks[0][0]`	`marks[0][1]`	`marks[0][2]`	`marks[0][3]`	`marks[0][4]`
Row 1	`marks[1][0]`	`marks[1][1]`	`marks[1][2]`	`marks[1][3]`	`marks[1][4]`
Row 2	`marks[2][0]`	`marks[2][1]`	`marks[2][2]`	`marks[2][3]`	`marks[2][4]`

Figure 11.29 Two-dimensional array

of the second and third column, i.e., the elements of the array are stored column by column where n elements of the first column will occupy the first nth locations. This is illustrated in Figure 11.32.



Figure 11.32 Elements of 2D array in column major order

In 1D arrays, we have seen that computer does not keep track of the address of every element in the array. It stores only the address of the first element and calculates the addresses of other elements from the base address (address of the first element). Same is the case with a two-dimensional array. Here also, the computer stores the base address and the address of the other elements is calculated using the following formula.

`Address(A[I][J]) = B_A + w{M(J - 1) + (I - 1)}`, if the array elements are stored in column major order, and

`Address(A[I][J]) = B_A + w{N (I - 1) + (J - 1)}`, if the array elements are stored in row major order.

where, `w` is the number of words stored per memory location, `m` is the number of columns, `n` is the number of rows, `I` and `J` are the subscripts of the array element, and `B_A` is the base address.

Example 11.7 Consider a `20 × 5` two-dimensional array `Marks` which has `base address = 1000` and number of words per memory location of the `array = 2`. Now compute the address of the element `Marks[18, 4]` assuming that the elements are stored in row major order.

Solution

```
Address(A[I][J])
   = Base_Address + w{N (I - 1) + (J - 1)}
Address(Marks[18,4])
   = 1000 + 2{5(18 - 1) + (4 - 1)}
   = 1000 + 2 {5(17) + 3}
   = 1000 + 2 (88)
   = 1000 + 176 = 1176
```

11.8.2 Initialization of Two-dimensional Arrays

Similar to other variables, declaring a 2D array only reserves space for the array in the memory. No values are stored in it. A 2D array is initialized in the same way as a 1D array. For example,

```
int marks[2][3] = {90, 87, 78, 68, 62, 71};
```

The initialization of a 2D array is done row by row. This statement can also be written as

```
int marks[2][3]={{90,87,78},{68, 62, 71}};
```

The given 2D array has two rows and three columns. Here, the elements in the first row are initialized first and then the elements of the second row are initialized. Therefore,

```
marks[0][0] = 90      marks[0][1] = 87
marks[0][2] = 78      marks[1][0] = 68
marks[1][1] = 62      marks[1][2] = 71
```

Therefore, in this example, each row is defined as a 1D array of three elements that are enclosed in braces. Commas are used to separate the elements in the row as well as to separate the elements of two rows.

In the case of a 1D array, if the array is completely initialized, we may omit the size of the array. The same concept can be applied to a 2D array, except that only the size of the first dimension can be omitted. Therefore, the following declaration statement is valid:

```
int marks[][3]={{90,87,78},{68, 62, 71}};
```

In order to initialize the entire 2D array to zero, simply specify the first value as zero, i.e., simply write

```
int marks[2][3] = {0};
```

If some values are missing in the initializer then it is automatically set to zero. For example, the following statement will initialize the values in the first row but the elements of the second row will be initialized to zero.

```
int marks[2][3] = {{50, 60, 70}};
```



An un-initialized array contains unpredictable contents.

11.8.3 Accessing the Elements

The elements in a multidimensional array are stored in contiguous memory locations. While accessing the elements, remember that the last subscript varies most rapidly whereas the first varies least rapidly.

In case of 1D arrays we used a single `for` loop to vary the index `i` in every pass, so that all the elements could be scanned. Similarly, since a 2D array contains two subscripts, we will use two `for` loops to scan the elements. The first `for` loop will loop for each row in the 2D array and the second `for` loop will scan individual columns for every row in the array.

However, individual elements of a 2D array can be initialized using the assignment operator as shown below.

```
marks[1][2] = 79; or
marks[1][2] = marks[1][1] + 10;
```

In order to input the values from the keyboard, you must use the following code.

```
for(i=0; i<2; i++)
   for(j=0; j<2; j++)
      scanf("%d", &arr[i][j]);
```

Look at the following code which prints the elements of a 2D array on the screen.

