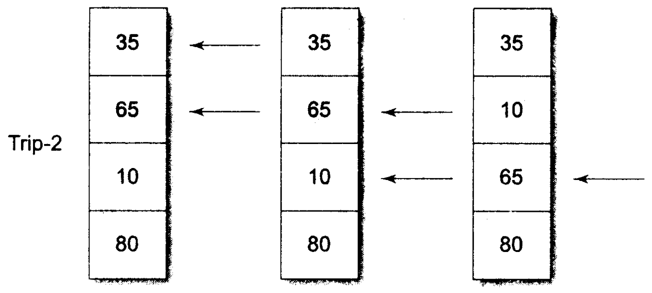
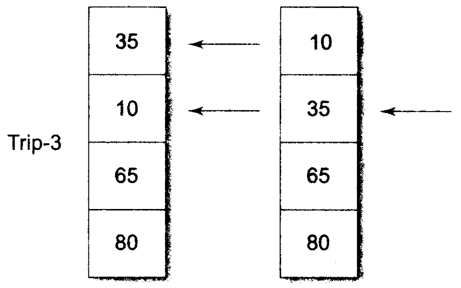




During the first trip, three pairs of items are compared and interchanged whenever needed. It should be noted that the number 80, the largest among the items, has been moved to the bottom at the end of the first trip. This means that the element 80 (the last item in the new list) need not be considered 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);
```

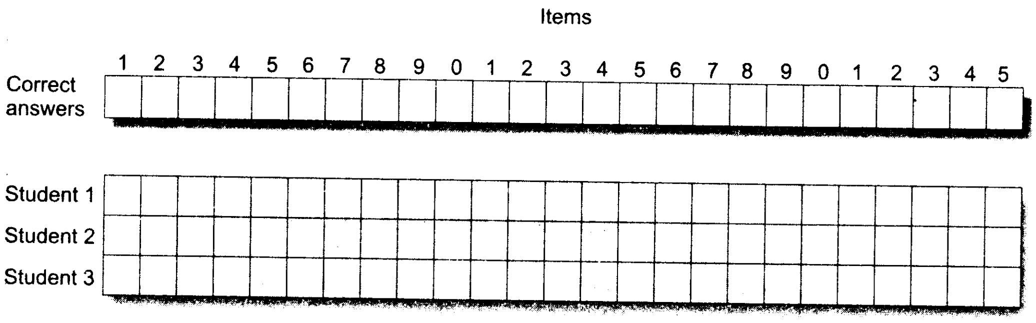
```
        printf("Mean : %f\n", mean);
        printf("Standard deviation : %f\n", stddeviation);
    }
Output
    Input values: input -1 to end
    65 9 27 78 12 20 33 49 -1

    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 <= STUDENTS ; 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 manufactured in ith week and Sij the number of jth product sold in ith week. We may also represent the 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?

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

PROGRAMMING EXERCISES

7.1 Write a program for fitting a straight line through a set of points (x_i,y_i), $i = 1,....,n$.
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:

	City			
Day	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.

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

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

The product of **A** and **B** is a third matrix C of size nxn where each 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.

Chapter 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.

8.2 DECLARING AND 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 string reading.

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

with the line of text

NEW YORK

will assign the string "NEW" to **adrl** 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 wordl[40], word2[40], word3[40], word4[40];

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

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

Output

```
Enter text :
Oxford Road, London M17ED

wordl = 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";
stringl = string2;
```

are not valid. If we really want to copy the characters in **string2** into **stringl**, 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 **stringl**. The loop is terminated when the *null* character is reached. Note that we are again assigning a null character to the **stringl.**

```
Program
    main( )
    {
        char stringl[80], string2[80];
        int i;
        printf("Enter a string \n");
        printf("?");
        scanf("%s", string2);
        for( i=0 ; string2[i] != '\0'; i++)
            stringl[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 screen. The format %s can be used to display an array of characters that is terminated by the null 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 specification

%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 **%*.1s** 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
   CProgramm
  CProgrammi
 CProgrammin
CProgramming
CProgramming
 CProgrammin
  CProgrammi
   CProgramm
    CProgram
     CProgra
      CProgr
       CProg
        CPro
         CPr
          CP
           C
```

