Loops

Mohamed Saied

Objectives

In this chapter you will:

- Learn about repetition (looping) control structures
- Explore how to construct and use count-controlled, sentinel-controlled, flag-controlled, and EOF-controlled repetition structures
- Examine `break` and `continue` statements
- Discover how to form and use nested control structures

Why Is Repetition Needed?

- Repetition allows you to efficiently use variables
- Can input, add, and average multiple numbers using a limited number of variables
- For example, to add five numbers:
  - Declare a variable for each number, input the numbers and add the variables together
  - Create a loop that reads a number into a variable and adds it to a variable that contains the sum of the numbers

The `while` Loop

- The general form of the `while` statement is:

```
while (expression)
    statement
```

  `while` is a reserved word
- Statement can be simple or compound
- Expression acts as a decision maker and is usually a logical expression
- Statement is called the body of the loop
- The parentheses are part of the syntax

The `while` Loop (continued)

- Expression provides an entry condition
- Statement executes if the expression initially evaluates to true
- Loop condition is then reevaluated
- Statement continues to execute until the expression is no longer true

The `while` Loop (continued)

- Infinite loop: continues to execute endlessly
- Can be avoided by including statements in the loop body that assure exit condition will eventually be `false`

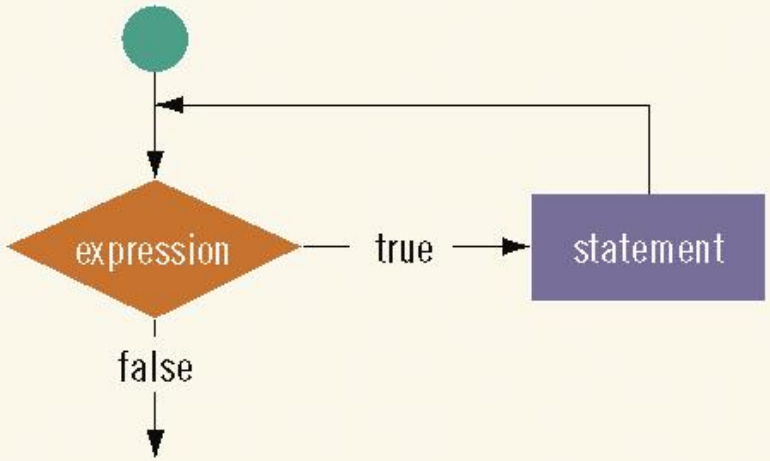

FIGURE 5-1 `while` loop

EXAMPLE 5-1

Consider the following C++ program segment:

```
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 5-2

Consider the following C++ program segment:

```
i = 20;                          //Line 1
while (i < 20)                   //Line 2
{
    cout << i << " ";            //Line 3
    i = i + 5;                   //Line 4
}
cout << endl;                    //Line 5
```

It is easy to overlook the difference between this example and Example 5-1. In this example, in Line 1, `i` is set to 20. Because `i` is 20, the expression `i < 20` in the **`while`** statement (Line 2) evaluates to **`false`**. Because initially the loop entry condition, `i < 20`, is **`false`**, the body of the **`while`** loop never executes. Hence, no values are output and the value of `i` remains 20.

Counter-Controlled `while` Loops

- If you know exactly how many pieces of data need to be read, the `while` loop becomes a counter-controlled loop

```
counter = 0;          //initialize the loop control variable

while (counter < N) //test the loop control variable
{
      .
      .
      .
     counter++;       //update the loop control variable
      .
      .
      .
}
```

Sentinel-Controlled `while` Loops

- Sentinel variable is tested in the condition and loop ends when sentinel is encountered

```
cin >> variable;                    //initialize the loop control variable

while (variable != sentinel)   //test the loop control variable
{
     .
     .
     .
     cin >> variable;          //update the loop control variable
     .
     .
     .
}
```

Flag-Controlled `while` Loops

- A flag-controlled `while` loop uses a `bool` variable to control the loop
- The flag-controlled `while` loop takes the form:

```
found = false;          //initialize the loop control variable

while (!found)          //test the loop control variable
{
    .
    .
    .
    if (expression)
        found = true; //update the loop control variable
    .
    .
    .
}
```

The `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 are called `for` loop control statements

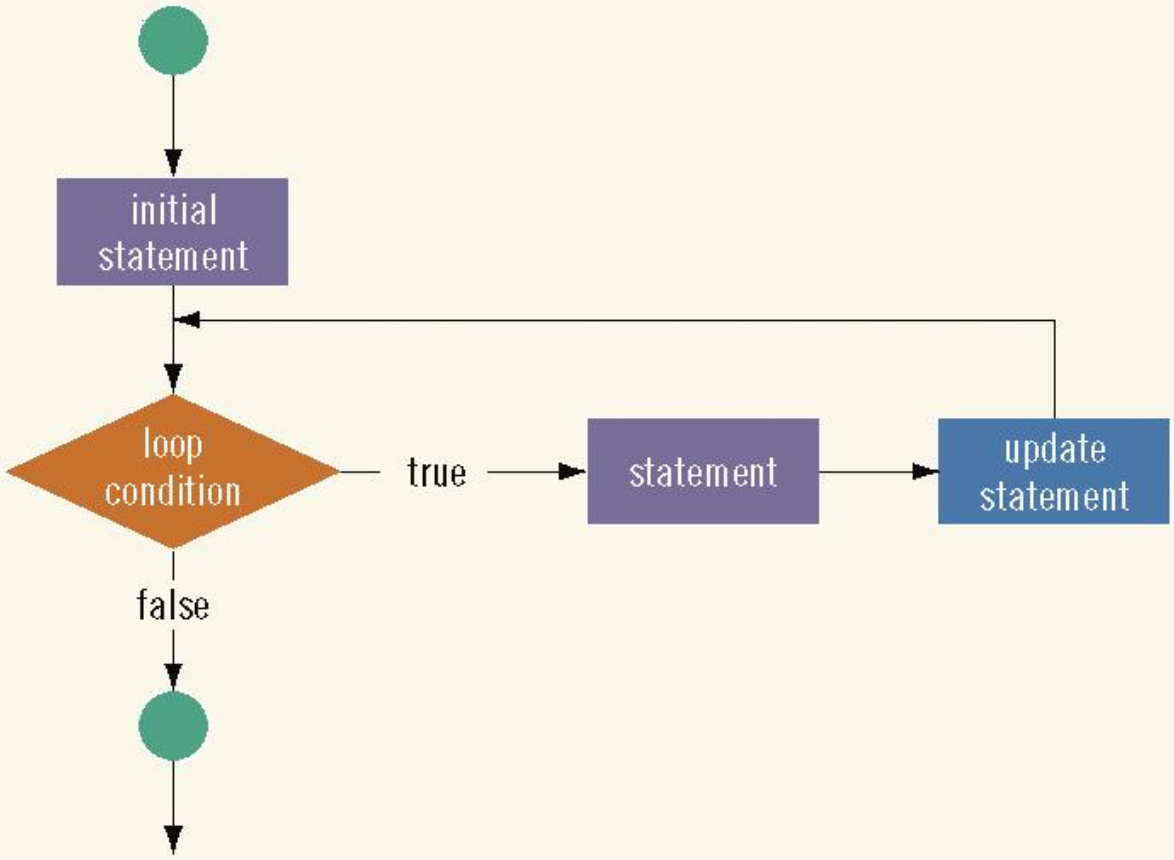

FIGURE 5-2 `for` loop

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.

EXAMPLE 5-7

The following `for` loop prints the first 10 non-negative integers:

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

EXAMPLE 5-8

The following `for` loop outputs `Hello!` and a star (on separate lines) five times:

```
for (i = 1; i <= 5; i++)
{
    cout << "Hello!" << endl;
    cout << "*" << endl;
}
```

Consider the following `for` loop:

```
for (i = 1; i <= 5; i++)
    cout << "Hello!" << endl;
    cout << "*" << endl;
```

This loop outputs `Hello!` five times and the star only once.

EXAMPLE 5-9

The following `for` loop executes five empty statements:

```
for (i = 0; i < 5; i++);        //Line 1
    cout << "*" << endl;        //Line 2
```

The `for` Loop (comments)

The following are some comments on `for` loops:

- If the loop condition is initially `false`, the loop body does not execute.
- The update expression, when executed, changes the value of the loop control variable (initialized by the initial expression), which eventually sets the value of the loop condition to false. The `for` loop body executes indefinitely if the loop condition is always `true`.
- C++ allows you to use fractional values for loop control variables of the `double` type (or any real data type). Because different computers can give these loop control variables different results, you should avoid using such variables.

The `for` Loop (comments)

- A semicolon at the end of the `for` statement (just before the body of the loop) is a semantic error. In this case, the action of the `for` loop is empty.
- In the `for` statement, if the loop condition is omitted, it is assumed to be `true`.
- In a `for` statement, you can omit all three statements—initial statement, loop condition, and update statement. The following is a legal `for` loop:
- `for (;;)`

```
    cout << "Hello" << endl;
```

EXAMPLE 5-10

```
for (i = 10; i >= 1; i--)
    cout << " " << i;
cout << endl;
```

The output is:

