TOPIC 4: CONTROL STRUCTURES (LOOPING)

At the end of the chapter, the students should be able to:

- *understand the requirements of a loop*
- *understand the loop control variable*
- *program loops with while, for and do-while statements*
- *understand counter-controlled loop and sentinel-controlled loop*
- *learn the use of break and continue statement in a loop*

C++ has three repetition or looping structures that let you **repeat statements over and over until certain conditions are met**. This chapter introduces all **three looping (repetition) structures**. The next section discusses the first repetition structure, called the *while* loop.

4.1 WHILE LOOP REPETITION STRUCTURE

The general form of the *while* statement is:

```
while (expression)
    statement
```

In C++, *while* is a reserved word. Of course, the statement can be either a simple or compound statement. The expression acts as a decision maker and is usually a logical expression. The statement is called the body of the loop. Note that the parentheses around the expression are part of the syntax.

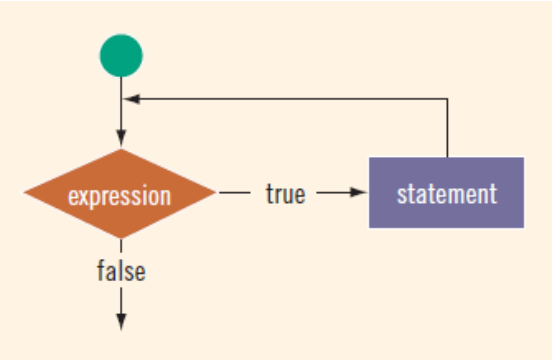

Figure 4.1 The *while* loop

The expression provides an entry condition. If it initially evaluates to *true*, the statement executes. The loop condition—the expression—is then reevaluated. If it again evaluates to *true*, the statement executes again. The statement (body of the loop) continues to execute until the expression is no longer true. A loop that continues to execute endlessly is called an **infinite loop**. To avoid an infinite loop, make sure that the loop's body contains statement(s) that assure that the exit condition—the expression in the while statement—will eventually be *false*.

Consider the following C++ program segment: (Assume that `i` is an `int` variable.)

```
i = 0;                      //Line 1

while (i <= 20)             //Line 2
{
    cout << i << " ";       //Line 3
    i = i + 5;              //Line 4
}

cout << endl;
```

Sample Run:

```
0 5 10 15 20
```

Example 4.1 The *while* loop

4.1.1 COUNTER-CONTROLLED WHILE LOOP

Suppose you **know exactly how many times** certain statements need to be executed. For example, suppose you know exactly how many pieces of data (or entries) need to be read. In such cases, the *while* loop assumes the form of a **counter-controlled *while* loop**.

Suppose the input is:

```
8 9 2 3 90 38 56 8 23 89 7 2
```

Suppose you want to add these numbers and find their average. Consider the following program:

```
//Program: Counter-Controlled Loop

#include <iostream>

using namespace std;

int main()
{
    int limit;    //store the number of data items
    int number;   //variable to store the number
    int sum;      //variable to store the sum
    int counter;  //loop control variable

    cout << "Line 1: Enter the number of "
         << "integers in the list: ";          //Line 1
    cin >> limit;                              //Line 2
    cout << endl;                              //Line 3
```

```
    sum = 0;                                          //Line 4
    counter = 0;                                      //Line 5

    cout << "Line 6: Enter " << limit
         << " integers." << endl;                     //Line 6

    while (counter < limit)                           //Line 7
    {
        cin >> number;                                //Line 8
        sum = sum + number;                           //Line 9
        counter++;                                    //Line 10
    }

    cout << "Line 11: The sum of the " << limit
         << " numbers = " << sum << endl;             //Line 11

    if (counter != 0)                                 //Line 12
        cout << "Line 13: The average = "
             << sum / counter << endl;                //Line 13
    else                                              //Line 14
        cout << "Line 15: No input." << endl;         //Line 15

    return 0;                                         //Line 16
}
```

Sample Run: In this sample run, the user input is shaded.

```
Line 1: Enter the number of integers in the list: 12

Line 6: Enter 12 integers.
8 9 2 3 90 38 56 8 23 89 7 2
Line 11: The sum of the 12 numbers = 335
Line 13: The average = 27
```

