6 Decision Making and Looping

6.1 INTRODUCTION

We have seen in the previous chapter that it is possible to execute a segment of a program repeatedly by introducing a counter and later testing it using the **if** statement. While this method is quite satisfactory for all practical purposes, we need to initialize and increment a counter and test its value at an appropriate place in the program for the completion of the loop. For example, suppose we want to calculate the sum of squares of all integers between 1 and 10, we can write a program using the **if** statement as follows:

```
           ..........
           ..........
           sum = 0;
           n = 1;
        -> loop:
   L       sum = sum + n*n;
   o       if (n == 10)
   o           goto print;--
   p       else             |  n = 10,
           {                |  end of loop
              n = n+1;      |
        ---   goto loop;    |
           }                |
           print: <---------
           ..........
           ..........
```

This program does the following things:

1. Initializes the variable **n**.
2. Computes the square of **n** and adds it to **sum**.
3. Tests the value of **n** to see whether it is equal to 10 or not. If it is equal to 10, then the program prints the results.
4. If **n** is less than 10, then it is incremented by one and the control goes back to compute the **sum** again.

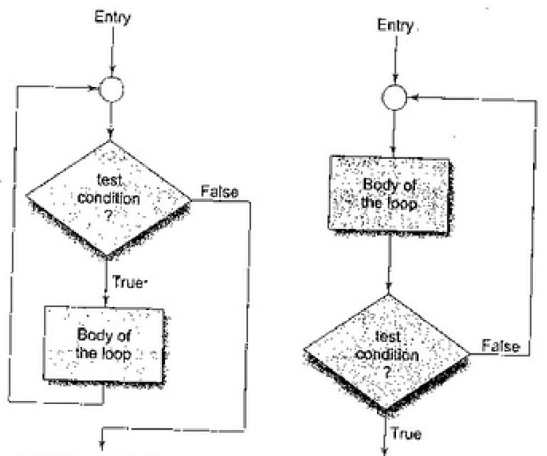
The program evaluates the statement

```
sum = sum + n*n;
```

10 times. That is, the loop is executed 10 times. This number can be increased or decreased easily by modifying the relational expression appropriately in the statement **if** (n == 10). On such occasions where the exact number of repetitions are known, there are more convenient methods of looping in C. These looping capabilities enable us to develop concise programs containing repetitive processes without the use of **goto** statements.

In looping, a sequence of statements are executed until some conditions for termination of the loop are satisfied. A *program loop* therefore consists of two segments, one known as the *body of the loop* and the other known as the *control statement*. The control statement tests certain conditions and then directs the repeated execution of the statements contained in the body of the loop.

Depending on the position of the control statement in the loop, a control structure may be classified either as the *entry-controlled loop* or as the *exit-controlled loop*. The flow charts in Fig. 6.1 illustrate these structures. In the entry-controlled loop, the control conditions are tested before the start of the loop execution. If the conditions are not satisfied, then the body of the loop will not be executed. In the case of an exit-controlled loop, the test is performed at the end of the body of the loop and therefore the body is executed unconditionally for the first time. The entry-controlled and exit-controlled loops are also known as *pre-test* and *post-test* loops respectively.



(a) Entry controlled loop (b) Exit controlled loop

Fig. 6.1 *Loop control structures*

The test conditions should be carefully stated in order to perform the desired number of loop executions. It is assumed that the test condition will eventually transfer the control out of the loop. In case, due to some reason it does not do so, the control sets up an *infinite loop* and the body is executed over and over again.

A looping process, in general, would include the following four steps:

1. Setting and initialization of a condition variable.
2. Execution of the statements in the loop.
3. Test for a specified value of the condition variable for execution of the loop.
4. Incrementing or updating the condition variable.

The test may be either to determine whether the loop has been repeated the spe number of times or to determine whether a particular condition has been met.

The C language provides for three *constructs* for performing *loop* operations. The

1. The **while** statement.
2. The **do** statement.
3. The **for** statement.

We shall discuss the features and applications of each of these statements in this cha

Sentinel Loops

Based on the nature of control variable and the kind of value assigned to it for testing the control expression, the loops may be classified into two general categories:

1. Counter-controlled loops
2. Sentinel-controlled loops

When we know in advance exactly how many times the loop will be executed we use a *counter-controlled loop*. We use a control variable known *as counter*. The counter must be initialized, tested and updated properly for the desired loop operations. The number of times we want to execute the loop may be a constant or a variable that is assigned a value. A counter-controlled loop is sometimes called *definite repetition loop*.

In a *sentinel-controlled loop*, a special value called a *sentinel* value is used to change the loop control expression from true to false. For example, when reading data we may indicate the "end of data" by a special value, like –1 and 999. The control variable is called **sentinel** variable. A sentinel-controlled loop is often called *indefinite repetition loop* because the number of repetitions is not known before the loop begins executing.

6.2 THE WHILE STATEMENT

The simplest of all the looping structures in C is the **while** statement. We have used w in many of our earlier programs. The basic format of the **while** statement is

```
while (test condition)
{
      body of the loop
}
```

The **while** is an *entry-controlled* loop statement. The *test-condition* is evaluated and if the condition is *true*, then the body of the loop is executed. After execution of the body, the test-condition is once again evaluated and if it is true, the body is executed once again. This process of repeated execution of the body continues until the test-condition finally becomes *false* and the control is transferred out of the loop. On exit, the program continues with the statement immediately after the body of the loop.

The body of the loop may have one or more statements. The braces are needed only if the body contains two or more statements. However, it is a good practice to use braces even if the body has only one statement.

We can rewrite the program loop discussed in Section 6.1 as follows:

```
        ========
        sum = 0;
        n = 1;                         /* Initialization */
      → while(n <= 10)                 /* Testing */
        {
loop      sum = sum + n * n;
            n = n+1;                   /* Incrementing */
      → }
        printf("sum = %d\n", sum);
        ========
```

The body of the loop is executed 10 times for n = 1, 2, ..., 10, each time adding the square of the value of n, which is incremented inside the loop. The test condition may also be written as n < 11; the result would be the same. This is a typical example of counter-controlled loops. The variable n is called ***counter*** or ***control variable***.

Another example of **while** statement, which uses the keyboard input is shown below:

```
=========
character = ' ' ;
while (character != 'Y')
       character = getchar();
xxxxxxx;
=========
```

First the **character** is initialized to ' '. The **while** statement then begins by testing whether **character** is not equal to Y. Since the **character** was initialized to ' ', the test is true and the loop statement

```
character = getchar();
```

is executed. Each time a letter is keyed in, the test is carried out and the loop statemen executed until the letter Y is pressed. When Y is pressed, the condition becomes false cause **character** equals Y, and the loop terminates, thus transferring the control to statement xxxxxxx;. This is a typical example of sentinel-controlled loops. The charac constant 'y' is called *sentinel* value and the variable **character** is the condition varia which often referred to as the *sentinel variable*.

Example 6.1 A program to evaluate the equation

$y = x^n$

when n is a non-negative integer, is given in Fig. 6.2

The variable **y** is initialized to 1 and then multiplied by **x, n** times using the **while** The loop control variable **count** is initialized outside the loop and incremented inside loop. When the value of **count** becomes **greater than n**, the control exists the loop.

```
Program
     main()
     {
        int count, n;
        float x, y;
        printf("Enter the values of x and n : ");
        scanf("%f %d", &x, &n);
        y = 1.0;
        count = 1;          /* Initialisation */
        /* LOOP BEGINs */
        while ( count <= n)  /* Testing */
        {
          y = y*x;
          count++;           /* Incrementing */
        }
        /* END OF LOOP */
        printf("\nx = %f; n = %d; x to power n = %f\n",x,n,y);
     }
Output
    Enter the values of x and n : 2.5 4
    x = 2.500000; n = 4; x to power n = 39.062500
    Enter the values of x and n : 0.5 4
    x = 0.500000; n = 4; x to power n = 0.062500
```

Fig. 6.2 *Program to compute* **x** *to the power n using* **while** *loop*

6.3 THE DO STATEMENT

The **while** loop construct that we have discussed in the previous section, makes a test of condition *before* the loop is executed. Therefore, the body of the loop may not be executed at all if the condition is not satisfied at the very first attempt. On some occasions it might be necessary to execute the body of the loop before the test is performed. Such situations can be handled with the help of the **do** statement. This takes the form:

```
do
{
   body of the loop
}
while (test-condition);
```

On reaching the **do** statement, the program proceeds to evaluate the body of the loop first. At the end of the loop, the *test-condition* in the **while** statement is evaluated. If the condition is true, the program continues to evaluate the body of the *loop* once again. This process continues as long as the *condition* is **true**. When the condition becomes false, the loop will be terminated and the control goes to the statement that appears immediately after the **while** statement.

Since the *test-condition* is evaluated at the bottom of the loop, the **do...while** construct provides an *exit-controlled* loop and therefore the body of the loop is *always executed at least once.*

A simple example of a **do...while** loop is:

```
         ............
           do
       ┌─▶ {
       │      printf ("Input a number\n");
 loop  │      number = getnum ( );
       │   }
       └── while (number > 0);
         ............
```

This segment of a program reads a number from the keyboard until a zero or a negative number is keyed in, and assigned to the *sentinel* variable **number**.

