Chapter **12**

LOOP CONSTRUCTS

12.1 OVERVIEW

We often encounter problems whose solution may require executing a statement or a set of statements repeatedly. In such situations, we need a structure that would allow repeating a set of statements up to fixed number of times or until a certain criterion is satisfied. In C, Loop statements fulfill this requirement. This chapter will provide the basis for writing iterative solutions to certain problems. Here, we shall introduce different loop constructs available in C.

Iteration is the third type of program control structure (sequence, selection, iteration), and the repetition of statements in a program is called a *loop*. There are three loop control statements in C, these are:

- *while*
- *do-while*
- *for*

12.2 WHILE STATEMENT

The *while* loop keeps repeating associated statements until the specified condition becomes false. This is useful where the programmer does not know in advance how many times the loop will be traversed. The syntax of the *while* statement is:

```
while (condition)
{
        statement(s);
}
```

The *condition* in the *while loop* controls the loop iteration. The statements, which are executed when the given condition is true, form the *body of the loop*. If the condition is *true*, the body of the loop is executed. As soon as it becomes false, the loop terminates immediately.

Example 1 **Write a program to print digits from 1 to 10.**

```
#include <stdio.h>

void main(void)
{
     int count;

     count = 1;
     while( count <= 10)
     {
          printf("%d\n", count);
          count = count + 1;
     }
}
```

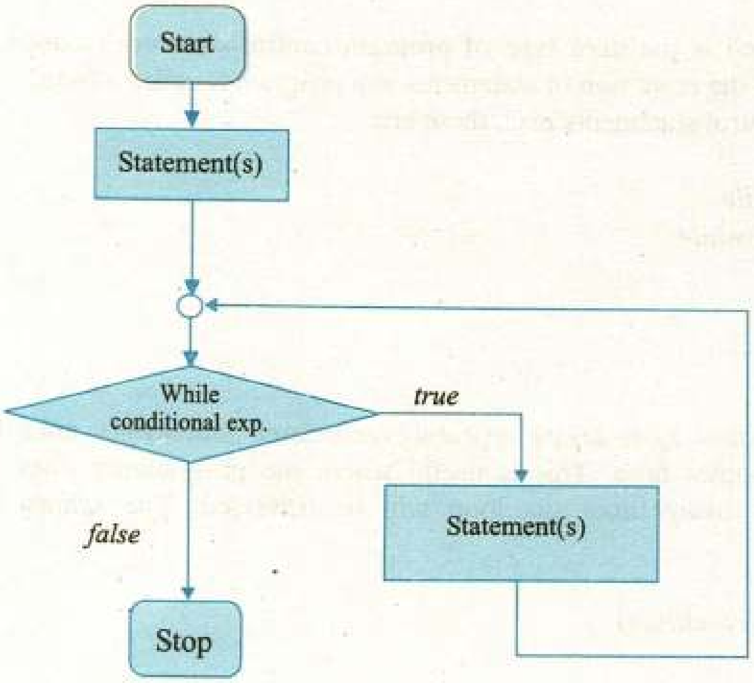

Fig. 12.1 Flowchart of the while loop

This is a simple program which demands iterative solution. It does not make sense to use ten printf statements to print ten digits; if so, what if we have to print digits from 1 to 1000? Should we write one thousand printf statements to accomplish the task? Certainly not, the right way to come up to the solution is to use a loop, which would execute ten times. Each time the loop executes, a number is printed which is incremented by one for every iteration until the required list of numbers is printed.

In this program, we use a variable *count* which is initialized to 1. The condition (count <= 10) depends on the value of this variable. Until the condition is *true*, the control will enter the *body of the loop*, and as soon as it becomes *false*, the control will exit from the loop and will jump to the next statement to the body of the loop. First time, when the condition is checked, it is found *true* as the value of *count* which is one, is less than ten. The control enters the body of the loop and the number "1" is printed. The next line of code increments the value of *count* by one, which becomes "2". After that, the control will immediately jump to the *while* statement where again the condition is tested which is still found *true*, as 2 is less than 10. The control again enters the body of the loop, and the number "2" is printed. The value of the variable *count* again increases by one and becomes "3". The control again transfer to the *while* statement. This process continues until the value of *count* becomes "11", making the condition *false*. When the condition becomes *false*, the control will exit from the loop.

The *count* is the *loop control variable*. A variable whose value controls the number of iterations is known as *loop control variable*. The compound statement, which is enclosed in braces, is the *body of the loop*.

In *while loop*, the *loop control variable* is always initialized outside the *body of the loop* and is incremented or decremented inside the *loop body*.

