7

Arrays

7.1 INTRODUCTION

So far we have used only the fundamental data types, namely **char, int, float, double** and variations of **int** and **double**. Although these types are very useful, they are constrained by the fact that a variable of these types can store only one value at any given time. Therefore, they can be used only to handle limited amounts of data. In many applications, however, we need to handle a large volume of data in terms of reading, processing and printing. To process such large amounts of data, we need a powerful data type that would facilitate efficient storing, accessing and manipulation of data items. C supports a derived data type known as *array* that can be used for such applications.

An array is a *fixed-size* sequenced collection of elements of the same data type. It is simply a grouping of like-type data. In its simplest form, an array can be used to represent a list of numbers, or a list of names. Some examples where the concept of an array can be used:

- List of temperatures recorded every hour in a day, or a month, or a year.
- List of employees in an organization.
- List of products and their cost sold by a store.
- Test scores of a class of students.
- List of customers and their telephone numbers.
- Table of daily rainfall data.

and so on.

Since an array provides a convenient structure for representing data, it is classified as one of the *data structures* in C. Other data structures include structures, lists, queues and trees. A complete discussion of all data structures is beyond the scope of this text. However, we shall consider structures in Chapter 10 and lists in Chapter 13.

As we mentioned earlier, an array is a sequenced collection of related data items that share a common name. For instance, we can use an array name *salary* to represent a *set of salaries* of a group of employees in an organization. We can refer to the individual salaries by writing a number called *index* or *subscript* in brackets after the array name. For example,

```
salary [10]
```

represents the salary of 10^{th} employee. While the complete set of values is referred to as an array, individual values are called *elements*.

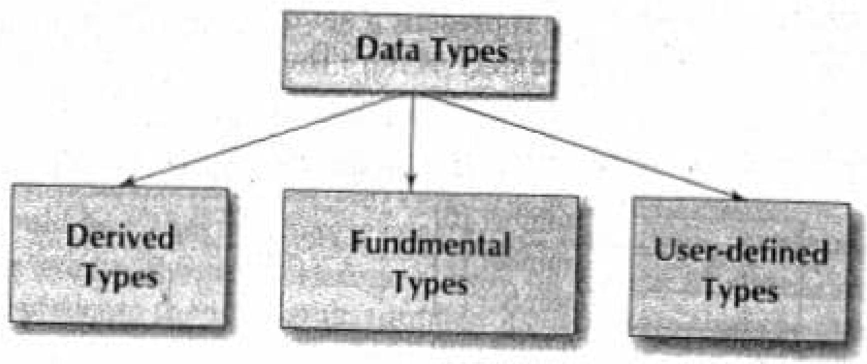
The ability to use a single name to represent a collection of items and to refer to an item by specifying the item number enables us to develop concise and efficient programs. For example, we can use a loop construct, discussed earlier, with the subscript as the control variable to read the entire array, perform calculations, and print out the results.

We can use arrays to represent not only simple lists of values but also tables of data in two, three or more dimensions. In this chapter, we introduce the concept of an array and discuss how to use it to create and apply the following types of arrays.

- One-dimensional arrays
- Two-dimensional arrays
- Multidimensional arrays

Data Structures

C supports a rich set of derived and user-defined data types in addition to a variety of fundamental types as shown below:



Derived Types	Fundmental Types	User-defined Types
- Arrays	- Integral Types	- Structures
- Functions	- Float Types	- Unions
- Pointers	- Character Types	- Enumerations

Arrays and structures are referred to as *structured data types* because they can be used to represent data values that have a structure of some sort. Structured data types provide an organizational scheme that shows the relationships among the individual elements and facilitate efficient data manipulations. In programming parlance, such data types are known as *data structures*.

In addition to arrays and structures, C supports creation and manipulation of the following data structures:

- Linked Lists
- Stacks
- Queues
- Trees

7.2 ONE-DIMENSIONAL ARRAYS

A list of items can be given one variable name using only one subscript and such a variable is called a *single-subscripted variable* or a *one-dimensional* array. In mathematics, we often deal with variables that are single-subscripted. For instance, we use the equation.

$$A = \frac{\sum_{i=1}^{n} x_i}{n}$$

to calculate the average of n values of x. The subscripted variable x_i refers to the ith element of x. In C, single-subscripted variable x_i can be expressed as

x[1], x[2], x[3],.........x[n]

The subscript can begin with number 0. That is

x[0]

is allowed. For example, if we want to represent a set of five numbers, say (35,40,20,57,19) by an array variable **number**, then we may declare the variable **number** as follows

```
int number[5];
```

and the computer reserves five storage locations as shown below:

	number [0]
	number [1]
	number [2]
	number [3]
	number [4]

The values to the array elements can be assigned as follows:

```
number[0] = 35;
number[1] = 40;
number[2] = 20;
number[3] = 57;
number[4] = 19;
```

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

number [0]	35
number [1]	40
number [2]	20
number [3]	57
number [4]	19

These elements may be used in programs just like any other C variable. For example, the following are valid statements:

```
a = number[0] + 10;
number[4] = number[0] + number [2];
number[2] = x[5] + y[10];
value[6] = number[i] * 3;
```

The subscripts of an array can be integer constants, integer variables like i, or expressions that yield integers. *C performs no bounds checking and, therefore, care should be exercised to ensure that the array indices are within the declared limits.*

7.3 DECLARATION OF ONE-DIMENSIONAL ARRAYS

Like any other variable, arrays must be declared before they are used so that the compiler can allocate space for them in memory. The general form of array declaration is

type *variable-name[size];*

The *type* specifies the type of element that will be contained in the array, such as **int, float,** or **char** and the *size* indicates the maximum number of elements that can be stored inside the array. For example,

```
float height[50];
```

declares the **height** to be an array containing 50 real elements. Any subscripts 0 to 49 are valid. Similarly,

```
int group[10];
```

declares the **group** as an array to contain a maximum of 10 integer constants. Remember:

- Any reference to the arrays outside the declared limits would not necessarily cause an error. Rather, it might result in unpredictable program results.
- The size should be either a numeric constant or a symbolic constant.

The C language treats character strings simply as arrays of characters. The *size* in a character string represents the maximum number of characters that the string can hold. For instance,

```
char name[10];
```

declares the **name** as a character array (string) variable that can hold a maximum of 10 characters. Suppose we read the following string constant into the string variable **name.**

"WELL DONE"

Each character of the string is treated as an element of the array **name** and is stored in the memory as follows:

'W'
'E'
'L'
'L'
' '
'D'
'O'
'N'
'E'
'\0'

When the compiler sees a character string, it terminates it with an additional null character. Thus, the element **name[10]** holds the null character '\0'. *When declaring character arrays, we must allow one extra element space for the null terminator.*

Example 7.1 Write a program using a single-subscripted variable to evaluate the following expressions:

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

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

Program in Fig. 7.1 uses a one-dimensional array **x** to read the values and compute the sum of their squares.

```
Program
    main()
    {
        int i ;
        float x[10], value, total ;
    /* . . . . . . .READING VALUES INTO ARRAY . . . . . . */
        printf("ENTER 10 REAL NUMBERS\n") ;

        for( i = 0 ; i < 10 ; i++ )
        {
            scanf("%f", &value) ;
            x[i] = value ;
        }
    /* . . . . . . . .COMPUTATION OF TOTAL . . . . . . .*/

        total = 0.0 ;
        for( i = 0 ; i < 10 ; i++ )
            total = total + x[i] * x[i] ;

    /*. . . . PRINTING OF x[i] VALUES AND TOTAL . . . */

        printf("\n");
        for( i = 0 ; i < 10 ; i++ )
            printf("x[%2d] = %5.2f\n", i+1, x[i]) ;

        printf("\ntotal = %.2f\n", total) ;
    }
Output
    ENTER 10 REAL NUMBERS
```

```
1.1 2.2 3.3 4.4 5.5 6.6 7.7 8.8 9.9 10.10

                x[ 1] = 1.10
                x[ 2] = 2.20
                x[ 3] = 3.30
                x[ 4] = 4.40
                x[ 5] = 5.50
                x[ 6] = 6.60
                x[ 7] = 7.70
                x[ 8] = 8.80
                x[ 9] = 9.90
                x[10] = 10.10

                Total = 446.86
```

Fig. 7.1 *Program to illustrate* ***one-dimensional array***

NOTE: C99 permits arrays whose size can be specified at run time. See Appendix "C99 Features".

7.4 INITIALIZATION OF ONE-DIMENSIONAL ARRAYS

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

- At compile time
- At run time

Compile Time Initialization

We can initialize the elements of arrays in the same way as the ordinary variables when they are declared. The general form of initialization of arrays is:

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

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

```
int number[3] = { 0,0,0 };
```

will declare the variable **number** as an array of size 3 and will assign zero to each element. If the number of values in the list is less than the number of elements, then only that many elements will be initialized. The remaining elements will be set to zero automatically. For instance,

```
float total[5] = {0.0,15.75,-10};
```

will initialize the first three elements to 0.0, 15.75, and –10.0 and the remaining two elements to zero.

The *size* may be omitted. In such cases, the compiler allocates enough space for all initialized elements. For example, the statement

```
int counter[ ] = {1,1,1,1};
```

will declare the **counter** array to contain four elements with initial values 1. This approach works fine as long as we initialize every element in the array.

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

```
char name[ ] = {'J','o', 'h', 'n', '\0'};
```

declares the **name** to be an array of five characters, initialized with the string "John" ending with the null character. Alternatively, we can assign the string literal directly as under:

```
char name [ ] = "John";
```

(Character arrays and strings are discussed in detail in Chapter 8.)

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

```
int number [5] = {10, 20};
```

will initialize the first two elements to 10 and 20 respectively, and the remaining elements to 0. Similarly, the declaration.

```
char city [5] = {'B'};
```

will initialize the first element to 'B' and the remaining four to NULL. It is a good idea, however, to declare the size explicitly, as it allows the compiler to do some error checking.

Remember, however, if we have more initializers than the declared size, the compiler will produce an error. That is, the statement

```
int number [3] = {10, 20, 30, 40};
```

will not work. It is illegal in C.

Run Time Initialization

An array can be explicitly initialized at run time. This approach is usually applied for initializing large arrays. For example, consider the following segment of a C program.

```
--------
--------
for (i = 0; i < 100; i = i+1)
{
    if   i < 50
         sum[i] = 0.0;          /* assignment statement */
    else
         sum[i] = 1.0;
}
--------
--------
```

The first 50 elements of the array **sum** are initialized to zero while the remaining 50 elements are initialized to 1.0 at run time.

We can also use a read function such as **scanf** to initialize an array. For example, the statements

```
int x [3];
scanf("%d%d%d", &x[0], &[1], &x[2]);
```

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

Example 7.2 Given below is the list of marks obtained by a class of 50 students in an annual examination.

43 65 51 27 79 11 56 61 82 09 25 36 07 49 55 63 74 81 49 37
40 49 16 75 87 91 33 24 58 78 65 56 76 67 45 54 36 63 12 21
73 49 51 19 39 49 68 93 85 59

Write a program to count the number of students belonging to each of following groups of marks: 0–9, 10–19, 20–29,......,100.

The program coded in Fig. 7.2 uses the array **group** containing 11 elements, one for each range of marks. Each element counts those values falling within the range of values it represents.

For any value, we can determine the correct group element by dividing the value by 10. For example, consider the value 59. The integer division of 59 by 10 yields 5. This is the element into which 59 is counted.