The test conditions may have compound relations as well. For instance, the statement

```
while (number > 0 && number < 100);
```

in the above example would cause the loop to be executed as long as the number keyed in lies between 0 and 100.

Consider another example:

```
    -------
    I = 1;                          /* Initializing */
    sum = 0;
    do
```

```
              {
              sum = sum + I;
      loop    I = I+2;                          /* Incrementing */
              }
              while(sum < 40 || I < 10);      /* Testing */
              printf("%d %d\n", I, sum);
              -------
```

The loop will be executed as long as one of the two relations is true.

Example 6.2 A program to print the multiplication table from 1 x 1 to 12 x 10 as sho below is given in Fig. 6.3.

```
1    2    3    4    ........   10
2    4    6    8    ........   20
3    6    9    12   ........   30
4                   ........   40
-
-
-
12                  ........  120
```

This program contains two **do.... while** loops in nested form. The outer loop is controlle the variable **row** and executed 12 times. The inner loop is controlled by the variable **colu** and is executed 10 times, each time the outer loop is executed. That is, the inner loop executed a total of 120 times, each time printing a value in the table.

```
Program:
    #define COLMAX 10
    #define ROWMAX 12
    main()
    {
        int row,column, y;
        row = 1;
        printf("    MULTIPLICATION TABLE     \n");
        printf("-----------------------------\n");
        do /*.....OUTER LOOP BEGINS........*/
        {
            column = 1;
            do /*.......INNER LOOP BEGINS.......*/
            {
              y = row * column;
              printf("%4d", y);
              column = column + 1;
            }
            while (column <= COLMAX); /*... INNER LOOP ENDS ...*/
            printf("\n");
            row = row + 1;
        }
        while (row <= ROWMAX);/*..... OUTER LOOP ENDS .....*/
        printf("-----------------------------------\n");
    }
Output
           MULTIPLICATION TABLE
  -----------------------------------------
  1   2   3   4   5   6   7   8   9   10
  2   4   6   8   10  12  14  16  18  20
  3   6   9   12  15  18  21  24  27  30
  4   8   12  16  20  24  28  32  36  40
  5   10  15  20  25  30  35  40  45  50
  6   12  18  24  30  36  42  48  54  60
  7   14  21  28  35  42  49  56  63  70
  8   16  24  32  40  48  56  64  72  80
  9   18  27  36  45  54  63  72  81  90
  10  20  30  40  50  60  70  80  90  100
  11  22  33  44  55  66  77  88  99  110
  12  24  36  48  60  72  84  96  108 120
  -----------------------------------------
```

Fig. 6.3 *Printing of a multiplication table using* ***do...while loop***

Notice that the **printf** of the inner loop does not contain any new line character (\n). This allows the printing of all row values in one line. The empty **printf** in the outer loop initiates a new line to print the next row.

6.4 THE FOR STATEMENT

Simple 'for' Loops

The **for** loop is another *entry-controlled* loop that provides a more concise loop control structure. The general form of the **for** loop is

```
for ( initialization ; test-condition ; increment)
{
    body of the loop
}
```

The execution of the **for** statement is as follows:

1. *Initialization* of the *control variables* is done first, using assignment statements such as i = 1 and count = 0. The variables **i** and **count** are known as loop-control variables.
2. The value of the control variable is tested using the test-condition. The *test-condition* is a relational expression, such as i < 10 that determines when the loop will exit. If the

```
      double q;
      printf("-----------------------------------\n");
      printf(" 2 to power n        n       2 to power -n\n");
      printf("-----------------------------------\n");
      p = 1;
      for (n = 0; n < 21 ; ++n) /* LOOP BEGINS */
      {
        if (n == 0)
          p = 1;
        else
          p = p * 2;
        q = 1.0/(double)p ;
        printf("%10ld %10d %20.12lf\n", p, n, q);
      }                           /* LOOP ENDS */
      printf("-----------------------------------\n");
   }
Output
```

```
   ----------------------------------------------
    2 to power n       n           2 to power -n
   ----------------------------------------------
             1         0          1.000000000000
             2         1          0.500000000000
             4         2          0.250000000000
             8         3          0.125000000000
            16         4          0.062500000000
            32         5          0.031250000000
            64         6          0.015625000000
           128         7          0.007812500000
           256         8          0.003906250000
           512         9          0.001953125000
          1024        10          0.000976562500
          2048        11          0.000488281250
          4096        12          0.000244140625
          8192        13          0.000122070313
         16384        14          0.000061035156
         32768        15          0.000030517578
         65536        16          0.000015258789
        131072        17          0.000007629395
        262144        18          0.000003814697
        524288        19          0.000001907349
       1048576        20          0.000000953674
   ----------------------------------------------
```

Fig. 6.4 Program to print 'Power of 2' table using for loop

Note that the initialization section has two parts **p** = 1 and **n** = 1 separated by a *comma*.

Like the initialization section, the increment section may also have more than one part. For example, the loop

```
for (n=1, m=50; n<=m; n=n+1, m=m-1)
{
    p = m/n;
    printf("%d %d %d\n", n, m, p);
}
```

is perfectly valid. The multiple arguments in the increment section are separated by *commas*.

The third feature is that the test-condition may have any compound relation and the testing need not be limited only to the loop control variable. Consider the example below:

```
sum = 0;
for (i = 1; i < 20 && sum < 100; ++i)
{
    sum = sum+i;
    printf("%d %d\n", i, sum);
}
```

The loop uses a compound test condition with the counter variable **i** and sentinel variable **sum**. The loop is executed as long as both the conditions **i** < 20 and **sum** < 100 are true. The **sum** is evaluated inside the loop.

It is also permissible to use expressions in the assignment statements of initialization and increment sections. For example, a statement of the type

```
for (x = (m+n)/2; x > 0; x = x/2)
```

is perfectly valid.

Another unique aspect of **for** loop is that one or more sections can be omitted, if necessary. Consider the following statements:

```
-------
m = 5;
for ( ; m != 100 ; )
{
    printf("%d\n", m);
    m = m+5;
}
-------
```

Both the initialization and increment sections are omitted in the **for** statement. The initialization has been done before the **for** statement and the control variable is incremented inside the loop. In such cases, the sections are left 'blank'. However, the semicolons separating the sections must remain. If the test-condition is not present, the **for** statement sets up an *'infinite'* loop. Such loops can be broken using **break** or **goto** statements in the loop.

We can set up *time delay loops* using the null statement as follows:

```
for ( j = 1000; j > 0; j = j-1)
    ;
```

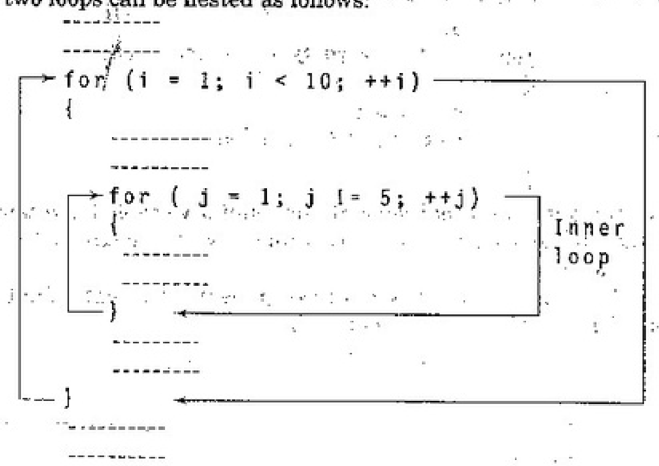
This loop is executed 1000 times without producing any output; it simply causes a time delay. Notice that the body of the loop contains only a semicolon, known as a *null* statement. This can also be written as

```
for (j=1000; j > 0; j = j-1)
```

This implies that the C compiler will not give an error message if we place a semicolon by mistake at the end of a **for** statement. The semicolon will be considered as a *null statement* and the program may produce some nonsense.

Nesting of for Loops

Nesting of loops, that is, one **for** statement within another **for** statement, is allowed in C. For example, two loops can be nested as follows:



```
----------
----------
for (i = 1; i < 10; ++i)
{
    ----------
    ----------
    for ( j = 1; j != 5; ++j)
    {
        ---------
        ---------
    }
    ---------
    ---------
}
----------
----------
```

The nesting may continue up to any desired level. The loops should be properly indented so as to enable the reader to easily determine which statements are contained within each **for** statement. (ANSI C allows up to 15 levels of nesting. However, some compilers permit more).

The program to print the multiplication table discussed in Example 6.2 can be written more concisely using nested for statements as follows:

```
--------------
for (row = 1; row <= ROWMAX ; ++row)
{
   for (column = 1; column <= COLMAX ; ++column)
   {
       y = row * column;
       printf("%4d", y);
   }
   printf("\n");
}
--------------
```

The outer loop controls the rows while the inner loop controls the columns.

Example 6.4 A class of **n** students take an annual examination in **m** subjects. A program to read the marks obtained by each student in various subjects and to compute and print the total marks obtained by each of them is given in Fig. 6.5.