(a) %12.*s

```
C|
CP|
CPr|
CPro|
CProg|
CProgr|
CProgra|
CProgram|
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|
      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 values 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 use this function repeatedly to output a string of characters stored in an array using a loop: 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** declared in the header file <*stdio.h*>. This is a one parameter function and invoked as under:

```
puts ( str );
```

where **str** is a string variable containing a string value. This prints the value of the string variable **str** and then moves the cursor to the beginning of the next line on the screen. For 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 uppercase alphabets and 97 to 122 represent lowercase 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 statements 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

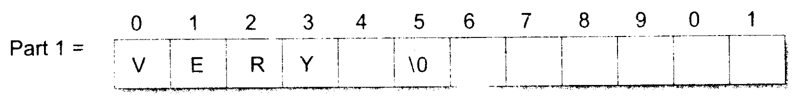
We shall discuss briefly how each of these functions can be used in the processing of strings.

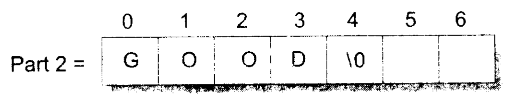
strcat() Function

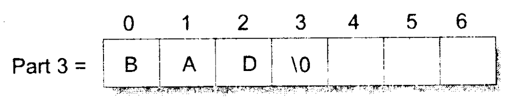
The **strcat** function joins two strings together. It takes the following form:

```
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:



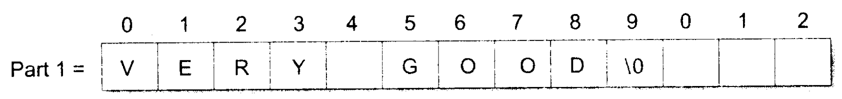


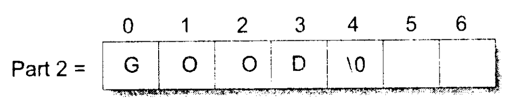


Execution of the statement

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

will result in:





while the statement

strcat(part 1, part 3);

will result in:

	0	1	2	3	4	5	6	7	8	9	0	1	2
Part 1 =	V	E	R	Y		B	A	D	\0				

	0	1	2	3	4	5	6
Part 3 =	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 8.8 **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 */
          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.

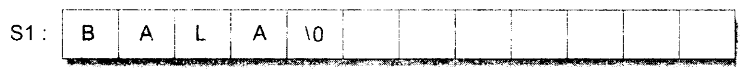
(a) 0 if they are equal,
(b) negative number, if s1 sub-string is less than s2, and
(c) positive number, otherwise.

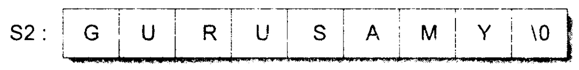
strncat

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

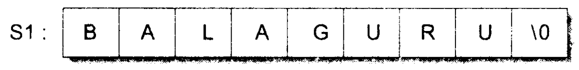
```
strncat (s1, s2, n);
```

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





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



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, list of the names of employees in an organization, list of places, etc. A list of names can be treated as a table of strings and a two-dimensional character array can be used to store the entire 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 declaration:

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

To access the name of the ith city in the list, we write

city[i-1]

and therefore **city[0]** denotes "Chandigarh", **city[1]** denotes "Madras" and so on. This shows that once an array is declared as two-dimensional, it can be used like a one-dimensional array in further manipulations. That is, the table can be treated as a column of strings.

Example 8.9 Write a program that would sort a list of names in alphabetical order.

A program to sort the list of strings in alphabetical order is given in Fig. 8.10. It employs the method of bubble sorting described in Case Study 1 in the previous chapter.

