Lecture Notes

Chapter 5

Control Structures II (Repetition)

ECE 111: Introduction to C and C++ Programming

Instructor: Dr. Shayan (Sean) Taheri

Gannon University (GU)

Personal Information

- Name: Shayan (Sean) Taheri.
- Date of Birth: July/28/1991.
- Current Position: Assistant Professor at Gannon University
- Previous Position: Postdoctoral Fellow at University of Florida.
- Ph.D. Degree: Electrical Engineering from the University of Central Florida.
- M.S. Degree: Computer Engineering from the Utah State University.
- University Profile: https://www.gannon.edu/FacultyProfiles.aspx?profile=taheri001

Objectives (1 of 2)

- In this chapter, you will:
  - Learn about repetition (looping) control structures
  - Learn how to use a **`while`** loop in a program
  - Explore how to construct and use counter-controlled, sentinel-controlled, flag-controlled, and EOF-controlled repetition structures
  - Learn how to use a **`for`** loop in a program
  - Learn how to use a **`do`**…**`while`** loop in a program

Objectives (2 of 2)

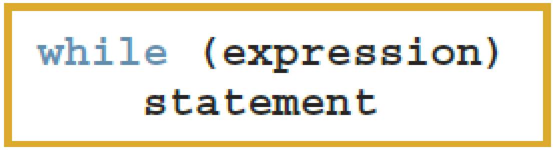
- Examine **`break`** and **`continue`** statements
- Discover how to form and use nested control structures
- Learn how to avoid bugs by avoiding patches
- Learn how to debug loops

Why Is Repetition Needed?

- Repetition allows efficient use of variables
- It is possible to input, add, and average multiple numbers using a limited number of variables
- Consider the code to determine the average number of calories burned each day doing regular exercise
  - Method 1: Declare a variable for each day and enter the number of calories burned, add the values and store in a variable for the week’s total, and divide the total by **7** to find the average
  - Method 2: Create a loop that reads a number into a variable and adds it to a variable that contains the sum of the numbers (only two variables needed)

while Looping (Repetition) Structure (1 of 3)

- A **while** loop is one of three repetition, or looping structures in C++
- Syntax of the **while** statement



```
while (expression)
    statement
```

- The **statement** can be simple or compound
- The **expression** acts as a decision maker and is usually a logical expression
- The **statement** is called the body of the loop
- The parentheses are part of the syntax

while Looping (Repetition) Structure (2 of 3)

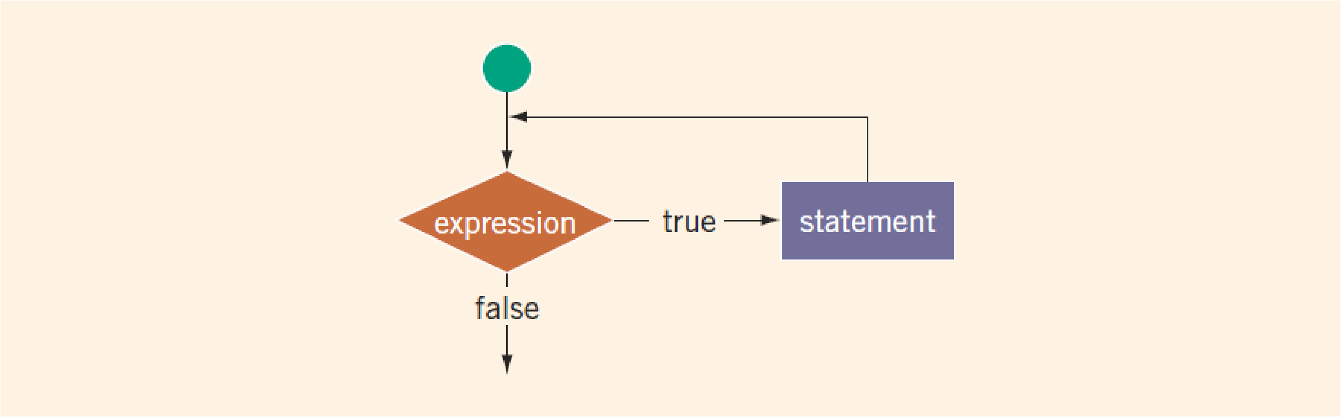
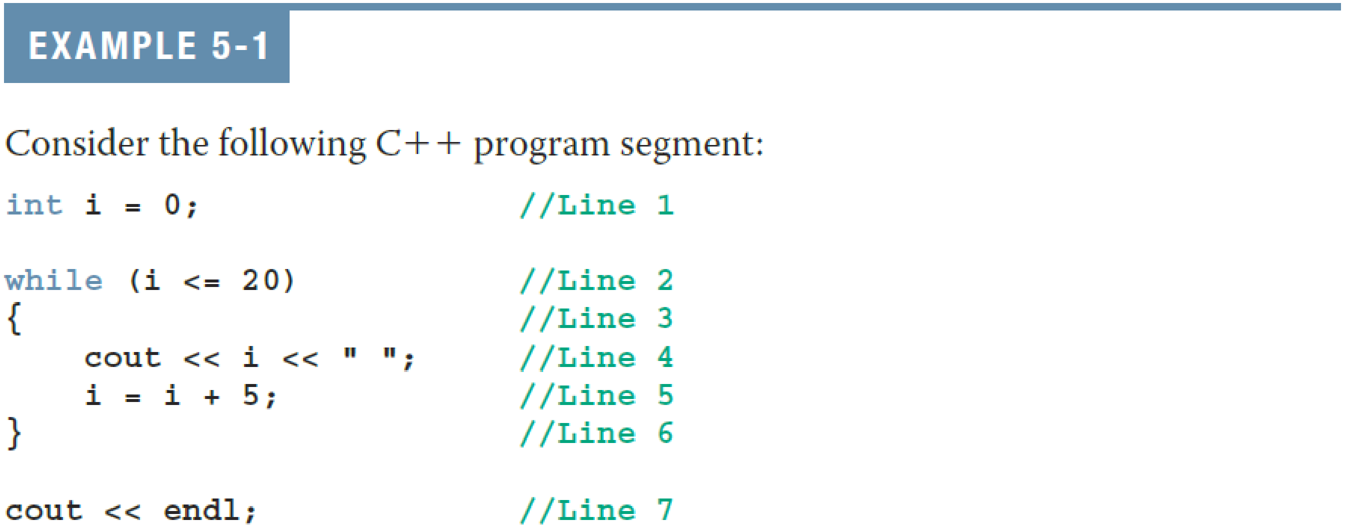