The program uses two **for** loops, one for controlling the number of students and the other for controlling the number of subjects. Since both the number of students and the number of subjects are requested by the program, the program may be used for a class of any size and any number of subjects.

The outer loop includes three parts:

(1) reading of roll-numbers of students, one after another;
(2) inner loop, where the marks are read and totalled for each student; and
(3) printing of total marks and declaration of grades.

```
Program
     #define FIRST 360
     #define SECOND 240
     main()
     {
          int n, m, i, j,
               roll_number, marks, total;
          printf("Enter number of students and subjects\n");
          scanf("%d %d", &n, &m);
          printf("\n");
          for (i = 1; i <= n ; ++i)
          {
               printf("Enter roll_number : ");
               scanf("%d", &roll_number);
               total = 0 ;
               printf("\nEnter marks of %d subjects for ROLL NO %d\n",
                         m,roll_number);
               for (j = 1; j <= m; j++)
               {
                    scanf("%d", &marks);
                    total = total + marks;
               }
               printf("TOTAL MARKS = %d ", total);
               if (total >= FIRST)
                  printf("( First Division )\n\n");
               else if (total >= SECOND)
                      printf("( Second Division )\n\n");
                 else
                      printf("( *** F A I L *** )\n\n");
          }
     }
```

```
Output
            Enter number of students and subjects
            3 6
            Enter roll_number : 8701
            Enter marks of 6 subjects for ROLL NO 8701
            81 75 83 45 61 59
            TOTAL MARKS = 404 ( First Division )
            Enter roll_number : 8702
            Enter marks of 6 subjects for ROLL NO 8702
            51 49 55 47 65 41
            TOTAL MARKS = 308 ( Second Division )
            Enter roll_number : 8704
            Enter marks of 6 subjects for ROLL NO 8704
            40 19 31 47 39 25
            TOTAL MARKS = 201 ( *** F A I L *** )
```

Fig. 6.5 *Illustration of nested* ***for*** *loops*

Selecting a Loop

Given a problem, the programmer's first concern is to decide the type of loop structure to be used. To choose one of the three loop supported by C, we may use the following strategy:

- Analyse the problem and see whether it required a pre-test or post-test loop.
- If it requires a post-test loop, then we can use only one loop, **do while**.
- If it requires a pre-test loop, then we have two choices: **for** and **while**.
- Decide whether the loop termination requires counter-based control or sentinel-based control.
- Use **for** loop if the counter-based control is necessary.
- Use **while** loop if the sentinel-based control is required.
- Note that both the counter-controlled and sentinel-controlled loops can be implemented by all the three control structures.

6.5 JUMPS IN LOOPS

Loops perform a set of operations repeatedly until the control variable fails to satisfy the test-condition. The number of times a loop is repeated is decided in advance and the test condition is written to achieve this. Sometimes, when executing a loop it becomes desirab to skip a part of the loop or to leave the loop as soon as a certain condition occurs. Fo example, consider the case of searching for a particular name in a list containing, say, 100 names. A program loop written for reading and testing the names 100 times must be ter

nated as soon as the desired name is found. C permits a *jump* from one statement to another within a loop as well as a *jump* out of a loop.

Jumping Out of a Loop

An early exiti from a loop can be accomplished by using the **break** statement or the **goto** statement. We have already seen the use of the **break** in the **switch** statement and the **goto** in the **if...else** construct. These statements can also be used within **while, do,** or **for** loops. They are illustrated in Fig. 6.6 and Fig. 6.7.

When a **break** statement is encountered inside a loop, the loop is immediately exited and the program continues with the statement immediately following the loop. When the loops are nested, the **break** would only exit from the loop containing it. That is, the **break** will exit only a single loop.

Since a **goto** statement can transfer the control to any place in a program, it is useful to provide branching within a loop. Another important use of **goto** is to exit from deeply nested loops when an error occurs. A simple **break** statement would not work here.

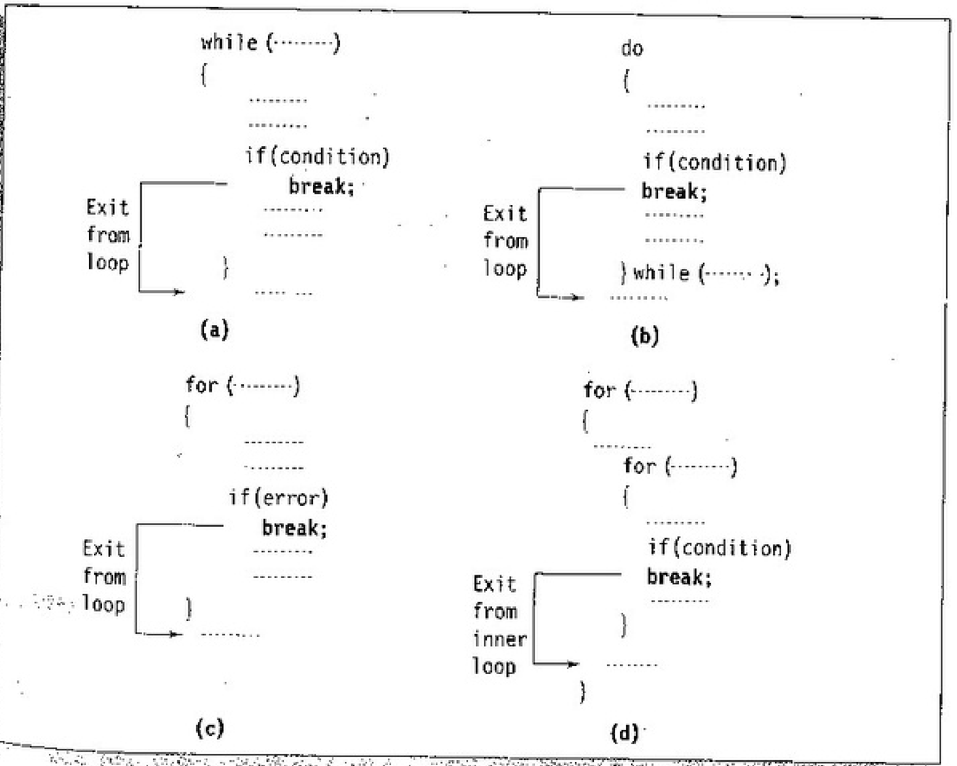


Fig. 6.6 *Exiting a loop with* ***break*** *statement*

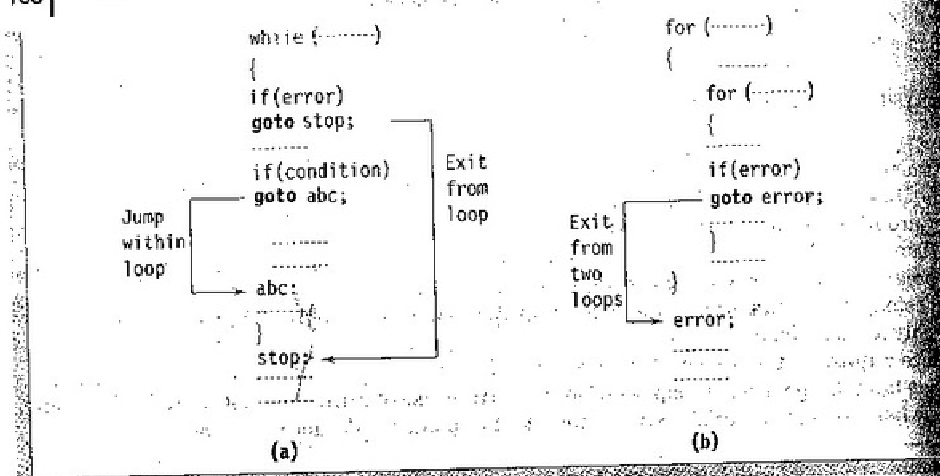


Fig. 6.7 Jumping within and exiting from the loops with **goto** statement

Example 6.5 The program in Fig. 6.8 illustrates the use of the break statement in a program.

The program reads a list of positive values and calculates their average. The **for** loop is written to read 1000 values. However, if we want the program to calculate the average of a set of values less than 1000, then we must enter a 'negative' number after the last value in the list, to mark the end of input.

```
Program
     main()
     {
          int m;
          float x, sum, average;

          printf("This  program computes the average of a
                        set of numbers\n");
          printf("Enter values one after another\n");
          printf("Enter a NEGATIVE number at the end.\n\n");
          sum = 0;
          for (m = 1 ; m < = 1000 ; ++m)
          {
            scanf("%f", &x);
            if (x < 0)
               break;
            sum += x ;
          }
          average = sum/(float)(m-1);
          printf("\n");
```

```
          printf("Number of values  = %d\n", m-1);
          printf("Sum               = %f\n", sum);
          printf("Average           = %f\n", average);

Output
     This program computes the average of a set of numbers
     Enter values one after another
     Enter a NEGATIVE number at the end.

     21 23 24 22 26 22 -1

     Number of values = 6
     Sum              = 138.000000
     Average          = 23.000000
```

Fig. 6.8 Use of ***break*** *in a program*

Each value, when it is read, is tested to see whether it is a positive number or not. If it is positive, the value is added to the **sum**; otherwise, the loop terminates. On exit, the average of the values read is calculated and the results are printed out.