Program

```
#define ITEMS 5
#define MAXCHAR 20
main( )
{
  char string[ITEMS][MAXCHAR], dummy[MAXCHAR];
  int i = 0, j = 0;
  /* Reading the list */
  printf ("Enter names of %d items \n ",ITEMS);
  while (i < ITEMS)
    scanf ("%s", string[i++]);
  /* Sorting begins */
  for (i=1; i < ITEMS; i++) /* Outer loop begins */
  {
    for (j=1; j <= ITEMS-i ; j++) /*Inner loop begins*/
    {
      if (strcmp (string[j-1], string[j]) > 0)
      { /* Exchange of contents */
        strcpy (dummy, string[j-1]);
        strcpy (string[j-1], string[j]);
        strcpy (string[j], dummy );
      }
    } /* Inner loop ends */
  } /* Outer loop ends */
  /* Sorting completed */
  printf ("\nAlphabetical list \n\n");
  for (i=0; i < ITEMS ; i++)
    printf ("%s", string[i]);
}
```

Output

```
Enter names of 5 items
London Manchester Delhi Paris Moscow

Alphabetical list

Delhi
London
Manchester
Moscow
Paris
```

Fig. 8.10 *Sorting of strings in alphabetical order*

Note that a two-dimensional array is used to store the list of strings. Each string is read using a **scanf** function with **%s** format. Remember, if any string contains a white space, then the part of the string after the white space will be treated as another item in the list by the **scanf.** In such cases, we should read the entire line as a string using a suitable algorithm. For example, we can use **gets** function to read a line of text containing a series of words. We may also use **puts** function in place of **scanf** for output.

8.10 OTHER FEATURES OF STRINGS

Other aspects of strings we have not discussed in this chapter include:

- Manipulating strings using pointers
- Using string as function parameters
- Declaring and defining strings as members of structures.

These topics will be dealt with later when we discuss functions, structures and pointers.

Just Remember

- Character constants are enclosed in single quotes and string constants are enclosed in double quotes.
- Allocate sufficient space in a character array to hold the null character at the end.
- Avoid processing single characters as strings.
- Using the address operator **&** with a **string** variable in the **scanf** function call is an error.
- It is a compile time error to assign a string to a character variable.
- Using a string variable name on the left of the assignment operator is illegal.
- When accessing individual characters in a string variable, it is logical error to access outside the array bounds.
- Strings cannot be manipulated with operators. Use string functions.
- Do not use string functions on an array **char** type that is not terminated with the null character.
- Do not forget to append the null character to the target string when the number of characters copied is less than or equal to the source string.
- Be aware the return values when using the functions **strcmp** and **strncmp** for comparing strings.
- When using string functions for copying and concatenating strings, make sure that the target string has enough space to store the resulting string. Otherwise memory overwriting may occur.
- The header file <stdio.h> is required when using standard I/O functions.
- The header file <ctype.h> is required when using character handling functions.
- The header file <stdlib.h> is required when using general utility functions
- The header file <string.h> is required when using string manipulation functions.

CASE STUDIES

1. Counting Words in a Text

One of the practical applications of string manipulations is counting the words in a text. We assume that a word is a sequence of any characters, except escape characters and blanks, and that two words are separated by one blank character. The algorithm for counting words is as follows:

1. Read a line of text.
2. Beginning from the first character in the line, look for a blank. If a blank is found, increment words by 1.
3. Continue steps 1 and 2 until the last line is completed.

The implementation of this algorithm is shown in Fig. 8.11. The first **while** loop will be executed once for each line of text. The end of text is indicated by pressing the 'Return' key an extra time after the entire text has been entered. The extra 'Return' key causes a newline character as input to the last line and as a result, the last line contains only the null character.
The program checks for this special line using the test

    if (line[0] == '\0')

and if the first (and only the first) character in the line is a null character, then counting is terminated. Note the difference between a null character and a blank character.