12.3 DO-WHILE LOOP

This is very similar to the *while loop* except that the test occurs at the end of the loop body. This guarantees that the loop is executed at least once. This loop is frequently used where data is to be read; the test then verifies the data, and loops back to read again if it was unacceptable. The syntax of the do-while statement is:

```
do
{
        statement(s);
}while (condition);
```

The important point about this loop is that unlike *while* loop, it ends with a semicolon. Omitting the semicolon will cause a syntax error. Let us re-write the program in *example 1* using *do-while* loop.

```
#include <stdio.h>

void main(void)
```

```
{
        int count;
        count = 1;
        do
        {
                printf("%d\n", count);
                count = count + 1;
        } while( count <= 10);
    }
```

Here, we achieve the same objective as in previous example but in a different way. The keyword do let the program flow to move into the *body of the loop* without checking any test condition. It means, whatever is written in the *loop body* always will be executed at least once. At the completion of execution of the *body of the loop*, the test condition is checked. If it is found *true*, the control is transferred to the first statement in the *body of the loop*, and if the condition is evaluated to *false*, the loop terminates immediately and the control moves to the very next instruction outside the loop.

The *do-while* loop is of great importance in situations where we need to execute certain statements at least once.

Example 2 **Your telephone connection may be in any of two states i.e., working or dead. Write a program that reads the current state of the telephone line; the user should enter w for working state and d for dead state. Any input, other than *w* or *d*, will be considered invalid. We want to force the user to enter a valid input value. This could be achieved by using a *do-while* loop. Let us consider the following program:**

```
#include <stdio.h>
void main( void )
{
        char state;
        do
        {
            printf("\nPlease enter the current state
            of the telephone line (enter \'w\' for
            working and \'d\' for dead) >");
            scanf("%c", &state);

        }while (state != 'w' && state != 'd');
    }
```

This program demonstrates a scenario where an invalid input is not processed. Until the user enters a valid input (d or *w*), the program repeatedly shows him (or her) the message for the valid input to be entered.

Here, the key point is the correct understanding of the test condition (state != 'w' && state != 'd'). It is a compound condition which is comprised of two sub conditions i.e. state != 'w' and state != 'd'. It should be noted that if two or more conditions are combined using logical AND operator to form a compound condition, the compound condition will be *true* only if all the sub conditions are *true* and if any of the sub conditions is *false*, the compound condition evaluates to *false*. Therefore, when the user enters an invalid input (suppose *e*), the first sub condition state != 'w' evaluates to *true* (because *e* is not equal to *w*), similarly the second sub condition state != 'd' also evaluates to *true*. Since both the sub conditions are true, therefore the compound condition also evaluates to *true* and the control flow returns back to the *printf* statement in the *body of the loop*. This process continues until the user enters a *w* or *d*. When the user enters a *d* or a *w*, one of the sub conditions evaluate to *false* causing the compound condition to be evaluated to *false* and the control flow exit the loop.

while vs do-while

- In *while* loop, the *body of the loop* may or may not execute depending on the evaluation of the test condition.
- In *do-while* loop, first the *body of the loop* is executed and then the test condition is checked. Hence it always executed at least once.

The *for* statement is another way of implementing loops in C. Because of its flexibility, most programmers prefer the *for* statement to implement loops. The syntax of the *for* loop is as follows:

for (*initialization expression; test condition; increment/decrement expression*)
{
    statement(s);
}

There are three expressions in *for* loop statement, these are

- Initialization of the loop control variable
- Test condition
- Change (increment or decrement) of the loop control variable

The *initialization expression* is executed in only the first iteration. Then the *loop condition* is tested. If it is *true*, the statements in the body of the loop are executed. After execution of the body of the loop, the increment/decrement expression is evaluated. It is very important to note that the initialization expression is only executed for the first iteration. For second and next iterations, the loop condition is tested, if it is true then the body of the loop is executed and then the increment/decrement expression is evaluated. After evaluation of the increment/decrement expression, the test condition is checked again and if it is true then the body of the loop executed. This process continues as long as the *loop condition* is true. When this condition is found to be *false*, the *for loop* is terminated, and the control transfers to the next statement following the *for loop*. Usually, we increment or decrement the loop control variable in the increment/decrement expression. Let us re-write the program in *example1* using *for* loop.

```
#include <stdio.h>
void main(void)
{
     int count;
     for (count = 1; count <= 10; count++)
          printf("%d\n", count);

}
```

There are three expressions in *for* loop statement, separated by semicolons. Two of the expressions i.e. *initialization expression* and *increment/decrement expression* are optional. We may omit these expressions. In this case, the *for* statement will be written as follows:

```
for (; count <= 10;)
```

The loop condition is mandatory. This can not be omitted. In this case we must have to initialize the loop control variable outside the *for* statement and it should be incremented or decremented inside the *loop body*.