Program

```
#define  MAXVAL   50
#define  COUNTER  11
main()
{
    float      value[MAXVAL];
    int        i, low, high;
    int  group[COUNTER] = {0,0,0,0,0,0,0,0,0,0,0};
    /* . . . . . . . .READING AND COUNTING . . . . . . .*/
    for( i = 0 ; i < MAXVAL ; i++ )
    {
    /*. . . . . . . . .READING OF VALUES . . . . . . . . */
      scanf("%f", &value[i]) ;
    /*. . . . . .COUNTING FREQUENCY OF GROUPS. . . . . */
      ++ group[ (int) ( value[i]) / 10] ;
    }
    /* . . . .PRINTING OF FREQUENCY TABLE . . . . . . .*/
    printf("\n");
    printf(" GROUP     RANGE     FREQUENCY\n\n") ;
    for( i = 0 ; i < COUNTER ; i++ )
    {
        low = i * 10 ;
        if(i == 10)
           high = 100 ;
```

```
          else
            high = low + 9 ;
          printf(" %2d %3d to %3d %d\n",
                i+1, low, high, group[i] ) ;
     }
  }
Output
  43 65 51 27 79 11 56 61 82 09 25 36 07 49 55 63 74
  81 49 37 40 49 16 75 87 91 33 24 58 78 65 56 76 67 (Input data)
  45 54 36 63 12 21 73 49 51 19 39 49 68 93 85 59
 GROUP              RANGE               FREQUENCY
  1                 0  to   9             2
  2                10  to  19             4
  3                20  to  29             4
  4                30  to  39             5
  5                40  to  49             8
  6                50  to  59             8
  7                60  to  69             7
  8                70  to  79             6
  9                80  to  89             4
  10               90  to  99             2
  11              100  to 100             0
```

Fig. 7.2 *Program for frequency counting*

Note that we have used an initialization statement.

```
int group [COUNTER] = {0,0,0,0,0,0,0,0,0,0,0};
```

which can be replaced by

```
int group [COUNTER] = {0};
```

This will initialize all the elements to zero.

Searching and Sorting

Searching and sorting are the two most frequent operations performed on arrays. Computer Scientists have devised several data structures and searching and sorting techniques that facilitate rapid access to data stored in lists.

Sorting is the process of arranging elements in the list according to their values, in ascending or descending order. A sorted list is called an *ordered list*. Sorted lists are especially important in list searching because they facilitate rapid search operations. Many sorting techniques are available. The three simple and most important among them are:

- Bubble sort
- Selection sort
- Insertion sort

Other sorting techniques include Shell sort, Merge sort and Quick sort.

Searching is the process of finding the location of the specified element in a list. The specified element is often called the *search key*. If the process of searching finds a match of the search key with a list element value, the search said to be successful; otherwise, it is unsuccessful. The two most commonly used search techniques are:

- Sequential search
- Binary search

A detailed discussion on these techniques is beyond the scope of this text. Consult any good book on data structures and algorithms.

7.5 TWO-DIMENSIONAL ARRAYS

So far we have discussed the array variables that can store a list of values. There could be situations where a table of values will have to be stored. Consider the following data table, which shows the value of sales of three items by four sales girls:

	Item1	*Item2*	*Item3*
Salesgirl #1	310	275	365
Salesgirl #2	210	190	325
Salesgirl #3	405	235	240
Salesgirl #4	260	300	380

The table contains a total of 12 values, three in each line. We can think of this table as a matrix consisting of four *rows* and three *columns*. Each row represents the values of sales by a particular salesgirl and each column represents the values of sales of a particular item.

In mathematics, we represent a particular value in a matrix by using two subscripts such as v_{ij}. Here **v** denotes the entire matrix and v_{ij} refers to the value in the i^{th} row and j^{th} column. For example, in the above table v_{23} refers to the value 325.

C allows us to define such tables of items by using two-dimensional arrays. The table discussed above can be defined in C as

v[4][3]

Two-dimensional arrays are declared as follows:

type *array_name* [row_size][column_size];

Note that unlike most other languages, which use one pair of parentheses with commas to separate array sizes, C places each size in its own set of brackets.

Two-dimensional arrays are stored in memory, as shown in Fig.7.3. As with the single-dimensional arrays, each dimension of the array is indexed from zero to its maximum size minus one; the first index selects the row and the second index selects the column within that row.

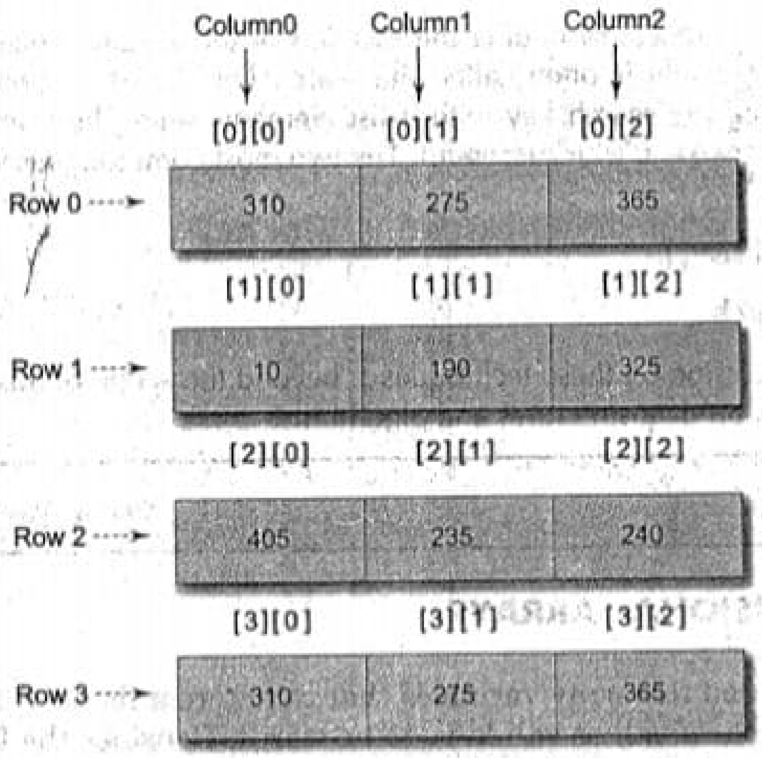


Fig. 7.3 *Representation of a **two-dimensional** array in memory*

Example 7.3 Write a program using a two-dimensional array to compute and print the following information from the table of data discussed above:
(a) Total value of sales by each girl.
(b) Total value of each item sold.
(c) Grand total of sales of all items by all girls.

The program and its output are shown in Fig. 7.4. The program uses the variable **value** in two-dimensions with the index i representing girls and j representing items. The following equations are used in computing the results:

(a) Total sales by m^{th} girl = $\sum_{j=0}^{2}$ value [m][j](girl_total[m])

(b) Total value of n^{th} item = $\sum_{i=0}^{3}$ value [i][n](item_total[n])

(c) Grand total = $\sum_{i=0}^{3}\sum_{j=0}^{2}$ value[i][j]

$$= \sum_{i=0}^{3} \text{girl\_total}[i]$$

$$= \sum_{j=0}^{2} \text{item\_total}[j]$$

Program

```
#define  MAXGIRLS 4
#define  MAXITEMS 3
main()
{
    int value[MAXGIRLS][MAXITEMS];
    int girl_total[MAXGIRLS] , item_total[MAXITEMS];
    int i, j, grand_total;
/*.......READING OF VALUES AND COMPUTING girl_total ...*/

    printf("Input data\n");
    printf("Enter values, one at a time, row-wise\n\n");

    for( i = 0 ; i < MAXGIRLS ; i++ )
    {
        girl_total[i] = 0;
        for( j = 0 ; j < MAXITEMS ; j++ )
        {
            scanf("%d", &value[i][j]);
            girl_total[i] = girl_total[i] + value[i][j];
        }
    }
/*.......COMPUTING item_total..........................*/

    for( j = 0 ; j < MAXITEMS ; j++ )
    {
        item_total[j] = 0;
        for( i =0 ; i < MAXGIRLS ; i++ )
            item_total[j] = item_total[j] + value[i][j];
    }
/*.......COMPUTING grand_total.........................*/

    grand_total = 0;
    for( i =0 ; i < MAXGIRLS ; i++ )
      grand_total = grand_total + girl_total[i];
/* .......PRINTING OF RESULTS...........................*/

    printf("\n GIRLS TOTALS\n\n");
```

```
        for( i = 0 ; i < MAXGIRLS ; i++ )
            printf("Salesgirl[%d] = %d\n", i+1, girl_total[i] );
        printf("\n ITEM TOTALS\n\n");
        for( j = 0 ; j < MAXITEMS ; j++ )
            printf("Item[%d] = %d\n", j+1 , item_total[j] );
        printf("\nGrand Total = %d\n", grand_total);
    }
Output
    Input data
    Enter values, one at a time, row_wise
    310 257 365
    210 190 325
    405 235 240
    260 300 380
    GIRLS TOTALS

    Salesgirl[1] = 950
    Salesgirl[2] = 725
    Salesgirl[3] = 880
    Salesgirl[4] = 940
    ITEM TOTALS

    Item[1] = 1185
    Item[2] = 1000
    Item[3] = 1310
    Grand Total = 3495
```

Fig. 7.4 *Illustration of two-dimensional arrays*

Example 7.4 Write a program to compute and print a multiplication table for numbers 1 to 5 as shown below:

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

The program shown in Fig. 7.5 uses a two-dimensional array to store the table values. Each value is calculated using the control variables of the nested for loops as follows:

product[i] [j] = row * column

where i denotes rows and j denotes columns of the product table. Since the indices i and j range from 0 to 4, we have introduced the following transformation:

row = i+1

column = j+1

Program

```
#define  ROWS     5
#define  COLUMNS  5
main()
{
        int row, column, product[ROWS][COLUMNS] ;
        int i, j ;
        printf(" MULTIPLICATION TABLE\n\n") ;
        printf(" ") ;
        for( j = 1 ; j <= COLUMNS ; j++ )
          printf("%4d" , j ) ;
        printf("\n") ;
        printf("————————————————————————————\n");
        for( i = 0 ; i < ROWS ; i++ )
        {
            row = i + 1 ;
            printf("%2d |", row) ;
            for( j = 1 ; j <= COLUMNS ; j++ )
            {
              column = j ;
              product[i][j] = row * column ;
              printf("%4d", product[i][j] ) ;
            }
            printf("\n") ;
        }
}
```

Output

```
   MULTIPLICATION TABLE
   1   2   3   4   5
 ---------------------
1 |   1   2   3   4   5
2 |   2   4   6   8  10
3 |   3   6   9  12  15
4 |   4   8  12  16  20
5 |   5  10  15  20  25
```

Fig. 7.5 *Program to print multiplication table using two-dimensional array*

7.6 INITIALIZING TWO-DIMENSIONAL ARRAYS

Like the one-dimensional arrays, two-dimensional arrays may be initialized by following their declaration with a list of initial values enclosed in braces. For example,

```
int table[2][3] = { 0,0,0,1,1,1};
```

initializes the elements of the first row to zero and the second row to one. The initialization is done row by row. The above statement can be equivalently written as

```
int table[2][3] = {{0,0,0}, {1,1,1}};
```

by surrounding the elements of the each row by braces.

We can also initialize a two-dimensional array in the form of a matrix as shown below:

```
int table[2][3] = {
                    {0,0,0},
                    {1,1,1}
                  };
```

Note the syntax of the above statements. Commas are required after each brace that closes off a row, except in the case of the last row.

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

```
int table [ ] [3] = {
                      { 0, 0, 0},
                      { 1, 1, 1}
                    };
```

is permitted.

If the values are missing in an initializer, they are automatically set to zero. For instance the statement

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

will initialize the first two elements of the first row to one, the first element of the second row to two, and all other elements to zero.

When all the elements are to be initialized to zero, the following short-cut method may be used.

```
int m[3][5] = { {0}, {0}, {0}};
```

The first element of each row is explicitly initialized to zero while other elements are automatically initialized to zero. The following statement will also achieve the same result:

```
int m [3] [5] = { 0, 0};
```

Example 7.5 A survey to know the popularity of four cars (Ambassador, Fiat, Dolphin and Maruti) was conducted in four cities (Bombay, Calcutta, Delhi and Madras). Each person surveyed was asked to give his city and the type of car he was using. The results, in coded form, are tabulated as follows:

M	1	C	2	B	1	D	3	M	2	B	4
C	1	D	3	M	4	B	2	D	1	C	3
D	4	D	4	M	1	M	1	B	3	B	3
C	1	C	1	C	2	M	4	M	4	C	2
D	1	C	2	B	3	M	1	B	1	C	2
D	3	M	4	C	1	D	2	M	3	B	4

Codes represent the following information:

M – Madras	1 – Ambassador
D – Delhi	2 – Fiat
C – Calcutta	3 – Dolphin
B – Bombay	4 – Maruti

Write a program to produce a table showing popularity of various cars in four cities.

