```
                |
                |************************************
Group-3         |************************************(35)
                |************************************
                |
                |********************
Group-4         |********************(20)
                |********************
                |
                |***********
Group-5         |***********(11)
                |***********
                |
```

Fig. 6.13 *Program to draw a* ***histogram***

3. Minimum Cost

Problem: The cost of operation of a unit consists of two components C1 and C2 which can be expressed as functions of a parameter p as follows:

$$C1 = 30 - 8p$$
$$C2 = 10 + p^2$$

The parameter p ranges from 0 to 10. Determine the value of p with an accuracy of + 0.1 where the cost of operation would be minimum.

Problem Analysis:

$$\text{Total cost} = C_1 + C_2 = 40 - 8p + p^2$$

The cost is 40 when p = 0, and 33 when p = 1 and 60 when p = 10. The cost, therefore, decreases first and then increases. The program in Fig. 6.14 evaluates the cost at successive intervals of p (in steps of 0.1) and stops when the cost begins to increase. The program employs **break** and **continue** statements to exit the loop.

```
Program
    main()
    {
      float p, cost, p1, cost1;
      for (p = 0; p <= 10; p = p + 0.1)
      {
           cost = 40 - 8 * p + p * p;
           if(p == 0)
           {
             cost1 = cost;
             continue;
           }
           if (cost >= cost1)
```

```
                    break;
                costl = cost;
                p1 = p;
            }
            p = (p + p1)/2.0;
            cost = 40 - 8 * p + p * p;
            printf("\nMINIMUM COST = %.2f AT p = %.1f\n",
                    cost, p);
        }
Output
    MINIMUM COST = 24.00 AT p = 4.0
```

Fig. 6.14 *Program of minimum cost problem*

4. Plotting of Two Functions

Problem: We have two functions of the type

$$y1 = \exp(-ax)$$

$$y2 = \exp(-ax^2/2)$$

Plot the graphs of these functions for x varying from 0 to 5.0.

Problem Analysis: Initially when x = 0, y1 = y2 =1 and the graphs start from the same point. The curves cross when they are again equal at x = 2.0. The program should have appropriate branch statements to print the graph points at the following three conditions:

1. y1 > y2
2. y1 < y2
3. y1 = y2

The functions y1 and y2 are normalized and converted to integers as follows:

$$y1 = 50 \exp(-ax) + 0.5$$

$$y2 = 50 \exp(-ax^2/2) + 0.5$$

The program in Fig. 6.15 plots these two functions simultaneously. (0 for y1, * for y2, and # for the common point).

```
Program
    #include <math.h>
    main()
    {
      int i;
      float a, x, y1, y2;
      a = 0.4;
      printf("                Y ----->                \n");
      printf(" 0 ------------------------------\n");
      for ( x = 0; x < 5; x = x+0.25)
      { /* BEGINNING OF FOR LOOP */
```

```
    /*......Evaluation of functions .......*/
      y1 = (int) ( 50 * exp( -a * x ) + 0.5 );
      y2 = (int) ( 50 * exp( -a * x * x/2 ) + 0.5 );
    /*......Plotting when y1 = y2.........*/
      if ( y1 == y2)
      {
        if ( x == 2.5)
           printf(" X   |");
        else
           printf("|");
        for ( i = 1; i <= y1 - 1; ++i)
           printf(" ");
        printf("#\n");
        continue;
      }
    /*..... Plotting when y1 > y2 .....*/
      if ( y1 > y2)
      {
        if ( x == 2.5 )
          printf(" X |");
        else
          printf("   |");
        for ( i = 1; i <= y2 -1 ; ++i)
          printf(" ");
        printf("*");
        for ( i = 1; i <= (y1 - y2 - 1); ++i)
          printf("-");
        printf("0\n");
        continue;
      }
    /*........ Plotting when y2 > y1.........*/
      if ( x == 2.5)
        printf(" X |");
      else
        printf("   |");
      for ( i = 1 ; i <= (y1 - 1); ++i )
        printf(" ");
      printf("0");
      for ( i = 1; i <= ( y2 - y1 - 1 ); ++i)
        printf("-");
      printf("*\n");
    } /*.......END OF FOR LOOP........*/
      printf("   |\n");
}
```