25. Write a program to print the elements of a 2D array.

```
#include <stdio.h>
#include <conio.h>
main()
{
   int arr[2][2] = {12, 34, 56, 32};
   int i, j;
   for(i=0; i<2; i++)
   {
      printf("\n");
      for(j=0; j<2; j++)
      printf("%d\t", arr[i][j]);
   }
   return 0;
}
```

Output

```
12 34
56 32
```

26. Write a program to generate Pascal's triangle.

```
#include <stdio.h>
#include <conio.h>
main()
{
   int arr[7][7]={0};
   int row=2, col, i, j;
   arr[0][0] = arr[1][0] = arr[1][1] = 1;
   while(row <= 7)
   {
      arr[row][0] = 1;
      for(col = 1; col <= row; col++)
      arr[row][col] = arr[row-1][col-1] +
            arr[row-1][col];
      row++;
   }
   for(i=0; i<7; i++)
   {
      printf("\n");
      for(j=0; j<=i; j++)
         printf("\t %d", arr[i][j]);
   }
   getch();
   return 0;
}
```

Output

```
1
1  1
1  2  1
1  3  3  1
1  4  6  4  1
1  5  10 10 5  1
1  6  15 20 15 6  1
```

27. In a small company there are five salesmen. Each salesman is supposed to sell three products. Write a program using a 2D array to print (i) the total sales by each salesman and (ii) total sales of each item.

```
#include <stdio.h>
#include <conio.h>
main()
{
   int sales[5][3], i, j, total_sales=0;
   //INPUT DATA
   printf("\n ENTER THE DATA");
   printf("\n *****************");
   for(i=0; i<5; i++)
   {
      printf("\n Enter the sales of 3 items
            sold by salesman %d: ", i);
      for(j=0; j<3; j++)
         scanf("%d", &sales[i][j]);
   }
   // PRINT TOTAL SALES BY EACH SALESMAN
   for(i=0; i<5; i++)
   {
      total_sales = 0;
      for(j=0; j<3; j++)
         total_sales += sales[i][j];
      printf("\n Total Sales By Salesman %d
            = %d", i, total_sales);
   }
   // TOTAL SALES OF EACH ITEM
   for(i=0; i<3; i++) // for each item
   {
      total_sales=0;
      for(j=0; j<5; j++) // for each
   salesman
         total_sales += sales[j][i];
      printf("\n Total sales of item %d
            = %d", i, total_sales);
   }
   getch();
   return 0;
}
```

Output

```
ENTER THE DATA
*****************
Enter the sales of 3 items sold by salesman
0: 23 23 45
Enter the sales of 3 items sold by salesman
1: 34 45 63
```

```
Enter the sales of 3 items sold by salesman
2: 36 33 43
Enter the sales of 3 items sold by salesman
3: 33 52 35
Enter the sales of 3 items sold by salesman
4: 32 45 64
Total Sales By Salesman 0 = 91
Total Sales By Salesman 1 = 142
Total Sales By Salesman 2 = 112
Total Sales By Salesman 3 = 120
Total Sales By Salesman 4 = 141
Total sales of item 0 = 158
Total sales of item 1 = 198
Total sales of item 2 = 250
```

28. In a class there are 10 students. Each student is supposed to appear in three tests. Write a program using two-dimensional arrays to print

(a) the marks obtained by each student in different subjects

(b) total marks and average obtained by each student

(c) store the average of each student in a separate 1D array so that it can be used to calculate the class average.

```
#include <stdio.h>
#include <conio.h>
main()
{
   int marks[10][3], i, j;
   int total_marks[10]={0};;
   float class_avg=0.0, total_avg = 0.0;
   float avg[10];

   //INPUT DATA
   printf("\n ENTER THE DATA");
   printf("\n *****************");
   for(i=0; i<10; i++)
   {
   printf("\n Enter the marks of student %d
           in 3 subjects : ", i);
      for(j=0; j<3; j++)
         scanf("%d", &marks[i][j]);
   }

   // CALCULATE TOTAL MARKS OF EACH STUDENT
   for(i=0; i<10; i++)
   {
      for(j=0; j<3; j++)
         total_marks[i] += marks[i][j];
   }
   // CALCULATE AVERAGE OF EACH STUDENT
   for(i=0; i<10; i++)
   {
      for(j=0; j<3; j++)
         avg[i] = (float)total_marks[i]/3.0;
   }
   // CALCULATE CLASS AVERAGE
   for(i=0; i<10; i++)
      total_avg += avg[i];
   class_avg = (float)total_avg/10;
   // DISPLAY RESULTS
   printf("\n\n STUD NO. MARKS OBTAINED IN
   THREE SUBJECTS TOTAL MARKS \t AVERAGE");
   printf("\n*********************************
   **********************");
   for(i=0; i<10; i++)
   {  printf("\n %4d", i);
      for(j=0; j<3; j++)
         printf("    %d", marks[i][j]);
      printf("%4d \t%2.2f", total_marks[i],
           avg[i]);
   }
   printf("\n\n CLASS AVERAGE = %f", class_
   avg);
   getch();
   return 0;
}
```