Example 6.6 A program to evaluate the series

$$\frac{1}{1-x} = 1 + x + x^2 + x^3 + \ldots + x^n$$

for $-1 < x < 1$ with 0.01 per cent accuracy is given in Fig. 6.9. The goto statement is used to exit the loop on achieving the desired accuracy.

We have used the **for** statement to perform the repeated addition of each of the terms in the series. Since it is an infinite series, the evaluation of the function is terminated when the term x^n reaches the desired accuracy. The value of n that decides the number of loop operations is not known and therefore we have decided arbitrarily a value of 100, which may or may not result in the desired level of accuracy.

```
Program
     #define  LOOP        100
     #define  ACCURACY    0.0001
     main()
     {
          int n;
          float x, term, sum;

          printf("Input value of x : ");
          scanf("%f", &x);
          sum = 0 ;
          for (term = 1, n = 1 ; n <= LOOP ; ++n)
          {
               sum += term ;
               if (term <= ACCURACY)
```

```
                goto output; /* EXIT FROM THE LOOP */
              term *= x ;
            }
            printf("\nFINAL VALUE OF N IS NOT SUFFICIENT\n");
            printf("TO ACHIEVE DESIRED ACCURACY\n");
            goto end;
            output:
            printf("\nEXIT FROM LOOP\n");
            printf("Sum = %f; No.of terms = %d\n", sum, n);
            end:
            ;      /* Null Statement */
      }
Output
      Input value of x : .21
      EXIT FROM LOOP
      Sum = 1.265800; No.of terms = 7
      Input value of x : .75
      EXIT FROM LOOP
      Sum = 3.999774; No.of terms = 34
      Input value of x : .99
      FINAL VALUE OF N IS NOT SUFFICIENT
      TO ACHIEVE DESIRED ACCURACY
```

Fig. 6.9 *Use of* **goto** *to exit from a loop*

The test of accuracy is made using an **if** statement and the **goto** statement exits the loop as soon as the accuracy condition is satisfied. If the number of loop repetitions is not large enough to produce the desired accuracy, the program prints an appropriate message.

Note that the **break** statement is not very convenient to use here. Both the normal exit and the **break** exit will transfer the control to the same statement that appears next to the loop. But, in the present problem, the normal exit prints the message

"FINAL VALUE OF N IS NOT SUFFICIENT

TO ACHIEVE DESIRED ACCURACY"

and the *forced exit* prints the results of evaluation. Notice the use of a *null* statement at the end. This is necessary because a program should not end with a label.

Structured Programming

Structured programming is an approach to the design and development of programs. It is a discipline of making a program's logic easy to understand by using only the basic three control structures:

- Sequence (straight line) structure
- Selection (branching) structure

- Repetition (looping) structure

While sequence and loop structures are sufficient to meet all the requirements of programming, the selection structure proves to be more convenient in some situations.

The use of structured programming techniques helps ensure well-designed programs that are easier to write, read, debug and maintain compared to those that are unstructured.

Structured programming discourages the implementation of unconditional branching using jump statements such as **goto**, **break** and **continue**. In its purest form, structured programming is synonymous with "*goto less programming*".

Do not go to **goto** statement!

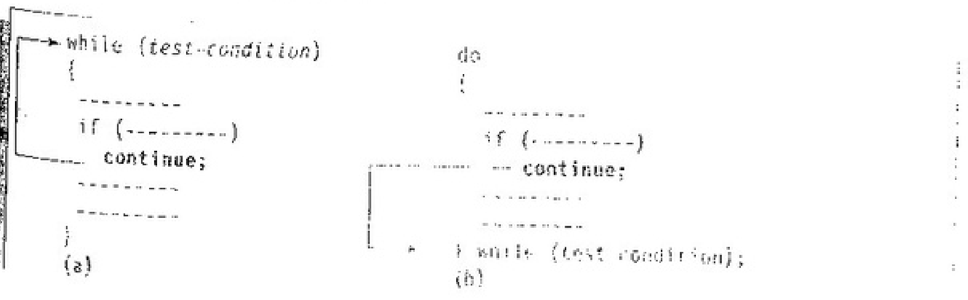
Skipping a Part of a Loop

During the loop operations, it may be necessary to skip a part of the body of the loop under certain conditions. For example, in processing of applications for some job, we might like to exclude the processing of data of applicants belonging to a certain category. On reading the category code of an applicant, a test is made to see whether his application should be considered or not. If it is not to be considered, the part of the program loop that processes the application details is skipped and the execution continues with the next loop operation.

Like the **break** statement, C supports another similar statement called the **continue** statement. However, unlike the **break** which causes the loop to be terminated, the **continue**, as the name implies, causes the loop to be continued with the next iteration after skipping any statements in between. The **continue** statement tells the compiler, "SKIP THE FOLLOWING STATEMENTS AND CONTINUE WITH THE NEXT ITERATION". The format of the **continue** statement is simply

```
continue;
```

The use of the **continue** statement in loops is illustrated in Fig. 6.10. In **while** and **do** loops, **continue** causes the control to go directly to the test-condition and then to continue the iteration process. In the case of **for** loop, the increment section of the loop is executed before the test-condition is evaluated.



```
while (test-condition)          do
{                               {
  ---------                       ---------
  if (---------)                  if (---------)
    continue;                       continue;
  ---------                       ---------
  ---------                       ---------
}                               } while (test-condition);
(a)                             (b)
```

```
    for (initialization; test condition; increment)
    {
      ---------
      if (---------)
        continue;
      ---------
      ---------
    }
      (c)
```

Fig. 6.10 Bypassing and continuing in loops

Example 6.7 The program in Fig. 6.11 illustrates the use of **continue** statement.

The program evaluates the square root of a series of numbers and prints the results. The process stops when the number 9999 is typed in.

In case, the series contains any negative numbers, the process of evaluation of square root should be bypassed for such numbers because the square root of a negative number is not defined. The **continue** statement is used to achieve this. The program also prints a message saying that the number is negative and keeps an account of negative numbers.

The final output includes the number of positive values evaluated and the number of negative items encountered.

```
Program:
    #include <math.h>
    main()
    {
      int count, negative;
      double number, sqroot;
      printf("Enter 9999 to STOP\n");
      count = 0 ;
      negative = 0 ;
      while (count < = 100)
      {
          printf("Enter a number : ");
          scanf("%lf", &number);
          if (number == 9999)
            break;      /* EXIT FROM THE LOOP */
          if (number < 0)
          {
            printf("Number is negative\n\n");
            negative++ ;
            continue; /* SKIP REST OF THE LOOP */
          }
```

```
          sqroot = sqrt(number);
          printf("Number     = %lf\n Square root = %lf\n\n",
                                    number, sqroot);
          count++ ;
      }
      printf("Number of items done = %d\n", count);
      printf("\n\nNegative items = %d\n", negative);
      printf("END OF DATA\n");
    }
Output
    Enter 9999 to STOP
    Enter a number : 25.0
    Number        = 25.000000
    Square root   = 5.000000
    Enter a number : 40.5
    Number        = 40.500000
    Square root   = 6.363961
    Enter a number : -9
    Number is negative
    Enter a number : 16
    Number        = 16.000000
    Square root   = 4.000000
    Enter a number : -14.75
    Number is negative
    Enter a number : 80
    Number        = 80.000000
    Square root   = 8.944272
    Enter a number : 9999
    Number of items done = 4
    Negative items       = 2
    END OF DATA
```

Fig. 6.11 Use of **continue** statement

Avoiding goto

As mentioned earlier, it is a good practice to avoid using **goto**. There are many reasons for this. When **goto** is used, many compilers generate a less efficient code. In addition, using many of them makes a program logic complicated and renders the program unreadable. It is possible to avoid using **goto** by careful program design. In case any **goto** is absolutely necessary, it should be documented. The **goto** jumps shown in Fig. 6.12 would cause problems and therefore must be avoided.

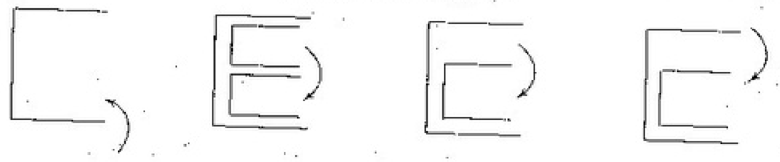


Fig. 6.12 *goto jumps to be avoided.*

Jumping out of the Program

We have just seen that we can jump out of a loop using either the **break** statement or **got** statement. In a similar way, we can jump out of a program by using the library functio **exit( )**. In case, due to some reason, we wish to break out of a program and return to th operating system, we can use the **exit( )** function, as shown below:

```
........
........
if (test-condition) exit(0) ;
........
........
```

The **exit( )** function takes an integer value as its argument. Normally *zero* is used t indicate normal termination and a *nonzero* value to indicate termination due to some err or abnormal condition. The use of **exit( )** function requires the inclusion of the header fil **<stdlib.h>**.