Program

```
#include <stdio.h>
main()
{
  char line[81], ctr;
  int i,c,
      end = 0,
      characters = 0,
      words = 0,
      lines = 0;
  printf("KEY IN THE TEXT.\n");
  printf("GIVE ONE SPACE AFTER EACH WORD.\n");
  printf("WHEN COMPLETED, PRESS 'RETURN'.\n\n");
  while( end == 0)
  {
    /* Reading a line of text */
    c = 0;
    while((ctr=getchar()) != '\n')
      line[c++] = ctr;
    line[c] = '\0';
    /* counting the words in a line */
    if(line[0] == '\0')
      break ;
    else
    {
      words++;
```

```
                for(i=0; line[i] != '\0';i++)
                    if(line[i] == ' ' || line[i] == '\t')
                      words++;
              }
              /* counting lines and characters */
              lines = lines +1;
              characters = characters + strlen(line);
            }
            printf ("\n");
            printf("Number of lines = %d\n", lines);
            printf("Number of words = %d\n", words);
            printf("Number of characters = %d\n", characters);
          }
Output
          KEY IN THE TEXT.
          GIVE ONE SPACE AFTER EACH WORD.
          WHEN COMPLETED, PRESS 'RETURN'.

          Admiration is a very short-lived passion.
          Admiration involves a glorious obliquity of vision.
          Always we like those who admire us but we do not
          like those whom we admire.
          Fools admire, but men of sense approve.

          Number of lines = 5
          Number of words = 36
          Number of characters = 205
```

Fig. 8.11 *Counting of characters, words and lines in a text*

The program also counts the number of lines read and the total number of characters in the text. Remember, the last line containing the null string is not counted.

After the first **while** loop is exited, the program prints the results of counting.

2. Processing of a Customer List

Telephone numbers of important customers are recorded as follows:

Full name	**Telephone number**
Joseph Louis Lagrange	869245
Jean Robert Argand	900823
Carl Freidrich Gauss	806788
-----	-----
-----	-----

It is desired to prepare a revised alphabetical list with surname (last name) first, followed by a comma and the initials of the first and middle names. For example,

Argand,J.R

We create a table of strings, each row representing the details of one person, such as first_name, middle_name, last_name, and telephone_number. The columns are interchanged as required and the list is sorted on the last_name. Figure 8.12 shows a program to achieve this.

Program

```
#define  CUSTOMERS   10

main( )
{
    char  first_name[20][10], second_name[20][10],
          surname[20][10], name[20][20],
          telephone[20][10], dummy[20];

   int   i,j;

    printf("Input names and telephone numbers \n");
    printf("?");
    for(i=0; i < CUSTOMERS ; i++)
    {
         scanf("%s %s %s %s", first_name[i],
               second_name[i], surname[i], telephone[i]);

        /* converting full name to surname with initials */

        strcpy(name[i], surname[i] );
        strcat(name[i], ",");
        dummy[0] = first_name[i][0];
        dummy[1] = '\0';
        strcat(name[i], dummy);
        strcat(name[i], ".");
        dummy[0] = second_name[i][0];
        dummy[1] = '\0';
        strcat(name[i], dummy);
    }
    /* Alphabetical ordering of surnames */

        for(i=1; i <= CUSTOMERS-1; i++)
           for(j=1; j <= CUSTOMERS-i; j++)
              if(strcmp (name[j-1], name[j]) > 0)
              {
              /* Swaping names */
                 strcpy(dummy, name[j-1]);
                 strcpy(name[j-1], name[j]);
                 strcpy(name[j], dummy);
```

```
                /* Swaping telephone numbers */
                   strcpy(dummy, telephone[j-1]);
                   strcpy(telephone[j-1],telephone[j]);
                   strcpy(telephone[j], dummy);
                }
         /* printing alphabetical list */
        printf("\nCUSTOMERS LIST IN ALPHABETICAL ORDER \n\n");
        for(i=0; i < CUSTOMERS ; i++)
           printf("   %-20s\t %-10s\n", name[i], telephone[i]);
    }
```

