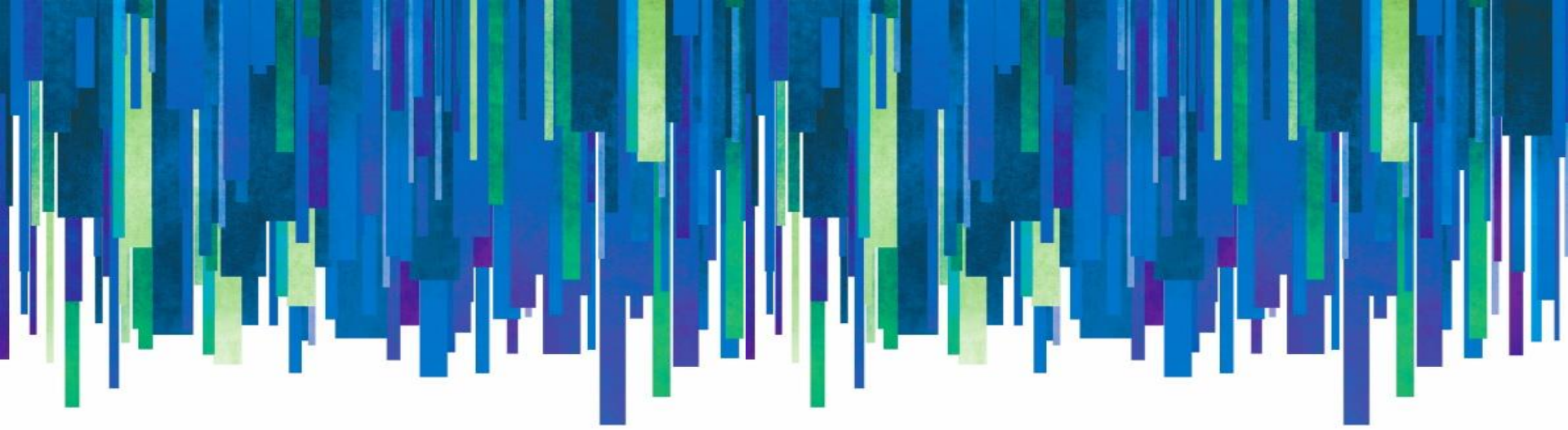

Introduction to Programming in C++
Seventh Edition

Chapter 7:
The Repetition Structure

Objectives

- Differentiate between a pretest loop and a posttest loop
- Include a pretest loop in pseudocode
- Include a pretest loop in a flowchart
- Code a pretest loop using the C++ `while` statement
- Utilize counter and accumulator variables
- Code a pretest loop using the C++ `for` statement

Repeating Program Instructions

- The **repetition structure**, or **loop**, processes one or more instructions repeatedly
- Every loop contains a Boolean condition that controls whether the instructions are repeated
- A **looping condition** says whether to continue looping through instructions
- A **loop exit condition** says whether to stop looping through the instructions
- Every looping condition can be expressed as a loop exit condition (its opposite)

Repeating Program Instructions (cont'd.)

- C++ uses looping conditions in repetition structures
- A repetition structure can be pretest or posttest
- In a **pretest loop**, the condition is evaluated *before* the instructions in the loop are processed
- In a **posttest loop**, the condition is evaluated *after* the instructions in the loop are processed
- In both cases, the condition is evaluated with each repetition

Repeating Program Instructions (cont'd.)

- The instructions in a posttest loop will always be processed at least once
- Instructions in a pretest loop may not be processed if the condition initially evaluates to false
- The repeatable instructions in a loop are called the **loop body**
- The condition in a loop must evaluate to a Boolean value

Repeating Program Instructions (cont'd.)

Problem specification

A superheroine named Isis must prevent a poisonous yellow spider from attacking King Khafra and Queen Rashida. Isis has one weapon at her disposal: a laser beam that shoots out from her right hand. Unfortunately, Isis gets only one shot at the spider, which is flying around the palace looking for the king and queen. Before taking the shot, she needs to position both her right arm and her right hand toward the spider. After taking the shot, she should return her right arm and right hand to their original positions. In addition, she should say "You are safe now. The spider is dead." if the laser beam hit the spider; otherwise, she should say "Run for your lives, my king and queen!"

```
1. position both your right arm and your right hand toward the spider
2. shoot a laser beam at the spider
3. return your right arm and right hand to their original positions
4. if (the laser beam hit the spider)
        say "You are safe now. The spider is dead."
   else
        say "Run for your lives, my king and queen!"
   end if
```

Figure 7-1 A problem that requires the sequence and selection structures

Repeating Program Instructions (cont'd.)