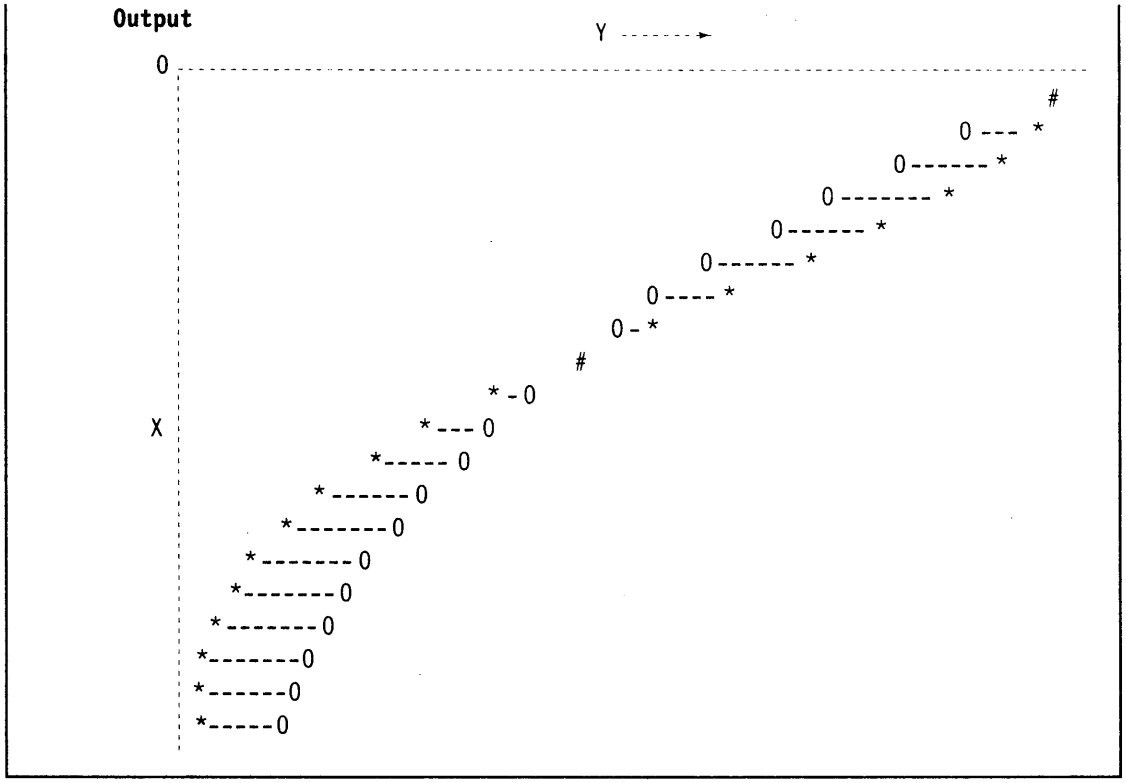


Fig. 6.15 *Plotting of two functions*

REVIEW QUESTIONS

6.1 State whether the following statements are *true* or *false*.

(a) The **do...while** statement first executes the loop body and then evaluate the loop control expression.

(b) In a pretest loop, if the body is executed n times, the test expression is executed n + 1 times.

(c) The number of times a control variable is updated always equals the number of loop iterations.

(d) Both the pretest loops include initialization within the statement.

(e) In a **for** loop expression, the starting value of the control variable must be less than its ending value.

(f) The initialization, test condition and increment parts may be missing in a **for** statement.

(g) **while** loops can be used to replace **for** loops without any change in the body of the loop.

(h) An exit-controlled loop is executed a minimum of one time.

(i) The use of **continue** statement is considered as unstructured programming.

(j) The three loop expressions used in a **for** loop header must be separated by commas.