Output

```
Input names and telephone numbers
?Gottfried Wilhelm Leibniz 711518
Joseph Louis Lagrange 869245
Jean Robert Argand 900823
Carl Freidrich Gauss 806788
Simon Denis Poisson 853240
Friedrich Wilhelm Bessel 719731
Charles Francois Sturm 222031
George Gabriel Stokes 545454
Mohandas Karamchand Gandhi 362718
Josian Willard Gibbs 123145

CUSTOMERS LIST IN ALPHABETICAL ORDER

       Argand,J.R   900823
       Bessel,F.W   719731
       Gandhi,M.K   362718
       Gauss,C.F    806788
       Gibbs,J.W    123145
       Lagrange,J.L 869245
       Leibniz,G.W  711518
       Poisson,S.D  853240
       Stokes,G.G   545454
       Sturm,C.F    222031
```

Fig. 8.12 *Program to alphabetize a customer list*

REVIEW QUESTIONS

8.1 State whether the following statements are *true* or *false*

(a) When initializing a string variable during its declaration, we must include the null character as part of the string constant, like "GOOD\0".

(b) The **gets** function automatically appends the null character at the end of the string read from the keyboard.

(c) When reading a string with **scanf**, it automatically inserts the terminating null character.
(d) String variables cannot be used with the assignment operator.
(e) We cannot perform arithmetic operations on character variables.
(f) We can assign a character constant or a character variable to an **int** type variable.
(g) The function **scanf** cannot be used in any way to read a line of text with the white-spaces.
(h) The ASCII character set consists of 128 distinct characters.
(i) In the ASCII collating sequence, the uppercase letters precede lowercase letters.
(j) In C, it is illegal to mix character data with numeric data in arithmetic operations.
(k) The function **getchar** skips white-space during input.
(l) In C, strings cannot be initialized at run time.
(m) The input function **gets** has one string parameter.
(n) The function call **strcpy(s2, s1);** copies string s2 into string s1.
(o) The function call **strcmp("abc", "ABC");** returns a positive number.

8.2 Fill in the blanks in the following statements.
(a) We can use the conversion specification _______in **scanf** to read a line of text.
(b) We can initialize a string using the string manipulation function_______.
(c) The function **strncat** has _____ parameters.
(d) To use the function **atoi** in a program, we must include the header file ________.
(e) The function _______does not require any conversion specification to read a string from the keyboard.
(f) The function _______ is used to determine the length of a string.
(g) The _________string manipulation function determines if a character is contained in a string.
(h) The function _____is used to sort the strings in alphabetical order.
(i) The function call **strcat (s2, s1);** appends _____ to ______.
(j) The **printf** may be replaced by ______function for printing strings.

8.3 Describe the limitations of using **getchar** and **scanf** functions for reading strings.

8.4 Character strings in C are automatically terminated by the *null* character. Explain how this feature helps in string manipulations.

8.5 Strings can be assigned values as follows:

(a) During type declaration	`char string[ ] = {"......."};`
(b) Using **strcpy** function	`strcpy(string, ".......");`
(c) Reading using **scanf** function	`scanf("%s", string);`
(d) Reading using **gets** function	`gets(string);`

Compare them critically and describe situations where one is superior to the others.

8.6 Assuming the variable **string** contains the value "The sky is the limit.", determine what output of the following program segments will be.

```
(a) printf("%s", string);
(b) printf("%25.10s", string);
(c) printf("%s", string[0]);
(d) for (i=0; string[i] != "."; i++)
        printf("%c", string[i]);
(e) for (i=0; string[i] != '\0'; i++;)
```

```
    printf("%d\n", string[i]);
(f) for (i=0; i <= strlen[string]; ;)
    {
       string[i++] = i;
       printf("%s\n", string[i]);
    }
(g) printf("%c\n", string[10] + 5);
(h) printf("%c\n", string[10] + 5')
```