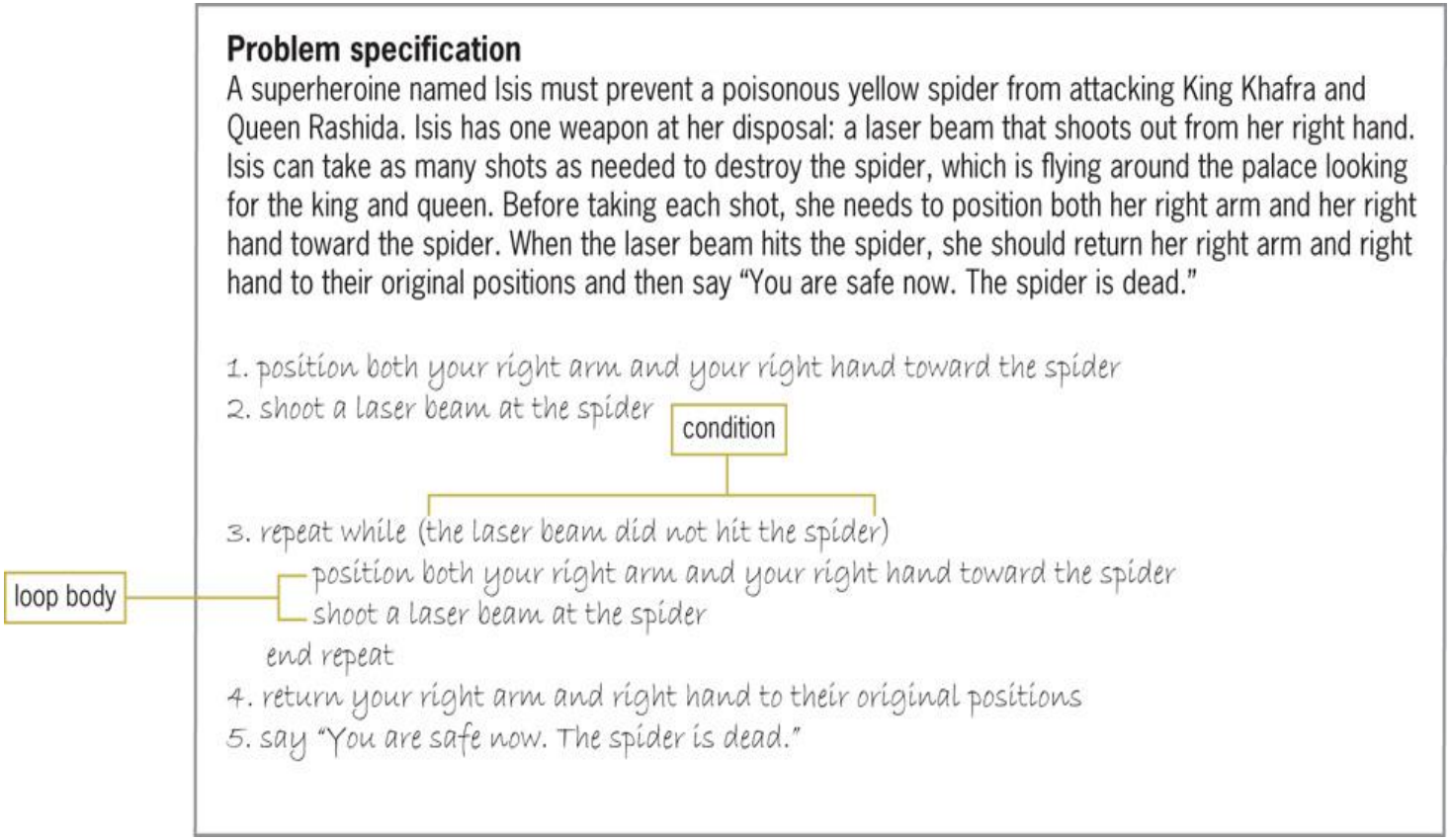

Figure 7-2 A problem that requires the sequence and repetition structures

Using a Pretest Loop to Solve a Real-World Problem

- Most loops have a condition and a loop body
- Some loops require the user to enter a special **sentinel value** to end the loop
- Sentinel values should be easily distinguishable from the valid data recognized by the program
- When a loop's condition evaluates to true, the instructions in the loop body are processed
- Otherwise, the instructions are skipped and processing continues with the first instruction after the loop

Using a Pretest Loop to Solve a Real-World Problem (cont.)

- After each processing of the loop body (iteration), the loop's condition is reevaluated to determine if the instructions should be processed again
- A **priming read** is an instruction that appears before a loop and is used to set up the loop with an initial value entered by the user
- An **update read** is an instruction that appears within a loop that allows the user to enter a new value at each iteration of the loop

Using a Pretest Loop to Solve a Real-World Problem (cont'd.)

Problem specification

Recently, the owner of the Totally Sweet Shoppe hired one part-time employee, who earns $10 per hour. The owner wants a program that calculates and displays the employee's weekly gross pay.

Input	Processing	Output
pay rate ($10 per hour) hours worked	Processing items: none	gross pay

Algorithm:

1. enter the hours worked
2. calculate the gross pay by multiplying the hours worked by the pay rate
3. display the gross pay

calculates and displays the gross pay for one employee only

Figure 7-4 Problem specification and IPO chart for the Totally Sweet Shoppe program

Using a Pretest Loop to Solve a Real-World Problem (cont'd.)

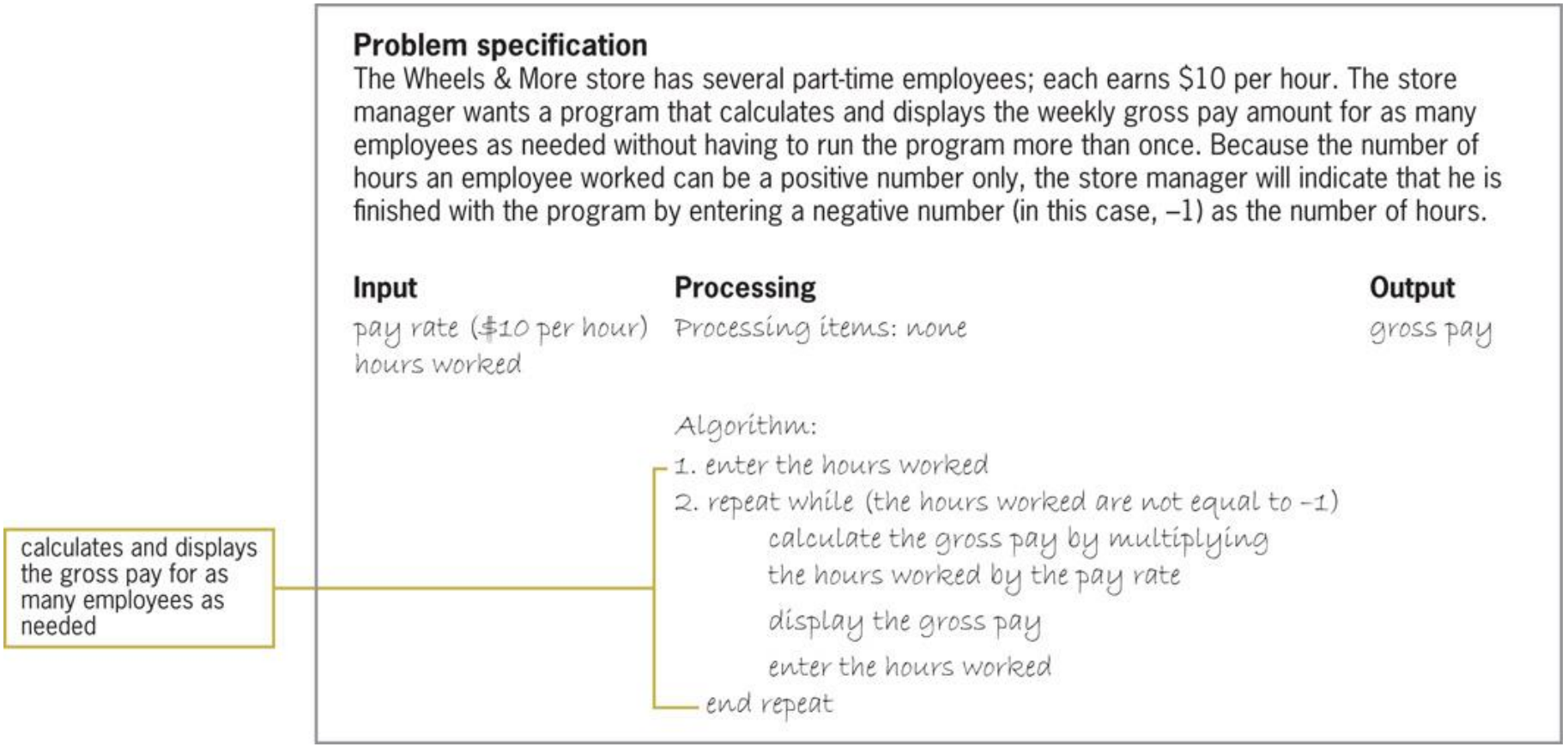

Figure 7-5 Problem specification and IPO chart for the Wheels & More program

Using a Pretest Loop to Solve a Real-World Problem (cont'd.)