Output

```
ENTER THE DATA
*****************
Enter the marks of student 0 in 3 subjects:
78 89 90
Enter the marks of student 1 in 3 subjects:
98 87 76
Enter the marks of student 2 in 3 subjects:
67 78 89
Enter the marks of student 3 in 3 subjects:
90 87 65
Enter the marks of student 4 in 3 subjects:
56 87 97
Enter the marks of student 5 in 3 subjects:
45 67 89
Enter the marks of student 6 in 3 subjects:
66 77 88
Enter the marks of student 7 in 3 subjects:
76 87 98
Enter the marks of student 8 in 3 subjects:
67 88 66
Enter the marks of student 9 in 3 subjects:
66 75 78
STUD NO. MARKS OBTAINED IN THREE SUBJECTS
TOTAL MARKS    AVERAGE
***************
0     78    89    90    257    85.67
1     98    87    76    261    87.00
2     67    78    89    234    78.00
3     90    87    65    242    80.67
4     56    87    97    240    80.00
5     45    67    89    201    67.00
```

```
6     66     77     88     231    77.00
7     76     87     98     261    87.00
8     67     88     66     221    73.67
9     66     75     78     210    73.00
CLASS AVERAGE = 78.90
```

29. Write a program to read a 2D array `marks` which stores the marks of five students in three subjects. Write a program to display the highest marks in each subject.

```
#include <stdio.h>
#include <conio.h>
main()
{
   int marks[5][3], i, j, max_marks;
   for(i=0; i<5; i++)
   {
      printf("\n Enter the marks obtained by
            student %d",i);
      for(j=0; j<3; j++)
      {
         printf("\n marks[%d][%d] = ", i, j);
         scanf("%d", &marks[i][j]);
      }
   }
   for(j=0; j<3; j++)
   {
      max_marks = -999;
      for(i=0; i<5; i++)
      {
         if(marks[i][j]>max_marks)
         max_marks = marks[i][j];
      }
      printf("\n The highest marks obtained
            in the subject %d = %d", j,
            max_marks);
   }
   getch();
   return 0;
}
```

Output

```
Enter the marks obtained by student 0
marks[0][0] = 89
marks[0][1] = 76
marks[0][2] = 100
Enter the marks obtained by student 1
marks[1][0] = 99
marks[1][1] = 90
marks[1][2] = 89
Enter the marks obtained by student 2
marks[2][0] = 67
marks[2][1] = 76
marks[2][2] = 56
Enter the marks obtained by student 3
marks[3][0] = 88
marks[3][1] = 77
marks[3][2] = 66
Enter the marks obtained by student 4
marks[4][0] = 67
marks[4][1] = 78
marks[4][2] = 89
The highest marks obtained in the subject 0
= 99
The highest marks obtained in the subject 1
= 90
The highest marks obtained in the subject 2
= 100
```

11.9 OPERATIONS ON TWO-DIMENSIONAL ARRAYS

We can perform the following operations on a two-dimensional array:

Transpose Transpose of a $m \times n$ matrix A is given as a $n \times m$ matrix B where, $B_{i,j} = A_{j,i}$

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

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

Product Two matrices can be multiplied with each other if the number of columns in the first matrix is equal to the number of rows in the second matrix. Therefore, $m \times n$ matrix A can be multiplied with a $p \times q$ matrix if $n = q$. Elements of the matrices can be multiplied by writing: $C_{i,j} = \sum A_{i,k}B_{j,k}$ for $k = 1$ to $k < n$

