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 shown 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 controlled by the variable **row** and executed 12 times. The inner loop is controlled by the variable **column** and is executed 10 times, each time the outer loop is executed. That is, the inner loop is 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 condition is *true*, the body of the loop is executed; otherwise the loop is terminated and the execution continues with the statement that immediately follows the loop.
3. When the body of the loop is executed, the control is transferred back to the **for** statement after evaluating the last statement in the loop. Now, the control variable is *incremented* using an assignment statement such as i = i+1 and the new value of the control variable is again tested to see whether it satisfies the loop condition. If the condition is satisfied, the body of the loop is again executed. This process continues till the value of the control variable fails to satisfy the test-condition.

Consider the following segment of a program:

```
        ┌→ for ( x = 0 ; x <= 9 ; x = x+1)
loop    │  {
        │      printf("%d", x);
        └─ }
           printf("\n");
```

This **for** loop is executed 10 times and prints the digits 0 to 9 in one line. The three sections enclosed within parentheses must be separated by semicolons. Note that there is no semicolon at the end of the *increment* section, x = x+1.

The **for** statement allows for *negative increments*. For example, the loop discussed above can be written as follows:

```
for ( x = 9 ; x >= 0 ; x = x-1 )
    printf("%d", x);
printf("\n");
```

This loop is also executed 10 times, but the output would be from 9 to 0 instead of 0 to 9. Note that braces are optional when the body of the loop contains only one statement.

Since the conditional test is always performed at the beginning of the loop, the body of the loop may not be executed at all, if the condition fails at the start. For example,

```
for (x = 9; x < 9; x = x-1)
    printf("%d", x);
```

will never be executed because the test condition fails at the very beginning itself.

Let us again consider the problem of sum of squares of integers discussed in Section 6.1. This problem can be coded using the **for** statement as follows:

```
-----------------
sum = 0;
for (n = 1; n <= 10; n = n+1)
{
        sum = sum+ n*n;
}
printf("sum = %d\n", sum);
-----------------
```

The body of the loop

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

is executed 10 times for n = 1, 2, ....., 10 each time incrementing the **sum** by the square of the value of n.

One of the important points about the **for** loop is that all the three actions, namely *initialization, testing*, and *incrementing*, are placed in the **for** statement itself, thus making them visible to the programmers and users, in one place. The **for** statement and its equivalent of **while** and **do** statements are shown in Table 6.1.

**Table 6.1** *Comparison of the Three Loops*

| *for* | *while* | *do* |
|---|---|---|
| for (n=1; n<=10; ++n) | n = 1; | n = 1; |
| { | **while** (n<=10) | **do** |
| ________ | { | { |
| ________ | ________ | ________ |
| } | ________ | ________ |
| | n = n+1; | n = n+1; |
| | } | } |
| | | **while**(n<=10); |

**Example 6.3** The program in Fig. 6.4 uses a **for** loop to print the "Powers of 2" table for the power 0 to 20, both positive and negative.

The program evaluates the value

$$p = 2^n$$

successively by multiplying 2 by itself n times.

$$q = 2^{-n} = \frac{1}{p}$$

Note that we have declared **p** as a ***long int*** and **q** as a **double**.

## Additional Features of for Loop

The **for** loop in C has several capabilities that are not found in other loop constructs. For example, more than one variable can be initialized at a time in the **for** statement. The statements

```
p = 1;
for (n=0; n<17; ++n)
```

can be rewritten as

```
for (p=1, n=0; n<17; ++n)
```

```
Program
  main()
  {
    long int p;
    int n;
    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*

Notice 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)
    ;
```