8.7 Which of the following statements will correctly store the concatenation of strings **s1** and **s2** in string **s3**?

```
(a) s3 = strcat (s1, s2);
(b) strcat (s1, s2, s3);
(c) strcat (s3, s2, s1);
(d) strcpy (s3, strcat (s1, s2));
(e) strcmp (s3, strcat (s1, s2));
(f) strcpy (strcat (s1, s2), s3);
```

8.8 What will be the output of the following statement?

```
        printf ("%d", strcmp ("push", "pull"));
```

8.9 Assume that s1, s2 and s3 are declared as follows:

```
        char s1[10] = "he", s2[20] = "she", s3[30], s4[30];
```

What will be the output of the following statements executed in sequence.

```
        printf("%s", strcpy(s3, s1));
        printf("%s", strcat(strcat(strcpy(s4, s1), "or"), s2));
        printf("%d %d", strlen(s2)+strlen(s3), strlen(s4));
```

8.10 Find errors, if any, in the following code segments;

```
(a) char str[10]
    strncpy(str, "GOD", 3);
    printf("%s", str);
(b) char str[10];
    strcpy(str, "Balagurusamy");
(c) if strstr("Balagurusamy", "guru") == 0);
    printf("Substring is found");
(d) char s1[5], s2[10],
    gets(s1, s2);
```

PROGRAMMING EXERCISES

8.1 Write a program, which reads your name from the keyboard and outputs a list of ASCII codes, which represent your name.

8.2 Write a program to do the following:
(a) To output the question "Who is the inventor of C ?"
(b) To accept an answer.
(c) To print out "Good" and then stop, if the answer is correct.
(d) To output the message 'try again', if the answer is wrong.

(e) To display the correct answer when the answer is wrong even at the third attempt and stop.

8.3 Write a program to extract a portion of a character string and print the extracted string. Assume that m characters are extracted, starting with the nth character.

8.4 Write a program which will read a text and count all occurrences of a particular word.

8.5 Write a program which will read a string and rewrite it in the alphabetical order. For example, the word STRING should be written as GINRST.

8.6 Write a program to replace a particular word by another word in a given string. For example, the word "PASCAL" should be replaced by "C" in the text "It is good to program in PASCAL language."

8.7 A Maruti car dealer maintains a record of sales of various vehicles in the following form:

Vehicle type	**Month of sales**	**Price**
MARUTI-800	02/01	210000
MARUTI-DX	07/01	265000
GYPSY	04/02	315750
MARUTI-VAN	08/02	240000

Write a program to read this data into a table of strings and output the details of a particular vehicle sold during a specified period. The program should request the user to input the vehicle type and the period (starting month, ending month).

8.8 Write a program that reads a string from the keyboard and determines whether the string is a *palindrome* or not. (A string is a palindrome if it can be read from left and right with the same meaning. For example, Madam and Anna are palindrome strings. Ignore capitalization).

8.9 Write program that reads the cost of an item in the form RRRR.PP (Where RRRR denotes Rupees and PP denotes Paise) and converts the value to a string of words that expresses the numeric value in words. For example, if we input 125.75, the output should be "ONE HUNDRED TWENTY FIVE AND PAISE SEVENTY FIVE".

8.10 Develop a program that will read and store the details of a list of students in the format

Roll No.	**Name**	**Marks obtained**
....	....	....
....	....	....
....	....	....

and produce the following output lits:

(a) Alphabetical list of names, roll numbers and marks obtained.

(b) List sorted on roll numbers.

(c) List sorted on marks (rank-wise list)

Chapter 9

User-Defined Functions

9.1 INTRODUCTION

We have mentioned earlier that one of the strengths of C language is that C functions are easy to define and use. We have used functions in every program that we have discussed so far. However, they have been primarily limited to the three functions, namely, **main, printf**, and **scanf**. In this chapter, we shall consider in detail how a function is designed, how two or more functions are put together and how they communicate with one another.