```
10 9 8 7 6 5 4 3 2 1
```

EXAMPLE 5-11

```
for (i = 1; i <= 20; i = i + 2)
    cout << " " << i;
cout << endl;
```

This `for` loop outputs the first 10 positive odd integers.

The `do...while` Loop

- The general form of a `do...while` statement is:

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

- 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

The `do...while` Loop (continued)

- To avoid an infinite loop, the loop body must contain a statement that makes the expression `false`
- The statement can be simple or compound
- If compound, it must be in braces
- `do...while` loop has an exit condition and always iterates at least once (unlike `for` and `while`)

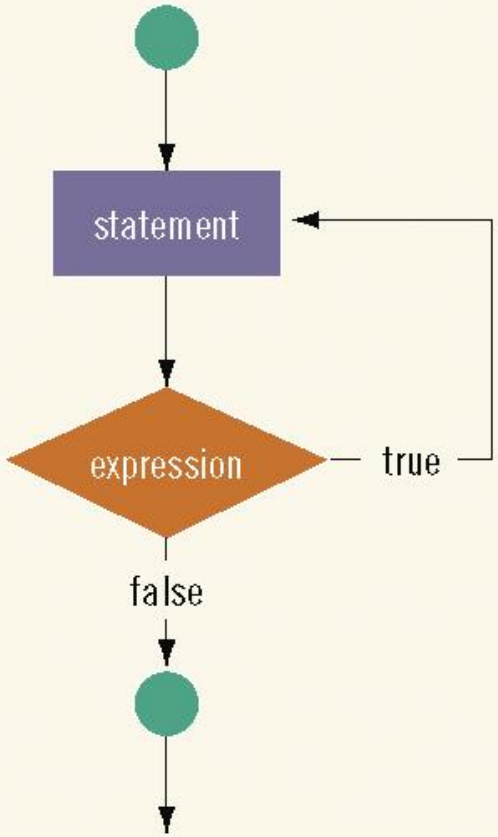

FIGURE 5-3 `do...while` loop

EXAMPLE 5-15

```
i = 0;

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

The output of this code is:

```
0 5 10 15 20
```

EXAMPLE 5-16

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;
```

`break` & `continue` Statements

- `break` and `continue` alter the flow of control
- When the `break` statement executes in a repetition structure, it immediately exits
- The `break` statement, in a `switch` structure, provides an immediate exit
- The `break` statement can be used in `while`, `for`, and `do`...`while` loops

`break` & `continue` Statements (continued)

- The `break` statement is used for two purposes:
  1. To exit early from a loop
  2. To skip the remainder of the switch structure
- After the `break` statement executes, the program continues with the first statement after the structure
- The use of a `break` statement in a loop can eliminate the use of certain (flag) variables

`break` & `continue` Statements (continued)

- `continue` is used in `while`, `for`, and `do`…`while` structures
- When executed in a loop
  - It skips remaining statements and proceeds with the next iteration of the loop

`break` & `continue` Statements (continued)

- In a `while` and `do`…`while` structure
  - Expression (loop-continue test) is evaluated immediately after the continue statement
- In a `for` structure, the update statement is executed after the `continue` statement
  - Then the loop condition executes

Nested Control Structures

- Suppose we want to create the following pattern

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

- In the first line, we want to print one star, in the second line two stars and so on

Nested Control Structures (continued)

- Since five lines are to be printed, we start with the following for statement

```
for (i = 1; i <= 5 ; i++)
```

- The value of `i` in the first iteration is `1`, in the second iteration it is `2`, and so on
- Can use the value of `i` as limit condition in another for loop nested within this loop to control the number of starts in a line

Nested Control Structures (continued)

- The syntax is:

```
for (i = 1; i <= 5 ; i++)
{
    for (j = 1; j <= i; j++)
        cout << "*";
    cout << endl;
}
```

Nested Control Structures (continued)

- What pattern does the code produce if we replace the first for statement with the following?

```
for (i = 5; i >= 1; i--)
```

- Answer:

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

Summary

- C++ has three looping (repetition) structures: `while`, `for`, and `do`…`while`
- `while`, `for`, and `do` are reserved words
- `while` and `for` loops are called pre-test loops
- `do`...`while` loop is called a post-test loop
- `while` and `for` may not execute at all, but `do`...`while` always executes at least once

Summary (continued)

- `while`: expression is the decision maker, and the statement is the body of the loop
- In a counter-controlled while loop,
  - Initialize counter before loop
  - Body must contain a statement that changes the value of the counter variable
- A sentinel-controlled `while` loop uses a sentinel to control the `while` loop

Summary (continued)

- `for` loop: simplifies the writing of a count-controlled while loop
- Executing a `break` statement in the body of a loop immediately terminates the loop
- Executing a `continue` statement in the body of a loop skips to the next iteration
- After a `continue` statement executes in a for loop, the update statement is the next statement executed