12.4 NESTED LOOP

Nested loop means a loop inside the body of another loop. Nesting can be done up to any level. But, as the level of nesting increases, the complexity of the *nested loop* also increases. There is no restriction on the type of loops (while, do-while, or for) that may be placed in the body of other loops. For example, we can place one or more *while* or *do-while* loops in the body of *for loop*. Similarly, one or more *for loops* can be placed in the body of while or do-while loop.

Example 3 **Write a program that will print asterisks (*) according to the pattern shown in the fig. 12.2.**

```
# include <stdio.h>

void main(void)
{
    int inner;

    for(int outer=7; outer>=1; outer--)
    {
        inner = 1;
        while( inner <= outer)
        {
            printf("*");
            inner++;
        }
        printf("\n");
    }
}
```

```
*******
******
*****
****
***
**
*
```

Fig. 12.2 asterisks pattern

In this program, a *while* loop is used inside the body of *for* loop, which shows a nested loop. The outer loop is controlled by the loop control variable i.e., *outer*. The outer loop is executed seven times. For each iteration of the outer loop, the inner loop executes until the value of the inner loop control variable i.e., *inner* is less than or equal to the value of the variable *outer*. It should be noted that each time a new iteration for the outer loop starts, the variables used in the inner loop are re-initialized and re-processed.

For the first iteration of the outer loop, the variable '*outer*' is initialized to 7, and in all next iterations it is decremented by 1. This process continues until the value of the variable is greater than or equal to 1. For the first iteration of the outer loop, the inner loop executes seven times, and for the 2nd iteration it executes six times, similarly for the last seventh iteration, the inner loop executes just one time. Each time when the inner loop is terminated, the statement `printf("\n")` moves the cursor to the start of the new line.

Note:

Many programs require a list of items to be entered by the user. Often, we don't know how many items the list will have. For example, to find the average marks of a class, we have to input the marks of every student of the class. Similarly to calculate the sum of a series, we have to input the list of numbers in the series. There are so many other situations where the solution demands to enter a list of items to process. Loops are very useful to develop solutions for such problems. Each time the loop body is repeated, one or more data items are input. But, often we don't know how many data items will be input by the user. Therefore, we must find some way to signal the program to stop reading and processing new data.

One way to do this is to instruct the user to enter a unique data value, called a *sentinel value*, after the last data item. The loop condition tests each data item and causes loop exit when the sentinel value is read. Choose the sentinel value carefully; it must be a value that could not normally occur as data. The general form of a sentinel-controlled loop is:

1. Get the first line of data
2. While the sentinel value has not been encountered
3. Process the data line
4. Get another line of data

Sentinel Value is an end marker that follows the last item in a list of items

Example 4 **Write a program to find the average marks of the students in a class.**

```
#include <stdio.h>

void main(void)
{
    int sum = 0, marks, total_students = 0;
    float average;
    do{
        printf("Enter marks of the student (or any -ve
        number to quit) >");
        scanf("%d", &marks);

        if (marks >= 0)
```

```
            {
                total_students++;
                sum += marks;
            }
        } while(marks >= 0);
        if (total_students > 0)
        {
            average = sum / (float)total_students;
            printf("The average marks of the class are:
            %f\n", average);
        }
        else
            printf("Please enter the marks of at least one
            student to calculate average\n");
}
```

This program demonstrates a typical implementation of sentinel loop. Size of the class does not matter, whatever it is, the average will be calculated in the same way. Here *any negative number* may act as the sentinel value because no student can have negative marks. However, zero would not be a wise option because there can be a student with zero marks.

The program reads the marks until the user enters a negative number. For every valid input (zero and +ve numbers) the control switches to the body of the while loop. In the loop body, the *total_students* is incremented by one and the *sum* is accumulated. As soon as a negative number is entered, the *sentinel while* loop is terminated. The next line to the end of the *while loop* is an *if* statement, which checks the count for total students i.e., *total_students* to ensure that the marks of at least one student have been entered. Omitting this *if* statement may crash the program. It is because of the formula for calculating average where the *sum* is divided by *total_students*. When marks of any students are not entered, the value of *total_students* is zero and calculating average for zero students will result in a runtime error of division by zero. So, to avoid this possible error first the value of the variable *total_students* is cheked; if its greater than zero then the average is calculated otherwise a message is shown to the user to enter the marks of at least one student. Now, notice the average formula i.e.,

```
        average = sum / (float)total_students;
```

We have used the keyword *float* in parenthesis before the variable *total_students*. The reason is that both the variables *sum* and *total_students* are integers. So, their division will be integral division in which the fractional part is truncated. Hence, the result will not be accurate. Writing *float* in parenthesis i.e. (*float*) before the integer variable name (i.e. *total*_students) causes the integer variable to temporarily act as a float variable for this particular calculation. It is done to preserve the fractional part in the result. The integer variable (*total_students*) will act as an integer for all other calculations. The effect of this change is strictly associated with that particular calculation. This phenomenon is known as *type casting*.

