C++ Programming: From Problem Analysis to Program Design, Fourth Edition

Chapter 5: Control Structures II *(Repetition)*

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

`while` Looping (Repetition) Structure

- 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

`while` Looping (Repetition) Structure (continued)

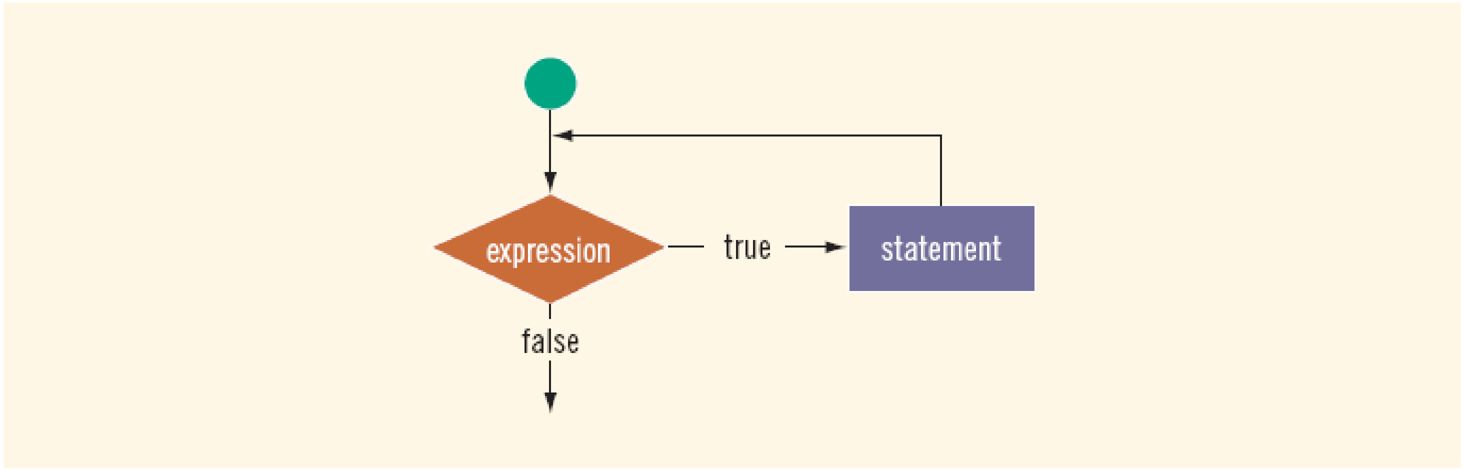
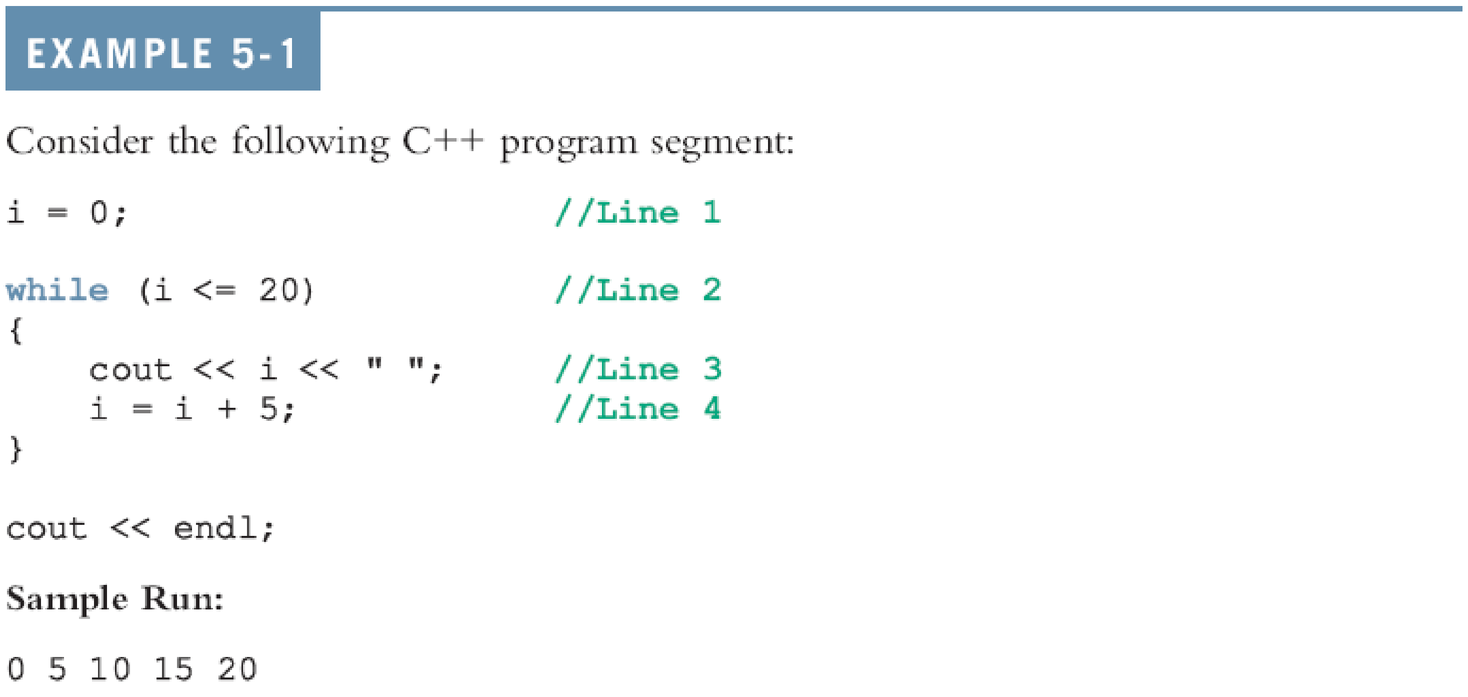