FIGURE 5-1 **while** loop

- The **expression** provides an entry condition to the loop
- The **statement** (body of the loop) continues to execute until the expression is no longer **true**
- An infinite loop continues to execute endlessly

while Looping (Repetition) Structure (3 of 3)



EXAMPLE 5-1

Consider the following C++ program segment:

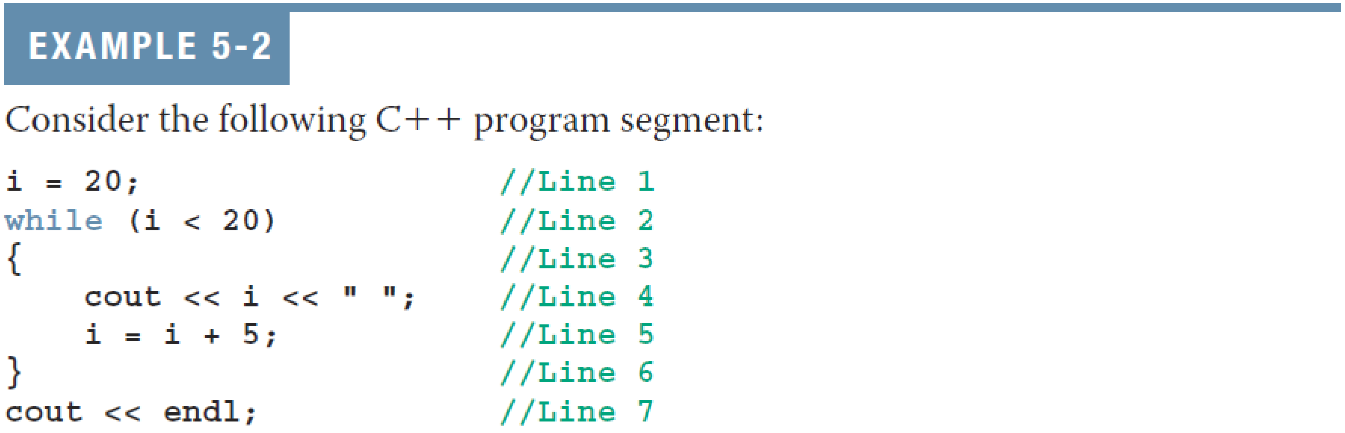
```
int i = 0;                  //Line 1

while (i <= 20)             //Line 2
{                           //Line 3
    cout << i << " ";       //Line 4
    i = i + 5;              //Line 5
}                           //Line 6

cout << endl;               //Line 7
```

- The preceding **while** loop produces the following output:

```
0 5 10 15 20
```

- The variable **i** in Example 5-1 is called the loop control variable (LCV)

while Looping (Repetition) Structure (cont’d.)



EXAMPLE 5-2

Consider the following C++ program segment:

```
i = 20;                        //Line 1
while (i < 20)                 //Line 2
{                              //Line 3
    cout << i << " ";          //Line 4
    i = i + 5;                 //Line 5
}                              //Line 6
cout << endl;                  //Line 7
```

Case 1: Counter-Controlled while Loops

- When you know exactly how many times the statements need to be executed
  - Use a counter-controlled **while** loop

```
counter = 0;         //initialize the loop control variable
while (counter < N) //test the loop control variable
{
    .
    .
    .
    counter++;       //update the loop control variable
    .
    .
    .
}
```

Case 2: Sentinel-Controlled `while` Loops

- A sentinel variable is tested in the condition
- The loop ends when the sentinel is encountered
- The following is an example of a sentinel-controlled **`while`** loop:

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

Example 5-5: Telephone Digits

- Example 5-5 provides an example of a sentinel-controlled loop
- The program converts uppercase letters to their corresponding telephone digit
- The sentinel value is **#**

Case 3: Flag-Controlled `while` Loops

- Flag-controlled **`while`** loop: uses a **`bool`** variable to control the loop

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

Number Guessing Game

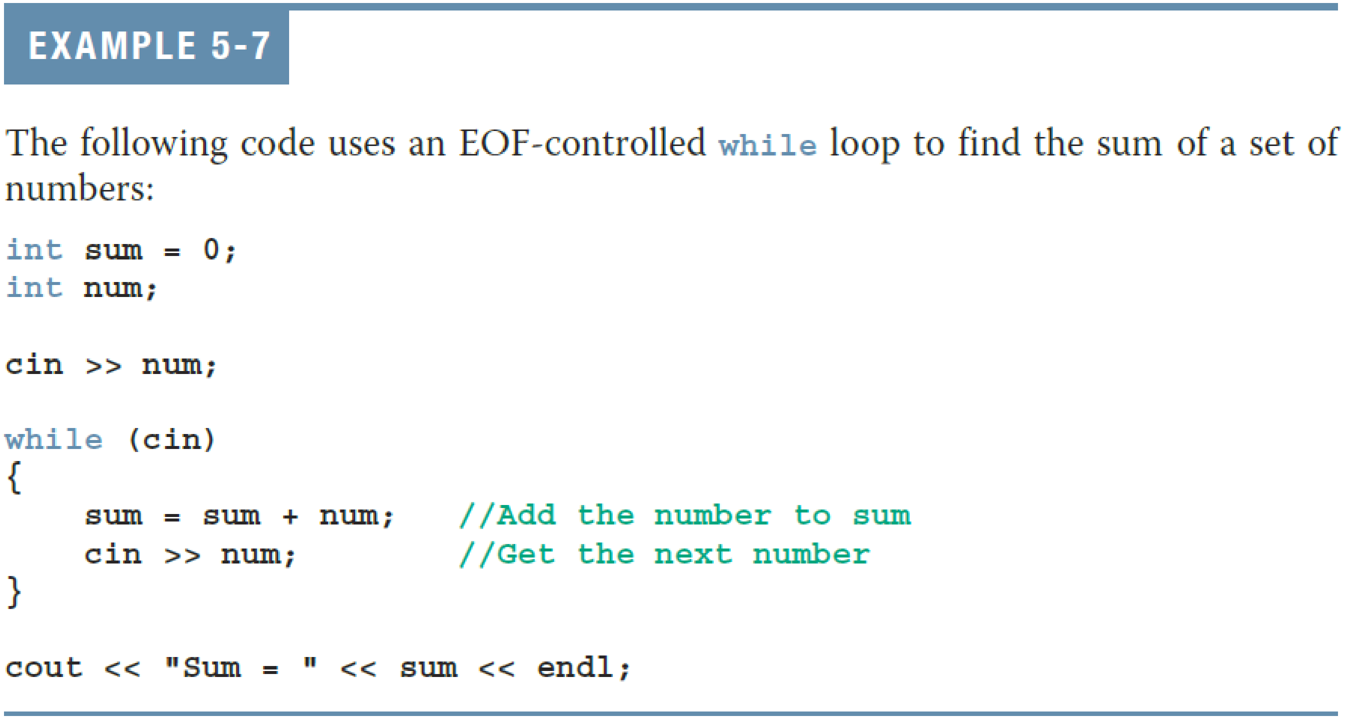
- Example 5-6 implements a number guessing game using a flag-controlled `while` loop
- Uses the function `rand` of the header file **`cstdlib`** to generate a random number
  - **`rand()`** returns an `int` value between **0** and **`32767`**
  - To convert to an integer **`>= 0`** and **`< 100`**:

```
rand() % 100
```