30. Write a program to read and display a 3×3 matrix.

```
#include <stdio.h>
#include <conio.h>
int main()
{
   int i, j, mat[3][3];
   clrscr();

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

   for(i=0; i<3; i++)
   {
      for(j=0; j<3; j++)
```

```
        scanf("%d", &mat[i][j]);
    }
    printf("\n The elements of the matrix are ");
    printf("\n *************************");

    for(i=0; i<3; i++)
    {
       printf("\n");
       for(j=0; j<3; j++)
          printf("\t %d %d %d",i, j, mat[i]
            [j]);
    }
    return 0;
}
```

Output

```
Enter the elements of the matrix
*************************
1 2 3 4 5 6 7 8 9
The elements of the matrix are
*************************
1 2 3
4 5 6
7 8 9
```

31. Write a program to transpose a 3×3 matrix.

```
#include <stdio.h>
#include <conio.h>
int main()
{
   int i, j, mat[3][3], transposed_mat[3][3];
   clrscr();
   printf("\n Enter the elements of the
   matrix");
   printf("\n *************************");
   for(i=0; i<3; i++)
   {
      for(j=0; j<3; j++)
      {
         printf("\n mat[%d][%d] = ", i, j);
         scanf("%d", &mat[i][j]);
      }
   }
   printf("\n The elements of the matrix are ");
   printf("\n *************************");
   for(i=0; i<3; i++)
   {
      printf("\n");
      for(j=0; j<3; j++)
         printf("\t mat[%d][%d] = %d", i, j,
                  mat[i][j]);
   }
   for(i=0; i<3; i++)
   {
      for(j=0; j<3; j++)
         transposed_mat[i][j] = mat[j][i];
   }
   printf("\n The elements of the transpose
   matrix are ");
   printf("\n *************************");

   for(i=0; i<3; i++)
   {
      printf("\n");
      for(j=0; j<3; j++)
         printf("\t transposed_mat[%d][%d]
               = %d", i, j, transposed_ mat[i
               [j]);
   }
   return 0;
}
```

Output

```
Enter the elements of the matrix
*************************
1 2 3 4 5 6 7 8 9
The elements of the matrix are
*************************
1 2 3
4 5 6
7 8 9
The elements of the transposed matrix are
*************************
1 4 7
2 5 8
7 8 9
```

32. Write a program to input two $m \times n$ matrices and then calculate the sum of their corresponding elements and store it in a third $m \times n$ matrix.

```
#include <stdio.h>
#include <conio.h>
int main()
{
   int i, j;
   int rows1, cols1, rows2, cols2, rows_sum,
   cols_sum;
   int mat1[5][5], mat2[5][5], sum[5][5];
   clrscr();

   printf("\n Enter the numbers of rows in
   the first matrix: ");
   scanf("%d", &rows1);

   printf("\n Enter the numbers of columns
   in the first matrix: ");
   scanf("%d", &cols1);
```

```
printf("\n Enter the numbers of rows in
the second matrix: ");
scanf("%d", &rows2);

printf("\n Enter the numbers of columns
in the second matrix:");
scanf("%d", &cols2);

if(rows1 != rows2 || cols1 != cols2)
{
   printf("\n The number of rows and
         columns of both the matrices must
         be equal");
   getch();
   exit();
}
rows_sum = rows1;
cols_sum = cols1;

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

for(i=0; i<rows1; i++)
{
   for(j=0; j<cols1; j++)
      scanf("%d",&mat1[i][j]);
}
printf("\n Enter the elements of the
second matrix");
printf("\n *************************");

for(i=0; i<rows2; i++)
{
   for(j=0; j<cols2; j++)
      scanf("%d",&mat2[i][j]);
}
for(i=0; i<rows_sum; i++)
{
   for(j=0; j<cols_sum; j++)
      sum[i][j] = mat1[i][j] + mat2[i][j];
}
printf("\n The elements of the resultant
matrix are");
printf("\n *************************");

for(i=0; i<rows_sum; i++)
{
   printf("\n");
   for(j=0; j<cols_sum; j++)
      printf("\t %d", sum[i][j]);
   }
   return 0;
}
```

Output

```
Enter the numbers of rows in the first
matrix: 2
Enter the numbers of columns in the first
matrix: 2
Enter the numbers of rows in the second
matrix: 2
Enter the numbers of columns in the second
matrix: 2
Enter the elements of the first matrix
*************************
1 2 3 4
Enter the elements of the second matrix
*************************
5 6 7 8
The elements of the resultant matrix are
*************************
6   8
10 12
```