FIGURE 5-1 `while` loop

- Infinite loop: continues to execute endlessly
  - Avoided by including statements in loop body that assure exit condition is eventually `false`

while Looping (Repetition) Structure (continued)



EXAMPLE 5-1

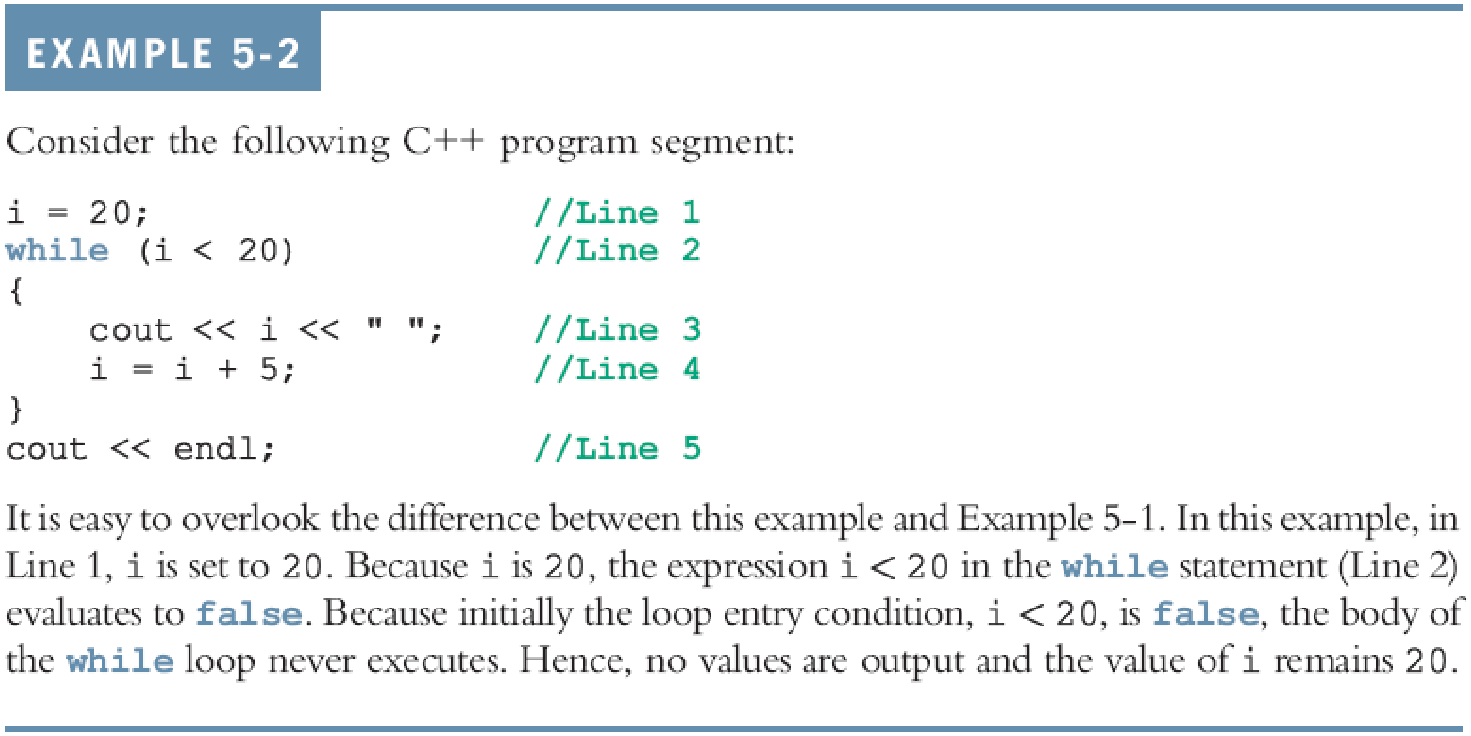
Consider the following C++ program segment:

```
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
```