C functions can be classified into two categories, namely, *library* functions and *user-defined* functions. **main** is an example of user-defined functions. **printf** and **scanf** belong to the category of library functions. We have also used other library functions such as **sqrt, cos, strcat,** etc. The main distinction between these two categories is that library functions are not required to be written by us whereas a user-defined function has to be developed by the user at the time of writing a program. However, a user-defined function can later become a part of the C program library. In fact, this is one of the strengths of C language.

9.2 NEED FOR USER-DEFINED FUNCTIONS

As pointed out earlier, **main** is a specially recognized function in C. Every program must have a **main** function to indicate where the program has to begin its execution. While it is possible to code any program utilizing only **main** function, it leads to a number of problems. The program may become too large and complex and as a result the task of debugging, testing, and maintaining becomes difficult. If a program is divided into functional parts, then each part may be independently coded and later combined into a single unit. These subprograms called 'functions' are much easier to understand, debug, and test.

There are times when certain type of operations or calculations is repeated at many points throughout a program. For instance, we might use the factorial of a number at several points in the program. In such situations, we may repeat the program statements wherever they are needed. Another approach is to design a function that can be called and used whenever required. This saves both time and space.

This "division" approach clearly results in a number of advantages.

1. It facilitates top-down modular programming as shown in Fig. 9.1. In this programming style, the high level logic of the overall problem is solved first while the details of each lower-level function are addressed later.
2. The length of a source program can be reduced by using functions at appropriate places. This factor is particularly critical with microcomputers where memory space is limited.
3. It is easy to locate and isolate a faulty function for further investigations.
4. A function may be used by many other programs. This means that a C programmer can build on what others have already done, instead of starting all over again from scratch.

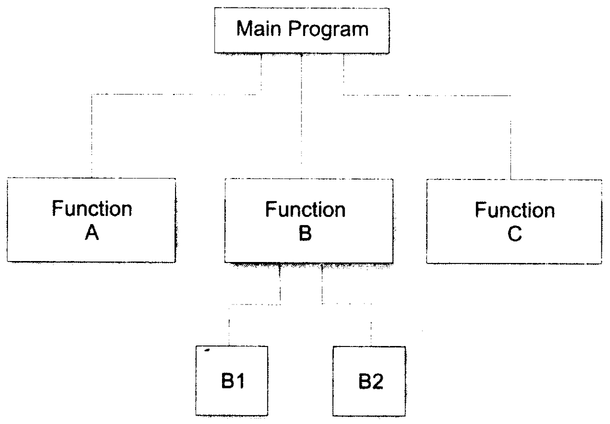


Fig. 9.1 *Top-down modular programming using functions*

9.3 A MULTI-FUNCTION PROGRAM

A function is a self-contained block of code that performs a particular task. Once a function has been designed and packed, it can be treated as a 'black box' that takes some data from the main program and returns a value. The inner details of operation are invisible to the rest of the program. All that the program knows about a function is: What goes in and what comes out. Every C program can be designed using a collection of these black boxes known as *functions*.

Consider a set of statements as shown below:

```
void  printline(void)
{
  int i;
  for (i=1; i<40; i++)
       printf("-");
  printf("\n");
}
```

The above set of statements defines a function called **printline,** which could print a line of 39-character length. This function can be used in a program as follows:

```
void printline(void);  /* declaration  */
  main( )
  {
       printline( );
       printf("This illustrates the use of C functions\n");
       printline();
  }
   void printline(void)
  {
    int i;
    for(i=1; i<40; i++)
    printf("-");
    printf("\n");
   }
```

This program will print the following output:

```
---------------------------------------
  This illustrates the use of C functions
---------------------------------------
```

The above program contains two user-defined functions:

main() function

printline() function

As we know, the program execution always begins with the **main** function. During execution of the **main,** the first statement encountered is

```
printline( );
```

which indicates that the function **printline** is to be executed. At this point, the program control is transferred to the function **printline.** After executing the **printline** function, which outputs a line of 39 character length, the control is transferred back to the **main.** Now, the execution continues at the point where the function call was executed. After executing the **printf** statement, the control is again transferred to the **printline** function for printing the line once more.