A two-dimensional array **frequency** is used as an accumulator to store the number of cars used, under various categories in each city. For example, the element **frequency** [i][j] denotes the number of cars of type j used in city i. The **frequency** is declared as an array of size 5 × 5 and all the elements are initialized to zero.

The program shown in Fig. 7.6 reads the city code and the car code, one set after another, from the terminal. Tabulation ends when the letter X is read in place of a city code.

Program

```
main()
{
  int i, j, car;
  int frequency[5][5] = { {0},{0},{0},{0},{0} };
  char city;
  printf("For each person, enter the city code \n");
  printf("followed by the car code.\n");
  printf("Enter the letter X to indicate end.\n");
/*. . . . . . TABULATION BEGINS . . . . . */
  for( i = 1 ; i < 100 ; i++ )
  {
    scanf("%c", &city );
    if( city == 'X' )
      break;
    scanf("%d", &car );
    switch(city)
    {
          case 'B' :  frequency[1][car]++;
                      break;
          case 'C' :  frequency[2][car]++;
                      break;
          case 'D' :  frequency[3][car]++;
                      break;
          case 'M' :  frequency[4][car]++;
```

```
                    break;
        }
    }
/*. . . . .TABULATION COMPLETED AND PRINTING BEGINS. . . .*/
    printf("\n\n");
    printf(" POPULARITY TABLE\n\n");
    printf("-------------------------------------------\n");
    printf("City Ambassador Fiat Dolphin Maruti \n");
    printf("-------------------------------------------\n");
    for( i = 1 ; i <= 4 ; i++ )
    {
        switch(i)
        {
              case 1 : printf("Bombay   ") ;
                    break ;
        case 2 : printf("Calcutta ") ;
                    break ;
        case 3 : printf("Delhi    ") ;
                    break ;
        case 4 : printf("Madras   ") ;
                    break ;
    }
    for( j = 1 ; j <= 4 ; j++ )
      printf("%7d", frequency[i][j] ) ;
    printf("\n") ;
  }
  printf("-------------------------------------------\n");
/*. . . . . . . . . . PRINTING ENDS. . . . . . . . . . .*/
}
```

Output

```
  For each person, enter the city code
  followed by the car code.
  Enter the letter X to indicate end.
  M 1  C 2  B 1  D 3  M 2  B 4
  C 1  D 3  M 4  B 2  D 1  C 3
  D 4  D 4  M 1  M 1  B 3  B 3
  C 1  C 1  C 2  M 4  M 4  C 2
  D 1  C 2  B 3  M 1  B 1  C 2
  D 3  M 4  C 1  D 2  M 3  B 4   X
```

POPULARITY TABLE

City	Ambassador	Fiat	Dolphin	Maruti
Bombay	2	1	3	2

```
Calcutta    4        5     1     0
Delhi       2        1     3     2
Madras      4        1     1     4
```

Fig. 7.6 *Program to tabulate a survey data*

Memory Layout

The subscripts in the definition of a two-dimensional array represent rows and columns. This format maps the way that data elements are laid out in the memory. The elements of all arrays are stored contiguously in increasing memory locations, essentially in a single list. If we consider the memory as a row of bytes, with the lowest address on the left and the highest address on the right, a simple array will be stored in memory with the first element at the left end and the last element at the right end. Similarly, a two-dimensional array is stored "row-wise, starting from the first row and ending with the last row, treating each row like a simple array. This is illustrated below.

Column

row	0	1	2
0	10	20	30
1	40	50	60
2	70	80	90

3 × 3 array

row 0			row 1			row 2		
10	20	30	40	50	60	70	80	90
[0][0]	[0][1]	[0][2]	[1][0]	[1][1]	[1][2]	[2][0]	[2][1]	[2][2]
1	2	3	4	5	6	7	8	9

Memory Layout

For a multi–dimensional array, the order of storage is that the first element stored has 0 in all its subscripts, the second has all of its subscripts 0 except the far right which has a value of 1 and so on.

The elements of a 2 x 3 x 3 array will be stored as under

1	2	3	4	5	6	7	8	9	
000	001	002	010	011	012	020	021	022	..

	10	11	12	13	14	15	16	17	18
..	100	101	102	110	111	112	120	121	122

The far right subscript increments first and the other subscripts increment in order from right to left. The sequence numbers 1, 2,......, 18 represents the location of that element in the list

7.7 MULTI-DIMENSIONAL ARRAYS

C allows arrays of three or more dimensions. The exact limit is determined by the compiler. The general form of a multi-dimensional array is

```
type array_name[s1][s2][s3]....[sm];
```

where s_i is the size of the ith dimension. Some example are:

```
int survey[3][5][12];
float table[5][4][5][3];
```

survey is a three-dimensional array declared to contain 180 integer type elements. Similarly **table** is a four-dimensional array containing 300 elements of floating-point type.

The array **survey** may represent a survey data of rainfall during the last three years from January to December in five cities.

If the first index denotes year, the second city and the third month, then the element **survey[2][3][10]** denotes the rainfall in the month of October during the second year in city-3.

Remember that a three-dimensional array can be represented as a series of two-dimensional arrays as shown below:

Year 1

month / city	1	2	....	12
1				
. . . . 5				

Year 2

month / city	1	2	....	12
1				
. . . . 5				

ANSI C does not specify any limit for array dimension. However, most compilers permit seven to ten dimensions. Some allow even more.

7.8 DYNAMIC ARRAYS

So far, we created arrays at compile time. An array created at compile time by specifying size in the source code has a fixed size and cannot be modified at run time. The process of allocating memory at compile time is known as *static memory allocation* and the arrays that receive static memory allocation are called *static arrays*. This approach works fine as long as we know exactly what our data requirements are.

Consider a situation where we want to use an array that can vary greatly in size. We must guess what will be the largest size ever needed and create the array accordingly. A difficult task in fact! Modern languages like C do not have this limitation. In C it is possible to allocate memory to arrays at run time. This feature is known as *dynamic memory allocation* and the arrays created at run time are called *dynamic* arrays. This effectively postpones the array definition to run time.

Dynamic arrays are created using what are known as *pointer variables* and *memory management functions* **malloc, calloc** and **realloc**. These functions are included in the header file **<stdlib.h>**. The concept of dynamic arrays is used in creating and manipulating data structures such as linked lists, stacks and queues. We discuss in detail pointers and pointer variables in Chapter 11 and creating and managing linked lists in Chapter 13.

7.9 MORE ABOUT ARRAYS

What we have discussed in this chapter are the basic concepts of arrays and their applications to a limited extent. There are some more important aspects of application of arrays. They include:

- using printers for accessing arrays;
- passing arrays as function parameters;
- arrays as members of structures;
- using structure type data as array elements;
- arrays as dynamic data structures; and
- manipulating character arrays and strings.

These aspects of arrays are covered later in the following chapters:

Chapter 8 : Strings
Chapter 9 : Functions
Chapter 10 : Structures
Chapter 11 : Pointers
Chapter 13 : Linked Lists

Just Remember

- We need to specify three things, namely, name, type and size, when we declare an array.
- Always remember that subscripts begin at 0 (not 1) and end at size –1.
- Defining the size of an array as a symbolic constant makes a program more scalable.
- Be aware of the difference between the "kth element" and the "element k". The kth element has a subscript k-1, whereas the element k has a subscript of k itself.
- Do not forget to initialize the elements; otherwise they will contain "garbage".
- Supplying more initializers in the initializer list is a compile time error.
- Use of invalid subscript is one of the common errors. An incorrect or invalid index may cause unexpected results.
- When using expressions for subscripts, make sure that their results do not go outside the permissible range of 0 to size –1. Referring to an element outside the array bounds is an error.
- When using control structures for looping through an array, use proper relational expressions to eliminate "off-by-one" errors. For example, for an array of size 5, the following **for** statements are wrong:

```
for (i = 1; i < =5; i+ +)
for (i = 0; i < =5; i+ +)
for (i = 0; i = =5; i+ +)
for (i = 0; i < 4;  i+ +)
```

- Referring a two-dimensional array element like x[i, j] instead of x[i][j] is a compile time error.
- When initializing character arrays, provide enough space for the terminating null character.
- Make sure that the subscript variables have been properly initialized before they are used.
- Leaving out the subscript reference operator [ ] in an assignment operation is compile time error.
- During initialization of multi–dimensional arrays, it is an error to omit the array size for any dimension other than the first.

Case Studies

1. Median of a List of Numbers

When all the items in a list are arranged in an order, the middle value which divides the items into two parts with equal number of items on either side is called the *median*. Odd number of items have just one middle value while even number of items have two middle values. The median for even number of items is therefore designated as the average of the two middle values.

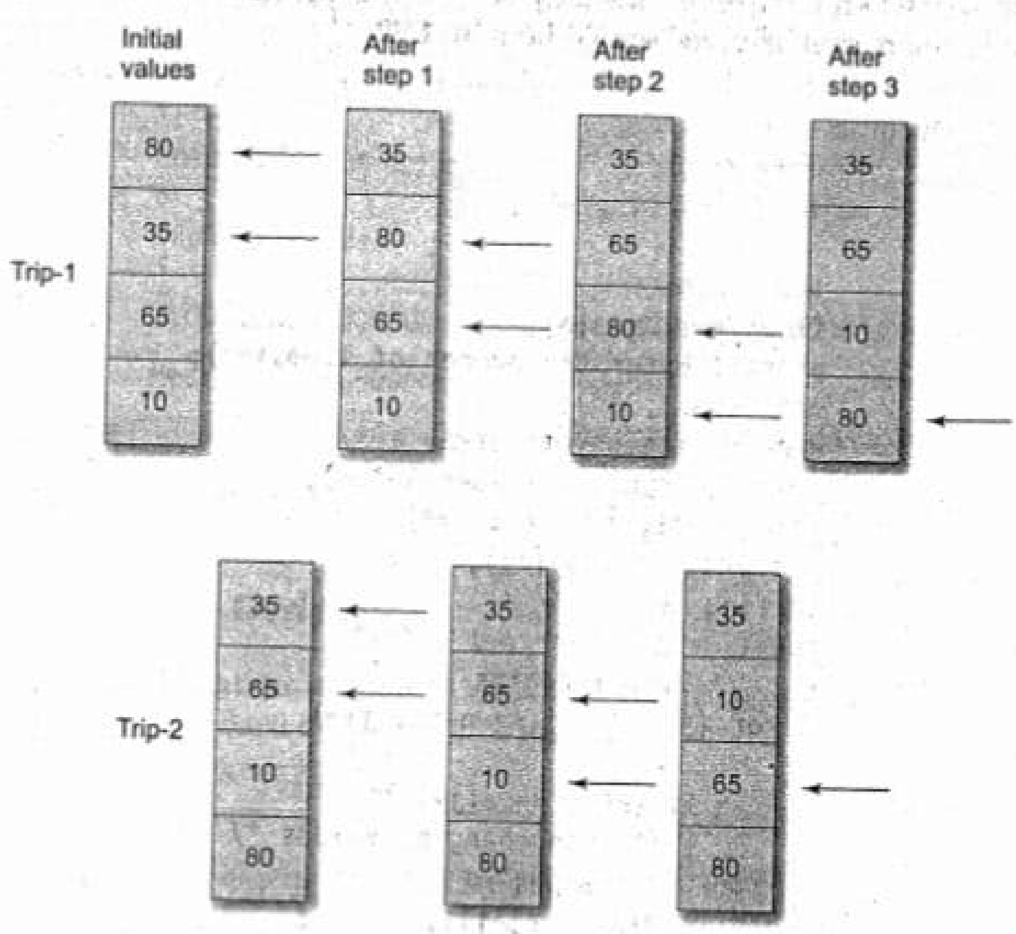
The major steps for finding the median are as follows:

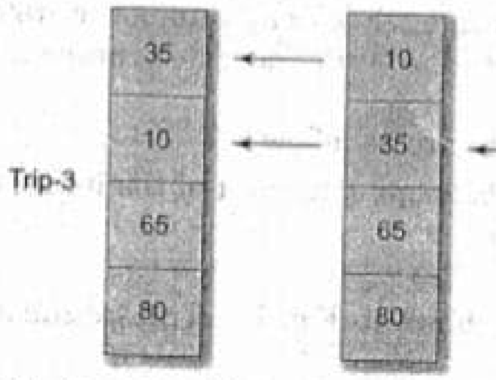
1. Read the items into an array while keeping a count of the items.
2. Sort the items in increasing order.
3. Compute median.