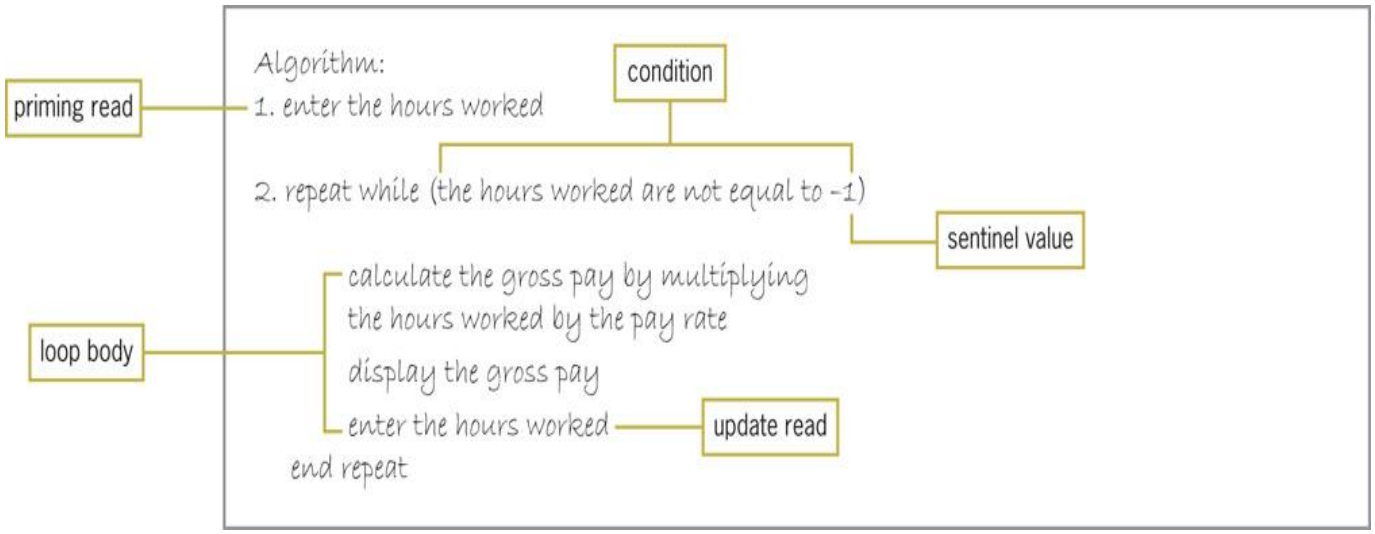

Figure 7-6 Components of the Wheels & More algorithm

Flowcharting a Pretest Loop

- The diamond symbol in a flowchart is the decision symbol – represents repetition structures
- A diamond representing a repetition structure contains a Boolean condition
- The condition determines whether the instructions in the loop are processed
- A diamond representing a repetition structure has one flowline leading into it and two leading out

Flowcharting a Pretest Loop (cont'd.)

- The flowlines leading out are marked "T" (true) and "F" (false)
- The "T" line points to the loop body
- The "F" line points to the instructions to be processed if the loop's condition evaluates to false
- The flowline entering the diamond and symbols and flowlines of the true path form a circle, or loop
- This distinguishes a repetition structure from a selection structure in a flowchart

Flowcharting a Pretest Loop (cont'd.)

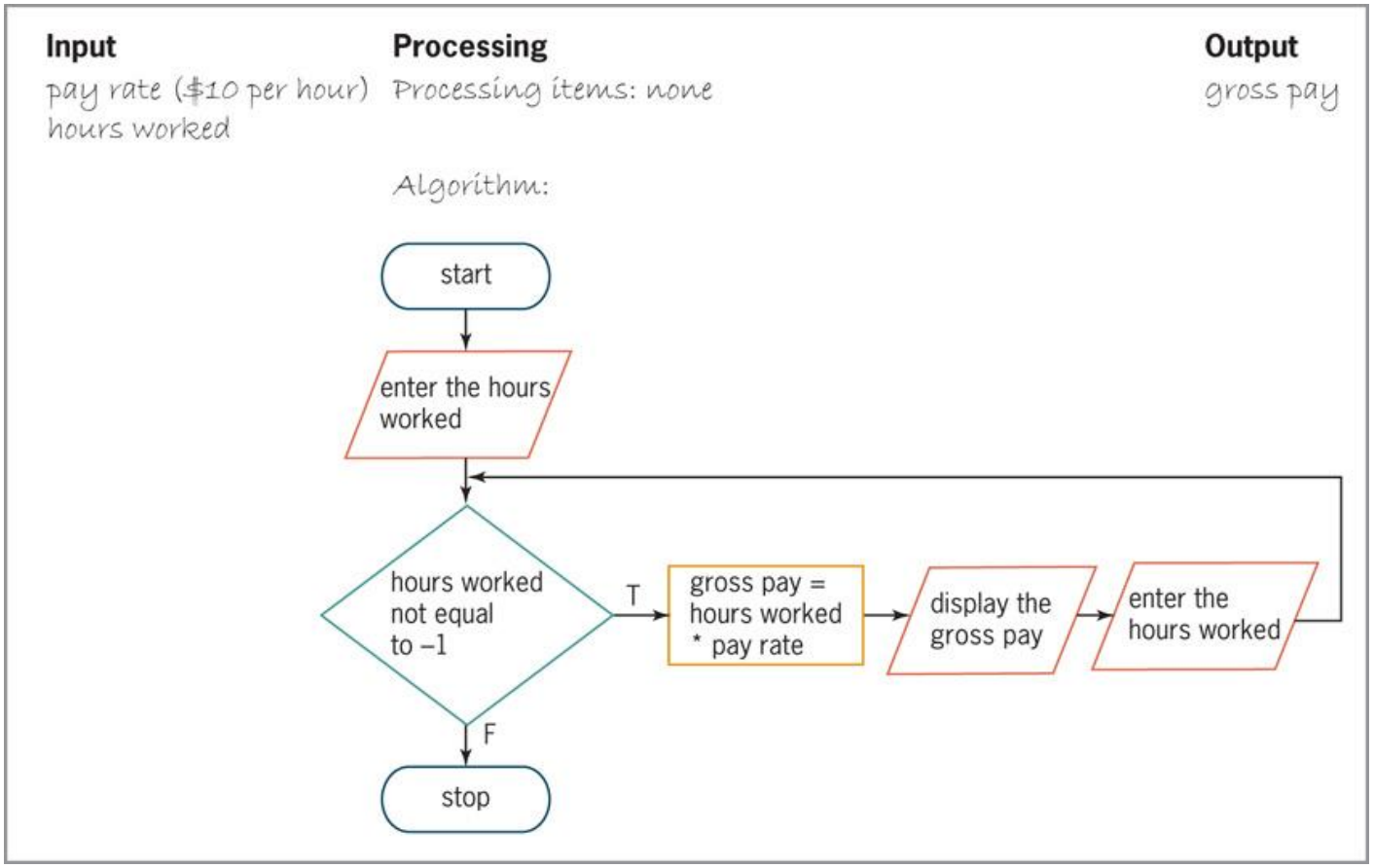

Figure 7-7 Wheels & More algorithm shown in flowchart form

Flowcharting a Pretest Loop (cont'd.)