33. Write a program to multiply two $m \times n$ matrices.

```
#include <stdio.h>
#include <conio.h>
int main()
{
   int i, j, k;
   int rows1, cols1, rows2, cols2, res_
   rows, res_cols;
   int mat1[5][5], mat2[5][5], res[5][5];
   clrscr();

   printf("\n Enter the number of rows in
   the first matrix: ");
   scanf("%d", &rows1);

   printf("\n Enter the number of columns
   in the first matrix: ");
   scanf("%d", &cols1);

   printf("\n Enter the number of rows in
   the second matrix: ");
   scanf("%d", &rows2);

   printf("\n Enter the number of columns in
   the second matrix:");
   scanf("%d", &cols2);

   if(cols1 != rows2)
   {
      printf("\n The number of columns in
            the first matrix must be equal
            to the number of rows in the
```

```
            second matrix");
      getch();
      exit();
   }
   res_rows = rows1;
   res_cols = cols2;

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

   for(i=0; i<rows1; i++)
   {
      for(j=0; j<cols1; j++)
         scanf("%d",&mat1[i][j]);
   }
   printf("\n Enter the elements of the
   second matrix");
   printf("\n *************************");

   for(i=0; i<rows2; i++)
   {
      for(j=0; j<cols2; j++)
         scanf("%d", &mat2[i][j]);
   }
   for(i=0; i<res_rows; i++)
   {
      j=0;
      for(j=0; j<res_cols; j++)
      {
         res[i][j]=0;
         for(k=0; k<res_cols; k++)
            res[i][j] += mat1[i][k] * mat2[k]
                         [j];
      }
   }
   printf("\n The elements of the product
   matrix are");
   printf("\n *************************");

   for(i=0; i<res_rows; i++)
   {
      printf("\n");
      for(j=0; j<res_cols; j++)
      printf("\t %d", res[i][j]);
   }
   return 0;
}
```

Output

```
Enter the number of rows in the first
matrix: 2
Enter the number of columns in the first
matrix: 2
Enter the number of rows in the second
matrix: 2
Enter the number of columns in the second
matrix: 2
Enter the elements of the first matrix
*************************
1 2 3 4
Enter the elements of the second matrix
*************************
5 6 7 8
The elements of the product matrix are
*************************
19    22
43    50
```

```
    for(i=0; i<r; i++)
    {
       for(j=0; j<r; j++)
       {
          if(i==j)
             mat[i][j] = 0;
          if(i>j)
             mat[i][j] = -1;
          else
             mat[i][j] = 1;
       }
    }
}
void display_matrix(int mat[5][5], int r)
{
    int i, j;
    for(i=0; i<r; i++)
    {
       printf("\n");
       for(j=0; j<r; j++)
          printf("\t mat[%d][%d] = %d", i, j,
                 mat[i][j]);
    }
}
```

Output

```
Enter the number of rows and columns of
the matrix: 2
 1   0
-1   1
```

11.11 MULTIDIMENSIONAL ARRAYS

A multidimensional array in simple terms is an array of arrays. Like we have one index in a 1D array, two indices in a 2D array, in the same way we have *n* indices in an, *n*-dimensional array or multi dimensional array. Conversely, an *n*-dimensional array is specified using n indices. An *n*-dimensional $m_1 \times m_2 \times m_3 \times \ldots m_n$ array is a collection $m_1 * m_2 * m_3 * \ldots * m_n$ elements. In a multidimensional array, a particular element is specified by using *n* subscripts as $A[I_1][I_2][I_3]\ldots[I_n]$, where,

$I_1 <= M_1,\ I_2 <= M_2,\ I_3 <= M_3,\ \ldots\ I_n <= M_n$

A multidimensional array can contain as many indices as needed and the requirement of the memory increases with the number of indices used. However, practically speaking we will hardly use more than three indices in any program. Figure 11.35 shows a three-dimensional (3D) array. The array has three pages, four rows, and two columns.

A multidimensional array is declared and initialized similar to one- and two-dimensional arrays.