The program and sample output are shown in Fig. 7.7. The sorting algorithm used is as follows:

1. Compare the first two elements in the list, say a[1], and a[2]. If a[2] is smaller than a[1], then interchange their values.
2. Compare a[2] and a[3]; interchange them if a[3] is smaller than a[2].
3. Continue this process till the last two elements are compared and interchanged.
4. Repeat the above steps n–1 times.

In repeated trips through the array, the smallest elements 'bubble up' to the top. Because of this bubbling up effect, this algorithm is called *bubble sorting*. The bubbling effect is illustrated below for four items.





During the first trip, three pairs of items are compared and interchanged whenever needed. It should be noted that the number 80, the largest among the items, has been moved to the bottom at the end of the first trip. This means that the element 80 (the last item in the new list) need not be considered any further. Therefore, trip-2 requires only two pairs to be compared. This time, the number 65 (the second largest value) has been moved down the list. Notice that each trip brings the smallest value 10 up by one level.

The number of steps required in a trip is reduced by one for each trip made. The entire process will be over when a trip contains only one step. If the list contains **n** elements, then the number of comparisons involved would be **n(n–1)/2.**

Program

```
#define N 10
main( )
{
   int i,j,n;
   float median,a[N],t;
   printf("Enter the number of items\n");
   scanf("%d", &n);
/* Reading items into array a */
   printf("Input %d values \n",n);
   for (i = 1; i <= n ; i++)
      scanf("%f", &a[i]);
/* Sorting begins */
   for (i = 1 ; i <= n-1 ; i++)
   {    /* Trip-i begins */
      for (j = 1 ; j <= n-i ; j++)
      {
           if (a[j] <= a[j+1])
           { /* Interchanging values */
              t = a[j];
              a[j] = a[j+1];
              a[j+1] = t;
           }
           else
              continue ;
```

```
        }
      } /* sorting ends */
    /* calculation of median */
      if ( n % 2 == 0)
         median = (a[n/2] + a[n/2+1])/2.0 ;
      else
         median = a[n/2 + 1];
    /* Printing */
      for (i = 1 ; i <= n ; i++)
           printf("%f ", a[i]);
      printf("\n\nMedian is %f\n", median);
    }
```

Output

```
Enter the number of items
5
Input 5 values
1.111 2.222 3.333 4.444 5.555
5.555000 4.444000 3.333000 2.222000 1.111000

Median is 3.333000

Enter the number of items
6
Input 6 values
3 5 8 9 4 6
9.000000 8.000000 6.000000 5.000000 4.000000 3.000000

Median is 5.500000
```

Fig. 7.7 *Program to sort a list of numbers and to determine median*

2. Calculation of Standard Deviation

In statistics, standard deviation is used to measure deviation of data from its mean. The formula for calculating standard deviation of **n** items is

$$s = \sqrt{\text{variance}}$$

where

$$\text{variance} = \frac{1}{n}\sum_{i=1}^{n}(x_i - m)^2$$

and

$$m = \text{mean} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

The algorithm for calculating the standard deviation is as follows:

1. Read **n** items.
2. Calculate sum and mean of the items.
3. Calculate variance.
4. Calculate standard deviation.

Complete program with sample output is shown in Fig. 7.8.

Program

```
#include <math.h>
#define MAXSIZE 100
main( )
{
    int i,n;
    float value [MAXSIZE], deviation,
          sum,sumsqr,mean,variance,stddeviation;
    sum = sumsqr = n = 0 ;
    printf("Input values: input -1 to end \n");
    for (i=1; i< MAXSIZE ; i++)
    {
      scanf("%f", &value[i]);
      if (value[i] == -1)
        break;
      sum += value[i];
      n += 1;
    }
    mean = sum/(float)n;
    for (i = 1 ; i<= n; i++)
    {
      deviation = value[i] - mean;
      sumsqr += deviation * deviation;
    }
    variance = sumsqr/(float)n ;
    stddeviation = sqrt(variance) ;
    printf("\nNumber of items : %d\n",n);
    printf("Mean : %f\n", mean);
    printf("Standard deviation : %f\n", stddeviation);
}
```

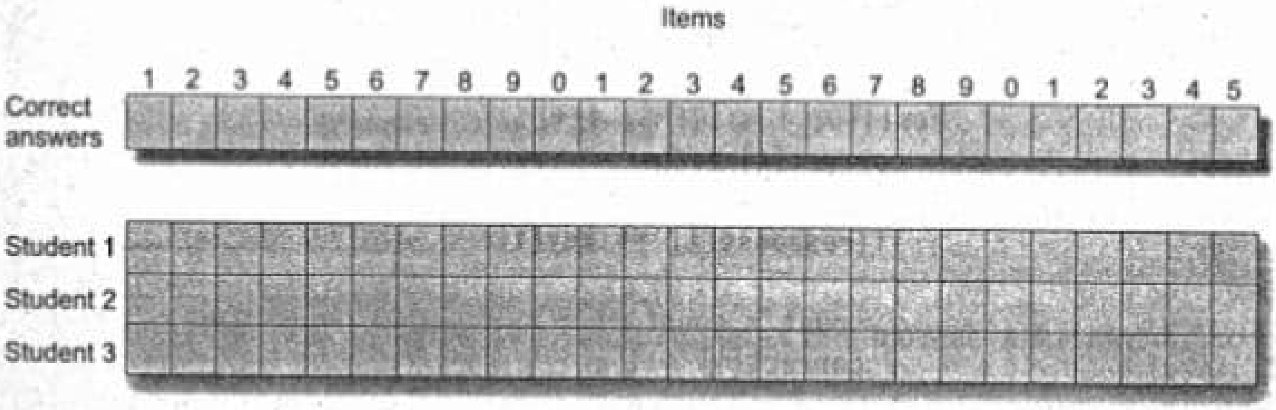
Output

```
Input values: input -1 to end
65 9 27 78 12 20 33 49 -1

Number of items : 8

Mean : 36.625000
Standard deviation : 23.510303
```

Fig. 7.8 *Program to calculate standard deviation*

3. Evaluating a Test

A test consisting of 25 multiple-choice items is administered to a batch of 3 students. Correct answers and student responses are tabulated as shown below:



The algorithm for evaluating the answers of students is as follows:

1. Read correct answers into an array.
2. Read the responses of a student and count the correct ones.
3. Repeat step-2 for each student.
4. Print the results.

A program to implement this algorithm is given in Fig. 7.9. The program uses the following arrays:

key[i] - To store correct answers of items

response[i] - To store responses of students

correct[i] - To identify items that are answered correctly.

Program

```
#define STUDENTS 3
#define ITEMS   25
main( )
{
  char key[ITEMS+1],response[ITEMS+1];
  int count, i, student,n,
      correct[ITEMS+1];
/* Reading of Correct answers */
  printf("Input key to the items\n");
  for(i=0; i < ITEMS; i++)
    scanf("%c",&key[i]);
  scanf("%c",&key[i]);
  key[i] = '\0';
/* Evaluation begins */
  for(student = 1; student <= STUD    ; student++)
  {
/*Reading student responses and counting correct ones*/
```

```
        count = 0;
        printf("\n");
        printf("Input responses of student-%d\n",student);
        for(i=0; i < ITEMS ; i++)
          scanf("%c",&response[i]);
        scanf("%c",&response[i]);
        response[i] = '\0';
        for(i=0; i < ITEMS; i++)
          correct[i] = 0;
        for(i=0; i < ITEMS ; i++)
          if(response[i] == key[i])
          {
            count = count +1 ;
            correct[i] = 1 ;
          }
        /* printing of results */
        printf("\n");
        printf("Student-%d\n", student);
        printf("Score is %d out of %d\n",count, ITEMS);
        printf("Response to the items below are wrong\n");
        n = 0;
        for(i=0; i < ITEMS ; i++)
          if(correct[i] == 0)
          {
               printf("%d ",i+1);
               n = n+1;
          }
        if(n == 0)
          printf("NIL\n");
        printf("\n");
        } /* Go to next student */
    /* Evaluation and printing ends */
    }
```

Output

```
Input key to the items
abcdabcdabcdabcdabcdabcda

Input responses of student-1
abcdabcdabcdabcdabcdabcda

Student-1
Score is 25 out of 25
Response to the following items are wrong
NIL

Input responses of student-2
abcddcbaabcdabcdddddddddd
```

```
Student-2
Score is 14 out of 25
Response to the following items are wrong
5 6 7 8 17 18 19 21 22 23 25

Input responses of student-3
aaaaaaaaaaaaaaaaaaaaaaaaa

Student-3
Score is 7 out of 25
Response to the following items are wrong
2 3 4 6 7 8 10 11 12 14 15 16 18 19 20 22 23 24
```

Fig. 7.9 *Program to evaluate responses to a multiple-choice test*

4. Production and Sales Analysis

A company manufactures five categories of products and the number of items manufactured and sold are recorded product-wise every week in a month. The company reviews its production schedule at every month-end. The review may require one or more of the following information:

(a) Value of weekly production and sales.
(b) Total value of all the products manufactured.
(c) Total value of all the products sold.
(d) Total value of each product, manufactured and sold.

Let us represent the products manufactured and sold by two two-dimensional arrays M and S respectively. Then,

M =

M11	M12	M13	M14	M15
M21	M22	M23	M24	M25
M31	M32	M33	M34	M35
M41	M42	M43	M44	M45

S =

S11	S12	S13	S14	S15
S21	S22	S23	S24	S25
S31	S32	S33	S34	S35
S41	S42	S43	S44	S45

where Mij represents the number of jth type product manufact l in ith week and Sij the number of jth product sold in ith week. We may also represent ... cost of each product by a single dimensional array C as follows:

C =

C1	C2	C3	C4	C5

where Cj is the cost of jth type product.

We shall represent the value of products manufactured and sold by two value arrays, namely, **Mvalue** and **Svalue**. Then,

Mvalue[i][j] = Mij x Cj

Svalue[i][j] = Sij x Cj

A program to generate the required outputs for the review meeting is shown in Fig. 7.10. The following additional variables are used:

Mweek[i] = Value of all the products manufactured in week i

$$= \sum_{j=1}^{5} \text{Mvalue[i][j]}$$

Sweek[i] = Value of all the products in week i

$$= \sum_{j=1}^{5} \text{Svalue[i][j]}$$

Mproduct[j] = Value of jth type product manufactured during the month

$$= \sum_{i=1}^{4} \text{Mvalue[i][j]}$$

Sproduct[j] = Value of jth type product sold during the month

$$= \sum_{i=1}^{4} \text{Svalue[i][j]}$$

Mtotal = Total value of all the products manufactured during the month

$$= \sum_{i=1}^{4} \text{Mweek[i]} = \sum_{j=1}^{5} \text{Mproduct[j]}$$

Stotal = Total value of all the products sold during the month

$$= \sum_{i=1}^{4} \text{Sweek[i]} = \sum_{j=1}^{5} \text{Sproduct[j]}$$

```
Program
    main( )
    {
      int M[5][6],S[5][6],C[6],
        Mvalue[5][6],Svalue[5][6],
        Mweek[5], Sweek[5],
        Mproduct[6], Sproduct[6],
        Mtotal, Stotal, i,j,number;
    /*  Input data    */
      printf (" Enter products manufactured week_wise \n");
      printf (" M11,M12,—, M21,M22,— etc\n");
```

```
    for(i=1; i<=4; i++)
      for(j=1;j<=5; j++)
        scanf("%d",&M[i][j]);
    printf (" Enter products sold week_wise\n");
    printf (" S11,S12,—, S21,S22,— etc\n");
    for(i=1; i<=4; i++)
      for(j=1; j<=5; j++)
        scanf("%d", &S[i][j]);
    printf(" Enter cost of each product\n");
      for(j=1; j <=5; j++)
        scanf("%d",&C[j]);
  /*Value matrices of production and sales */
    for(i=1; i<=4; i++)
      for(j=1; j<=5; j++)
      {
        Mvalue[i][j] = M[i][j] * C[j];
        Svalue[i][j] = S[i][j] * C[j];
      }
  /*Total value of weekly production and sales */
    for(i=1; i<=4; i++)
    {
      Mweek[i] = 0 ;
      Sweek[i] = 0 ;
      for(j=1; j<=5; j++)
      {
        Mweek[i] += Mvalue[i][j];
        Sweek[i] += Svalue[i][j];
      }
    }
  /*Monthly value of product_wise production and sales */
    for(j=1; j<=5; j++)
    {
      Mproduct[j] = 0 ;
      Sproduct[j] = 0 ;
      for(i=1; i<=4; i++)
      {
        Mproduct[j] += Mvalue[i][j];
        Sproduct[j] += Svalue[i][j];
      }
    }
  /*Grand total of production and sales values */
    Mtotal = Stotal = 0;
    for(i=1; i<=4; i++)
    {
      Mtotal += Mweek[i];
      Stotal += Sweek[i];
```

```
}
/***********************************************
   Selection and printing of information required
***********************************************/
printf("\n\n");
printf(" Following is the list of things you can\n");
printf(" request for. Enter appropriate item number\n");
printf(" and press RETURN Key\n\n");
printf(" 1.Value matrices of production & sales\n");
printf(" 2.Total value of weekly production & sales\n");
printf(" 3.Product_wise monthly value of production &");
printf(" sales\n");
printf(" 4.Grand total value of production & sales\n");
printf(" 5.Exit\n");
number = 0;
while(1)
{   /* Beginning of while loop */
  printf("\n\n ENTER YOUR CHOICE:");
  scanf("%d",&number);
  printf("\n");
  if(number == 5)
  {
    printf(" G O O D  B Y E\n\n");
    break;
  }
  switch(number)
  { /* Beginning of switch */
/* V A L U E   M A T R I C E S */
  case 1:
    printf(" VALUE MATRIX OF PRODUCTION\n\n");
    for(i=1; i<=4; i++)
    {
      printf(" Week(%d)\t",i);
      for(j=1; j <=5; j++)
        printf("%7d", Mvalue[i][j]);
      printf("\n");
    }
    printf("\n VALUE MATRIX OF SALES\n\n");
    for(i=1; i <=4; i++)
    {
      printf(" Week(%d)\t",i);
      for(j=1; j <=5; j++)
        printf("%7d", Svalue[i][j]);
      printf("\n");
    }
```

```
            break;
      /* W E E K L Y   A N A L Y S I S */
          case 2:
            printf(" TOTAL WEEKLY  PRODUCTION &  SALES\n\n");
            printf("               PRODUCTION    SALES\n");
            printf("               -----         --   \n");
            for(i=1; i <=4; i++)
            {
              printf(" Week(%d)\t", i);
              printf("%7d\t%7d\n", Mweek[i], Sweek[i]);
            }
            break;
      /* P R O D U C T   W I S E   A N A L Y S I S */
          case 3:
            printf(" PRODUCT_WISE TOTAL PRODUCTION &");
            printf(" SALES\n\n");
            printf("               PRODUCTION SALES\n");
            printf("               -----      --   \n");
            for(j=1; j <=5; j++)
            {
              printf(" Product(%d)\t", j);
              printf("%7d\t%7d\n",Mproduct[j],Sproduct[j]);
            }
            break;
      /* G R A N D   T O T A L S */
          case 4:
            printf(" GRAND TOTAL OF PRODUCTION & SALES\n");
            printf("\n Total production = %d\n", Mtotal);
            printf(" Total sales = %d\n", Stotal);
            break;
      /* D E F A U L T */
          default :
            printf(" Wrong choice, select again\n\n");
            break;
        } /* End of switch */
      } /* End of while loop */
      printf(" Exit from the program\n\n");
    } /* End of main */
```

Output

```
    Enter products manufactured week_wise
      M11, M12, ----, M21, M22, ---- etc
      11   15   12   14   13
      13   13   14   15   12
      12   16   10   15   14
      14   11   15   13   12
```

```
Enter products sold week_wise
S11,S12,----, S21,S22,---- etc
10  13  9   12  11
12  10  12  14  10
11  14  10  14  12
12  10  13  11  10
Enter cost of each product
10 20 30 15 25

Following is the list of things you can
request for. Enter appropriate item number
and press RETURN key
1.Value matrices of production & sales
2.Total value of weekly production & sales
3.Product_wise monthly value of production & sales
4.Grand total value of production & sales
5.Exit
ENTER YOUR CHOICE:1
VALUE MATRIX OF PRODUCTION
     Week(1)     110    300    360    210    325
     Week(2)     130    260    420    225    300
     Week(3)     120    320    300    225    350
     Week(4)     140    220    450    185    300
VALUE MATRIX OF SALES
     Week(1)     100    260    270    180    275
     Week(2)     120    200    360    210    250
     Week(3)     110    280    300    210    300
     Week(4)     120    200    390    165    250
ENTER YOUR CHOICE:2
TOTAL WEEKLY  PRODUCTION &  SALES
              PRODUCTION    SALES
              __________    _____
   Week(1)       1305        1085
   Week(2)       1335        1140
   Week(3)       1315        1200
   Week(4)       1305        1125
ENTER YOUR CHOICE:3
PRODUCT_WISE TOTAL PRODUCTION &  SALES
                   PRODUCTION    SALES
                   __________    _____
   Product(1)         500         450
   Product(2)        1100         940
   Product(3)        1530        1320
   Product(4)         855         765
   Product(5)        1275        1075

ENTER YOUR CHOICE:4

GRAND TOTAL OF PRODUCTION & SALES
```

```
Total production     = 5260
Total sales     = 4550
ENTER YOUR CHOICE:5
G O O D  B Y E
Exit from the program
```

Fig. 7.10 *Program for production and sales analysis*

Review Questions

7.1 State whether the following statements are *true* or *false*.
   (a) The type of all elements in an array must be the same.
   (b) When an array is declared, C automatically initializes its elements to zero.
   (c) An expression that evaluates to an integral value may be used as a subscript.
   (d) Accessing an array outside its range is a compile time error.
   (e) A **char** type variable cannot be used as a subscript in an array.
   (f) An unsigned long int type can be used as a subscript in an array.
   (g) In C, by default, the first subscript is zero.
   (h) When initializing a multidimensional array, not specifying all its dimensions is an error.
   (i) When we use expressions as a subscript, its result should be always greater than zero.
   (j) In C, we can use a maximum of 4 dimensions for an array.
   (k) In declaring an array, the array size can be a constant or variable or an expression.
   (l) The declaration int x[2] = {1,2,3}; is illegal.

7.2 Fill in the blanks in the following statements.
   (a) The variable used as a subscript in an array is popularly known as ________ variable.
   (b) An array can be initialized either at compile time or at ________.
   (c) An array created using **malloc** function at run time is referred to as ______ array.
   (d) An array that uses more than two subscript is referred to as ______ array.
   (e) _______ is the process of arranging the elements of an array in order.

7.3 Identify errors, if any, in each of the following array declaration statements, assuming that ROW and COLUMN are declared as symbolic constants:
   (a) `int score (100);`
   (b) `float values [10,15];`
   (c) `float average[ROW],[COLUMN];`
   (d) `char name[15];`
   (e) `int sum[ ];`
   (f) `double salary [i + ROW]`
   (g) `long int number [ROW]`
   (h) `int array x[COLUMN];`

7.4 Identify errors, if any, in each of the following initialization statements.

(a) `int number[ ] = {0,0,0,0,0};`
(b) `float item[3][2] = {0,1,2,3,4,5};`
(c) `char word[ ] = {'A','R', 'R', 'A', 'Y'};`
(d) `int m[2,4] = {(0,0,0,0)(1,1,1,1)};`
(e) `float result[10] = 0;`

7.5 Assume that the arrays A and B are declared as follows:

```
int A[5][4];
float B[4];
```

Find the errors (if any) in the following program segments.

(a)
```
for (i=1; i<=5; i++)
for(j=1; j<=4; j++)
A[i][j] = 0;
```
(b)
```
for (i=1; i<4; i++)
scanf("%f", B[i]);
```
(c)
```
for (i=0; i<=4; i++)
B[i] = B[i]+i;
```
(d)
```
for (i=4; i>=0; i--)
for (j=0; j<4; j++)
A[i][j] = B[j] + 1.0;
```

7.6 Write a **for** loop statement that initializes all the diagonal elements of an array to one and others to zero as shown below. Assume 5 rows and 5 columns.

1	0	0	0	0	.....	0
0	1	0	0	0	.....	0
0	0	1	0	0	.....	0
.	.	.	.	.		.
.	.	.	.	.		.
.	.	.	.	.		.
.	.	.	.	.		.
.	.	.	.	.		.
0	0	0	0	0	.....	1

7.7 We want to declare a two-dimensional integer type array called **matrix** for 3 rows and 5 columns. Which of the following declarations are correct?

(a) `int maxtrix [3],[5];`
(b) `int matrix [5] [3];`
(c) `int matrix [1+2] [2+3];`
(d) `int matrix [3,5];`
(e) `int matrix [3] [5];`

7.8 Which of the following initialization statements are correct?

(a) `char str1[4] = "GOOD";`
(b) `char str2[ ] = "C";`
(c) `char str3[5] = "Moon";`

(d) `char str4[ ] = {'S', 'U', 'N'};`

(e) `char str5[10] = "Sun";`

7.9 What is a data structure? Why is an array called a data structure?

10 What is a dynamic array? How is it created? Give a typical example of use of a dynamic array.

11 What is the error in the following program?

```
main ( )
{
        int x ;
        float y [ ] ;
        ......
}
```

12 What happens when an array with a specified size is assigned

(a) with values fewer than the specified size; and

(b) with values more than the specified size.

13 Discuss how initial values can be assigned to a multidimensional array.

14 What is the output of the following program?

```
main ( )
{
        int m [ ] = { 1,2,3,4,5 }
        int x, y = 0;
        for (x = 0; x < 5; x++ )
                y = y + m [ x ];
        printf("%d", y) ;
}
```

15 What is the output of the following program?

```
main ( )
{
        chart string [ ] = "HELLO WORLD" ;
        int m;
        for (m = 0; string [m] != '\0'; m++ )
             if ( (m%2) == 0)
                     printf("%c", string [m] );
}
```

Programming Exercises

7.1 Write a program for fitting a straight line through a set of points (x_i, y_i), $i = 1, \ldots, n$. The straight line equation is

$$y = mx + c$$

and the values of m and c are given by

$$m = \frac{n\,\Sigma(x_i y_i) - (\Sigma x_i)(\Sigma y_i)}{n(\Sigma x_i^2) - (\Sigma x_i)^2}$$

$$c = \frac{1}{n}\,(\Sigma\, y_i - m\,\Sigma\, x_i)$$

All summations are from 1 to n.

7.2 The daily maximum temperatures recorded in 10 cities during the month of January (for all 31 days) have been tabulated as follows:

Day \ City	1	2	3 - - - - - - - - - - - - - - - -	10
1			- - - - - - - - - - - - - - - -	
2				
3				
-				
-				
-				
-				
31				

Write a program to read the table elements into a two-dimensional array **temperature**, and to find the city and day corresponding to

(a) the highest temperature and

(b) the lowest temperature.

7.3 An election is contested by 5 candidates. The candidates are numbered 1 to 5 and the voting is done by marking the candidate number on the ballot paper. Write a program to read the ballots and count the votes cast for each candidate using an array variable **count**. In case, a number read is outside the range 1 to 5, the ballot should be considered as a 'spoilt ballot' and the program should also count the number of spoilt ballots.

7.4 The following set of numbers is popularly known as Pascal's triangle.

1						
1	1					
1	2	1				
1	3	3	1			
1	4	6	4	1		
1	5	10	10	5	1	
-	-	-	-	-	-	-
-	-	-	-	-	-	-

If we denote rows by i and columns by j, then any element (except the boundary elements) in the triangle is given by

$$p_{ij} = p_{i-1,\, j-1} + p_{i-1,\, j}$$

Write a program to calculate the elements of the Pascal triangle for 10 rows and print the results.