Case 4: EOF-Controlled `while` Loops (1 of 2)

- An end-of-file (EOF)-controlled **`while`** loop is a good choice when it is difficult to select a sentinel value
- The logical value returned by **`cin`** can determine if there is no more input

Case 4: EOF-Controlled `while` Loops (2 of 2)



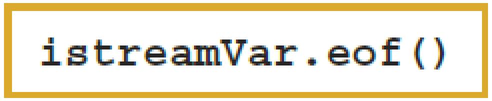
EXAMPLE 5-7

The following code uses an EOF-controlled `while` loop to find the sum of a set of numbers:

```
int sum = 0;
int num;

cin >> num;

while (cin)
{
    sum = sum + num;    //Add the number to sum
    cin >> num;         //Get the next number
}

cout << "Sum = " << sum << endl;
```

eof Function

- The function **eof** can determine the end of file status
- **eof** is a member of data type **istream**
- Syntax for the function **eof**



```
istreamVar.eof()
```

- **istreamVar** is an input stream variable, such as **cin**

More on Expressions in `while` Statements

- The expression in a **`while`** statement can be complex
  - Example

```
while ((noOfGuesses < 5) && (!isGuessed))
{
     . . .
}
```

Programming Example: Fibonacci Number (1 of 3)

- Consider the following sequence of numbers:

```
1, 1, 2, 3, 5, 8, 13, 21, 34, ....
```

- Called the Fibonacci sequence
- Given the first two numbers of the sequence (say, a_1 and a_2)
  - *n*th number a_n, $n >= 3$, of this sequence is given by: $a_n = a_{n-1} + a_{n-2}$

Programming Example: Fibonacci Number (2 of 3)

- Fibonacci sequence
  - *n*th Fibonacci number
  - a_2 = 1
  - a_1 = 1
  - Determine the *n*th number a_n, n >= 3

Programming Example: Fibonacci Number (3 of 3)

- Suppose a_2 = 6 and a_1 = 3
  - $a_3 = a_2 + a_1 = 6 + 3 = 9$
  - $a_4 = a_3 + a_2 = 9 + 6 = 15$
- Write a program that determines the *n*th Fibonacci number, given the first two numbers

Programming Example: Input and Output

- Input: first two Fibonacci numbers and the desired Fibonacci number
- Output: *n*th Fibonacci number

Programming Example: Problem Analysis and Algorithm Design

- Algorithm
  - Get the first two Fibonacci numbers
  - Get the desired Fibonacci number
    - Get the position, *n*, of the number in the sequence
  - Calculate the next Fibonacci number
    - Add the previous two elements of the sequence
  - Repeat Step 3 until the *n*th Fibonacci number is found
  - Output the *n*th Fibonacci number

Programming Example: Variables

```
int previous1; //variable to store the first Fibonacci number
int previous2; //variable to store the second Fibonacci number
int current; //variable to store the current Fibonacci number
int counter; //loop control variable
int nthFibonacci; //variable to store the desired
                  //Fibonacci number
```

Programming Example: Main Algorithm (1 of 4)

- Prompt the user for the first two numbers—that is, **`previous1`** and **`previous2`**
- Read (input) the first two numbers into **`previous1`** and **`previous2`**
- Output the first two Fibonacci numbers
- Prompt the user for the position of the desired Fibonacci number

Programming Example: Main Algorithm (2 of 4)

- Read the position of the desired Fibonacci number into `nthFibonacci`
  - **`if (nthFibonacci == 1)`**
    The desired Fibonacci number is the first Fibonacci number; copy the value of **`previous1`** into **`current`**
  - **`else if (nthFibonacci == 2)`**
    The desired Fibonacci number is the second Fibonacci number; copy the value of **`previous2`** into **`current`**

Programming Example: Main Algorithm (3 of 4)

- `else` calculate the desired Fibonacci number as follows:
  - Start by determining the third Fibonacci number
  - Initialize **`counter`** to **3** to keep track of the calculated Fibonacci numbers.
  - Calculate the next Fibonacci number, as follows:

    ```
    current = previous2 + previous1;
    ```

Programming Example: Main Algorithm (4 of 4)

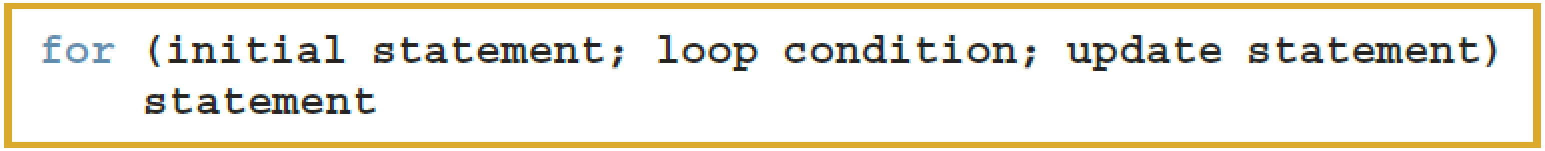
- Assign the value of `previous2` to `previous1`
- Assign the value of `current` to `previous2`
- Increment `counter`
- Repeat until Fibonacci number is calculated:

```
while (counter <= nthFibonacci)
{
   current = previous2 + previous1;
   previous1 = previous2;
   previous2 = current;
   counter++;
}
```

- Output the *n*th Fibonacci number, which is `current`

for Looping (Repetition) Structure (1 of 7)

- **for** loop: called a counted or indexed **for** loop
- Syntax of the **for** statement



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

- The **initial statement**, **loop condition**, and **update statement** are called **for** loop control statements

for Looping (Repetition) Structure (2 of 7)