Example 4.2 The counter-controlled *while* loop

4.1.2 SENTINEL-CONTROLLED WHILE LOOP

You do not always know how many pieces of data (or entries) need to be read, but you may know that the **last entry is a special value**, called a **sentinel**. In this case, you read the first item before the *while* statement. If this item does not equal the sentinel, the body of the *while* statement executes. The *while* loop continues to execute as long as the program has not read the sentinel. Such a while loop is called a **sentinel-controlled while loop**.

```
//Program:  Sentinel-Controlled Loop

#include <iostream>

using namespace std;

const int SENTINEL = -999;

int main()
{
    int number;     //variable to store the number
    int sum = 0;    //variable to store the sum
    int count = 0;  //variable to store the total
                    //numbers read
```

```
    cout << "Line 1: Enter integers ending with "
         << SENTINEL << endl;                          //Line 1
    cin >> number;                                     //Line 2

    while (number != SENTINEL)                         //Line 3
    {
        sum = sum + number;                            //Line 4
        count++;                                       //Line 5
        cin >> number;                                 //Line 6
    }

    cout << "Line 7: The sum of the " << count
         << " numbers is " << sum << endl;             //Line 7

    if (count != 0)                                    //Line 8
        cout << "Line 9: The average is "
             <<  sum / count << endl;                  //Line 9
    else                                               //Line 10
        cout << "Line 11: No input." << endl;          //Line 11

    return 0;
}
```

Sample Run: In this sample run, the user input is shaded.

```
Line 1: Enter integers ending with -999
34 23 9 45 78 0 77 8 3 5 -999
Line 7: The sum of the 10 numbers is 282
Line 9: The average is 28
```

Example 4.3 The sentinel-controlled *while* loop

4.2 FOR LOOP REPETITION STRUCTURE

The while loop discussed in the previous section is general enough to implement most forms of repetitions. The C++ for looping structure discussed here is a specialized form of the while loop. Its primary purpose is to **simplify the writing of counter-controlled loops**. For this reason, the *for* loop is typically called **a counted or indexed *for* loop**.

The general form of the *for* statement is:

```
for (initial statement; loop condition; update statement)
    statement
```

The initial statement, loop condition, and update statement (called for **loop control statements**) enclosed within the parentheses control the body (statement) of the *for* statement.

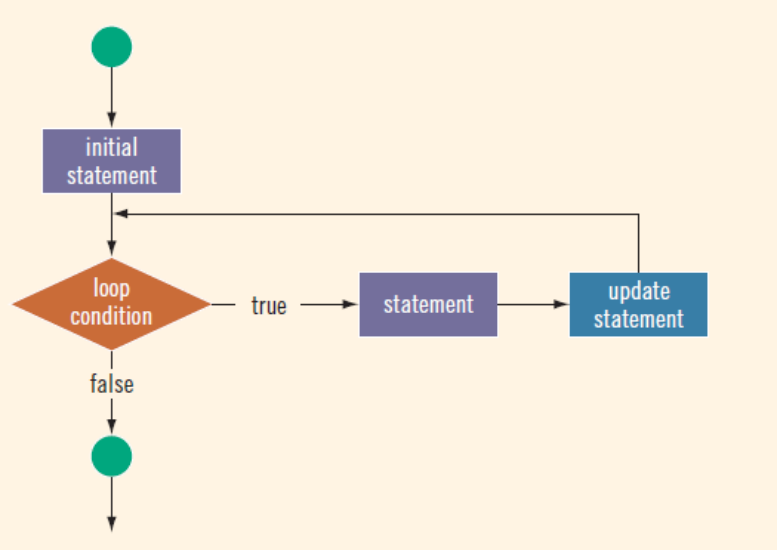

Figure 4.2 The *for* loop

The *for* loop executes as follows:

1. The initial statement executes.
2. The loop condition is evaluated. If the loop condition evaluates to *true*:
   i. Execute the *for* loop statement.
   ii. Execute the update statement (the third expression in the parentheses).
3. Repeat Step 2 until the loop condition evaluates to false.

The initial statement usually initializes a variable (called the *for* loop control, or *for* indexed, variable).

In C++, *for* is a reserved word.

The following `for` loop prints the first 10 nonnegative integers:

```
for (i = 0; i < 10; i++)
    cout << i << " ";
cout << endl;
```

Example 4.4 The *for* loop

In this example, a `for` loop reads five numbers and finds their sum and average. Consider the following program code, in which `i`, `newNum`, `sum`, and `average` are `int` variables.

```
sum = 0;

for (i = 1; i <= 5; i++)
{
   cin >> newNum;
   sum = sum + newNum;
}

average = sum / 5;
cout << "The sum is " << sum << endl;
cout << "The average is " << average << endl;
```

Example 4.5 The *for* loop

4.3 DO..WHILE REPETITION STRUCTURE

The general form of a *do. . .while* statement is as follows:

```
do
    statement
while (expression);
```