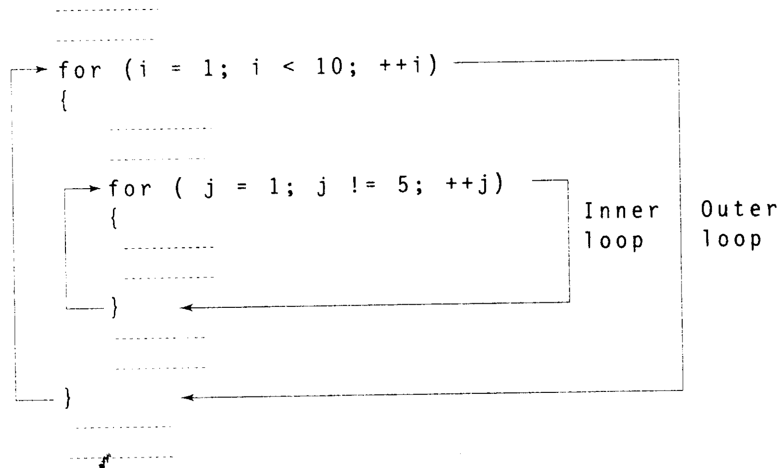
This loop is executed 1000 times without producing any output; it simply causes a time delay. Notice that the body of the loop contains only a semicolon, known as a *null* statement. This can also be written as

```
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:




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 desirable to skip a part of the loop or to leave the loop as soon as a certain condition occurs. For 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 terminated 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 exiting 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.

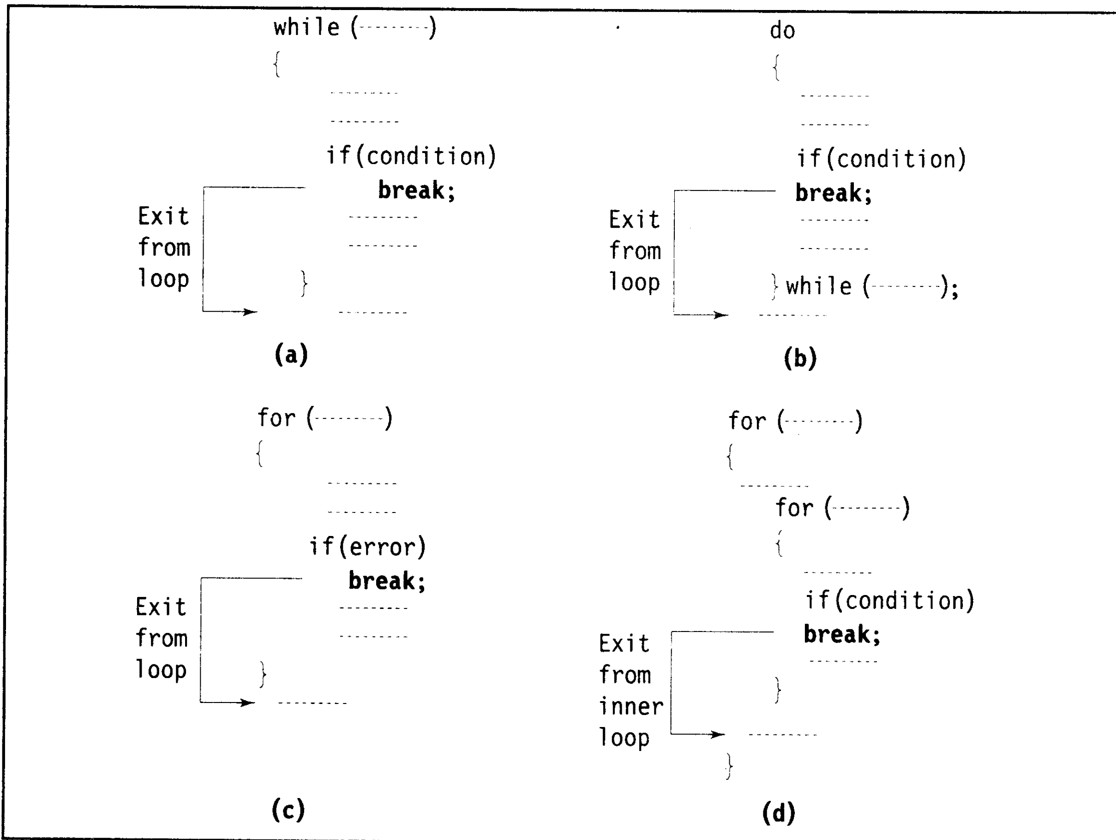



**Fig. 6.6** *Exiting a loop with **break** statement*

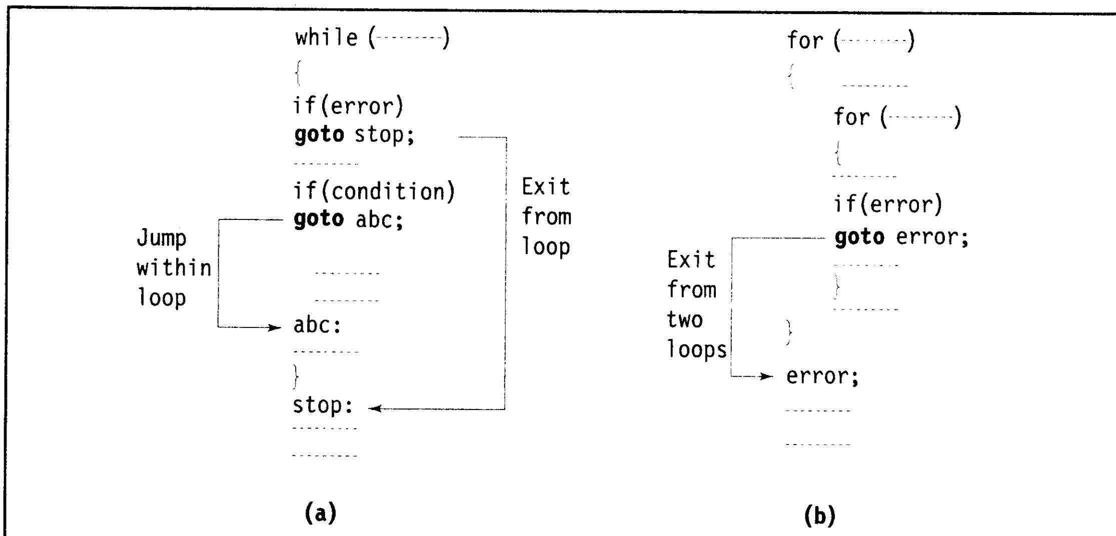



**Fig. 6.7** *Jumping within and exiting from the loops with **goto** statement*

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