12.5 GOTO STATEMENT

The *goto* statement performs an unconditional transfer of control to the named label. The label must be in the same function. A label is meaningful only to a goto statement; in any other context, the labeled statement is executed without regard to the label.

The general form of the goto statement is as follows:

goto label;

....

....

label: statement

Example 5 Write a program to calculate the square root of a positive number. Also handle negative numbers properly.

```
#include <math.h>
#include <stdio.h>

void main()
{
     float num;

positive:
     printf("Please Enter a positive number: ");
     scanf("%f", &num);

     if (num < 0)
          goto positive;
     else
          printf("Square root of %0.2f is %0.2f", num,
          sqrt(num));
}
```

If the user enters a negative number, the control transfers to the label *positive*.

Exercise 12c

1. Fill in the blanks:
   - (i) There are _______________ types of loop in C.
   - (ii) The loop condition controls the loop _______________.
   - (iii) In _______________ loop, first the body of the loop is executed and then the test condition is checked.
   - (iv) _________________ means a loop within the body of another loop.
   - (v) The ___________ statement performs an unconditional transfer of control to the named label.
   - (vi) Repetition of statements in a program is called _______________.
   - (vii) There are _________________ expressions in for loop statement.
   - (viii) The body of the while loop executes only if the specified condition is ____________.
   - (ix) Increase in the level of nesting increases the _______________ of the nested loop.
   - (x) A _______________ is meaningful only to a goto statement.
2. Write T for true and F for false statement.
   - (i) There is no difference between while and do-while loop.
   - (ii) The body of a while loop may or may not execute.
   - (iii) The do-while loop always executes at least once.
   - (iv) var++ is an example of prefix increment.
   - (v) The condition of an infinite loop never becomes true.
   - (vi) Initialization expression is optional in *for* loop.
   - (vii) `for( i = 1; i <= 10; i++);` is an infinite loop.
   - (viii) Loop is a decision making construct.
   - (ix) A *while* loop can not be used in the body of a *for* loop.
   - (x) In type casting, a variable of one type behaves as the variable of another type temporarily.

3. Define a loop. How many loops are available in C? Compare the following loops:
   a) while loop and do-while loop
   b) while loop and for loop

4. What is a sentinel controlled loop and how it is implemented? Discuss some of the situations where it can be useful.

5. Write the output of the following program fragments:
   (a)
```
k = 0;
        while (k <= 5)
        {
                printf("%3d %3d\n", k, 10 – k)
                k++;
        }
```

   (b) Trace the output of the following piece of code.
```
        j = 10;
        for (int i = 1; i <= 5; ++i)
        {
                printf("%d  %d\n", i, j);
                j -= 2;
        }
```

6. Correct the following code segments according to the given instructions:
   a) Insert braces where they are needed and correct errors if any. The corrected code should accept five integers and should display their sum.

```
count = 0;
while (count <= 5);
count += 1;
printf("Next Number >");
scanf("%d", &next_num);
next_num += sum;
printf("%d numbers were added; \n", count);
printf("Their sum is %d.\n", sum);
```

b) Rewrite the following code segment using a do-while statement

```
sum = 0;
for (odd = 1; odd < n; odd = odd + 2)
        sum = sum + odd;
printf("Sum of the positive odd numbers less than %d is %d\n", n,
sum);
```

7. Trace the output of the following program segments, assuming m is 3 and n is 5:

a)

```
for ( k = 1; k <= n;  ++k)
{
        for (j = 0; j < k; ++j)
        {
                printf("*");
        }
        printf("\n");
}
```

b)

```
for (k = n; k > 0; --k)
{
        for (j = m; j > 0; --j)
        {
                printf("*");
        }
        printf("\n");
}
```

c) (a) Re-write the program in example3 by replacing the inner *while* loop with

- a *for* loop
- a *do-while* loop

(b) Re-write the program in example3 by replacing the outer *for* loop with

- a *while* loop
- a *do-while* loop

9. Write a program that inputs a number and displays the message "Prime number" if it is a prime number, otherwise displays "Not a prime number".

10. Write a program that displays the first 15 even numbers.

11. Write a program that inputs a number, and displays its table according to the following format:

    Suppose the number entered is 5, the output will be as follows:

```
        5 * 1 = 5
        5 * 2 = 10
        5 * 3 = 15
.
.
.
5 * 10 = 50
```

12. Write a program using do-while loop that repeatedly prompts for and takes input until a value in the range 0 through 15 inclusive is input. The program should add all the values before exiting the loop and displays their sum at the end.

13. Write a program that produces the following output:

```
0
0  1
0  1  2
0  1  2  3
0  1  2  3  4
0  1  2  3  4  5
```

14. Write a program the produces the following output:

```
0     1
1     2
2     4
3     8
4     16
5     32
6     64
```