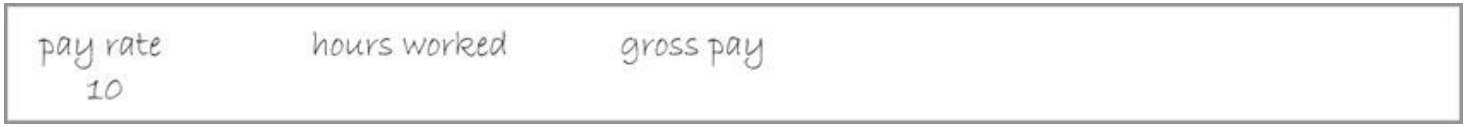

Figure 7-8 Input and output items entered in the desk-check table

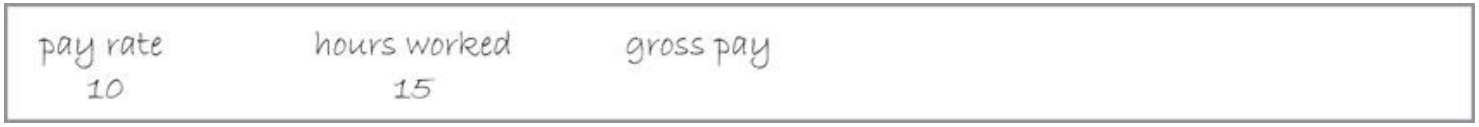

Figure 7-9 First hours worked entry in the desk-check table

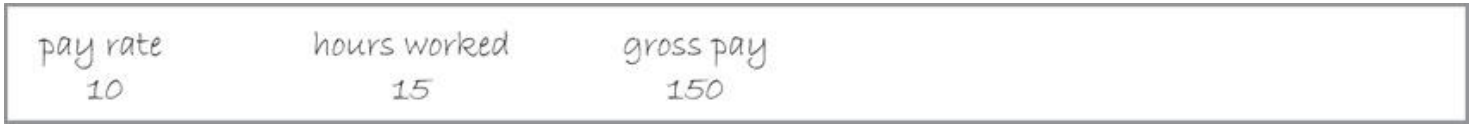

Figure 7-10 First employee's information recorded in the desk-check table

Flowcharting a Pretest Loop (cont'd.)

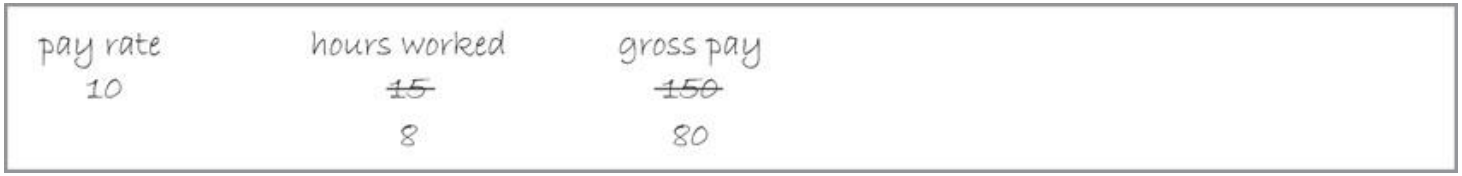

Figure 7-11 Second employee's information recorded in the desk-check table

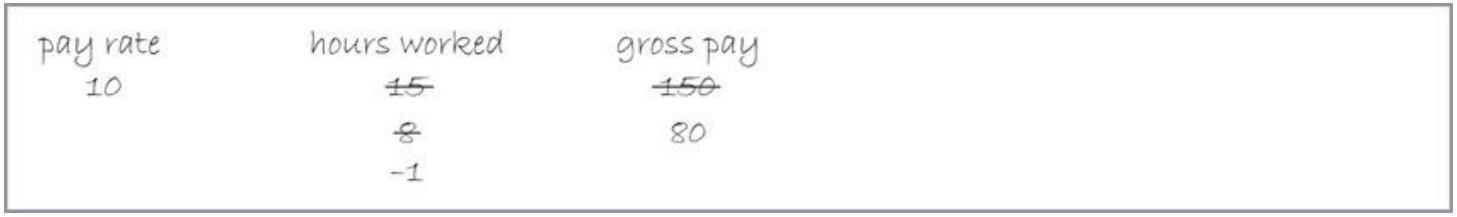

Figure 7-12 Completed desk-check table

The `while` Statement

- You can use the `while` statement to code a pretest loop in C++
- Syntax is:

  `while` (*condition*)

  one statement or a statement block to be processed as long as the condition is true
- Must supply looping condition (Boolean expression)
- *condition* can contain constants, variables, functions, arithmetic operators, comparison operators, and logical operators

The `while` Statement (cont'd.)

- Must also provide loop body statements, which are processed repeatedly as long as condition is true
- If more than one statement in loop body, must be entered as a statement block (enclosed in braces)
- Can include braces even if there is only one statement in the statement block
- Good programming practice to include a comment, such as `//end while`, to mark the end of the `while` statement

The `while` Statement (cont'd.)

- A loop whose instructions are processed indefinitely is called an **infinite loop** or **endless loop**
- You can usually stop a program that has entered an infinite loop by pressing Ctrl+c

The `while` Statement (cont'd.)

HOW TO Use the `while` Statement

Syntax

while (*condition*)
either one statement or a statement block to be processed as long as the condition is true
`//end while`

Example 1

```
int age = 0;

cout << "Enter age: ";
cin >> age;
while (age > 0)
{
    cout << "You entered " << age << endl;
    cout << "Enter age: ";
    cin >> age;
}   //end while
```

Figure 7-13 How to use the `while` statement

The `while` Statement (cont'd.)

Example 2

```
char makeEntry = ' ';
double sales = 0.0;

cout << "Enter a sales amount? (Y/N)";
cin >> makeEntry;
while (makeEntry == 'Y' || makeEntry == 'y')
{
    cout << "Enter the sales: ";
    cin >> sales;
    cout << "You entered " << sales << endl;
    cout << "Enter a sales amount? (Y/N)";
    cin >> makeEntry;
}   //end while
```

An alternate example using the `while` statement

The `while` Statement (cont'd.)

IPO chart information	C++ instructions
Input	
pay rate ($10 per hour)	`const double RATE = 10.0;`
hours worked	`double hours = 0.0;`
Processing	
none	
Output	
gross pay	`double gross = 0.0;`
Algorithm	
1. enter the hours worked	`cout << "First hours worked (-1 to stop): ";` `cin >> hours;`
2. repeat while (the hours worked are not equal to -1)	`while (hours != -1)` `{`
calculate the gross pay by multiplying the hours worked by the pay rate	`gross = hours * RATE;`
display the gross pay	`cout << "Gross pay: $" << gross;` `cout << endl << endl;`
enter the hours worked	`cout << "Next hours worked (-1 to stop): ";` `cin >> hours;`
end repeat	`}  //end while`

Figure 7-14 IPO chart information and C++ instructions for the Wheels & More program

The `while` Statement (cont'd.)