6.6 CONCISE TEST EXPRESSIONS

We often use test expressions in the **if, for, while** and **do** statements that are evaluated and compared with zero for making branching decisions. Since every integer expression has a true/false value, we need not make explicit comparisons with zero. For instance, the expres sion x is true whenever x is not zero, and false when x is zero. Applying! operator, we can write concise test expressions without using any relational operators.

if (expression ==0)

is equivalent to

if(!expression)

Similarly,

if (expression! = 0)

is equivalent to

if (expression)

For example,

if (m%5==0 && n%5==0) is same as **if (!(m%5)&&!(n%5))**

Just Remember

- ✍ Do not forget to place the semicolon at the end of **do ....while** statement.
- ✍ Placing a semicolon after the control expression in a **while** or **for** state ment is not a syntax error but it is most likely a logic error.
- ✍ Using commas rather than semicolon in the header of a **for** statement is an error.
- ✍ Do not forget to place the *increment* statement in the body of a **while** or **do...while** loop.
- ✍ It is a common error to use wrong relational operator in test expressions. Ensure that the loop is evaluated exactly the required number of times.
- ✍ Avoid a common error using = in place of = = operator.
- ✍ Do not change the control variable in both the **for** statement and the body of the loop. It is a logic error.
- ✍ Do not compare floating-point values for equality.
- ✍ Avoid using **while** and **for** statements for implementing exit-controlled (post-test) loops. Use **do...while** statement. Similarly, do not use **do...while** for pre-test loops.
- ✍ When performing an operation on a variable repeatedly in the body of a loop, make sure that the variable is initialized properly before entering the loop.
- ✍ Although it is legally allowed to place the initialization, testing and incre- ment sections outside the header of a **for** statement, avoid them as far as possible.
- ✍ Although it is permissible to use arithmetic expressions in initialization and increment section, be aware of round off and truncation errors during their evaluation.
- ✍ Although statements preceding a **for** and statements in the body can be placed in the **for** header, avoid doing so as it makes the program more difficult to read.
- ✍ The use of **break** and **continue** statements in any of the loops is consid- ered unstructured programming. Try to eliminate the use of these jump statements, as far as possible.
- ✍ Avoid the use of **goto** anywhere in the program.
- ✍ Indent the statements in the body of loops properly to enhance readability and understandability.
- ✍ Use of blank spaces before and after the loops and terminating remarks are highly recommended.
- ✍ Use the function **exit()** only when breaking out of a program is necessary.

Case Studies

1. Table of Binomial Coefficients

Problem: Binomial coefficients are used in the study of binomial distributions and reliability of multicomponent redundant systems. It is given by

$$B(m,x) = \binom{m}{x} = \frac{m!}{x!(m-x)!}, \; m >= x$$

A table of binomial coefficients is required to determine the binomial coefficient for any set of m and x.

Problem Analysis: The binomial coefficient can be recursively calculated as follows:

$$B(m,o) = 1$$

$$B(m,x) = B(m,x-1)\left[\frac{m-x+1}{x}\right], x = 1,2,3,...,m$$

Further,

$$B(o,o) = 1$$

That is, the binomial coefficient is one when either x is zero or m is zero. The program in Fig. 6.12 prints the table of binomial coefficients for m = 10. The program employs one do loop and one **while** loop.

```
Program
  #define MAX 10
  main()
  {
      int m, x, binom;
      printf(" m x");
      for (m = 0; m <= 10 ; ++m)
          printf("%4d", m);
      printf("\n--------------------------------\n");
      m = 0;
      do
      {
          printf("%2d ", m);
          x = 0; binom = 1;
          while (x <= m)
          {
              if(m == 0 || x == 0)
                printf("%4d", binom);
              else
                {
                    binom = binom * (m - x + 1)/x;
                    printf("%4d", binom);
                }
```

```
                    x = x + 1;
                }
                printf("\n");
                m = m + 1;
            }
            while (m <= MAX);
            printf("-------------------------------------\n");
        }
Output
    mx   0   1   2   3   4   5   6   7   8   9  10
  ------------------------------------------------
     0   1
     1   1   1
     2   1   2   1
     3   1   3   3   1
     4   1   4   6   4   1
     5   1   5  10  10   5   1
     6   1   6  15  20  15   6   1
     7   1   7  21  35  35  21   7   1
     8   1   8  28  56  70  56  28   8   1
     9   1   9  36  84 126 126  84  36   9   1
    10   1  10  45 120 210 252 210 120  45  10   1
  ------------------------------------------------
```

Fig. 6.12 *Program to print* ***binomial coefficient*** *table*

2. Histogram

Problem: In an organization, the employees are grouped according to their basic pay for the purpose of certain perks. The pay-range and the number of employees in each group are as follows:

Group	Pay-Range	Number of Employees
1	750 – 1500	12
2	1501 – 3000	23
3	3001 – 4500	35
4	4501 – 6000	20
5	above 6000	11

Draw a histogram to highlight the group sizes.

Problem Analysis: Given the size of groups, it is required to draw bars representing the sizes of various groups. For each bar, its group number and size are to be written.

Program in Fig. 6.13 reads the number of employees belonging to each group and draws a histogram. The program uses four **for** loops and two **if.....else** statements.

```
Program:
    #define  N 5
    main()
    {
        int  value[N];
        int i, j, n, x;
        for (n=0; n < N; ++n)
        {
          printf("Enter employees in Group - %d : ",n+1);
          scanf("%d", &x);
          value[n] = x;
          printf("%d\n", value[n]);
        }
        printf("\n");
        printf("|\n");
        for (n = 0 ; n < N ; ++n)
        {
          for (i = 1 ; i <= 3 ; i++)
          {
            if ( i == 2)
              printf("Group-%1d |",n+1);
            else
              printf("|");
            for (j = 1 ; j <= value[n]; ++j)
              printf("*");
            if (i == 2)
              printf("(%d)\n", value[n]);
            else
              printf("\n");
          }
          printf("|\n");
        }
    }
```

Output

```
    Enter employees in Group - 1 : 12
    12
    Enter employees in Group - 2 : 23
    23
    Enter employees in Group - 3 : 35
    35
    Enter employees in Group - 4 : 20
    20
    Enter Employees in Group - 5 : 11
    11
               |
               |************
    Group-1    |************(12)
               |************
               |
               |***********************
    Group-2    |***********************(23)
               |***********************
               |
               |***********************************
    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;
            cost1 = 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");
}
```

```
Output
                              Y ------>
  0 ----------------------------------------------------------------------
    |                                                                  #
    |                                                        0 --- *
    |                                                   0------ *
    |                                              0-------- *
    |                                          0------ *
    |                                     0------ *
    |                                 0---- *
    |                              0 - *
    |                           #
    |                     * - 0
    |                  *--- 0
X   |              *----- 0
    |          *------ 0
    |       *-------- 0
    |     *------- 0
    |    *------- 0
    |   *------- 0
    |  *------- 0
    |  *------ 0
    |  *----- 0
```

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.

6.16 What is the output of following code?
```
int m = 100, n = 0;
while ( n == 0 )
{
        if ( m < 10 )
                break;
        m = m-10;
```

6.17 What is the output of the following code?
```
int m = 0 ;
do
{
        if (m > 10 )
                continue ;
        m = m + 10 ;
} while ( m < 50 ) ;
printf("%d", m);
```

6.18 What is the output of the following code?

```
int n = 0, m = 1 ;
do
{
        printf(m) ;
        m++ ;
}
while (m <= n) ;
```

6.19 What is the output of the following code?

```
int n = 0, m ;
for (m = 1; m <= n + 1 ; m++ )
        printf(m);
```

6.20 When do we use the following statement?

```
for (; ; )
```

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              (b) * * * * *
    2 2                 * * * *
    3 3 3                * * *
    4 4 4 4               * *
    5 5 5 5 5              *
```

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					
.					
.					
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.

6.16 Write a program to print a square of size 5 by using the character **S** as shown below.

```
(a) S S S S S          (b) S S S S S
    S S S S S              S       S
    S S S S S              S       S
    S S S S S              S       S
    S S S S S              S S S S S
```

6.17 Write a program to graph the function

$$y = \sin(x)$$

in the interval 0 to 180 degrees in steps of 15 degrees. Use the concepts discussed in the Case Study 4 in Chapter 6.

6.18 Write a program to print all integers that are **not divisible** by either 2 or 3 and lie between 1 and 100. Program should also account the number of such integers and print the result.

6.19 Modify the program of Exercise 6.16 to print the character O instead of S at the center of the square as shown below.