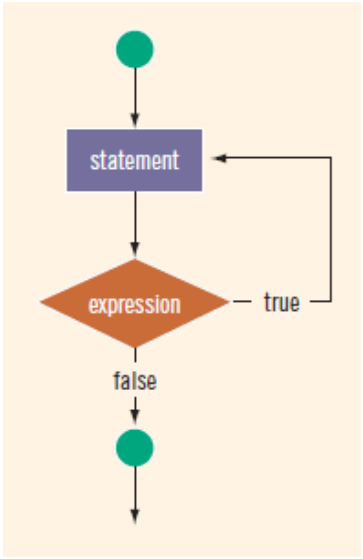

Figure 4.3 The *do..while* loop

In C++, *do* is a reserved word.

The statement executes first, and then the expression is evaluated. If the expression evaluates to *true*, the statement executes again. As long as the expression in a *do...while* statement is *true*, the statement executes. To avoid an infinite loop, you must, once again, make sure that the loop body contains a statement that ultimately makes the expression *false* and assures that it exits properly.

```
i = 0;

do
{
    cout << i << " ";
    i = i + 5;
}
while (i <= 20);
```

The output of this code is:

```
0 5 10 15 20
```

Example 4.6 The *do..while* statement

In a *while* and *for* loop, the loop condition is evaluated before executing the body of the loop. Therefore, *while* and *for* loops are called **pretest loops**. On the other hand, the loop condition in a *do. . .while* loop is evaluated after executing the body of the loop. Therefore, *do. . .while* loops are called **posttest loops**.

Because the *while* and *for* loops both have entry conditions, these loops may never activate. The *do...while* loop, on the other hand, has an exit condition and therefore always executes the statement **at least once**.

Consider the following two loops:

a.
```
i = 11;
while (i <= 10)
{
    cout << i << " ";
    i = i + 5;
}
cout << endl;
```

b.
```
i = 11;
do
{
    cout << i << " ";
    i = i + 5;
}
while (i <= 10);

cout << endl;
```

In (a), the `while` loop produces nothing. In (b), the `do...while` loop outputs the number `11` and also changes the value of `i` to 16.

Example 4.7 Comparison between *while* and *do..while* loop

All three loops have their place in C++. If you **know**, or the program can determine in advance, the number of repetitions needed, the *for* loop is the correct choice. If you **do not know**, and the program cannot determine in advance the number of repetitions needed, and it could be zero, the *while* loop is the right choice. If you **do not know**, and the program cannot determine in advance the number of repetitions needed, and it is **at least one**, the *do...while* loop is the right choice.

4.4 BREAK AND CONTINUE STATEMENTS

The *break* statement, when executed in a *switch* structure, provides an **immediate exit** from the *switch* structure. Similarly, you can use the break statement in *while*, *for*, and *do. . .while* loops. When the *break* statement executes in a repetition structure, it immediately exits from the structure.

The *break* statement is typically used for two purposes:

- To **exit** early from a loop.
- To **skip** the remainder of the switch structure.

After the *break* statement executes, the program continues to execute with the first statement after the structure. The use of a *break* statement in a loop can eliminate the use of certain (flag) variables.

```
sum = 0;
cin >> num;

while (cin)
{
    if (num < 0)    //if num is negative, terminate the loop
    {
        cout << "Negative number found in the data." << endl;
        break;
    }

    sum = sum + num;
    cin >> num;
}
```

Example 4.7 Example of while loop with *break* statement

The *break* statement is an effective way to avoid extra variables to control a loop and produce an elegant code. However, *break* statements must be used very sparingly within a loop. An excessive use of these statements in a loop will produce **spaghetti-code** (loops with many exit conditions) that can be very hard to understand and manage. You should be extra careful in using *break* statements and ensure that the use of the *break* statements makes the code more readable and not less readable. If you are not sure, do not use *break* statements.

The *continue* statement is used in *while*, *for*, and *do. . .while* structures. When the *continue* statement is executed in a loop, it **skips the remaining statements in the loop and proceeds with the next iteration of the loop**. In a *while* and *do. . .while* structure, the expression (that is, the loop-continue test) is evaluated immediately after the *continue* statement. In a *for* structure, the update statement is executed after the *continue* statement, and then the loop condition (that is, the loop-continue test) executes.

```
while (cin)
{
    if (num < 0)
    {
        cout << "Negative number found in the data." << endl;
        cin >> num;
        continue;
    }

    sum = sum + num;
    cin >> num;
}
```

Example 4.8 *while* loop with *continue* statement