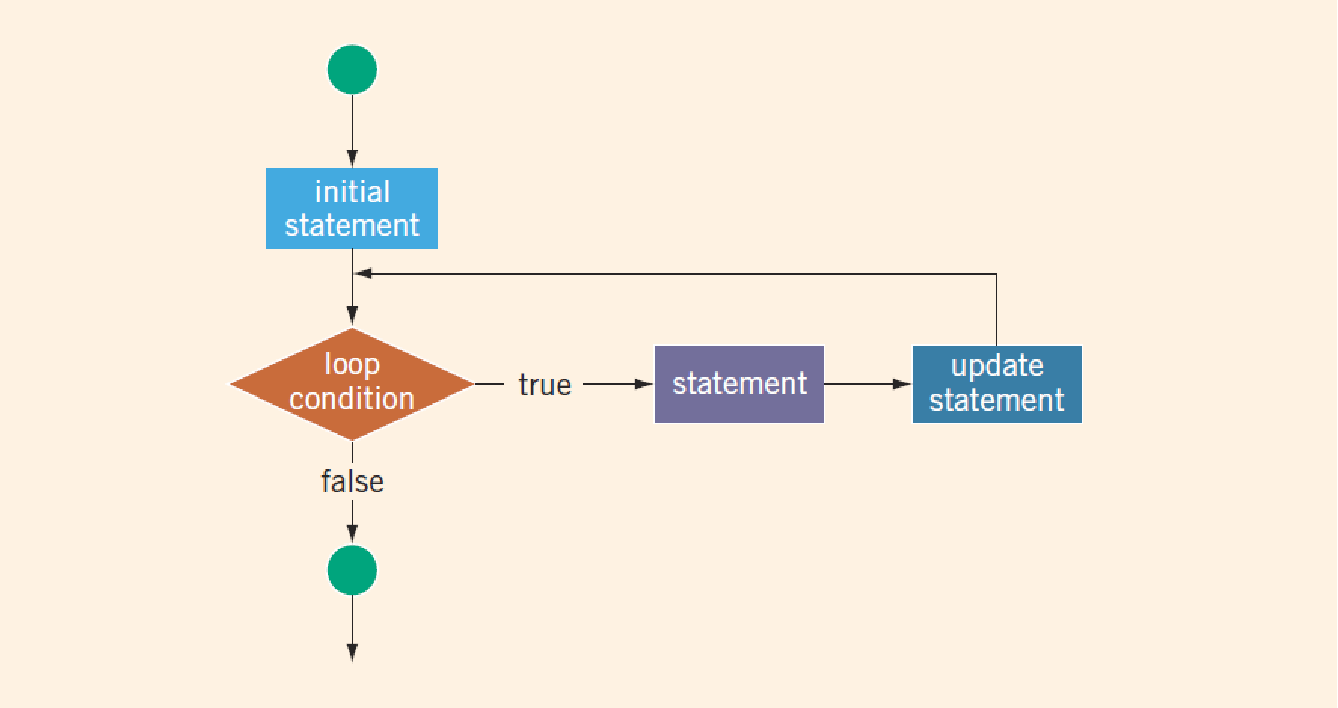
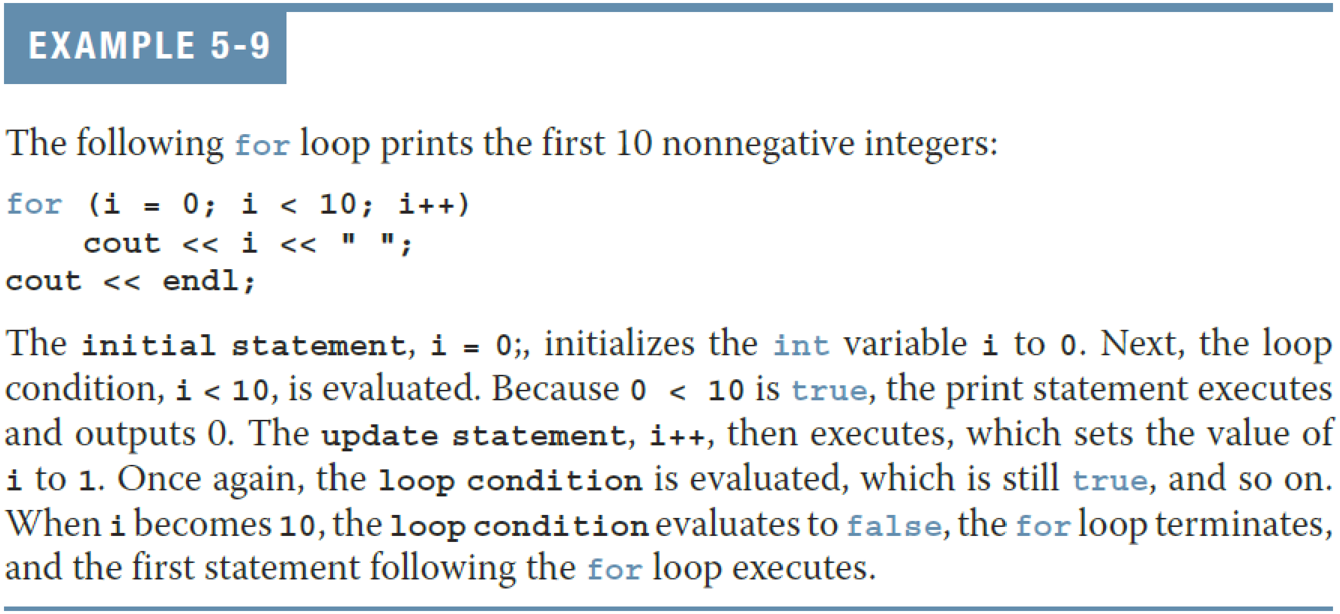


FIGURE 5-2 **for** loop

`for` Looping (Repetition) Structure (3 of 7)