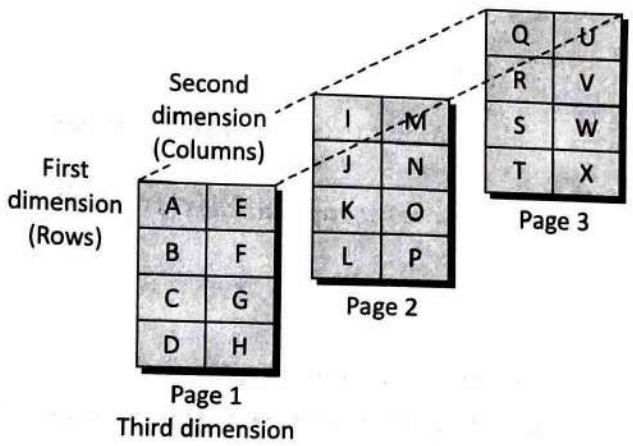
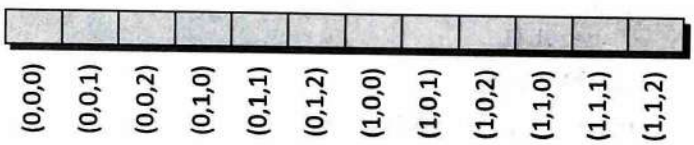


Figure 11.35 Three-dimensional array

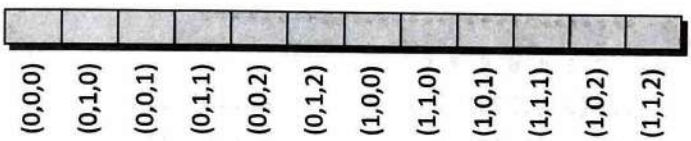
Example 11.8 Consider a 3D array defined as int A[2][3][2]. Calculate the number of elements in the array. Give the memory representation of the array in row major order and column major order.

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

Solution



(a) Row major order



(b) Column major order

The three dimensional array will contain $2 \times 3 \times 2 = 12$ elements.

36. Write a program to read and display a $2 \times 2 \times 2$ array.

```
#include <stdio.h>
#include <conio.h>
int main()
{
   int array1[3][3][3], i, j, k;
   clrscr();

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

```
        for(i=0; i<2; i++)
        {
           for(j=0; j<2; j++)
           {
              for(k=0; k<2; k++)
              {
                 printf("\n array[%d][%d][%d]
                        = ", i, j, k);
                 scanf("%d", &array1[i][j][k]);
              }
           }
        }
        printf("\n The matrix is: ");
        printf("\n ******************************");
        for(i=0; i<2; i++)
        {
           printf("\n\n");
           for(j=0; j<2; j++)
           {
              printf("\n");
              for(k=0; k<2; k++)
        printf("\tarray[%d][%d][%d] = %d", i, j,
                 k, array1[i][j][k]);
           }
        }
        getch();
        return 0;
     }
```

Output