6.2 Fill in the blanks in the following statements.
(a) In an exit-controlled loop, if the body is executed n times, the test condition is evaluated _____ times.
(b) The _____ statement is used to skip a part of the statements in a loop.
(c) A **for** loop with the no test condition is known as _____ loop.
(d) The sentinel-controlled loop is also known as _____ loop.
(e) In a counter-controlled loop, variable known as _____ is used to count the loop operations.

6.3 Can we change the value of the control variable in **for** statements? If yes, explain its consequences.

6.4 What is a null statement? Explain a typical use of it.

6.5 Use of **goto** should be avoided. Explain a typical example where we find the application of **goto** becomes necessary.

6.6 How would you decide the use of one of the three loops in C for a given problem?

6.7 How can we use **for** loops when the number of iterations are not known?

6.8 Explain the operation of each of the following **for** loops.

(a)
```
for ( n = 1; n != 10; n += 2)
    sum = sum + n;
```

(b)
```
for (n = 5; n <= m; n -=1)
    sum = sum + n;
```

(c)
```
for (n = 1; n <= 5;)
    sum = sum + n;
```

(d)
```
for ( n = 1; ; n = n + 1)
    sum = sum + n;
```

(e)
```
for (n = 1; n < 5; n ++)
    n = n -1
```

6.9 What would be the output of each of the following code segments?

(a)
```
count = 5;
while (count -- > 0)
printf(count);
```

(b)
```
count = 5;
while ( -- count > 0)
printf(count);
```

(c)
```
count = 5;
do printf(count);
while (count > 0);
```

(d)
```
for (m = 10; m > 7, m -=2)
printf(m);
```

6.10 Compare, in terms of their functions, the following pairs of statements:
(a) while and do...while
(b) while and for
(c) break and goto
(d) break and continue
(e) continue and goto

6.11 Analyse each of the program segments that follow and determine how many times the body of each loop will be executed.

(a)
```
x = 5;
y = 50;
while ( x <= y)
{
    x = y/x;
    ---
    ---
}
```

(b)
```
m = 1;
do
{
    ---
    ---
    m = m+2;
}
while (m < 10);
```

(c)
```
int i;
for (i = 0; i <= 5; i = i+2/3)
{
    ---
    ---
    ---
}
```

(d)
```
int m = 10;
int n = 7;
while ( m % n >= 0)
{
    ---
    m = m + 1;
    n = n + 2;
    ---
}
```

6.12 Find errors, if any, in each of the following looping segments. Assume that all the variables have been declared and assigned values.

(a)
```
while (count != 10);
{
    count = 1;
    sum = sum + x;
    count = count + 1;
}
```

(b)
```
name = 0;
do { name = name + 1;
```

```
    printf("My name is John\n");}
    while (name = 1)
(c) do;
    total = total + value;
    scanf("%f", &value);
    while (value != 999);
(d) for (x = 1, x > 10; x = x + 1)
    {
        ---
        ---
        ---
    }
(e) m = 1;
    n = 0;
    for ( ; m+n < 10; ++n);
    printf("Hello\n");
    m = m+10
(f) for (p = 10; p > 0;)
    p = p - 1;
    printf("%f", p);
```

6.13 Write a **for** statement to print each of the following sequences of integers:

(a) 1, 2, 4, 8, 16, 32
(b) 1, 3, 9, 27, 81, 243
(c) – 4, –2, 0, 2, 4
(d) –10, –12, –14, –18, –26, – 42

6.14 Change the following **for** loops to **while** loops:

```
(a) for (m = 1; m < 10; m = m + 1)
    printf(m);
(b) for ( ; scanf("%d", & m) != -1;)
    printf(m);
```

6.15 Change the **for** loops in Exercise 6.14 to **do** loops.

PROGRAMMING EXERCISES

6.1 Given a number, write a program using **while** loop to reverse the digits of the number. For example, the number

12345

should be written as

54321

(**Hint:** Use modulus operator to extract the last digit and the integer division by 10 to get the n–1 digit number from the n digit number.)