The **main** function calls the user-defined **printline** function two times and the library function **printf** once. We may notice that the **printline** function itself calls the library function **printf** 39 times repeatedly.

Any function can call any other function. In fact, it can call itself. A 'called function' can also call another function. A function can be called more than once. In fact, this is one of the main features of using functions. Figure 9.2 illustrates the flow of control in a multi-function program.

Except the starting point, there are no other predetermined relationships, rules of precedence, or hierarchies among the functions that make up a complete program. The functions can be placed in any order. A called function can be placed either before or after the calling function. However, it is the usual practice to put all the called functions at the end. See the box "Modular Programming"

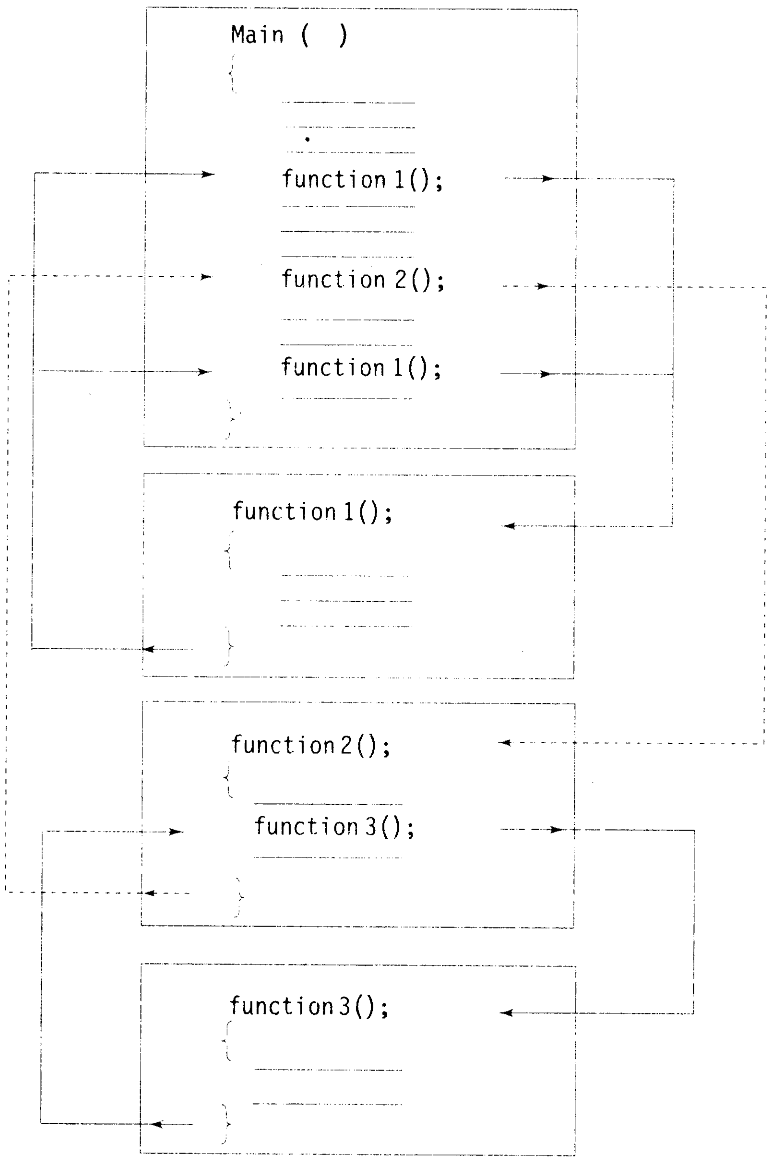


Fig. 9.2 *Flow of control in a multi-function program*