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

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.

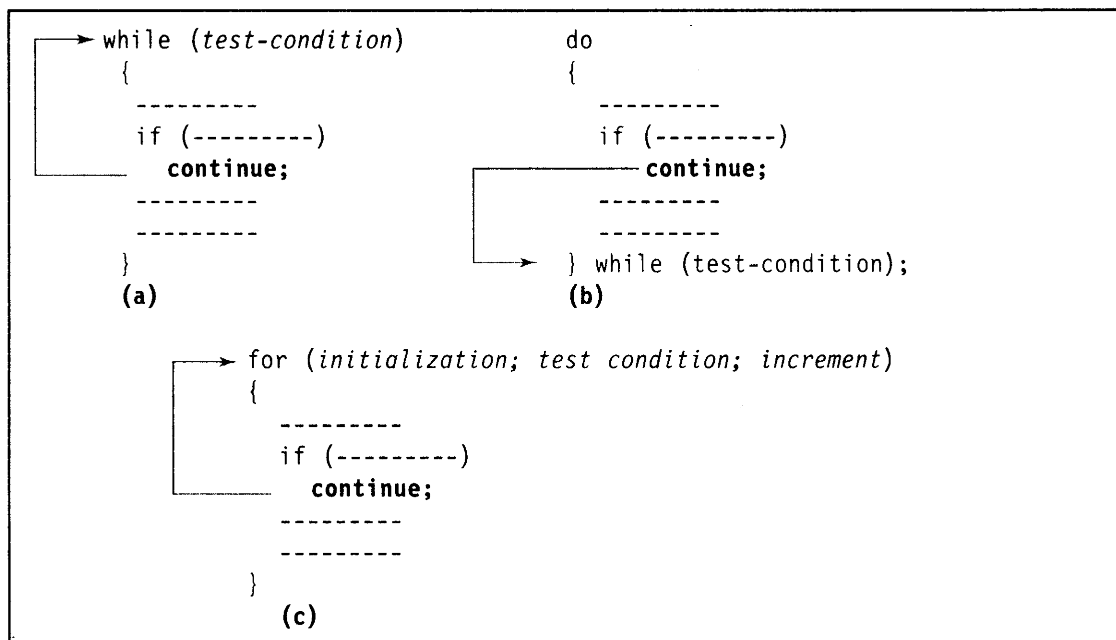



**Fig. 6.10** *Bypassing and continuing in loops*

**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**

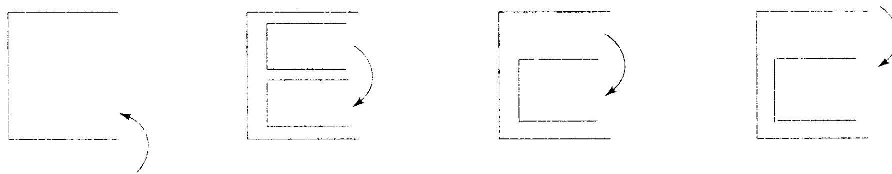
```
Enter 9999 to STOP
Enter a number : 25.0

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 **gotos**. 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 following **goto** jumps would cause problems and therefore must be avoided.



## 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 expression 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** statement 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 increment 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 considered 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.

## 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;
```