```
S S S S S
S S S S S
S S O S S
S S S S S
S S S S S
```

6.20 Given a set of 10 two-digit integers containing both positive and negative values, write a program using **for** loop to compute the sum of all positive values and print the sum and the number of values added. The program should use **scanf** to read the values and terminate when the sum exceeds 999. Do not use **goto** statement.

7

Arrays

7.1 INTRODUCTION

So far we have used only the fundamental data types, namely **char, int, float, double** an variations of **int** and **double**. Although these types are very useful, they are constrained b 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, w need to handle a large volume of data in terms of reading, processing and printing. To proc ess such large amounts of data, we need a powerful data type that would facilitate efficien storing, accessing and manipulation of data items. C supports a derived data type known a *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 simpl a grouping of like-type data. In its simplest form, an array can be used to represent a list o 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 on 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, w 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 tha share a common name. For instance, we can use an array name *salary* to represent a *set o salaries* of a group of employees in an organization. We can refer to the individual salaries b writing a number called *index* or *subscript* in brackets after the array name. For example

```
salary [10]
```

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

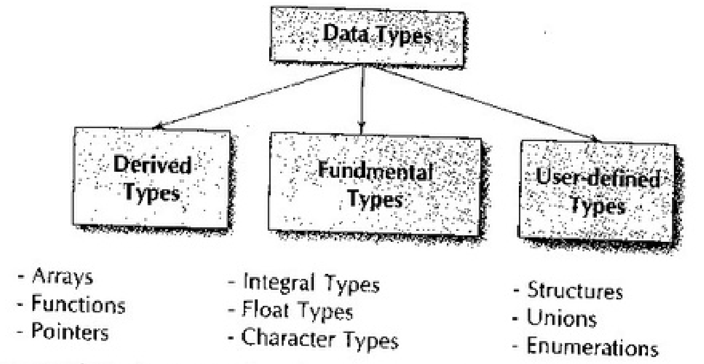
The ability to use a single name to represent a collection of items and to refer to an item by specifying the item number enables us to develop concise and efficient programs. For example, we can use a loop construct, discussed earlier, with the subscript as the control variable to read the entire array,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

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

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

Data Structures

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



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 elemen 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, th following are valid statements:

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

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

7.3 DECLARATION OF ONE-DIMENSIONAL ARRAYS

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

type *variable-name[size];*

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

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

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

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

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

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

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

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

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

"WELL DONE"

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

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

When the compiler sees a character string, it terminates it with an additional null charter. Thus, the element **name[10]** holds the null character '\0'. *When declaring charac 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 fo lowing 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 th sum of their squares.

```
Program
    main()
    {
        int i ;
        float x[10], value, total ;

    /* . . . . . .READING VALUES INTO ARRAY . . . . . . */

        printf("ENTER 10 REAL NUMBERS\n") ;

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

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

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

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

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

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

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

                  Total = 446.86
```

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

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

7.4 INITIALIZATION OF ONE-DIMENSIONAL ARRAYS

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

- At compile time
- At run time

Compile Time Initialization

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Compile time initialization may be partial. That is, the number of initializers may be less than the declared size. In such cases, the remaining elements are initialized 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 t 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 wi 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 initia izing 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 remainin 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 $\mathbf{v}_{ij}$ refers to the value in the ith row and jth 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.

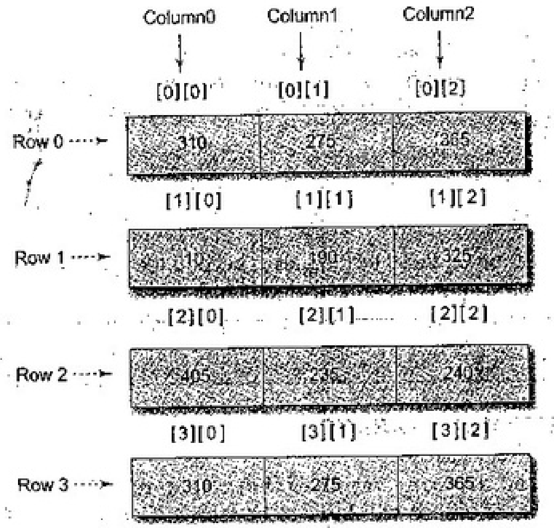


Fig. 7.3 Representation of a two-dimensional array in memory

Example 7.3 Write a program using a two-dimensional array to compute and prin 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 followin equations are used in computing the results:

(a) Total sales by mth 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 ; j++ )
              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 nu bers 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. Ea 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 multiplicatio ble 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 clos 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 t 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 instan 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 r to two, and all other elements to zero.

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

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

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

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

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

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
```

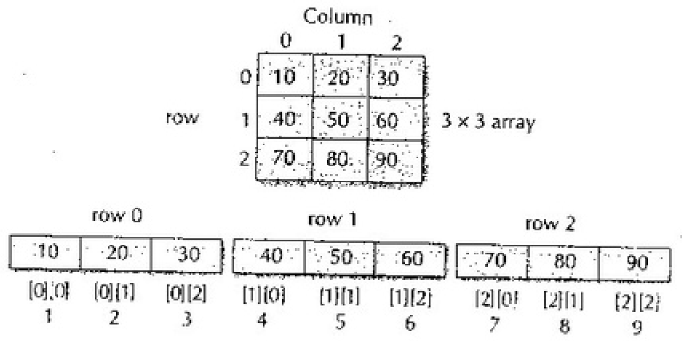
POPULARITY TABLE

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

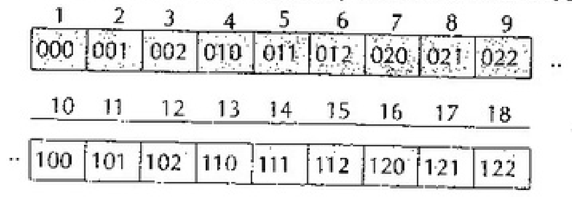
The subscripts in the definition of a two-dimensional array represent rows and columns. This format maps the way that data elements are laid out in the memory. The elements of all arrays are stored contiguously in increasing memory locations, essentially in a single list. If we consider the memory as 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 multi-dimensional array, the order of storage is that the first element stored has 0 in all its subscripts, the second has all of its subscripts 0 except the far right which has a value of 1 and so on.

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



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 compile
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. Sim larly **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 elem **survey[2][3][10]** denotes the rainfall in the month of October during the second year city-3.

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

Year 1

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

Year 2

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

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

7.8 DYNAMIC ARRAYS

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

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

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

7.9 MORE ABOUT ARRAYS

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

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

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

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

Just Remember

- We need to specify three things, namely, name, type and size, when we declare an array.
- Always remember that subscripts begin at 0 (not 1) and end at size –1.
- Defining the size of an array as a symbolic constant makes a program more scalable.
- Be aware of the difference between the "kth element" and the "element k". The kth element has a subscript k-1, whereas the element k has a sub script of k itself.
- Do not forget to initialize the elements; otherwise they will contain "gar bage".
- 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 prop 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 termi nating null character.
- Make sure that the subscript variables have been properly initialized be fore they are used.
- Leaving out the subscript reference operator [ ] in an assignment opera tion is compile time error.
- During initialization of multi–dimensional arrays, it is an error to omit the array size for any dimension other than the first.

Case Studies

1. Median of a List of Numbers

When all the items in a list are arranged in an order, the middle value which divides the items into two parts with equal number of items on either side is called the *median*. Od

number of items have just one middle value while even number of items have two middle values. The median for even number of items is therefore designated as the average of the two middle values.

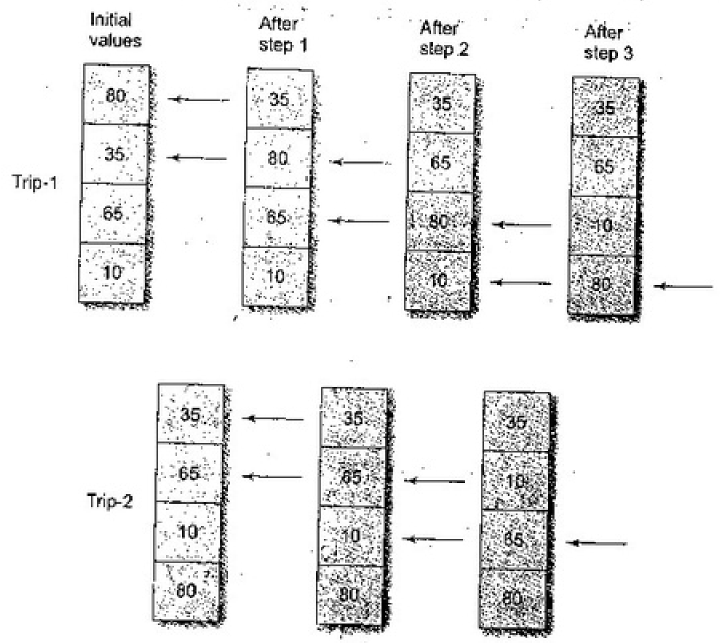
The major steps for finding the median are as follows:

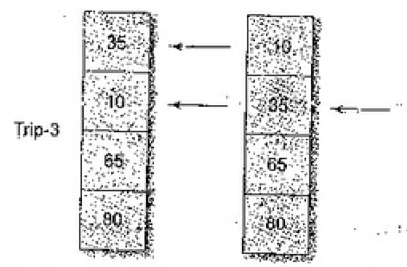
1. Read the items into an array while keeping a count of the items.
2. Sort the items in increasing order.
3. Compute median.

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

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

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





During the first trip, three pairs of items are compared and interchanged wheneve 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 th new list) need not be considered any further. Therefore, trip-2 requires only two pairs to b compared. This time, the number 65 (the second largest value) has been moved down th 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 entir process will be over when a trip contains only one step. If the list contains **n** elements, then the number of comparisons involved would be **n(n–1)/2**.