6.2 The factorial of an integer m is the product of consecutive integers from 1 to m. That is,

factorial m = m! = m x (m–1) x ..... x 1.

Write a program that computes and prints a table of factorials for any given m.

6.3 Write a program to compute the sum of the digits of a given integer number.

6.4 The numbers in the sequence

1 1 2 3 5 8 13 21 .....

are called Fibonacci numbers. Write a program using a **do....while** loop to calculate and print the first m Fibonacci numbers.

(**Hint:** After the first two numbers in the series, each number is the sum of the two preceding numbers.)

6.5 Rewrite the program of the Example 6.1 using the **for** statement.

6.6 Write a program to evaluate the following investment equation

$V = P(1+r)^n$

and print the tables which would give the value of V for various combination of the following values of P, r, and n.

P : 1000, 2000, 3000,....., 10,000

r : 0.10, 0.11, 0.12, ..., 0.20

n : 1, 2, 3, ..., 10

(**Hint:** P is the principal amount and V is the value of money at the end of n years. This equation can be recursively written as

$V = P(1+r)$

$P = V$

That is, the value of money at the end of first year becomes the principal amount for the next year and so on.)

6.7 Write programs to print the following outputs using **for** loops.

(a)
```
1
2 2
3 3 3
4 4 4 4
5 5 5 5 5
```

(b)
```
* * * * *
 * * * *
  * * *
   * *
    *
```

6.8 Write a program to read the age of 100 persons and count the number of persons in the age group 50 to 60. Use **for** and **continue** statements.

6.9 Rewrite the program of case study 6.4 (plotting of two curves) using **else...if** constructs instead of **continue** statements.

6.10 Write a program to print a table of values of the function

y = exp (-x)

for x varying from 0.0 to 10.0 in steps of 0.10. The table should appear as follows:

Table for Y = EXP(–X)

x	*0.1*	*0.2*	*0.3*	.....	*0.9*
0.0					
1.0					
2.0					
3.0					
.					

(Contd.)

x	0.1	0.2	0.3	.....	0.9
.					
.					
9.0					

6.11 Write a program that will read a positive integer and determine and print its binary equivalent.
(**Hint:** The bits of the binary representation of an integer can be generated by repeatedly dividing the number and the successive quotients by 2 and saving the remainder, which is either 0 or 1, after each division.)

6.12 Write a program using **for** and **if** statement to display the capital letter S in a grid of 15 rows and 18 columns as shown below.

```
* * * * * * * * * * * * * * * * * *
* * - - - - - - - - - - - - - - - - * *
* * * * * * * * * - - - - - - - - - * *
* * * *
* * * *
* * * *
* * * * * - - - - - - - - - - - - - * * * *
- - - - - - - - - - - - - - - - - - * * * *
- - - - - - - - - - - - - - - - - - * * * *
                                   * * * *
                                   * * * *
                                   * * * *
* * * * - - - - - - - - - - - - - - * * * *
* * * - - - - - - - - - - - - - - - * * * *
* * - - - - - - - - - - - - - - - - * * * *
```

6.13 Write a program to compute the value of Euler's number e, that is used as the base of natural logarithms. Use the following formula.

$$e = 1 + 1/1! + 1/2! + 1/3! + \ldots\ldots + 1/n!$$

Use a suitable loop construct. The loop must terminate when the difference between two successive values of e is less than 0.00001.

6.14 Write programs to evaluate the following functions to 0.0001% accuracy.

(a) $\sin x = x - x^3/3! + x^5/5! - x^7/7! + \ldots\ldots$

(b) $\cos x = 1 - x^2/2! + x^4/4! - x^6/6! + \ldots\ldots$

(c) $SUM = 1 + (1/2)^2 + (1/3)^3 + (1/4)^4 + \ldots\ldots$

6.15 The present value (popularly known as book value) of an item is given by the relationship.

$$P = c\,(1-d)^n$$

where
c = original cost
d = rate of depreciation (per year)
n = number of years
p = present value after y years.

If P is considered the scrap value at the end of useful life of the item, write a program to compute the useful life in years given the original cost, depreciation rate, and the scrap value. The program should request the user to input the data interactively.