Designing `while` Loops



EXAMPLE 5-2

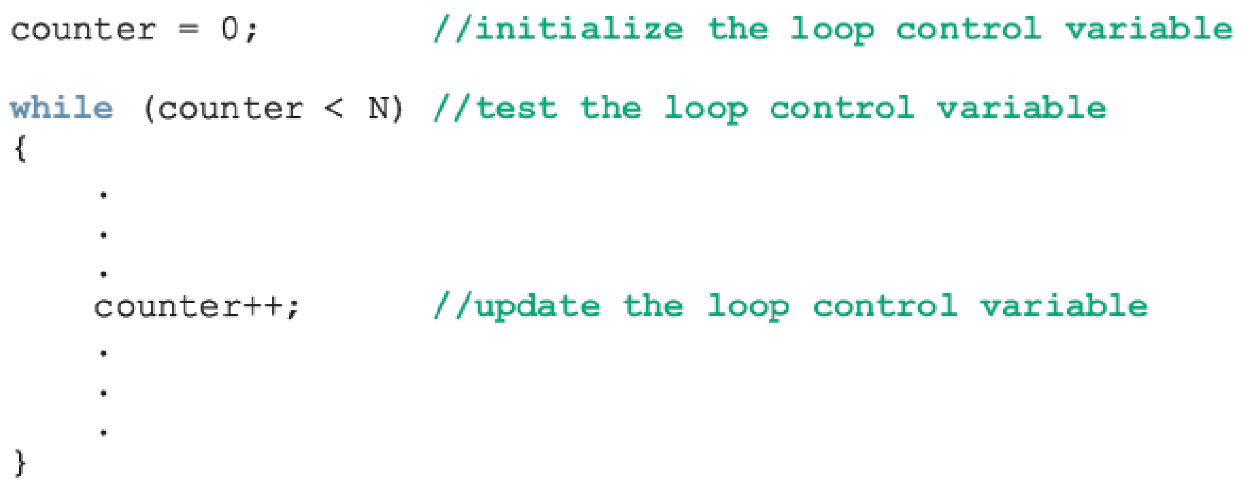
Consider the following C++ program segment:

```
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`.

Case 1: 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
      .
      .
      .
}
```

Case 2: 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
    .
    .
    .
}
```

Telephone Digits

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

Case 3: 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
     .
     .
     .
}
```

Number Guessing Game

- Example 5-6 implements a number guessing game using a flag-controlled `while` loop
- The program 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 it to an integer greater than or equal to 0 and less than 100:
    - `rand() % 100`

Case 4: EOF-Controlled `while` Loops

- Use an EOF (End Of File)-controlled `while` loop
- The logical value returned by `cin` can determine if the program has ended input

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

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

`eof` Function

- The function `eof` can determine the end of file status
- Like other I/O functions (`get`, `ignore`, `peek`), `eof` is a member of data type `istream`
- The syntax for the function `eof` is:

```
istreamVar.eof()
```

where `istreamVar` is an input stream variable, such as `cin`

More on Expressions in `while` Statements

- The expression in a `while` statement can be complex
  - For example:

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

Programming Example: Checking Account Balance

- A local bank in your town needs a program to calculate a customer’s checking account balance at the end of each month
- Data are stored in a file in the following form:

```
467343 23750.40
W 250.00
D 1200
W 75.00
I 120.74
```

Programming Example: Checking Account Balance (continued)

- The first line of data shows the account number followed by the account balance at the beginning of the month
- Thereafter each line has two entries:
  - Transaction code
  - Transaction amount
- Transaction codes
  - `W` or `w` means withdrawal
  - `D` or `d` means deposit
  - `I` or `i` means interest paid by the bank