```
Enter the elements of the matrix
******************************
1 2 3 4 5 6 7 8
arr[0][0][0] = 1          arr[0][0][1] = 2
arr[0][1][0] = 3          arr[0][1][1] = 4
arr[1][0][0] = 5          arr[1][0][1] = 6
arr[1][1][0] = 7          arr[1][1][1] = 8
```

11.12 SPARSE MATRICES

Sparse matrix is a matrix that has many elements with a value zero. In order to efficiently utilize the memory, specialized algorithms and data structures that take advantage of the sparse structure of the matrix should be used. If we apply operations using standard matrix structures and algorithms to sparse matrices, then execution will slow down and the matrix will consume large amounts of memory. Sparse data can be easily compressed which in turn can significantly reduce memory usage.

Basically, there are two types of sparse matrices. In the first type of sparse matrix, all elements above the main diagonal have a value zero. This type of sparse matrix is also called a *lower triangular matrix* because if you see it pictorially, all the elements with a non-zero value appear below the diagonal. In a lower triangular matrix, $A_{i,j} = 0$ where $i >= j$. An $n \times n$ lower triangular matrix A has one non-zero element in the first row, two non-zero element, in the second row and likewise, n non-zero elements in the n^{th} row. Figure 11.36 shows a lower triangular matrix.

$$\begin{bmatrix} 1 & & & & \\ 5 & 3 & & & \\ 2 & 7 & -1 & & \\ 3 & 1 & 4 & 2 & \\ -9 & 2 & -8 & 1 & 7 \end{bmatrix}$$

Figure 11.36 Lower triangular matrix

To store the lower triangular matrix efficiently in memory, we can use a one-dimensional array which stores only the non-zero elements. The mapping between a 2D matrix and a 1D array can be done in any one of the following ways:

- Row wise mapping—here the contents of array `A[]` will be {1, 5, 3, 2, 7, -1, 3, 1, 4, 2, -9, 2, -8, 1, 7}
- Column wise mapping—here the contents of array `A[]` will be {1, 5, 2, 3, -9, 3, 7, 1, 2, -1, 4, -8, 2, 1, 7}

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ & 3 & 6 & 7 & 8 \\ & & -1 & 9 & 1 \\ & & & 9 & 3 \\ & & & & 7 \end{bmatrix}$$

Figure 11.37 Upper triangular matrix

Like a lower triangular matrix, we also have an *upper triangular matrix* in which $A_{i,j} = 0$ where $i<=j$. An $n \times n$ upper triangular matrix A has n non zero elements in the first row, $n-1$ non-zero elements in the second row and likewise, 1 non-zero elements in the n^{th} row. Figure 11.37 shows an upper triangular matrix.

In the second variant of a sparse matrix, elements with a non-zero value can appear only on the diagonal or immediately above or below the diagonal. This type of matrix is also called a *tridiagonal matrix*. Hence in a tridiagonal matrix, $A_{i,j} = 0$ where $|i - j| > 1$. Mathematically, in a tridiagonal matrix, if elements are present on the main diagonal, then it contains non-zero elements for $i = j$. In all there will be n elements diagonal by below the main

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

Figure 11.38 Tridiagonal matrix

diagonal, it contains non-zero elements for `i = j + 1`. In all there will be `n - 1` elements diagonal above the main diagonal, it contains non zero elements for `i = j - 1`. In all there will be `n - 1` elements. Figure 11.38 shows a tridiagonal matrix.

To store the tridiagonal matrix efficiently in memory, we can use a one-dimensional array which stores only the non-zero elements. The mapping between a 2D matrix and a 1D array can be done in any one of the following ways:

- Row wise mapping—here the contents of array `A[]` will be {4, 1, 5, 1, 2, 9, 3, 1, 4, 2, 2, 5, 1, 9, 8, 7}
- Column wise mapping—here the contents of array `A[]` will be {4, 5, 1, 1, 9, 2, 3, 4, 1, 2, 5, 2, 1, 8, 9, 7}
- Diagonal wise mapping—here the contents of array `A[]` will be {5, 9, 4, 5, 8, 4, 1, 3, 2, 1, 7, 1, 2, 1, 2, 9}

SUMMARY

- An array is a collection of similar data elements of the same data type.
- The elements of the array are stored in consecutive memory locations and are referenced by an index (also known as the subscript). Subscript indicates an ordinal number of the elements counted from the beginning of the array.
- Declaring an array means specifying three things: data type, name, and its size.
- The name of the array is a symbolic reference for the address of the first byte of the array. Therefore, whenever we use the array name, we are actually referring to the first byte of that array. The index specifies an offset from the beginning of the array to the element being referenced.
- A 2D array is specified using two subscripts where one subscript denotes row and the other denotes column. C considers the 2D array as an array of a 1D array.
- A multidimensional array in simple terms is an array of arrays. Like we have one index in a 1D array, two indices in a 2D array, in the same way we have n indices in an *n*-dimensional or multidimensional array. Conversely, an *n*-dimensional array is specified using *n* indices.
- Sparse matrix is a matrix that has many elements with a value zero. Basically, there are two types of sparse matrices. In the first type of sparse matrix, all elements above the main diagonal have a value zero. This type of sparse matrix is called a lower triangular matrix.
- In the second variant of a sparse matrix, elements with a non-zero value can appear only on the diagonal or immediately above or below the diagonal. This type of sparse matrix is called a tridiagonal matrix.

GLOSSARY

Array An array is a collection of similar data elements.

Array index Location of an item in an array.

Binary search Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

k-dimensional array An array with exactly *k* orthogonal axes or *k* dimensions.

Linear search Search an array by checking elements one at a time.

Lower triangular matrix A matrix $A_{i,j}$ forms a lower triangular when $i \geq j$.

Matrix A two-dimensional array in which the first index is the row and the second index is the column.

One-dimensional array An array with one dimension or one subscript/index.

Rectangular matrix An $n \times m$ matrix whose size may not be the same in both dimensions.

Sparse matrix A matrix that has relatively few non-zero elements.

Three-dimensional array An array with three dimensions or three subscripts/indices.

Two-dimensional array An array with two dimensions or two subscripts/indices.

Upper triangular matrix A matrix $A_{i,j}$ forms an upper triangular when $i \leq j$.