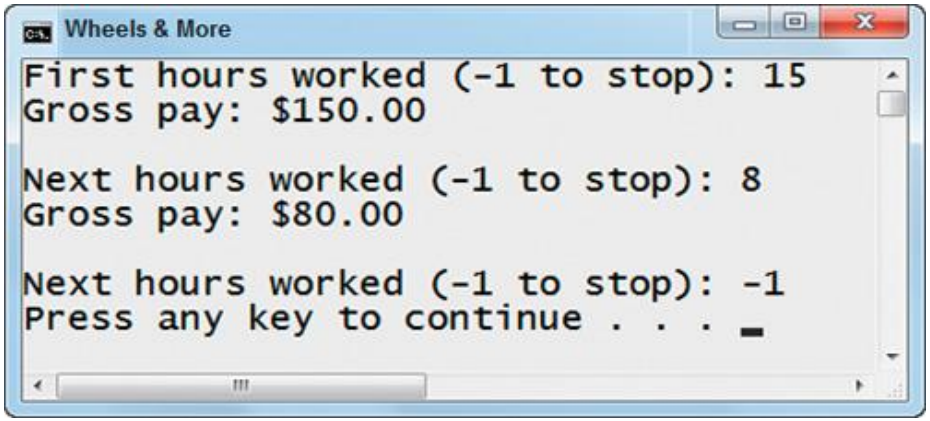

Figure 7-15 A sample run of the Wheels & More program

Using Counters and Accumulators

- Some problems require you to calculate a total or average
- To do this, you use a counter, accumulator, or both
- A **counter** is a numeric variable used for counting something
- An **accumulator** is a numeric variable used for accumulating (adding together) multiple values
- Two tasks are associated with counters and accumulators: initializing and updating

Using Counters and Accumulators (cont'd.)

- **Initializing** means assigning a beginning value to a counter or accumulator (usually 0) – happens once, before the loop is processed
- **Updating** (or **incrementing**) means adding a number to the value of a counter or accumulator
- A counter is updated by a constant value (usually 1)
- An accumulator is updated by a value that varies
- Update statements are placed in the body of a loop since they must be performed at each iteration

The Sales Express Program

- Example problem and program solution (following slides)
  - Program takes in a sequence of sales amounts from the keyboard and outputs their average
  - Uses a counter to keep track of the number of sales entered and an accumulator to keep track of the total sales
  - Both are initialized to 0
  - The loop ends when the user enters a sentinel value (-1)

The Sales Express Program (cont'd.)

Problem specification

Sales Express wants a program that displays the average amount the company sold during the prior year. The sales manager will enter each salesperson's sales. The program will use a counter to keep track of the number of sales amounts entered and an accumulator to total the sales amounts. When the sales manager has finished entering the sales amounts, the program will calculate the average sales amount by dividing the value stored in the accumulator by the value stored in the counter. It then will display the average sales amount on the screen. The sales manager will indicate that she is finished with the program by entering a negative number as the sales amount. If the sales manager does not enter any sales amounts, the program should display the "No sales entered" message.

Figure 7-17 Problem specification for the Sales Express program

The Sales Express Program (cont'd.)

IPO chart information	C++ instructions
Input	
sales	`double sales = 0.0;`
Processing	
number of sales entered (counter)	`int numSales = 0;`
total sales (accumulator)	`double totalSales = 0.0;`
Output	
average sales	`double average = 0.0;`
Algorithm	
1. enter the sales	`cout << "First sales amount (negative number to stop): ";` `cin >> sales;`
2. repeat while (the sales are at least 0)	`while (sales >= 0.0)`
	`{`
add 1 to the number of sales entered	`numSales = numSales + 1;`
add the sales to the total sales	`totalSales = totalSales + sales;`
enter the sales	`cout << "Next sales amount (negative number to stop): ";` `cin >> sales;`
end repeat	`} //end while`

Figure 7-17 IPO chart information and C++ instructions for the Sales Express program

The Sales Express Program (cont'd.)

3. if (the number of sales entered is greater than 0)	`if (numSales > 0)`
	`{`
calculate the average sales by dividing the total sales by the number of sales entered	`average = totalSales / numSales;`
display the average sales	`cout << "Average: $" << average << endl;`
	`}`
else	`else`
display "No sales entered" message	`cout << "No sales entered" << endl;`
end if	`//end if`

Figure 7-17 IPO chart information and C++ instructions for the Sales Express program (cont'd)

The Sales Express Program (cont'd.)