Programming Example: Checking Account Balance (continued)

- Program updates balance after each transaction
- During the month, if at any time the balance goes below $1000.00, a $25.00 service fee is charged

Programming Example: Checking Account Balance (continued)

- Program prints the following information:
  - Account number
  - Balance at the beginning of the month
  - Balance at the end of the month
  - Interest paid by the bank
  - Total amount of deposit
  - Number of deposits
  - Total amount of withdrawal
  - Number of withdrawals
  - Service charge if any

Programming Example: Input and Output

- Input: file consisting of data in the previous format
- Output is of the following form:

```
Account Number: 467343
Beginning Balance: $23750.40
Ending Balance: $24611.49
Interest Paid: $366.24
Amount Deposited: $2230.50
Number of Deposits: 3
Amount Withdrawn: $1735.65
Number of Withdrawals: 6
```

Programming Example: Program Analysis

- The first entry in the input file is the account number and the beginning balance
- Program first reads account number and beginning balance
- Thereafter, each entry in the file is of the following form:

```
transactionCode transactionAmount
```

- To determine account balance, process each entry that contains transaction code and transaction amount

Programming Example: Program Analysis (continued)

- Begin with starting balance and then update account balance after processing each entry
- If transaction code is `D`, `d`, `I`, or `i`, transaction amount is added to the account balance
- If the transaction code is `W` or `w`, transaction amount is subtracted from the balance
- Keep separate counts of withdrawals and deposits

Programming Example: Analysis Algorithm

- Algorithm:
  - Declare the variables
  - Initialize the variables
  - Get the account number and beginning balance
  - Get transaction code and transaction amount
  - Analyze transaction code and update the appropriate variables
  - Repeat Steps 4 and 5 for all data
  - Print the result

Programming Example: Variables and Constants

```
int acctNumber;
double beginningBalance;
double accountBalance;

double amountDeposited;
int numberOfDeposits;
double amountWithdrawn;
int numberOfWithdrawals;

double interestPaid;

char transactionCode;
double transactionAmount;

bool isServiceCharged;

ifstream infile;   //input file stream variable
ofstream outfile;  //output file stream variable

const double MINIMUM_BALANCE = 1000.00;
const double SERVICE_CHARGE = 25.00;
```

Programming Example: Steps

- Declare variables as discussed previously
- Initialize variables
  - `isServiceCharged` is initialized to `false`
  - Read the beginning balance in the variable `beginningBalance` from the file and initialize the variable `accountBalance` to the value of the variable `beginningBalance`
  - Since the data will be read from a file, you need to open input file

Programming Example: Steps (continued)

- Get account number and starting balance

```
infile >> acctNumber >> beginningBalance;
```

- Get transaction code and transaction amount

```
infile >> transactionCode
       >> transactionAmount;
```

- Analyze transaction code and update appropriate variables

Programming Example: Steps (continued)

- Repeat Steps 4 and 5 until there is no more data
  - Since the number of entries in the input file is not known, use an EOF-controlled while loop
- Print the result

Programming Example: Main Algorithm

- Declare and initialize variables
- Open input file
- If input file does not exist, exit
- Open output file
- Output numbers in appropriate formats
- Read `accountNumber` and `beginningBalance`

Programming Example: Main Algorithm (continued)

- Set `accountBalance` to `beginningBalance`
- Read `transactionCode` and `transactionAmount`

```
while (not end of input file)
    if transactionCode is 'D' or 'd'
        Add transactionAmount to accountBalance
        Increment numberOfDeposits
    if transactionCode is 'I' or 'i'
        Add transactionAmount to accountBalance
        Add transactionAmount to interestPaid
```

Programming Example: Main Algorithm (continued)

```
if transactionCode is 'W' or 'w'
    Subtract transactionAmount from
        accountBalance
    Increment numberOfWithdrawals
    if (accountBalance < MINIMUM_BALANCE
            && !isServicedCharged)
        Subtract SERVICE_CHARGE  from accountBalance
        Set isServiceCharged to true
if transactionCode is other than 'D', 'd', 'I',
    'i', 'W', or 'w', output an error message
```

- Output the results

`for` Looping (Repetition) Structure

- 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
  - `initial statement` usually initializes a variable (called the `for` **loop control**, or `for` **indexed**, **variable**)
- In C++, `for` is a reserved word

for Looping (Repetition) Structure (continued)