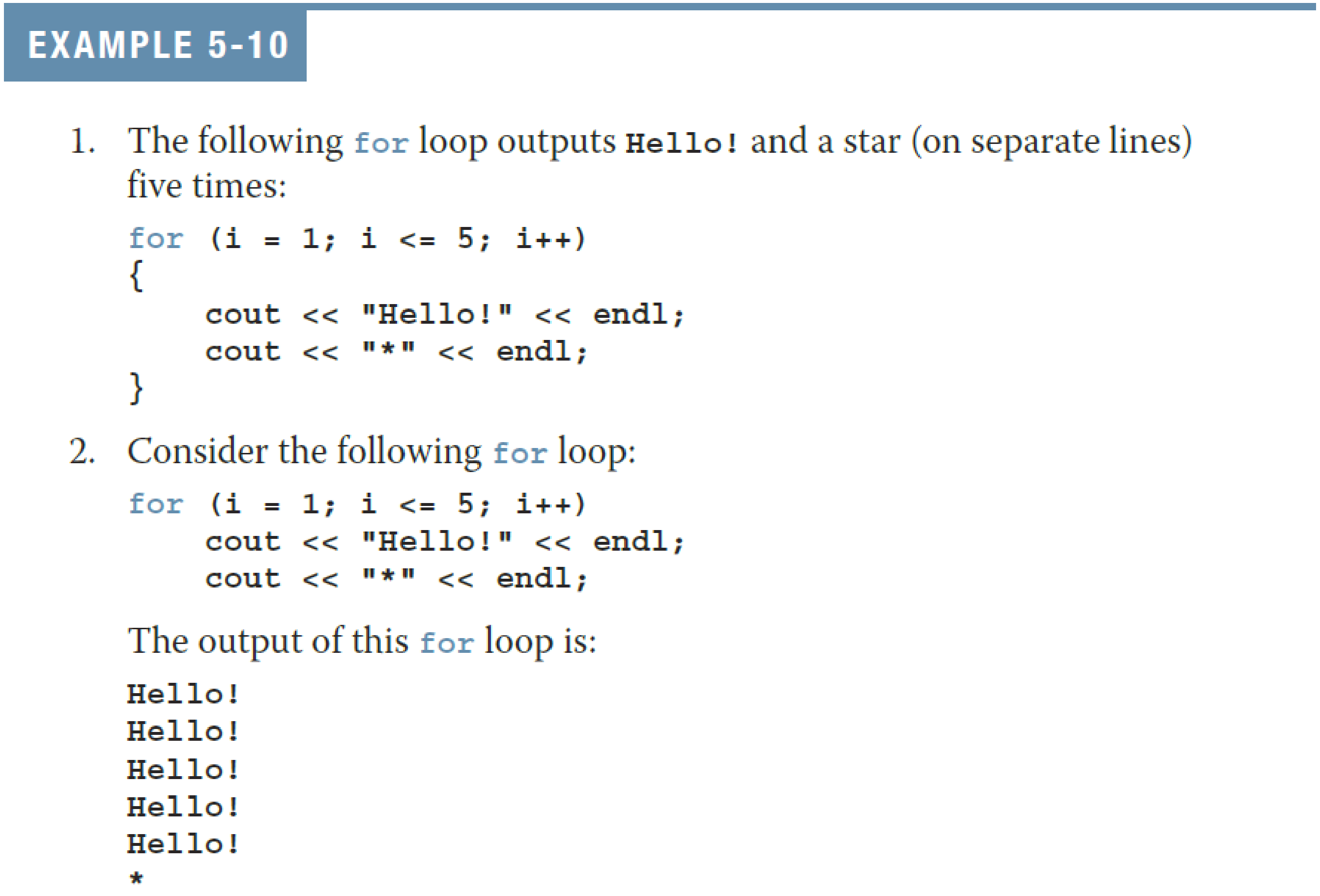
EXAMPLE 5-9

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

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

The **`initial statement`**, `i = 0;`, initializes the `int` variable `i` to `0`. Next, the loop condition, `i < 10`, is evaluated. Because `0 < 10` is `true`, the print statement executes and outputs 0. The **`update statement`**, `i++`, then executes, which sets the value of `i` to `1`. Once again, the **`loop condition`** is evaluated, which is still `true`, and so on. When `i` becomes `10`, the **`loop condition`** evaluates to `false`, the `for` loop terminates, and the first statement following the `for` loop executes.

for Looping (Repetition) Structure (4 of 7)



EXAMPLE 5-10

1. 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;
}
```

2. Consider the following for loop:

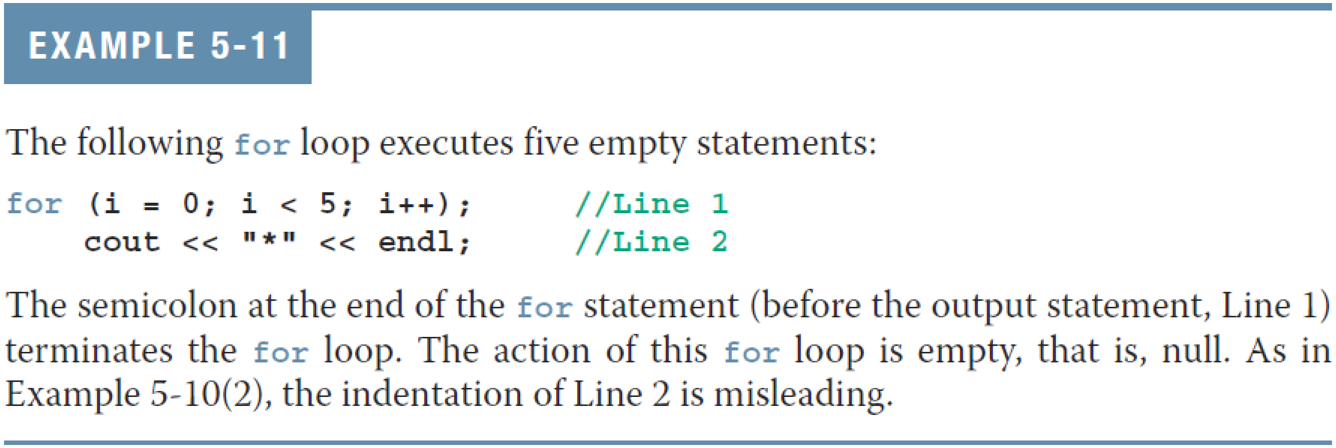
```
for (i = 1; i <= 5; i++)
    cout << "Hello!" << endl;
    cout << "*" << endl;
```

The output of this for loop is:

```
Hello!
Hello!
Hello!
Hello!
Hello!
*
```

for Looping (Repetition) Structure (5 of 7)

- The following is a semantic error:



EXAMPLE 5-11

The following for loop executes five empty statements:

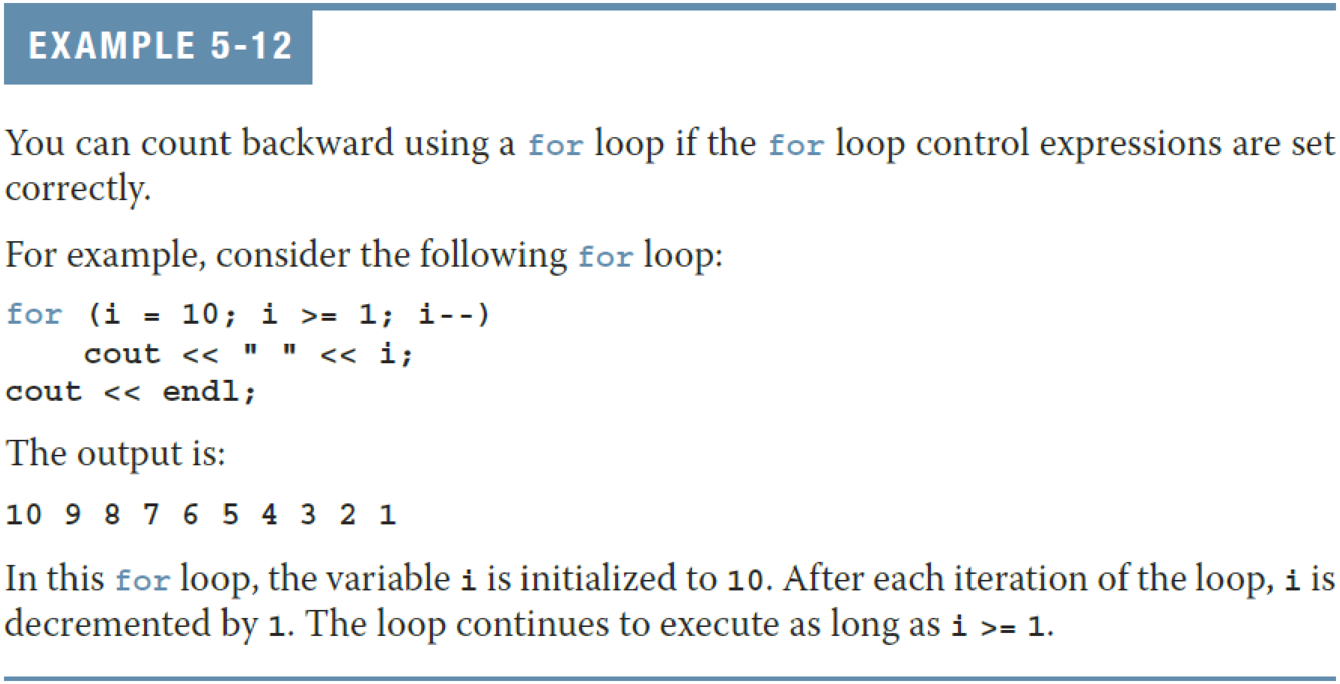
```
for (i = 0; i < 5; i++);       //Line 1
    cout << "*" << endl;       //Line 2
```

The semicolon at the end of the for statement (before the output statement, Line 1) terminates the for loop. The action of this for loop is empty, that is, null. As in Example 5-10(2), the indentation of Line 2 is misleading.

- The following is a legal (but infinite) for loop:

```
    for (;;)
        cout << "Hello" << endl;
```

for Looping (Repetition) Structure (6 of 7)



EXAMPLE 5-12

You can count backward using a `for` loop if the `for` loop control expressions are set correctly.

For example, consider the following `for` loop:

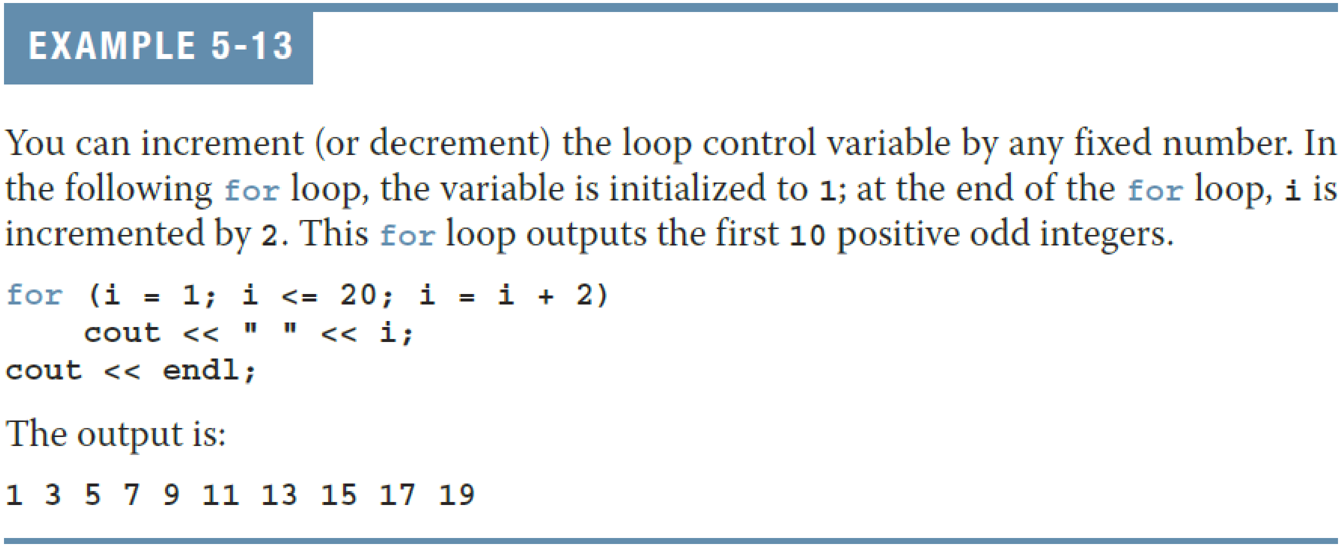
```
for (i = 10; i >= 1; i--)
    cout << " " << i;
cout << endl;
```

The output is:

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

In this `for` loop, the variable `i` is initialized to `10`. After each iteration of the loop, `i` is decremented by `1`. The loop continues to execute as long as `i >= 1`.

for Looping (Repetition) Structure (7 of 7)



EXAMPLE 5-13

You can increment (or decrement) the loop control variable by any fixed number. In the following `for` loop, the variable is initialized to `1`; at the end of the `for` loop, `i` is incremented by `2`. This `for` loop outputs the first `10` positive odd integers.

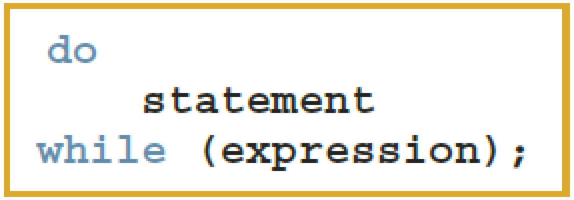
```
for (i = 1; i <= 20; i = i + 2)
    cout << " " << i;
cout << endl;
```

The output is:

```
1 3 5 7 9 11 13 15 17 19
```

do…while Looping (Repetition) Structure (1 of 6)

- Syntax of a `do...while` loop



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

- The **`statement`** executes first, and then the **`expression`** is evaluated
  - As long as **`expression`** is `true`, loop continues
- To avoid an infinite loop, body must contain a statement that makes the **`expression`** `false`

do…while Looping (Repetition) Structure (2 of 6)

- The statement can be simple or compound
- Loop always iterates at least once

`do…while` Looping (Repetition) Structure (3 of 6)

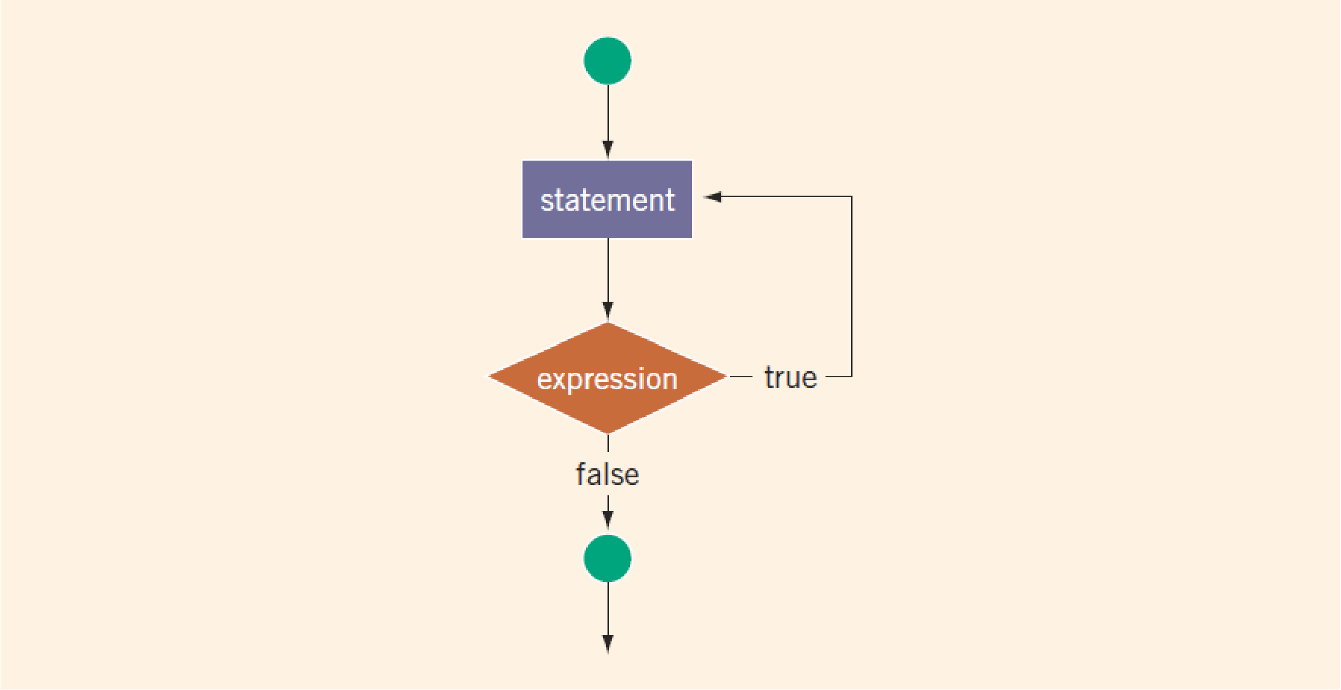
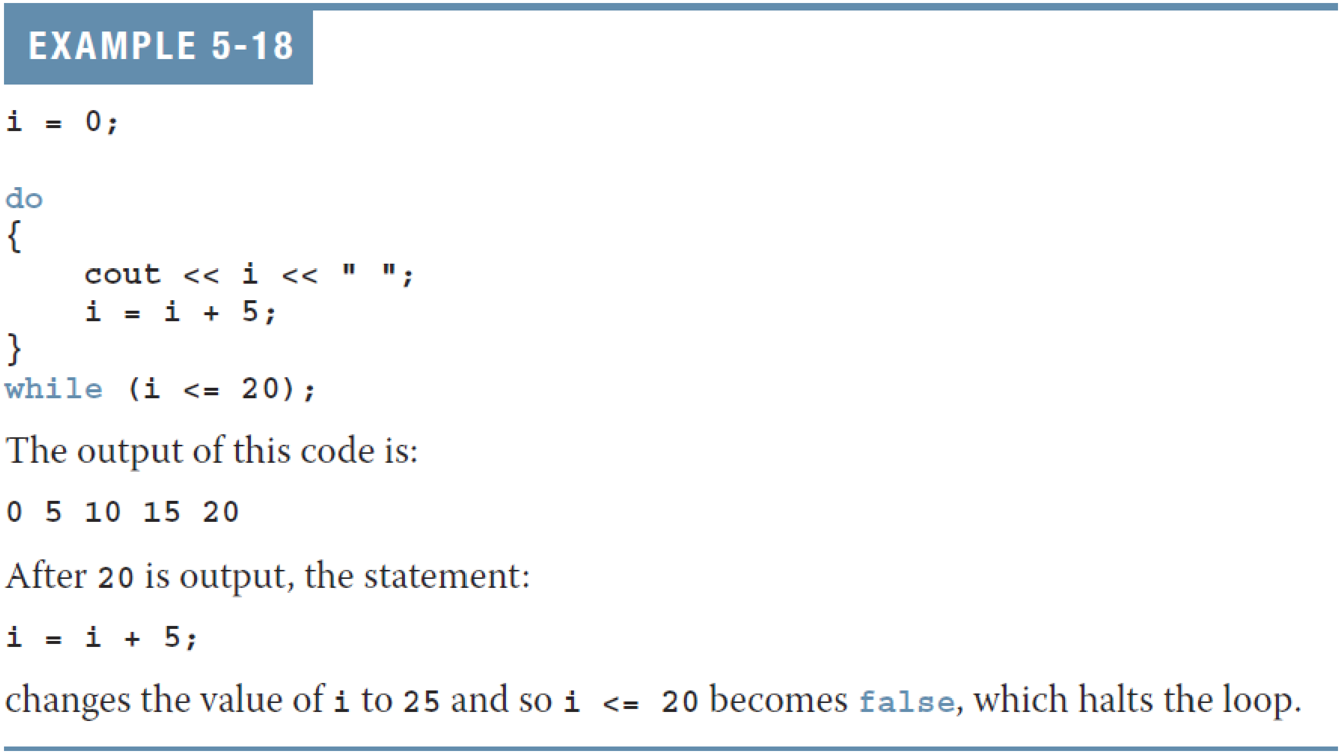


FIGURE 5-3 `do. . .while` loop

do…while Looping (Repetition) Structure (4 of 6)



EXAMPLE 5-18