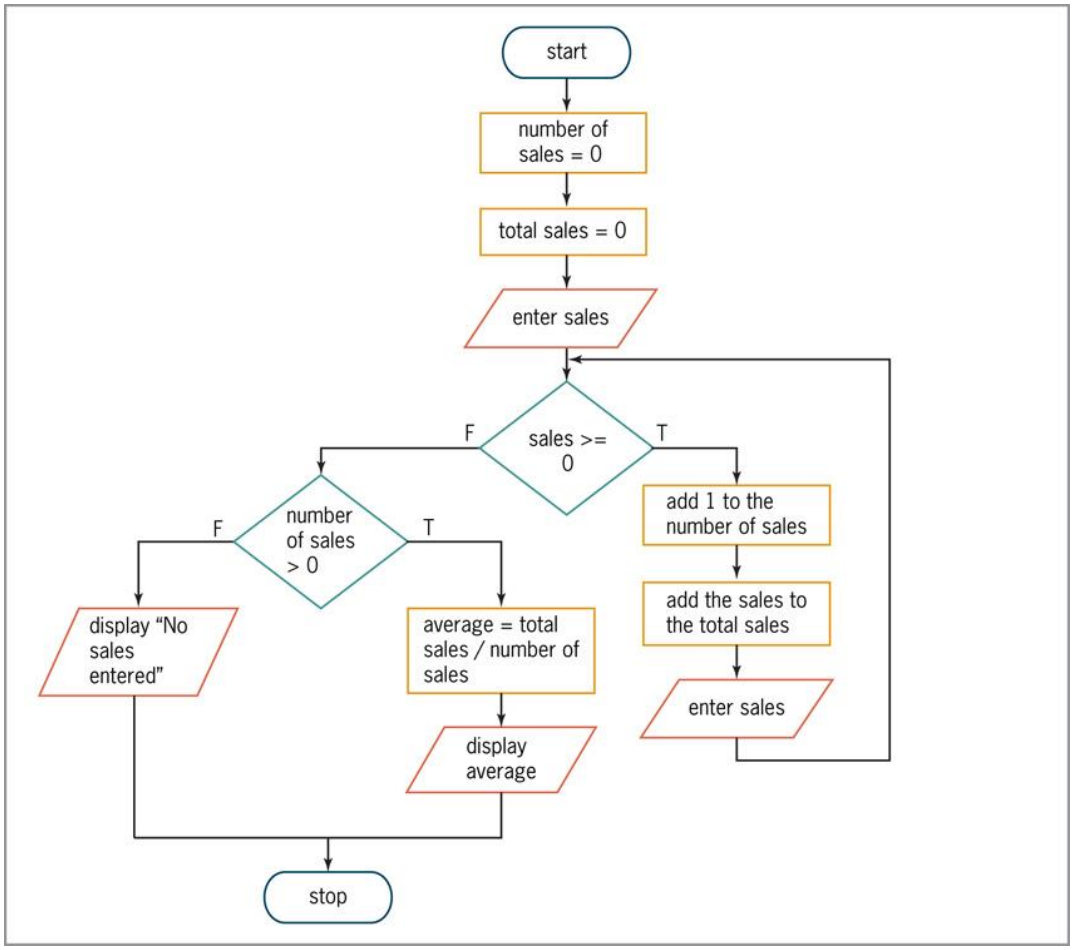

Figure 7-18 Flowchart for the Sales Express program

The Sales Express Program (cont'd.)

sales	numSales	totalSales	average
~~0.0~~	0	0.0	0.0
30000.0			

Figure 7-19 Desk-check table after the first sales amount is entered

sales	numSales	totalSales	average
~~0.0~~	~~0~~	~~0.0~~	0.0
30000.0	1	30000.0	

Figure 7-20 Desk-check showing the first update to the counter and accumulator variables

The Sales Express Program (cont'd.)

sales	numSales	totalSales	average
~~0.0~~	~~0~~	~~0.0~~	0.0
~~30000.0~~	~~1~~	~~30000.0~~	
40000.0	2	70000.0	

Figure 7-21 Desk-check table after the second update to the counter and accumulator variables

sales	numSales	totalSales	average
~~0.0~~	~~0~~	~~0.0~~	~~0.0~~
~~30000.0~~	~~1~~	~~30000.0~~	35000.0
~~40000.0~~	2	70000.0	
-3.0			

Figure 7-22 Completed desk-check for the Sales Express program

The Sales Express Program (cont'd.)

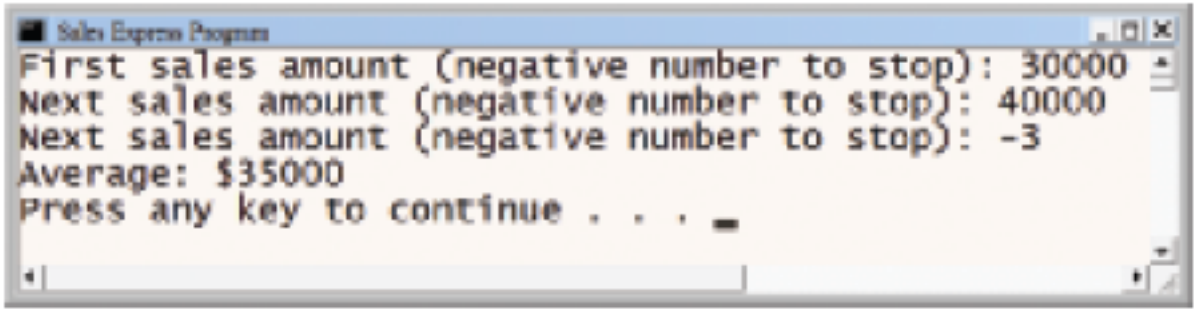

Figure 7-23 First sample run of the Sales Express program

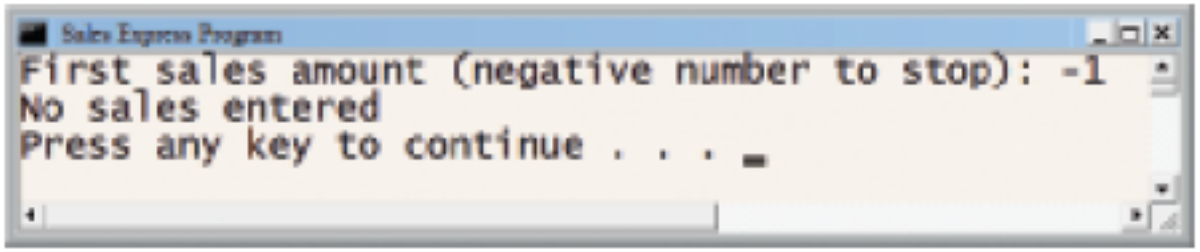

Figure 7-24 Second sample run of the Sales Express program

Counter-Controlled Pretest Loops

- Loops can be controlled using a counter rather than a sentinel value
- These are called **counter-controlled loops**
- Example problem and program solution (following slides)
  - Counter-controlled loop is used that totals the quarterly sales from three regions
  - Loop repeats three times, once for each region, using a counter to keep track

Counter-Controlled Pretest Loops (cont'd.)

Problem specification
The sales manager at Jasper Music Company wants a program that allows him to enter the quarterly sales amount made in each of three regions: Region 1, Region 2, and Region 3. The program should calculate the total quarterly sales and then display the result on the screen. The program will use a counter to ensure that the sales manager enters exactly three sales amounts. It will use an accumulator to total the sales amounts.

Figure 7-25 Problem specification for the Jasper Music Company program

Counter-Controlled Pretest Loops (cont'd.)

IPO chart information	C++ instructions
Input	
region's quarterly sales	`int regionSales = 0;`
Processing	
number of regions (counter: 1 to 3)	`int numRegions = 1;`
Output	
total quarterly sales (accumulator)	`int totalSales = 0;`
Algorithm	
1. repeat while (the number of regions is less than 4)	`while (numRegions < 4)`
	`{`
enter the region's quarterly sales	`cout << "Enter region " << numRegions << "'s quarterly sales: ";` `cin >> regionSales;`
add the region's quarterly sales to the total quarterly sales	`totalSales += regionSales;`
add 1 to the number of regions	`numRegions += 1;`
end repeat	`} //end while`
2. display the total quarterly sales	`cout << "Total quarterly sales: $" << totalSales << endl;`

Figure 7-25 IPO chart information and C++ instructions for the Jasper Music Company program

Counter-Controlled Pretest Loops (cont'd.)