7.5 The annual examination results of 100 students are tabulated as follows:

Roll No.	*Subject 1*	*Subject 2*	*Subject 3*
.			
.			

Write a program to read the data and determine the following:

(a) Total marks obtained by each student.

(b) The highest marks in each subject and the Roll No. of the student who secured it.

(c) The student who obtained the highest total marks.

7.6 Given are two one-dimensional arrays A and B which are sorted in ascending order. Write a program to **merge** them into a single sorted array C that contains every item from arrays A and B, in ascending order.

7.7 Two matrices that have the same number of rows and columns can be multiplied to produce a third matrix. Consider the following two matrices.

$$A = \begin{bmatrix} a_{11}\ a_{12} \ldots\ldots a_{1n} \\ a_{12}\ a_{22} \ldots\ldots a_{2n} \\ . \qquad\qquad . \\ . \qquad\qquad . \\ . \qquad\qquad . \\ a_{n1} \ldots\ldots a_{nn} \end{bmatrix}$$

$$B = \begin{bmatrix} b_{11}\ b_{12} \ldots\ldots b_{1n} \\ b_{12}\ b_{22} \ldots\ldots b_{2n} \\ . \qquad\qquad . \\ . \qquad\qquad . \\ . \qquad\qquad . \\ b_{n1} \ldots\ldots b_{nn} \end{bmatrix}$$

The product of **A** and **B** is a third matrix C of size n×n where ..ch element of C is given by the following equation.

$$C_{ij} = \sum_{k=1}^{n} a_{ik}b_{kj}$$

Write a program that will read the values of elements of A and B and produce the product matrix **C.**

7.8 Write a program that fills a five-by-five matrix as follows:

- Upper left triangle with +1s
- Lower right triangle with –1s
- Right to left diagonal with zeros

Display the contents of the matrix using not more than two **printf** statements

7.9 Selection sort is based on the following idea:

Selecting the largest array element and swapping it with the last array element leaves an unsorted list whose size is 1 less than the size of the original list. If we repeat this step again on the unsorted list we will have an ordered list of size 2 and an unordered list size n–2 . When we repeat this until the size of the unsorted list becomes one, the result will be a sorted list.

Write a program to implement this algorithm.

7.10 Develop a program to implement the binary search algorithm. This technique compares the search key value with the value of the element that is midway in a "sorted" list. Then;

(a) If they match, the search is over.

(b) If the search key value is less than the middle value, then the first half of the list contains the key value.

(c) If the search key value is greater than the middle value, then the second half contains the key value.

Repeat this "divide-and-conquer" strategy until we have a match. If the list is reduced to one non-matching element, then the list does not contain the key value.

Use the sorted list created in Exercise 7.9 or use any other sorted list.

7.11 Write a program that will compute the length of a given character string.

7.12 Write a program that will count the number occurrences of a specified character in a given line of text. Test your program.

7.13 Write a program to read a matrix of size m × n and print its transpose.

7.14 Every book published by international publishers should carry an International Standard Book Number (ISBN). It is a 10 character 4 part number as shown below.

0-07-041183-2

The first part denotes the region, the second represents publisher, the third identifies the book and the fourth is the check digit. The check digit is computed as follows:

Sum = (1 × first digit) + (2 × second digit) + (3 × third digit) + - - - - + (9 × ninth digit).

Check digit is the remainder when sum is divided by 11. Write a program that reads a given ISBN number and checks whether it represents a valid ISBN.

7.15 Write a program to read two matrices A and B and print the following:

(a) A + B; and

(b) A – B.

8

Character Arrays and Strings

8.1 INTRODUCTION

A string is a sequence of characters that is treated as a single data item. We have used strings in a number of examples in the past. Any group of characters (except double quote sign) defined between double quotation marks is a string constant. Example:

"Man is obviously made to think."

If we want to include a double quote in the string to be printed, then we may use it with a back slash as shown below.

"\" Man is obviously made to think,\" said Pascal."

For example,

```
printf ("\" Well Done !"\");
```

will output the string

" Well Done !"

while the statement

```
printf(" Well Done !");
```

will output the string

Well Done !

Character strings are often used to build meaningful and readable programs. The common operations performed on character strings include:

- Reading and writing strings.
- Combining strings together.
- Copying one string to another.
- Comparing strings for equality.
- Extracting a portion of a string.

In this chapter we shall discuss these operations in detail and examine library functions that implement them.

D INITIALIZING STRING VARIABLES

C does not support strings as a data type. However, it allows us to represent strings as character arrays. In C, therefore, a string variable is any valid C variable name and is always declared as an array of characters. The general form of declaration of a string variable is:

char string_name[*size*];

The *size* determines the number of characters in the string_name. Some examples are:

```
char city[10];
char name[30];
```

When the compiler assigns a character string to a character array, it automatically supplies a *null* character ('\0 ') at the end of the string. Therefore, the *size* should be equal to the maximum number of characters in the string *plus* one.

Like numeric arrays, character arrays may be initialized when they are declared. C permits a character array to be initialized in either of the following two forms:

```
char city [9] = " NEW YORK ";
char city [9]={'N','E','W',' ','Y','O','R','K','\0'};
```

The reason that **city** had to be 9 elements long is that the string NEW YORK contains 8 characters and one element space is provided for the null terminator. Note that when we initialize a character array by listing its elements, we must supply explicitly the null terminator.

C also permits us to initialize a character array without specifying the number of elements. In such cases, the size of the array will be determined automatically, based on the number of elements initialized. For example, the statement

```
char string [ ] = {'G','O','O','D','\0'};
```

defines the array **string** as a five element array.

We can also declare the size much larger than the string size in the initializer. That is, the statement.

```
char str[10] = "GOOD";
```

is permitted. In this case, the computer creates a character array of size 10, places the value "GOOD" in it, terminates with the null character, and initializes all other elements to NULL. The storage will look like:

G	O	O	D	\0	\0	\0	\0	\0	\0

However, the following declaration is illegal.

```
char str2[3] = "GOOD";
```

This will result in a compile time error. Also note that we cannot separate the initialization from declaration. That is,

```
char str3[5];
str3 = "GOOD";
```

is not allowed. Similarly,

```
char s1[4] = "abc";
char s2[4];
s2 = s1; /* Error */
```

is not allowed. An array name cannot be used as the left operand of an assignment operator.

Terminating Null Character

You must be wondering, "why do we need a terminating null character?" As we know, a string is not a data type in C, but it is considered a data structure stored in an array. The string is a variable-length structure and is stored in a fixed-length array. The array size is not always the size of the string and most often it is much larger than the string stored in it. Therefore, the last element of the array need not represent the end of the string. We need some way to determine the end of the string data and the null character serves as the "end-of-string" marker.

8.3 READING STRINGS FROM TERMINAL

Using scanf Function

The familiar input function **scanf** can be used with **%s** format specification to read in a string of characters. Example:

```
char address[10]
scanf("%s", address);
```

The problem with the **scanf** function is that it terminates its input on the first white space it finds. A white space includes blanks, tabs, carriage returns, form feeds, and new lines. Therefore, if the following line of text is typed in at the terminal,

NEW YORK

then only the string "NEW" will be read into the array **address**, since the blank space after the word 'NEW' will terminate the reading of string.

The **scanf** function automatically terminates the string that is read with a null character and therefore the character array should be large enough to hold the input string plus the null character. Note that unlike previous **scanf** calls, in the case of character arrays, the ampersand (&) is not required before the variable name.

The **address** array is created in the memory as shown below:

N	E	W	\0	?	?	?	?	?	?
0	1	2	3	4	5	6	7	8	9

Note that the unused locations are filled with garbage.

If we want to read the entire line "NEW YORK", then we may use two character arrays of appropriate sizes. That is,

```
char adr1[5], adr2[5];
scanf("%s %s", adr1, adr2);
```

with the line of text

NEW YORK

will assign the string "NEW" to **adr1** and "YORK" to **adr2**.

Example 8.1 Write a program to read a series of words from a terminal using **scanf** function.

The program shown in Fig. 8.1 reads four words and displays them on the screen. Note that the string 'Oxford Road' is treated as *two words* while the string 'Oxford-Road' as *one word*.

Program

```
main( )
{
    char word1[40], word2[40], word3[40], word4[40];

    printf("Enter text : \n");
    scanf("%s %s", word1, word2);
    scanf("%s", word3);
    scanf("%s", word4);

    printf("\n");
    printf("word1 = %s\nword2 = %s\n", word1, word2);
    printf("word3 = %s\nword4 = %s\n", word3, word4);
}
```

Output

```
Enter text :
Oxford Road, London M17ED

word1 = Oxford
word2 = Road,
word3 = London
word4 = M17ED

Enter text :
Oxford-Road, London-M17ED United Kingdom
word1 = Oxford-Road
```

```
word2 = London-M17ED
word3 = United
word4 = Kingdom
```

Fig. 8.1 *Reading a series of words using **scanf** function*

We can also specify the field width using the form %ws in the **scanf** statement for reading a specified number of characters from the input string . Example:

```
scanf("%ws", name);
```

Here, two things may happen.

1. The width **w** is equal to or greater than the number of characters typed in. The entire string will be stored in the string variable.
2. The width **w** is less than the number of characters in the string. The excess characters will be truncated and left unread.

Consider the following statements:

```
char name[10];
scanf("%5s", name);
```

The input string RAM will be stored as:

R	A	M	\0	?	?	?	?	?	?
0	1	2	3	4	5	6	7	8	9

The input string KRISHNA will be stored as:

K	R	I	S	H	\0	?	?	?	?
0	1	2	3	4	5	6	7	8	9

Reading a Line of Text

We have seen just now that **scanf** with **%s** or **%ws** can read only strings without whitespaces. That is, they cannot be used for reading a text containing more than one word. However, C supports a format specification known as the *edit set conversion code* %[. .] that can be used to read a line containing a variety of characters, including whitespaces. Recall that we have used this conversion code in Chapter 4. For example,
the program segment

```
char line [80];
scanf("%[^\n]", line);
printf("%s", line);
```

will read a line of input from the keyboard and display the same on the screen. We would very rarely use this method, as C supports an intrinsic string function to do this job. This is discussed in the next section.

Using *getchar* and *gets* Functions

We have discussed in Chapter 4 as to how to read a single character from the terminal, using the function **getchar.** We can use this function repeatedly to read successive single characters from the input and place them into a character array. Thus, an entire line of text can be read and stored in an array. The reading is terminated when the newline character ('\n') is entered and the null character is then inserted at the end of the string. The **getchar** function call takes the form:

```
char ch;
ch = getchar( );
```

Note that the **getchar** function has no parameters.

Example 8.2 Write a program to read a line of text containing a series of words from the terminal.

The program shown in Fig. 8.2 can read a line of text (up to a maximum of 80 characters) into the string **line** using **getchar** function. Every time a character is read, it is assigned to its location in the string **line** and then tested for *newline* character. When the *newline* character is read (signalling the end of line), the reading loop is terminated and the *newline* character is replaced by the null character to indicate the end of character string.

When the loop is exited, the value of the index **c** is one number higher than the last character position in the string (since it has been incremented after assigning the new character to the string). Therefore the index value **c-1** gives the position where the *null* character is to be stored.

```
Program
    #include <stdio.h>
    main( )
    {
      char line[81], character;
      int c;
      c = 0;
      printf("Enter text. Press <Return> at end\n");
      do
      {
        character = getchar();
        line[c] = character;
        c++;
      }
      while(character != '\n');
      c = c - 1;
      line[c] = '\0';
      printf("\n%s\n", line);
    }
```

```
Output
    Enter text. Press <Return> at end
    Programming in C is interesting.
    Programming in C is interesting.
    Enter text. Press <Return> at end
    National Centre for Expert Systems, Hyderabad.
    National Centre for Expert Systems, Hyderabad.
```

Fig. 8.2 *Program to **read a line of text** from terminal*

Another and more convenient method of reading a string of text containing whitespaces is to use the library function **gets** available in the *<stdio.h>* header file. This is a simple function with one string parameter and called as under:

gets (str);

str is a string variable declared properly. It reads characters into **str** from the keyboard until a new-line character is encountered and then appends a null character to the string. Unlike **scanf**, it does not skip whitespaces. For example the code segment

```
char line [80];
gets (line);
printf ("%s", line);
```

reads a line of text from the keyboard and displays it on the screen. The last two statements may be combined as follows:

```
printf("%s", gets(line));
```

(Be careful not to input more character that can be stored in the string variable used. Since C does not check array-bounds, it may cause problems.)