```
i = 0;

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

The output of this code is:

```
0 5 10 15 20
```

After `20` is output, the statement:

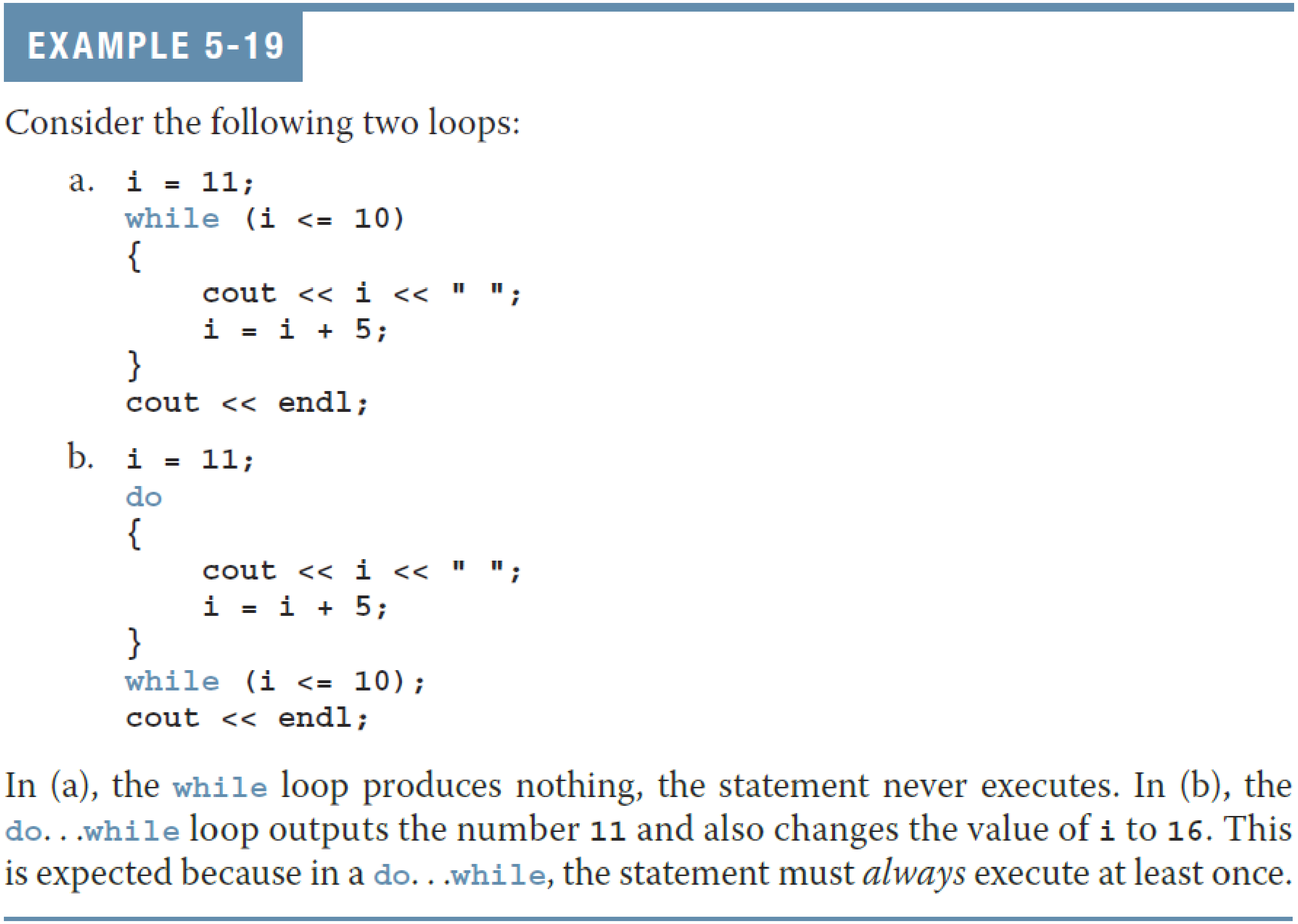
```
i = i + 5;
```

changes the value of `i` to `25` and so `i <= 20` becomes `false`, which halts the loop.

do…while Looping (Repetition) Structure (5 of 6)

- Note that **while** and **for** loops are pretest loops
  - It is possible that these loops many never activate due to entry conditions
- In contrast, **do. . .while** loops are posttest loops
  - These loops always execute at least once

do...while Looping (Repetition) Structure (6 of 6)



EXAMPLE 5-19

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, the statement never executes. In (b), the `do...while` loop outputs the number `11` and also changes the value of `i` to `16`. This is expected because in a `do...while`, the statement must *always* execute at least once.

Choosing the Right Looping Structure

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

break and continue Statements (1 of 2)

- **break** and **continue** alter the flow of control
- **break** statement is used for two purposes:
  - To exit early from a loop
  - To skip the remainder of a **switch** structure
- After **break** executes, the program continues with the first statement after the structure
- A **break** statement in a loop can eliminate the use of certain (flag) variables

`break` and `continue` Statements (2 of 2)

- **`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

Nested Control Structures (1 of 2)

- To create the following pattern:

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

- We can use the following code:

```
for (i = 1; i <= 5; i++)        //Line 1
{                               //Line 2
    for (j = 1; j <= i; j++)    //Line 3
        cout << "*";            //Line 4
    cout << endl;               //Line 5
}                               //Line 6
```

CENGAGE Learning

Nested Control Structures (2 of 2)

- What is the result if we replace the first **for** statement with this?

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

- Answer:

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

Avoiding Bugs by Avoiding Patches

- A software patch is a piece of code written on top of an existing piece of code
  - Intended to fix a bug in the original code
- Some programmers address the symptom of the problem by adding a software patch
- A programmer should instead resolve the underlying issue

Debugging Loops

- Loops are harder to debug than sequence and selection structures
- Use a loop invariant
  - Set of statements that remains true each time the loop body is executed
- The most common error associated with loops is off-by-one

Quick Review (1 of 3)

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

Quick Review (2 of 3)

- In a **`while`** loop:
  - The **`expression`** is the decision maker
  - The **`statement`** is the body of the loop
- A **`while`** loop can be:
  - Counter-controlled
  - Sentinel-controlled
  - EOF-controlled
- In the Windows console environment, the end-of-file marker is entered using **`Ctrl+z`**

Quick Review (3 of 3)

- A **for** loop simplifies the writing of a counter-controlled **while** loop
  - Putting a semicolon at the end of the **for** loop is a semantic error
- 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