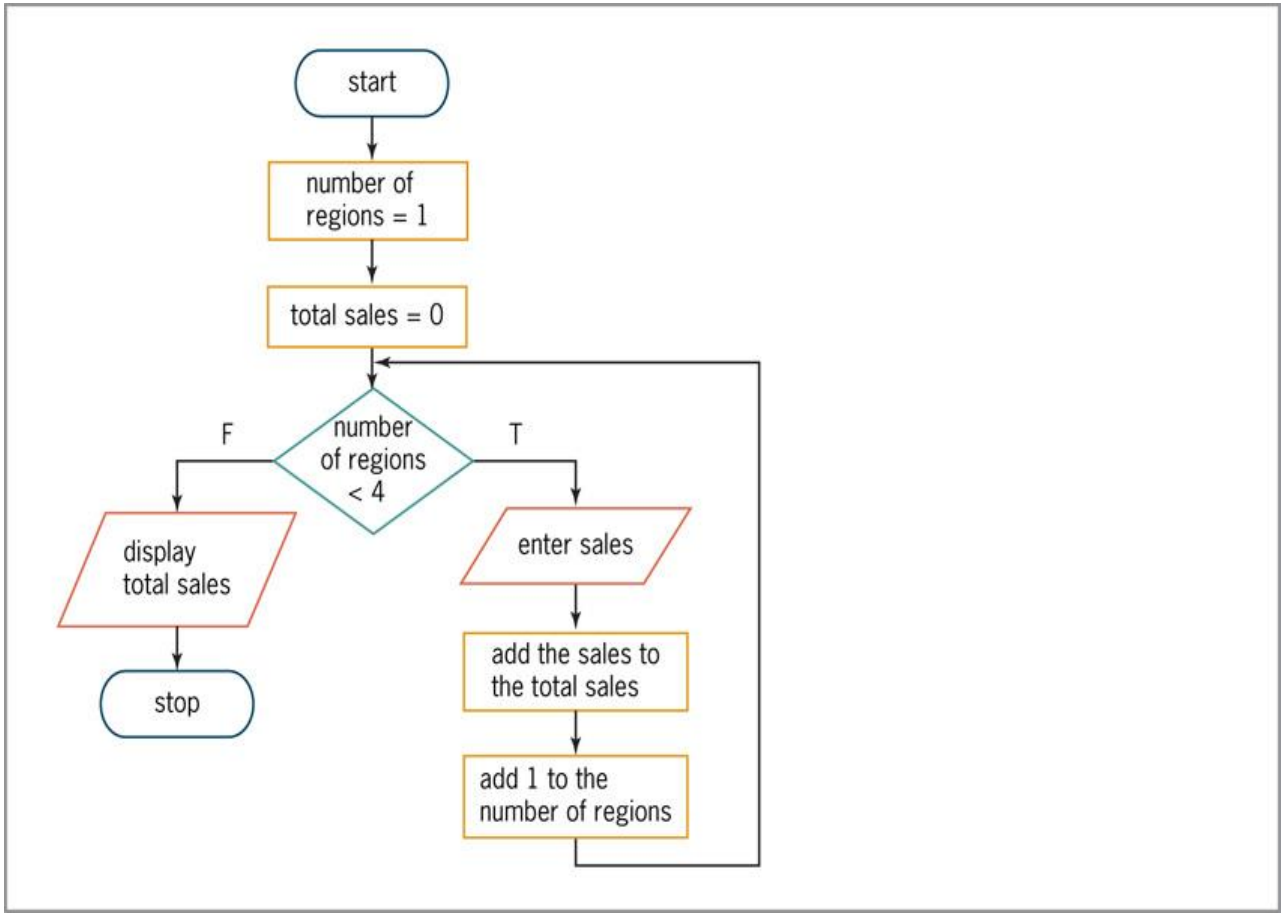

Figure 7-26 Flowchart for the Jasper Music Company program

Counter-Controlled Pretest Loops (cont'd.)

regionSales	numRegions	totalSales
0	1	0

Figure 7-27 Desk-check table after the variable declaration statements are processed

regionSales	numRegions	totalSales
~~0~~	~~1~~	~~0~~
2500	2	2500

Figure 7-28 Results of processing loop body instructions first time

regionSales	numRegions	totalSales
~~0~~	~~1~~	~~0~~
~~2500~~	~~2~~	~~2500~~
6000	3	8500

Figure 7-29 Results of processing loop body instructions second time

Counter-Controlled Pretest Loops (cont'd.)

regionSales	numRegions	totalSales
0	1	0
2500	2	2500
6000	3	8500
2000	4	10500

Figure 7-30 Results of processing loop body instructions third time

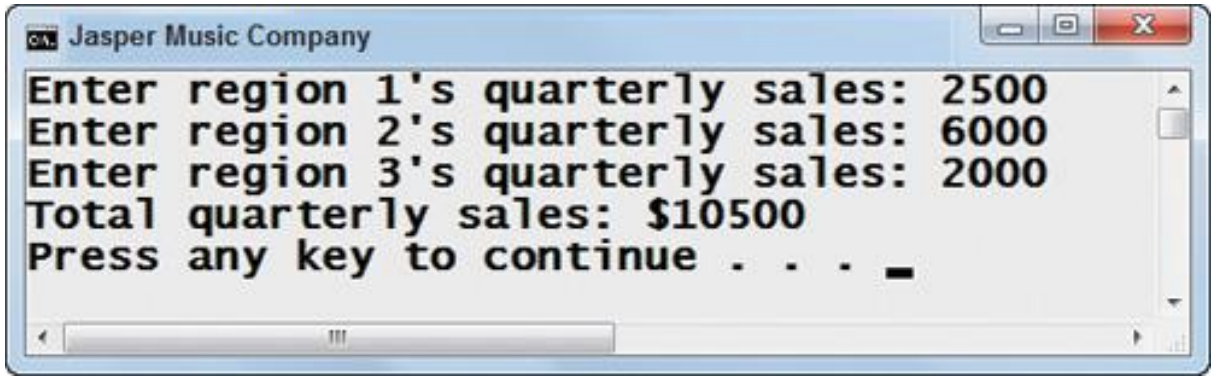

Figure 7-31 Sample run of Jasper Music Company program

The `for` Statement

- The `for` statement can also be used to code any pretest loop in C++
- Commonly used to code counter-controlled pretest loops (more compact than `while` in this case)
- Syntax:

  `for` (*[initialization]; condition; [update]*)

  one statement or a statement block to be processed as long as condition is true

- *initialization* and *update* arguments are optional

- Make one for loop count by 3's – to 100
- Make one for loop count by 5's – to 100

- Make program say 50 and print something that will make me laugh.
- Print out

```
0
 0
  0
   0
```

- Write an application that displays a perfect number.
- A perfect number is a number that is divisible by 1, 2, and 3.
- But it cannot be greater than 12.
- Count 1 to 1000 has to be positive

The `for` Statement (cont'd.)

- *initialization* argument usually creates and initializes a counter variable
- Counter variable is local to `for` statement (can only be used inside the loop)
- *condition* argument specifies condition that must be true for the loop body to be processed
- *condition* must be a Boolean expression
  - May contain variables, constants, functions, arithmetic operators, comparison operators, and logical operators

The `for` Statement (cont'd.)

- Loop ends when *condition* evaluates to false
- *update* argument usually contains an expression that updates the counter variable
- Loop body follows the `for` clause
  - Must be placed in braces if it contains more than one statement
  - Braces can be used even if the loop body contains only one statement
- Good programming practice to place a comment, such as `//end for`, to mark the end of a `for` loop

The `for` Statement (cont'd.)