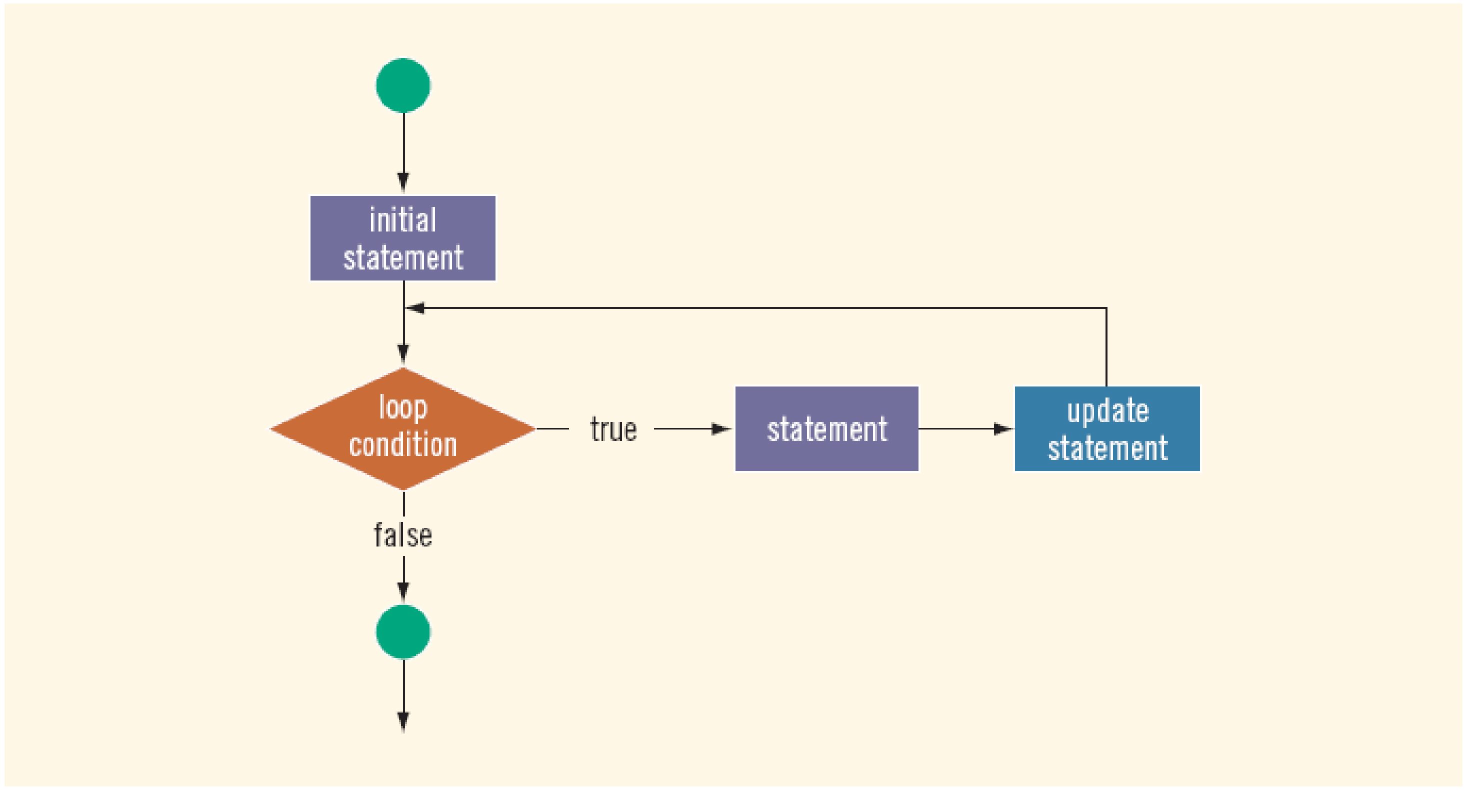


FIGURE 5-2 for loop

`for` Looping (Repetition) Structure (continued)