Program

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

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

Output

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

Median is 3.333000

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

Median is 5.500000
```

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

2. Calculation of Standard Deviation

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

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

where

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

and

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

The algorithm for calculating the standard deviation is as follows:

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

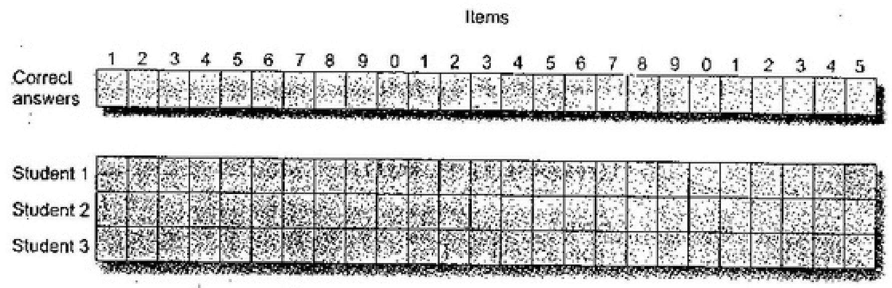
Complete program with sample output is shown in Fig. 7.8.

```
Program
     #include <math.h>
     #define MAXSIZE 100
     main( )
     {
          int i,n;
          float value [MAXSIZE], deviation,
                sum,sumsqr,mean,variance,stddeviation;
          sum = sumsqr = n = 0 ;
          printf("Input values: input -1 to end \n");
          for (i=1; i< MAXSIZE ; i++)
          {
            scanf("%f", &value[i]);
            if (value[i] == -1)
              break;
            sum += value[i];
            n += 1;
          }
          mean = sum/(float)n;
          for (i = 1 ; i<= n; i++)
          {
            deviation = value[i] - mean;
            sumsqr += deviation * deviation;
          }
          variance = sumsqr/(float)n ;
          stddeviation = sqrt(variance) ;
          printf("\nNumber of items : %d\n",n);
          printf("Mean : %f\n", mean);
          printf("Standard deviation : %f\n", stddeviation);
     }
Output
     Input values: input -1 to end
     65 9 27 78 12 20 33 49 -1

     Number of items : 8

     Mean : 36.625000
     Standard deviation : 23.510303
```

Fig. 7.8 *Program to calculate standard deviation*

3. Evaluating a Test

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



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

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

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

key[i]	- To store correct answers of items
response[i]	- To store responses of students
correct[i]	- To identify items that are answered correctly.

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

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

Output

```
    Input key to the items
    abcdabcdabcdabcdabcdabcda

    Input responses of student-1
    abcdabcdabcdabcdabcdabcda

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

    Input responses of student-2
    abcddcbaabcdabcdddddddddd
```

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

    Input responses of student-3
    aaaaaaaaaaaaaaaaaaaaaaaaa

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

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

4. Production and Sales Analysis

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

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

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

M =

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

S =

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

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

C =

C1	C2	C3	C4	C5

where Cj is the cost of jth type product.

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

Mvalue[i][j] = Mij x Cj

Svalue[i][j] = Sij x Cj

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
      for(i=1; i<=4; i++)
        for(j=1;j<=5; j++)
          scanf("%d",&M[i][j]);
      printf (" Enter products sold week_wise\n");
      printf (" S11,S12,—, S21,S22,— etc\n");
      for(i=1; i<=4; i++)
        for(j=1; j<=5; j++)
          scanf("%d", &S[i][j]);
      printf(" Enter cost of each product\n");
        for(j=1; j <=5; j++)
          scanf("%d",&C[j]);
    /*Value matrices of production and sales */
      for(i=1; i<=4; i++)
        for(j=1; j<=5; j++)
        {
          Mvalue[i][j] = M[i][j] * C[j];
          Svalue[i][j] = S[i][j] * C[j];
        }
    /*Total value of weekly production and sales */
      for(i=1; i<=4; i++)
      {
        Mweek[i] = 0 ;
        Sweek[i] = 0 ;
        for(j=1; j<=5; j++)
        {
          Mweek[i] += Mvalue[i][j];
          Sweek[i] += Svalue[i][j];
        }
      }
    /*Monthly value of product_wise production and sales */
      for(j=1; j<=5; j++)
      {
        Mproduct[j] = 0 ;
        Sproduct[j] = 0 ;
        for(i=1; i<=4; i++)
        {
          Mproduct[j] += Mvalue[i][j];
          Sproduct[j] += Svalue[i][j];
        }
      }
    /*Grand total of production and sales values */
      Mtotal = Stotal = 0;
      for(i=1; i<=4; i++)
      {
        Mtotal += Mweek[i];
        Stotal += Sweek[i];
        }
        /***********************************************
          Selection and printing of information required
        ***********************************************/
        printf("\n\n");
        printf(" Following is the list of things you can\n");
        printf(" request for. Enter appropriate item number\n");
        printf(" and press RETURN Key\n\n");
        printf(" 1.Value matrices of production & sales\n");
        printf(" 2.Total value of weekly production & sales\n");
        printf(" 3.Product_wise monthly value of production &");
        printf(" sales\n");
        printf(" 4.Grand total value of production & sales\n");
        printf(" 5.Exit\n");
        number = 0;
        while(1)
        {   /* Beginning of while loop */
          printf("\n\n ENTER YOUR CHOICE:");
          scanf("%d",&number);
          printf("\n");
          if(number == 5)
          {
            printf(" G O O D  B Y E\n\n");
            break;
          }
          switch(number)
          { /* Beginning of switch */
       /* V A L U E   M A T R I C E S */
          case 1:
            printf(" VALUE MATRIX OF PRODUCTION\n\n");
            for(i=1; i<=4; i++)
            {
              printf(" Week(%d)\t",i);
              for(j=1; j <=5; j++)
                printf("%7d", Mvalue[i][j]);
              printf("\n");
            }
            printf("\n VALUE MATRIX OF SALES\n\n");
            for(i=1; i <=4; i++)
            {
              printf(" Week(%d)\t",i);
              for(j=1; j <=5; j++)
                printf("%7d", Svalue[i][j]);
              printf("\n");
            }
```

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

Output

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

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

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

GRAND TOTAL OF PRODUCTION & SALES
```

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

Fig. 7.10 *Program for production and sales analysis*

Review Questions

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

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

7.2 Fill in the blanks in the following statements.

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

7.3 Identify errors, if any, in each of the following array declaration statements, assuming that ROW and COLUMN are declared as symbolic constants:

(a) `int score (100);`
(b) `float values [10,15];`
(c) `float average[ROW],[COLUMN];`
(d) `char name[15];`
(e) `int sum[ ];`
(f) `double salary [i + ROW]`
(g) `long int number [ROW]`
(h) `int array x[COLUMN];`

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

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

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

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

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

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

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

1	0	0	0	0	.....	0
0	1	0	0	0	.....	0
0	0	1	0	0	.....	0
⋮	⋮	⋮	⋮	⋮		⋮
0	0	0	0	0	.....	1

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

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

7.8 Which of the following initialization statements are correct?

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

```
(d) char str4[ ] = {'S', 'U', 'N'};
(e) char str5[10] = "Sun";
```

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

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

11 What is the error in the following program?

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

12 What happens when an array with a specified size is assigned
(a) with values fewer than the specified size; and
(b) with values more than the specified size.

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

14 What is the output of the following program?

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

15 What is the output of the following program?

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

rogramming Exercises

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

$$y = mx + c$$

and the values of m and c are given by

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

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

All summations are from 1 to n.

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

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

Write a program to read the table elements into a two-dimensional array **temper ture**, 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 voting is done by marking the candidate number on the ballot paper. Write a prog to read the ballots and count the votes cast for each candidate using an array varia **count**. In case, a number read is outside the range 1 to 5, the ballot should be con ered as a 'spoilt ballot' and the program should also count the number of spoilt ballo

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 bounda ments) 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 the results.

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

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

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

(a) Total marks obtained by each student.

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

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

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

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

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

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

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

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

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

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

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

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

7.9 Selection sort is based on the following idea:

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

Write a program to implement this algorithm.

7.10 Develop a program to implement the binary search algorithm. This technique compares the search key value with the value of the element that is midway in a "sorted" list. Then;
   (a) If they match, the search is over.
   (b) If the search key value is less than the middle value, then the first half of the list contains the key value.
   (c) If the search key value is greater than the middle value, then the second half contains the key value.

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

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

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

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

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

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

   0-07-041183-2

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

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

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

7.15 Write a program to read two matrices A and B and print the following:
   (a) A + B; and
   (b) A – B.

8 Character Arrays and Strings

8.1 INTRODUCTION

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

"Man is obviously made to think."

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

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

For example,

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

will output the string

" Well Done !"

while the statement

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

will output the string

Well Done !

Character strings are often used to build meaningful and readable programs. The common operations performed on character strings include:
- Reading and writing strings.
- Combining strings together.
- Copying one string to another.
- Comparing strings for equality.
- Extracting a portion of a string.

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

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 sup-plies 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 per-mits 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 termi-nator.

C also permits us to initialize a character array without specifying the number of ele-ments. 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 initializa-tion from declaration. That is,

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

is not allowed. Similarly,

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

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

Terminating Null Character

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

8.3 READING STRINGS FROM TERMINAL

Using scanf Function

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

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

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

NEW YORK

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

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

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

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

Note that the unused locations are filled with garbage.

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

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

with the line of text

NEW YORK

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

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

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

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

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

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

Output
  Enter text :
  Oxford Road, London M17ED

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

  Enter text :
  Oxford-Road, London-M17ED United Kingdom
  word1 = Oxford-Road
  word2 = London-M17ED
  word3 = United
  word4 = Kingdom
```

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

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

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

Here, two things may happen.