C does not provide operators that work on strings directly. For instance we cannot assign one string to another directly. For example, the assignment statements.

```
string = "ABC";
string1 = string2;
```

are not valid. If we really want to copy the characters in **string2** into **string1**, we may do so on a character-by-character basis.

Example 8.3 Write a program to copy one string into another and count the number of characters copied.

The program is shown in Fig. 8.3. We use a **for** loop to copy the characters contained inside **string2** into the **string1**. The loop is terminated when the *null* character is reached. Note that we are again assigning a null character to the **string1**.

```
Program
    main( )
    {
        char string1[80], string2[80];
        int i;
        printf("Enter a string \n");
        printf("?");
        scanf("%s", string2);
        for( i=0 ; string2[i] != '\0'; i++)
            string1[i] = string2[i];
        string1[i] = '\0';
        printf("\n");
        printf("%s\n", string1);
        printf("Number of characters = %d\n", i );
    }
Output
    Enter a string
    ?Manchester
    Manchester
    Number of characters = 10
    Enter a string
    ?Westminster
    Westminster
    Number of characters = 11
```

Fig. 8.3 *Copying one string into another*

8.4 WRITING STRINGS TO SCREEN

Using printf Function

We have used extensively the **printf** function with %s format to print strings to the scre[en]. The format %s can be used to display an array of characters that is terminated by the n[ull] character. For example, the statement

```
printf("%s", name);
```

can be used to display the entire contents of the array **name**.

We can also specify the precision with which the array is displayed. For instance, the spe[ci]fication

```
%10.4
```

indicates that the *first four* characters are to be printed in a field width of 10 columns.

However, if we include the minus sign in the specification (e.g., %-10.4s), the string will be printed left-justified. The Example 8.4 illustrates the effect of various %s specifications.

Example 8.4 Write a program to store the string "United Kingdom" in the array **country** and display the string under various format specifications.

The program and its output are shown in Fig. 8.4. The output illustrates the following features of the %s specifications.

1. When the field width is less than the length of the string, the entire string is printed.
2. The integer value on the right side of the decimal point specifies the number of characters to be printed.
3. When the number of characters to be printed is specified as zero, nothing is printed.
4. The minus sign in the specification causes the string to be printed left-justified.
5. The specification % .ns prints the first n characters of the string.

Program

```
main()
{
  char country[15] = "United Kingdom";
  printf("\n\n");
  printf("*123456789012345*\n");
  printf(" ----- \n");
  printf("%15s\n", country);
  printf("%5s\n", country);
  printf("%15.6s\n", country);
  printf("%-15.6s\n", country);
  printf("%15.0s\n", country);
  printf("%.3s\n", country);
  printf("%s\n", country);
  printf("----- \n");
}
```

Output

```
*123456789012345*
-----
United Kingdom
United Kingdom
      United
United

Uni
United Kingdom
-----
```

Fig. 8.4 *Writing strings using %s format*

The **printf** on UNIX supports another nice feature that allows for variable field width or precision. For instance

```
printf("%*.*s\n", w, d, string);
```

prints the first **d** characters of the string in the field width of **w**.

This feature comes in handy for printing a sequence of characters. Example 8.5 illustrates this.

Example 8.5 Write a program using **for loop** to print the following output:

```
C
CP
CPr
CPro
.....
.....
CProgramming
CProgramming
.....
.....
CPro
CPr
CP
C
```

The outputs of the program in Fig. 8.5, for variable specifications **%12.*s, %.*s, and %*.1** are shown in Fig. 8.6, which further illustrates the variable field width and the precision specifications.

```
Program
    main()
    {
      int c, d;
      char string[] = "CProgramming";
      printf("\n\n");
      printf("------------\n");
      for( c = 0 ; c <= 11 ; c++ )
      {
        d = c + 1;
        printf("|%-12.*s|\n", d, string);
      }
      printf("|------------|\n");
      for( c = 11 ; c >= 0 ; c-- )
      {
```

```
        d = c + 1;
        printf("|%-12.*s|\n", d, string);
    }
    printf("-------------\n");
}
```

Output

```
C
CP
CPr
CPro
CProg
CProgr
CProgra
CProgram
CProgramm
CProgrammi
CProgrammin
CProgramming
CProgramming
CProgrammin
CProgrammi
CProgramm
CProgram
CProgra
CProgr
CProg
CPro
CPr
CP
C
```

Fig. 8.5 *Illustration of variable field specifications by printing sequences of characters*

```
       C
      CP
     CPr
    CPro
   CProg
  CProgr
 CProgra
CProgram
```

```
C|
CP|
CPr|
CPro|
CProg|
CProgr|
CProgra|
CProgram|
```

```
C|
 C|
  C|
   C|
    C|
     C|
      C|
       C|
```

```
      CProgramm
      CProgrammi
     CProgrammin
    CProgramming
    CProgramming
     CProgrammin
      CProgrammi
       CProgramm
        CProgram
         CProgra
          CProgr
           CProg
            CPro
             CPr
              CP
               C
```

(a) %12.*s

```
CProgramm|
CProgrammi|
CProgrammin|
CProgramming|
CProgramming|
CProgrammin|
CProgrammi|
CProgramm|
CProgram|
CProgra|
CProgr|
CProg|
CPro|
CPr|
CP|
C|
```

(b) %.*s

```
                C|
                C|
                 C|
                 C|
                 C|
                C|
               C|
              C|
             C|
            C|
           C|
          C|
         C|
        C|
       C|
      C|
```

(c) %*.1s

Fig. 8.6 *Further illustrations of variable specifications*

Using putchar and puts Functions

Like **getchar**, C supports another character handling function **putchar** to output the va ues of character variables. It takes the following form:

```
char ch = 'A';
putchar (ch);
```

The function **putchar** requires one parameter. This statement is equivalent to:

```
printf("%c", ch);
```

We have used **putchar** function in Chapter 4 to write characters to the screen. We can u this function repeatedly to output a string of characters stored in an array using a loo Example:

```
char name[6] = "PARIS"
for (i=0, i<5; i++)
    putchar(name[i];
putchar('\n');
```

Another and more convenient way of printing string values is to use the function **puts** d clared in the header file *<stdio.h>*. This is a one parameter function and invoked as und

```
puts ( str );
```

where **str** is a string variable containing a string value. This prints the value of the stri variable **str** and then moves the cursor to the beginning of the next line on the screen. F example, the program segment

```
char line [80];
gets (line);
puts (line);
```

reads a line of text from the keyboard and displays it on the screen. Note that the syntax is very simple compared to using the **scanf** and **printf** statements.

8.5 ARITHMETIC OPERATIONS ON CHARACTERS

C allows us to manipulate characters the same way we do with numbers. Whenever a character constant or character variable is used in an expression, it is automatically converted into an integer value by the system. The integer value depends on the local character set of the system.

To write a character in its integer representation, we may write it as an integer. For example, if the machine uses the ASCII representation, then,

```
x = 'a';
printf("%d\n",x);
```

will display the number 97 on the screen.

It is also possible to perform arithmetic operations on the character constants and variables. For example,

```
x = 'z'-1;
```

is a valid statement. In ASCII, the value of 'z' is 122 and therefore, the statement will assign the value 121 to the variable **x.**

We may also use character constants in relational expressions. For example, the expression

```
ch >= 'A' && ch <= 'Z'
```

would test whether the character contained in the variable **ch** is an upper-case letter.

We can convert a character digit to its equivalent integer value using the following relationship:

```
x = character - '0';
```

where **x** is defined as an integer variable and **character** contains the character digit. For example, let us assume that the **character** contains the digit '7'.

Then,

x = ASCII value of '7' – ASCII value of '0'
= 55 – 48
= 7

The C library supports a function that converts a string of digits into their integer values. The function takes the form

x = atoi(string);

x is an integer variable and **string** is a character array containing a string of digits. Consider the following segment of a program:

```
number = "1988";
year = atoi(number);
```

number is a string variable which is assigned the string constant "1988". The function **atoi** converts the string "1988" (contained in **number**) to its numeric equivalent 1988 and assigns it to the integer variable **year**. String conversion functions are stored in the header file <std.lib.h>.

Example 8.6 Write a program which would print the alphabet set a to z and A to Z in decimal and character form.

The program is shown in Fig. 8.7. In ASCII character set, the decimal numbers 65 to 90 represent upper case alphabets and 97 to 122 represent lower case alphabets. The values from 91 to 96 are excluded using an **if** statement in the **for** loop.

Program

```
main()
{
   char c;
   printf("\n\n");
   for( c = 65 ; c <= 122 ; c = c + 1 )
   {
     if( c > 90 && c < 97 )
        continue;
     printf("|%4d - %c ", c, c);
   }
   printf("|\n");
}
```

Output

```
| 65 - A | 66 - B | 67 - C | 68 - D | 69 - E | 70 - F
| 71 - G | 72 - H | 73 - I | 74 - J | 75 - K | 76 - L
| 77 - M|  78 - N|  79 - O|  80 - P|  81 - Q|  82 - R
| 83 - S|  84 - T|  85 - U|  86 - V|  87 - W|  88 - X
| 89 - Y|  90 - Z|  97 - a|  98 - b|  99 - c| 100 - d
|101 - e| 102 - f| 103 - g| 104 - h| 105 - i| 106 - j
|107 - k| 108 - l| 109 - m| 110 - n| 111 - o| 112 - p
|113 - q| 114 - r| 115 - s| 116 - t| 117 - u| 118 - v
|119 - w| 120 - x| 121 - y| 122 - z|
```

Fig. 8.7 *Printing of the alphabet set in decimal and character form*

8.6 PUTTING STRINGS TOGETHER

Just as we cannot assign one string to another directly, we cannot join two strings together by the simple arithmetic addition. That is, the statements such as

```
string3 = string1 + string2;
string2 = string1 + "hello";
```

are not valid. The characters from **string1** and **string2** should be copied into the **string3** one after the other. The size of the array **string3** should be large enough to hold the total characters.

The process of combining two strings together is called *concatenation*. Example 8.7 illustrates the concatenation of three strings.

Example 8.7 The names of employees of an organization are stored in three arrays, namely **first_name**, **second_name**, and **last_name**. Write a program to concatenate the three parts into one string to be called **name**.

The program is given in Fig. 8.8. Three **for** loops are used to copy the three strings. In the first loop, the characters contained in the **first_name** are copied into the variable **name** until the *null* character is reached. The *null* character is not copied; instead it is replaced by a *space* by the assignment statement

```
name[i] = ' ' ;
```

Similarly, the **second_name** is copied into **name**, starting from the column just after the space created by the above statement. This is achieved by the assignment statement

```
name[i+j+1] = second_name[j];
```

If **first_name** contains 4 characters, then the value of **i** at this point will be 4 and therefore the first character from **second_name** will be placed in the *fifth cell* of **name.** Note that we have stored a space in the *fourth cell*.