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

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

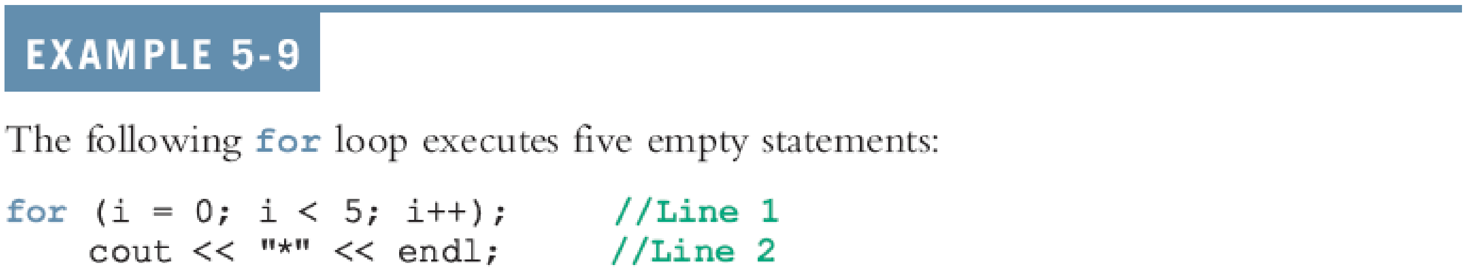
   ```
   for (i = 1; i <= 5; i++)
   {
       cout << "Hello!" << endl;
       cout << "*" << endl;
   }
   ```

2. Consider the following **for** loop:

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

for Looping (Repetition) Structure (continued)

- C++ allows you to use fractional values for loop control variables of the double type
  - Results may differ
- The following is a semantic error:



EXAMPLE 5-9

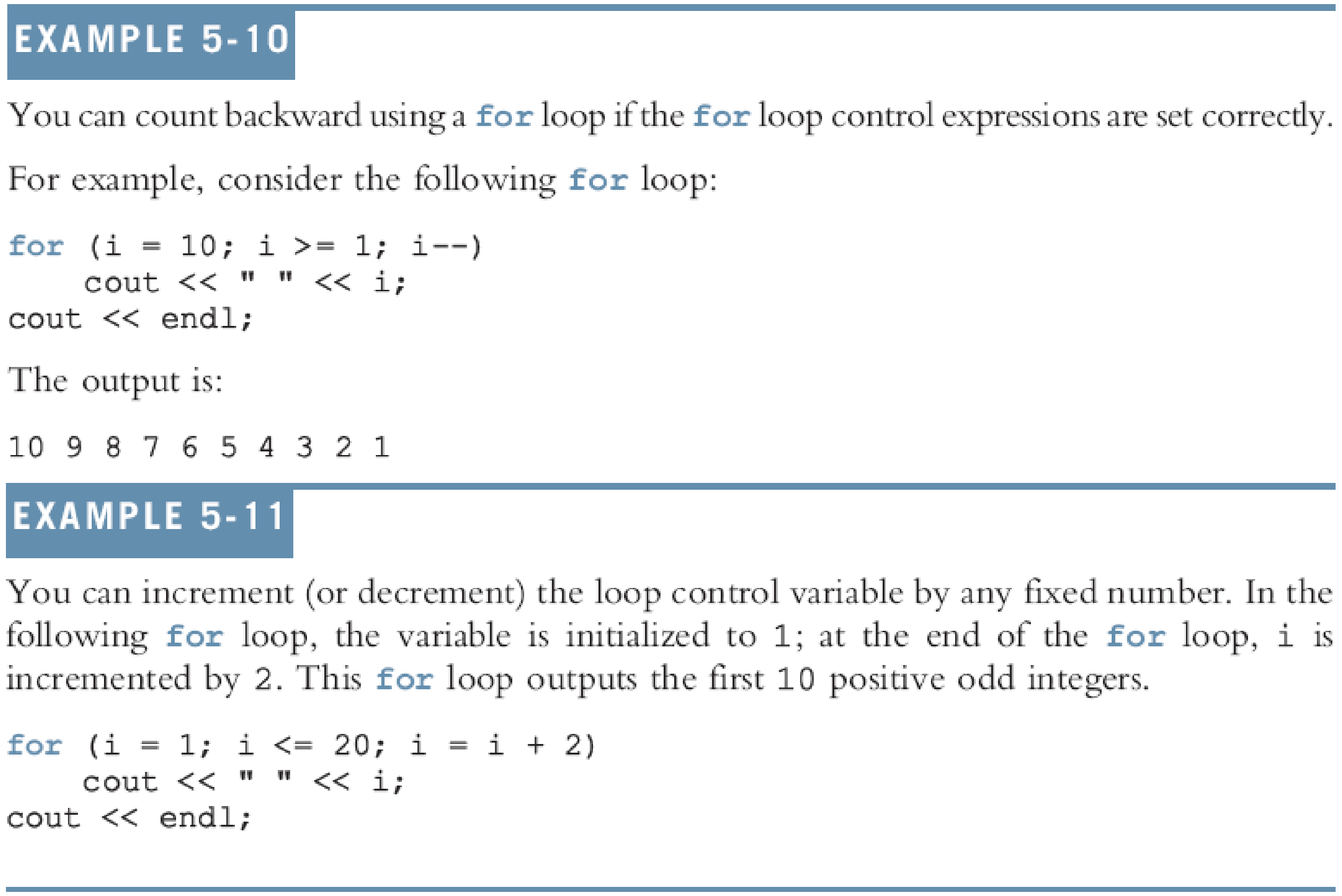
The following **for** loop executes five empty statements:

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

- The following is a legal for loop:

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

`for` Looping (Repetition) Structure (continued)



EXAMPLE 5-10

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
```

EXAMPLE 5-11

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;
```

do...while Looping (Repetition) Structure

- General form of a do...while:

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

- The statement executes first, and then the expression is evaluated
- To avoid an infinite loop, body must contain a statement that makes the expression false
- The statement can be simple or compound
- Loop always iterates at least once

do…while Looping (Repetition) Structure (continued)

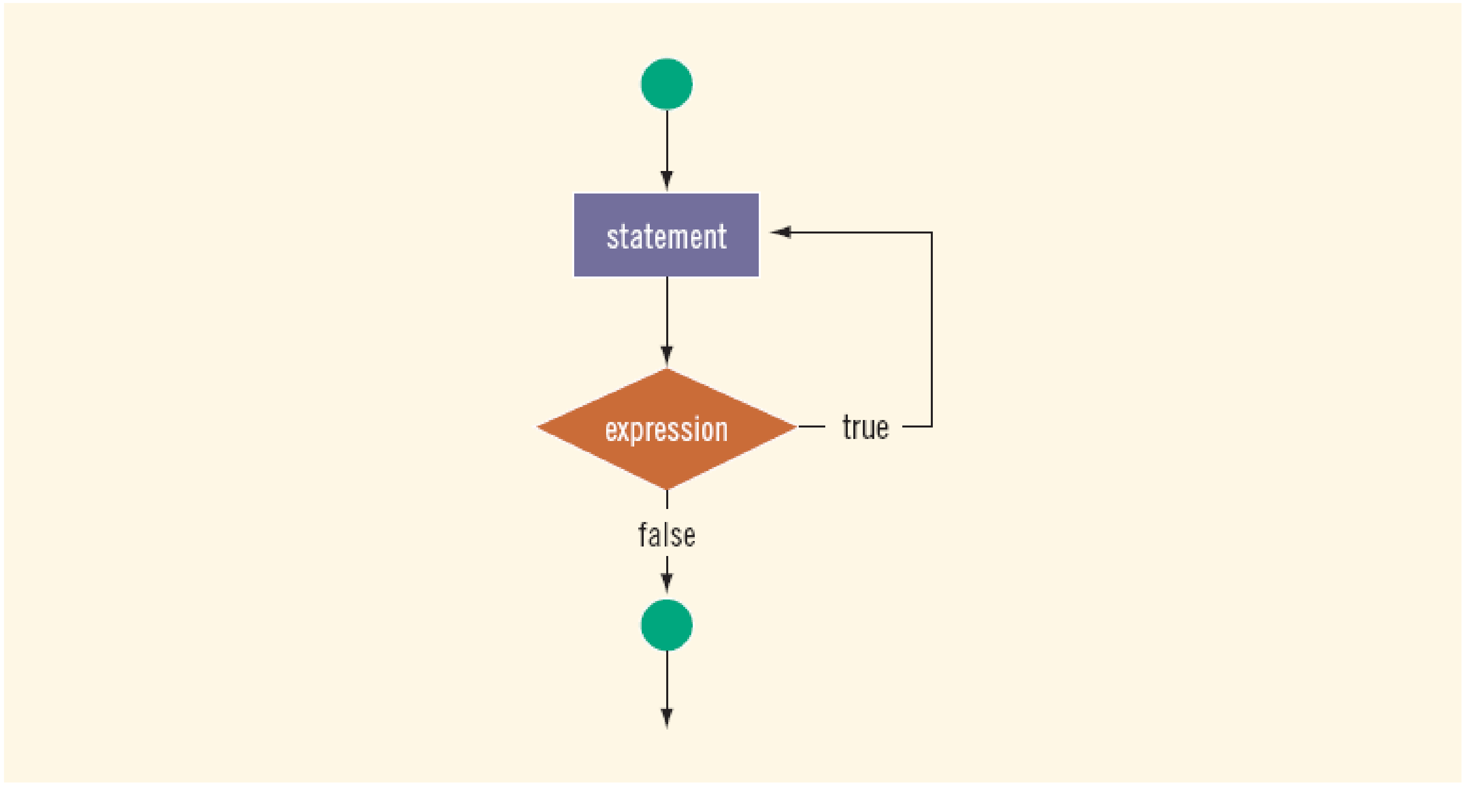
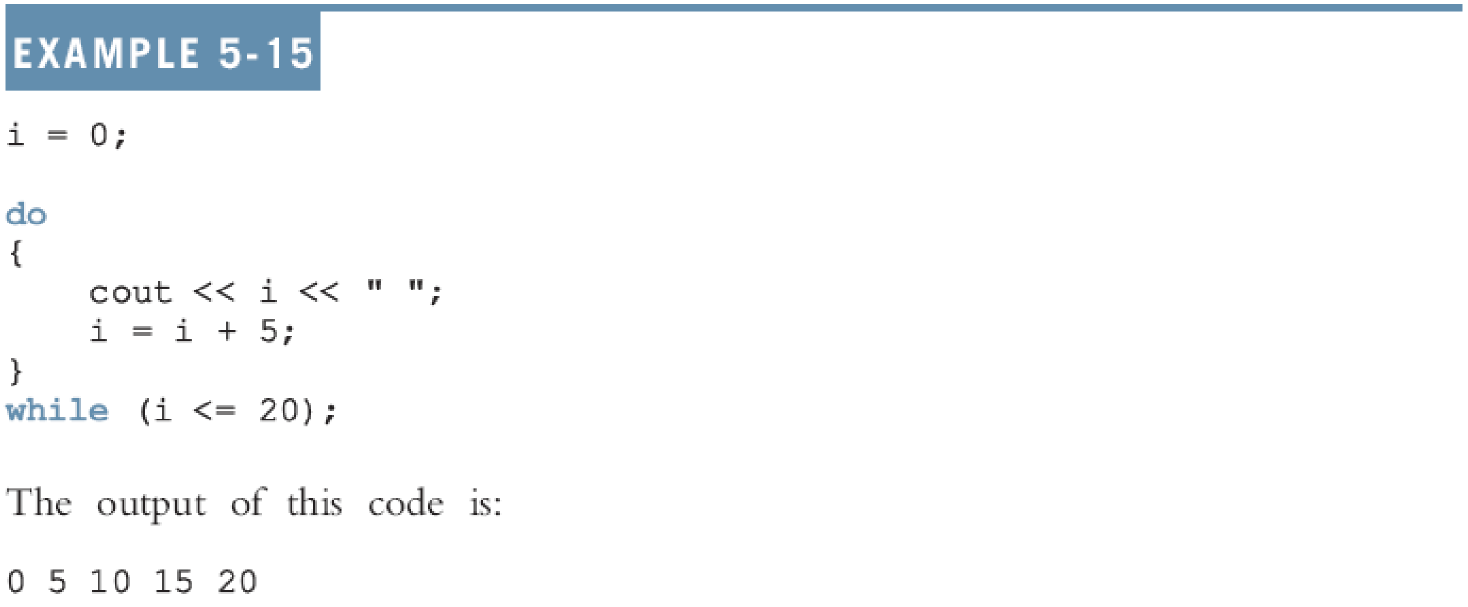


FIGURE 5-3 do…while loop

`do...while` Looping (Repetition) Structure (continued)



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

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.

Divisibility Test by 3 and 9

```
sum = 0;

do
{
    sum = sum + num % 10; //extract the last digit
                          //and add it to sum
    num = num / 10;       //remove the last digit
}
while (num > 0);

cout << "The sum of the digits = " << sum << endl;

if (sum % 3 == 0)
    cout << temp << " is divisible by 3" << endl;
else
    cout << temp << " is not divisible by 3" << endl;

if (sum % 9 == 0)
    cout << temp << " is divisible by 9" << endl;
else
    cout << temp << " is not divisible by 9" << endl;
```

Choosing the Right Looping Structure

- All three loops have their place in C++
  - If you know or 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

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

`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

Nested Control Structures

- To create the following pattern:

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

- We can use the following code:

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

Nested Control Structures (continued)

- What is the result 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 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

Summary (continued)

- `while`: expression is the decision maker, and 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`

Summary (continued)

- `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