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

Consider the following statements:

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

The input string RAM will be stored as:

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

The input string KRISHNA will be stored as:

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

Reading a Line of Text

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

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

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

Using *getchar* and *gets* Functions

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

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

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

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

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

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

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

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

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

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

```
gets (str);
```

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

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

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

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

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

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

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

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

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

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

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

Fig. 8.3 *Copying one string into another*

8.4 WRITING STRINGS TO SCREEN

Using printf Function

We have used extensively the **printf** function with %s format to print strings to the scre
The format %s can be used to display an array of characters that is terminated by the nu
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 s
fication

%10.4

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

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

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

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

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

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

    Uni
    United Kingdom
    -----
```

Fig. 8.4 *Writing strings using %s format*

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

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

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

This feature comes in handy for printing a sequence of characters. Example 8.5 illustrate 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 % are shown in Fig. 8.6, which further illustrates the variable field width and the preci specifications.

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

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

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

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

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

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

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

(a) %12.*s

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

(b) %.*s

```
             C|
             C|
              C|
              C|
-------------
             C|
            C|
           C|
          C|
         C|
        C|
       C|
      C|
     C|
    C|
   C|
  C|
 C|
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 v
ues of character variables. It takes the following form:

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

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

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

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

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

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

```
puts ( str );
```

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

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

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

8.5 ARITHMETIC OPERATIONS ON CHARACTERS

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

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

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

will display the number 97 on the screen.

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

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

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

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

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

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

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

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

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

Then,

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

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

x = atoi(string);

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

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

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

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

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

```
Program
    main()
    {
      char c;
      printf("\n\n");
      for( c = 65 ; c <= 122 ; c = c + 1 )
      {
        if( c > 90 && c < 97 )
          continue;
        printf("|%4d - %c ", c, c);
      }
      printf("|\n");
    }
Output
    | 65 - A | 66 - B | 67 - C | 68 - D | 69 - E | 70 - F
    | 71 - G | 72 - H | 73 - I | 74 - J | 75 - K | 76 - L
    | 77 - M|  78 - N|  79 - O|  80 - P|  81 - Q|  82 - R
    | 83 - S|  84 - T|  85 - U|  86 - V|  87 - W|  88 - X
    | 89 - Y|  90 - Z|  97 - a|  98 - b|  99 - c| 100 - d
    |101 - e| 102 - f| 103 - g| 104 - h| 105 - i| 106 - j
    |107 - k| 108 - l| 109 - m| 110 - n| 111 - o| 112 - p
    |113 - q| 114 - r| 115 - s| 116 - t| 117 - u| 118 - v
    |119 - w| 120 - x| 121 - y| 122 - z|
```

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

8.6 PUTTING STRINGS TOGETHER

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

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

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

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

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

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

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

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

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

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

In the same way, the statement

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

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

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

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

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

Fig. 8.8 Concatenation of strings

8.7 COMPARISON OF TWO STRINGS

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

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

are not permitted. It is therefore necessary to compare the two strings to be tested, charact by character. The comparison is done until there is a mismatch or one of the strings term nates 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 used to carry out many of the string manipulations discussed so far. Following are the mo commonly used string-handling functions.

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

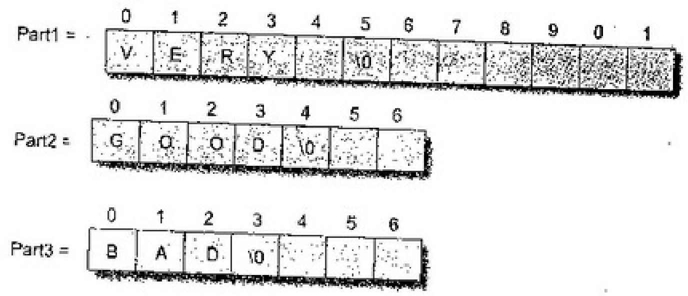
We shall discuss briefly how each of these functions can be used in the processing of strings.

strcat() Function

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

strcat(string1, string2);

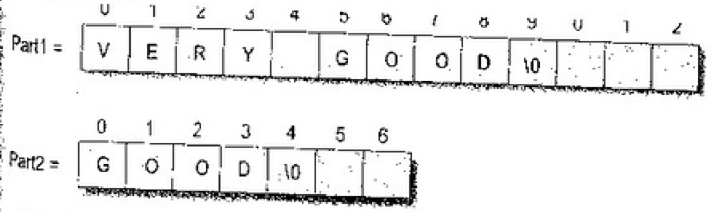
string1 and **string2** are character arrays. When the function **strcat** is executed, **string2** is appended to **string1**. It does so by removing the null character at the end of **string1** and placing **string2** from there. The string at **string2** remains unchanged. For example, consider the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



Execution of the statement

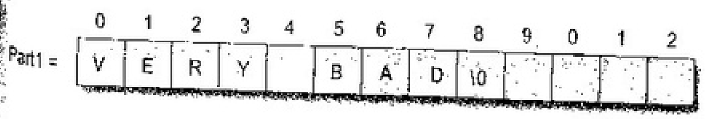
strcat(part1, part2);

will result in:



while the statement

will result in:



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

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

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

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

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

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

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

strcmp() Function

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

```
strcmp(string1, string2);
```

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

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

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

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

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

strcpy() Function

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

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

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

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

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

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

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

strlen() Function

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

n = strlen(string);

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

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

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

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

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

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

The program outputs all the three strings with their lengths.

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

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

```
        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 fun tion **strncpy** that copies only the left-most n characters of the source string to the targ 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 plac 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 para eters as illustrated in the function call below:

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

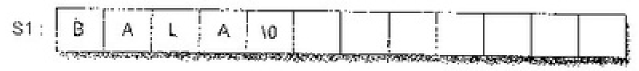
this compares the left-most n characters of **s1** to **s2** and returns.

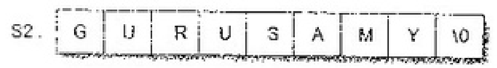
(a) 0 if they are equal;
(b) negative number, if s1 sub-string is less than s2; and
(c) positive number, otherwise.

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

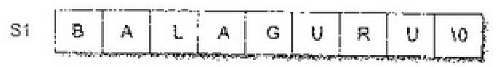
```
strncat (s1, s2, n);
```

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, lis of the names of employees in an organization, list of places, etc. A list of names can be treat as a table of strings and a two-dimensional character array can be used to store the enti list. For example, a character array **student[30][15]** may be used to store a list of 30 nam 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 de ration:

```
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);
```

8.11 What will be the output of the following segment?

```
char s1[ ] = "Kolkotta" ;
char s2[ ] = "Pune" ;
strcpy (s1, s2) ;
printf("%s", s1) ;
```

8.12 What will be the output of the following segment?

```
char s1[ ] = "NEW DELHI" ;
char s2[ ] = "BANGALORE" ;
strncpy (s1, s2, 3) ;
printf("%s", s1) ;
```

8.13 What will be the output of the following code?

```
char s1[ ] = "Jabalpur" ;
char s2[ ] = "Jaipur" ;
printf(strncmp(s1, s2, 2) );
```

8.14 What will be the output of the following code?

```
char s1[ ] = "ANIL KUMAR GUPTA";
char s2[ ] = "KUMAR";
printf (strstr (s1, s2) );
```

8.15 Compare the working of the following functions:

(a) strcpy and strncpy;

(b) strcat and strncat; and

(c) strcmp and strncmp.

rogramming 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 pr gram in PASCAL language."
8.7 A Maruti car dealer maintains a record of sales of various vehicles in the followin 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 particular vehicle sold during a specified period. The program should request the us 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 th string is a *palindrome* or not. (A string is a palindrome if it can be read from left an right with the same meaning. For example, Madam and Anna are palindrome strin Ignore capitalization).
8.9 Write program that reads the cost of an item in the form RRRR.PP (Where RRR denotes Rupees and PP denotes Paise) and converts the value to a string of words th expresses the numeric value in words. For example, if we input 125.75, the outp 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 th 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)
8.11 Write a program to read two strings and compare them using the function **strncm** and print a message that the first string is equal, less, or greater than the secon
8.12 Write a program to read a line of text from the keyboard and print out the numb occurrences of a given substring using the function **strstr ( )**.
8.13 Write a program that will copy m consecutive characters from a string s1 beginnin position n into another string s2.

8.14 Write a program to create a directory of students with roll numbers. The program should display the roll number for a specified name and vice-versa.
8.15 Given a string

char str [ ] = "123456789" ;

Write a program that displays the following:

```
    1
   2 3 2
  3 4 5 4 3
 4 5 6 7 6 5 4
5 6 7 8 9 8 7 6 5
```