Chapter 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*.

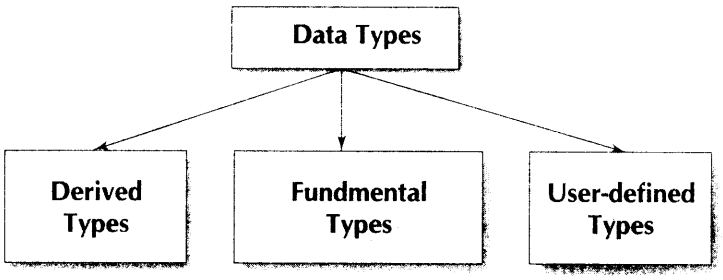
The ability to use a single name to represent a collection of items and to refer to an item by specifying the item number enables us to develop concise and efficient programs. For example, we can use a loop construct discussed earlier with the subscript as the control variable to read the entire array,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



- 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. 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***

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.

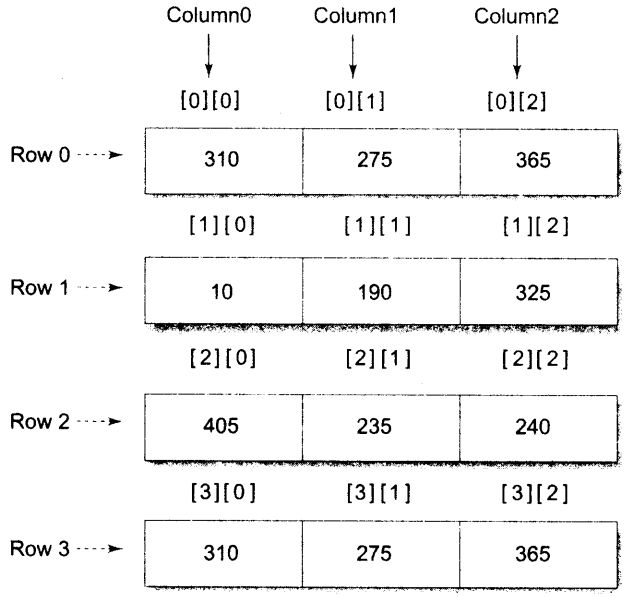


Fig. 7.3 *Representation of a* ***two-dimensional*** *array in memory*

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 nth 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]

$= \sum_{i=0}^{3}$ girl_total[i]

$= \sum_{j=0}^{2}$ 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
```

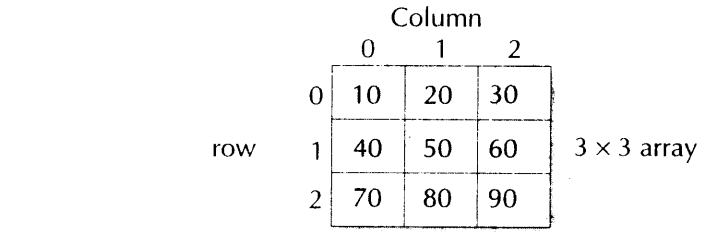
POPULARITY TABLE

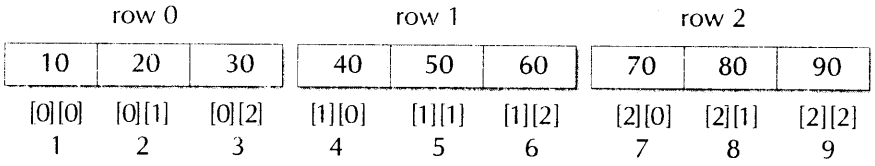
City	Ambassador	Fiat	Dolphin	Maruti
Bombay	2	1	3	2
Calcutta	4	5	1	0
Delhi	2	1	3	2
Madras	4	1	1	4

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.





Memory Layout

For a multidimensional 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
- 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 multidimensional arrays, it is an error to omit the array size for any dimension other than the first.

CASE STUDIES

7.1 Median of a List of Numbers

When all the items in a list are arranged in 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.