In the same way, the statement

```
name[i+j+k+2] = last_name[k];
```

is used to copy the characters from **last_name** into the proper locations of **name.**

At the end, we place a null character to terminate the concatenated string **name**. In this example, it is important to note the use of the expressions **i+j+1** and **i+j+k+2.**

```
Program
    main()
    {
      int i, j, k ;
      char first_name[10] = {"VISWANATH"} ;
      char second_name[10] = {"PRATAP"} ;
      char last_name[10] = {"SINGH"} ;
      char name[30] ;
    /* Copy first_name into name */
      for( i = 0 ; first_name[i] != '\0' ; i++ )
        name[i] = first_name[i] ;
    /* End first_name with a space */
      name[i] = ' ' ;
    /* Copy second_name into name */
      for( j = 0 ; second_name[j] != '\0' ; j++ )
        name[i+j+1] = second_name[j] ;
    /* End second_name with a space */
```

```
    name[i+j+1] = ' ' ;
  /* Copy last_name into name */
    for( k = 0 ; last_name[k] != '\0'; k++ )
      name[i+j+k+2] = last_name[k] ;
  /* End name with a null character */
    name[i+j+k+2] = '\0' ;
    printf("\n\n") ;
    printf("%s\n", name) ;
  }
```

Output

```
VISWANATH PRATAP SINGH
```

Fig. 8.8 *Concatenation of strings*

8.7 COMPARISON OF TWO STRINGS

Once again, C does not permit the comparison of two strings directly. That is, the statement such as

```
if(name1 == name2)
if(name == "ABC")
```

are not permitted. It is therefore necessary to compare the two strings to be tested, character by character. The comparison is done until there is a mismatch or one of the strings terminates into a null character, whichever occurs first. The following segment of a program illustrates this.

```
i=0;
while(str1[i] == str2[i] && str1[i] != '\0'
      && str2[i] != '\0')
  i = i+1;
if (str1[i] == '\0' && str2[i] == '\0')
    printf("strings are equal\n");
  else
    printf("strings are not equal\n");
```

8.8 STRING-HANDLING FUNCTIONS

Fortunately, the C library supports a large number of string-handling functions that can be used to carry out many of the string manipulations discussed so far. Following are the most commonly used string-handling functions.

Function	Action
strcat()	concatenates two strings
strcmp()	compares two strings
strcpy()	copies one string over another
strlen()	finds the length of a string

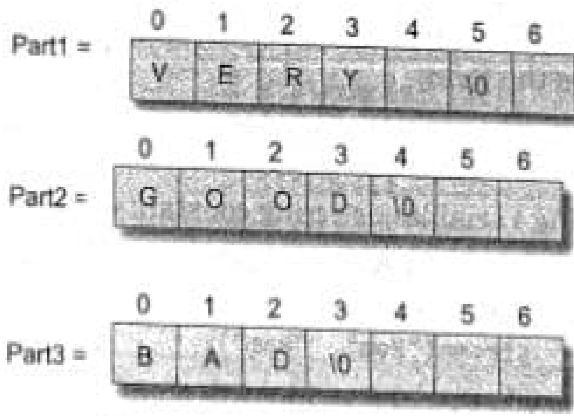
We shall discuss briefly how each of these functions can be used in the processing of strings.

strcat() Function

The **strcat** function joins two strings together. It takes the following form:

```
strcat(string1, string2);
```

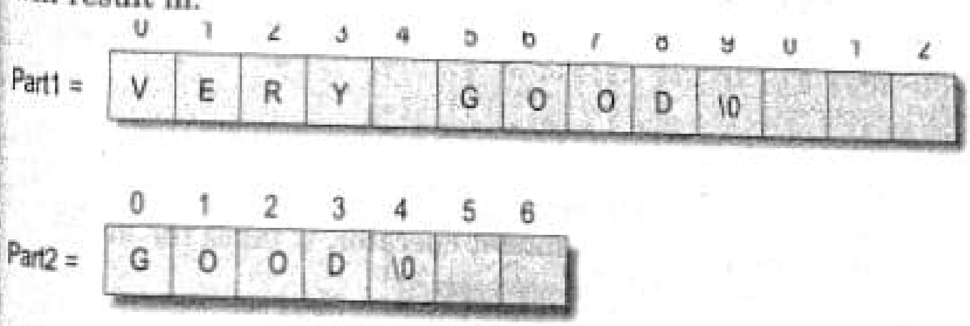
string1 and **string2** are character arrays. When the function **strcat** is executed, **string2** is appended to **string1**. It does so by removing the null character at the end of **string1** and placing **string2** from there. The string at **string2** remains unchanged. For example, consider the following three strings:



Execution of the statement

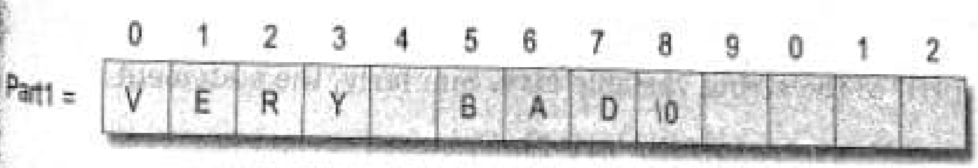
```
strcat(part1, part2);
```

will result in:



while the statement

will result in:



	0	1	2	3	4	5	6
Part3 =	B	A	D	\0			

We must make sure that the size of **string1** (to which **string2** is appended) is large enough to accommodate the final string.

strcat function may also append a string constant to a string variable. The following is valid:

```
strcat(part1,"GOOD");
```

C permits nesting of **strcat** functions. For example, the statement

```
strcat(strcat(string1,string2), string3);
```

is allowed and concatenates all the three strings together. The resultant string is stored in **string1**.

strcmp() Function

The **strcmp** function compares two strings identified by the arguments and has a value 0 if they are equal. If they are not, it has the numeric difference between the first nonmatching characters in the strings. It takes the form:

strcmp(string1, string2);

string1 and **string2** may be string variables or string constants. Examples are:

```
strcmp(name1, name2);
strcmp(name1, "John");
strcmp("Rom", "Ram");
```

Our major concern is to determine whether the strings are equal; if not, which is alphabetically above. The value of the mismatch is rarely important. For example, the statement

```
strcmp("their", "there");
```

will return a value of –9 which is the numeric difference between ASCII "i" and ASCII "r". That is, "i" minus "r" in ASCII code is –9. If the value is negative, **string1** is alphabetically above **string2**.

strcpy() Function

The **strcpy** function works almost like a string-assignment operator. It takes the form:

strcpy(string1, string2);

and assigns the contents of **string2** to **string1**. **string2** may be a character array variable or a string constant. For example, the statement

```
strcpy(city, "DELHI");
```

will assign the string "DELHI" to the string variable **city.** Similarly, the statement

```
strcpy(city1, city2);
```

will assign the contents of the string variable **city2** to the string variable **city1**. The size of the array **city1** should be large enough to receive the contents of **city2**.

strlen() Function

This function counts and returns the number of characters in a string. It takes the form

n = strlen(string);

Where **n** is an integer variable, which receives the value of the length of the **string**. The argument may be a string constant. The counting ends at the first null character.

Example 8.6 **s1**, **s2**, and **s3** are three string variables. Write a program to read two string constants into **s1** and **s2** and compare whether they are equal or not. If they are not, join them together. Then copy the contents of **s1** to the variable **s3**. At the end, the program should print the contents of all the three variables and their lengths.

The program is shown in Fig. 8.9. During the first run, the input strings are "New" and "York". These strings are compared by the statement

```
x = strcmp(s1, s2);
```

Since they are not equal, they are joined together and copied into **s3** using the statement

```
strcpy(s3, s1);
```

The program outputs all the three strings with their lengths.

During the second run, the two strings **s1** and **s2** are equal, and therefore, they are not joined together. In this case all the three strings contain the same string constant "London".

```
Program
    #include <string.h>
    main()
    { char s1[20], s2[20], s3[20];
      int x, l1, l2, l3;
      printf("\n\nEnter two string constants \n");
      printf("?");
      scanf("%s %s", s1, s2);
    /* comparing s1 and s2 */
      x = strcmp(s1, s2);
      if(x != 0)
      {    printf("\n\nStrings are not equal \n");
           strcat(s1, s2); /* joining s1 and s2 */
      }
      else
           printf("\n\nStrings are equal \n");
    /*copying s1 to s3
      strcpy(s3, s1);
    /*Finding length of strings */
```

strcpy() Function

The **strcpy** function works almost like a string-assignment operator. It takes the form:

```
strcpy(string1, string2);
```

and assigns the contents of **string2** to **string1**. **string2** may be a character array variable or a string constant. For example, the statement

```
strcpy(city, "DELHI");
```

will assign the string "DELHI" to the string variable **city**. Similarly, the statement

```
strcpy(city1, city2);
```



```
        l1 = strlen(s1);
        l2 = strlen(s2);
        l3 = strlen(s3);
    /* output */
        printf("\ns1 = %s\t length = %d characters\n", s1, l1);
        printf("s2 = %s\t length = %d characters\n", s2, l2);
        printf("s3 = %s\t length = %d characters\n", s3, l3);
    }
Output
    Enter two string constants
    ? New York
    Strings are not equal
    s1 = NewYork  length = 7 characters
    s2 = York     length = 4 characters
    s3 = NewYork  length = 7 characters

    Enter two string constants
    ? London London
    Strings are equal
    s1 = London length = 6 characters
    s2 = London length = 6 characters
    s3 = London length = 6 characters
```

Fig. 8.9 *Illustration of string handling functions*

Other String Functions

The header file **<string.h>** contains many more string manipulation functions. They might be useful in certain situations.

strncpy

In addition to the function **strcpy** that copies one string to another, we have another function **strncpy** that copies only the left-most n characters of the source string to the target string variable. This is a three-parameter function and is invoked as follows:

```
strncpy(s1, s2, 5);
```

This statement copies the first 5 characters of the source string **s2** into the target string **s1**. Since the first 5 characters may not include the terminating null character, we have to place it explicitly in the 6th position of **s2** as shown below:

```
s1[6] ='\0';
```

Now, the string **s1** contains a proper string.

strncmp

A variation of the function **strcmp** is the function **strncmp**. This function has three parameters as illustrated in the function call below:

```
strncmp (s1, s2, n);
```

this compares the left-most n characters of **s1** to **s2** and returns.

(a) 0 if they are equal;
(b) negative number, if s1 sub-string is less than s2; and
(c) positive number, otherwise.

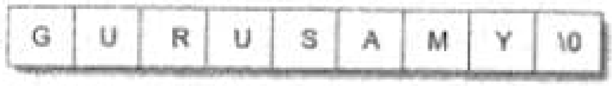
strncat

This is another concatenation function that takes three parameters as shown below:

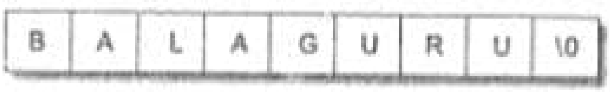
```
strncat (s1, s2, n);
```

This call will concatenate the left-most n characters of **s2** to the end of **s1.** Example:

S1 :


S2 :


After **strncat** (s1, s2, 4); execution:

S1


strstr

It is a two-parameter function that can be used to locate a sub-string in a string. This takes the forms:

```
strstr (s1, s2);
strstr (s1, "ABC");
```

The function **strstr** searches the string **s1** to see whether the string **s2** is contained in **s1**. If yes, the function returns the position of the first occurrence of the sub-string. Otherwise, it returns a NULL pointer. Example.

```
if (strstr (s1, s2) == NULL)
   printf("substring is not found");
else
   printf("s2 is a substring of s1");
```

We also have functions to determine the existence of a character in a string. The function call

```
strchr(s1, 'm');
```

will locate the first occurrence of the character 'm' and the call

```
strrchr(s1, 'm');
```

will locate the last occurrence of the character 'm' in the string **s1**.

Warnings

- When allocating space for a string during declaration, remember to count the terminating null character.
- When creating an array to hold a copy of a string variable of unknown size, we can compute the size required using the expression

  strlen (stringname) + 1.
- When copying or concatenating one string to another, we must ensure that the target (destination) string has enough space to hold the incoming characters. Remember that no error message will be available even if this condition is not satisfied. The copying may overwrite the memory and the program may fail in an unpredictable way.
- When we use **strncpy** to copy a specific number of characters from a source string, we must ensure to append the null character to the target string, in case the number of characters is less than or equal to the source string.

8.9 TABLE OF STRINGS

We often use lists of character strings, such as a list of the names of students in a class, lis of the names of employees in an organization, list of places, etc. A list of names can be treat as a table of strings and a two-dimensional character array can be used to store the ent list. For example, a character array **student[30][15]** may be used to store a list of 30 names each of length not more than 15 characters. Shown below is a table of five cities:

C	h	a	n	d	i	g	a	r	h
M	a	d	r	a	s				
A	h	m	e	d	a	b	a	d	
H	y	d	e	r	a	b	a	d	
B	o	m	b	a	y				

This table can be conveniently stored in a character array city by using the following decl ration:

```
char city[ ] [ ]
        {
          "Chandigarh",
          "Madras",
          "Ahmedabad",
          "Hyderabad",
          "Bombay"
        } ;
```