HOW TO Use the `for` Statement

Syntax

for ([*initialization*]; *condition*; [*update*])
    either one statement or a statement block to be processed as long as the condition is true
`//end for`

semicolons

Example 1: displays the numbers 1, 2, and 3 on separate lines on the screen

```
for (int x = 1; x < 4; x += 1)
    cout << x << endl;
//end for
```

you also can use `x = x + 1`

Example 2: displays the numbers 3, 2, and 1 on separate lines on the screen

```
for (int x = 3; x > 0; x = x - 1)
    cout << x << endl;
//end for
```

you also can use `x -= 1`

Figure 7-32 How to use the `for` statement

The `for` Statement (cont'd.)

Processing steps for Example 1

1. The *initialization* argument (`int x = 1`) tells the computer to create a variable named x and initialize it to the number 1.
2. The *condition* argument (`x < 4`) tells the computer to check whether the x variable's value is less than 4. It is, so the computer processes the statement in the loop body. That statement displays the x variable's value (1) on the screen.
3. The *update* argument (`x += 1`) tells the computer to add the number 1 to the contents of the x variable, giving 2.
4. The *condition* argument tells the computer to check whether the x variable's value is less than 4. It is, so the computer processes the statement in the loop body. That statement displays the x variable's value (2) on the screen.
5. The *update* argument tells the computer to add the number 1 to the contents of the x variable, giving 3.
6. The *condition* argument tells the computer to check whether the x variable's value is less than 4. It is, so the computer processes the statement in the loop body. That statement displays the x variable's value (3) on the screen.
7. The *update* argument tells the computer to add the number 1 to the contents of the x variable, giving 4.
8. The *condition* argument tells the computer to check whether the x variable's value is less than 4. It's not, so the computer stops processing the `for` loop and removes its local x variable. Processing continues with the statement following the end of the loop.

Figure 7-33 Processing steps for the code shown in Example 1 in Figure 7-32

The Holmes Supply Program

- Extended example of a problem and program solution (following slides)
  - Program totals up the sales from three stores using a `for` loop

Problem specification

The payroll manager at Holmes Supply Company wants a program that allows her to enter the payroll amount for each of three stores: Store 1, Store 2, and Store 3. The program should calculate the total payroll and then display the result on the screen. The program will use a counter to ensure that the payroll manager enters exactly three payroll amounts. It will use an accumulator to total the amounts.

Figure 7-34 Problem specification for the Holmes Supply Company program

The Holmes Supply Program (cont'd.)

IPO chart information	C++ instructions
Input	
store's payroll	`int storePayroll = 0;`
Processing	
number of stores (counter: 1 to 3)	this variable is created and initialized in the for clause
Output	
total payroll (accumulator)	`int totalPayroll = 0;`
Algorithm	
1. repeat for (number of stores from 1 to 3)	`for (int numStores = 1;`
	`num Stores <= 3; numStores += 1)`
	`{`
enter the store's payroll	`cout << "Store " << numStores`
	`<< " payroll: ";`
	`cin >> storePayroll;`
add the store's payroll to the total payroll	`totalPayroll += storePayroll;`
end repeat	`} //end for`
2. display the total payroll	`cout << "Total payroll: $"`
	`<< totalPayroll << endl;`

Figure 7-34 IPO chart information and C++ instructions for the Holmes Supply Company program

The Holmes Supply Program (cont'd.)

storePayroll	totalPayroll	numStores
0	0	1

Figure 7-35 Results of processing the declaration statements and *initialization* argument

storePayroll	totalPayroll	numStores
~~0~~	~~0~~	~~1~~
15000	15000	2

Figure 7-36 Desk-check table after *update* argument is processed first time

storePayroll	totalPayroll	numStores
~~0~~	~~0~~	~~1~~
~~15000~~	~~15000~~	~~2~~
25000	40000	3

Figure 7-37 Desk-check table after *update* argument is processed second time

The Holmes Supply Program (cont'd.)

storePayroll	totalPayroll	numStores
~~0~~	~~0~~	~~1~~
~~15000~~	~~15000~~	~~2~~
~~25000~~	~~40000~~	~~3~~
60000	100000	4

Figure 7-38 Desk-check table after *update* argument is processed third time

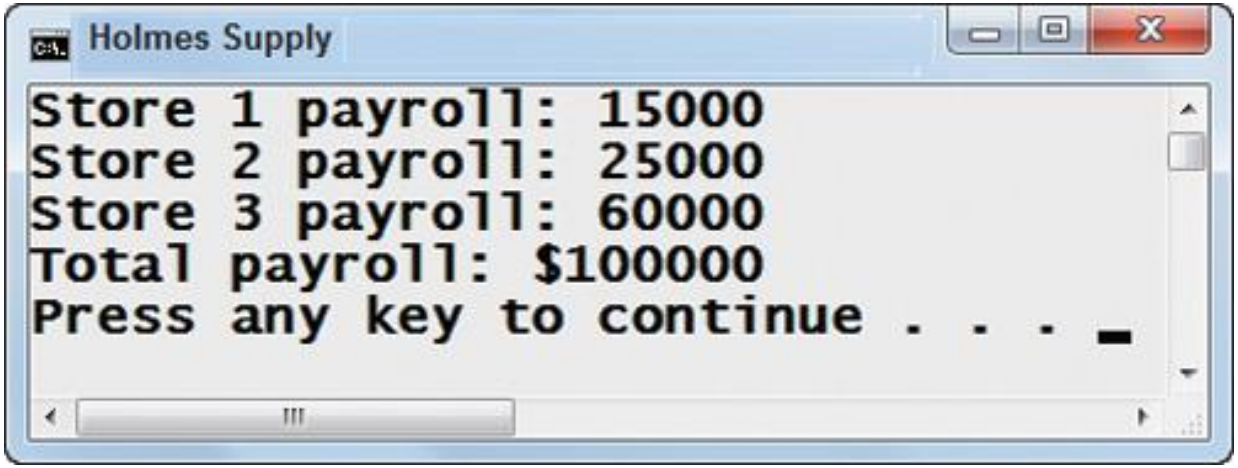

Figure 7-39 Sample run of Holmes Supply Company program

The Colfax Sales Program

- Extended example of a problem and program solution (following slides)
  - Calculates the commission for a given sales amount using four different rates
  - A `for` loop keeps track of each of the four rates

Problem specification
The sales manager at Colfax Sales wants a program that allows him to enter a sales amount. The program should calculate and display the appropriate commission using rates of 10%, 15%, 20%, and 25%. The program will use a counter to keep track of the four rates.

Figure 7-40 Problem specification for the Colfax Sales program

The Colfax Sales Program (cont'd.)

IPO chart information	C++ instructions
Input	
sales amount	`double sales = 0.0;`
Processing	
rate (counter: 10% to 25% in increments of 5%)	this variable is created and initialized in the for clause
Output	
commission	`double commission = 0.0;`
Algorithm	
1. enter the sales amount	`cout << "Enter the sales: ";` `cin >> sales;`

2. repeat for (rate from 10% to 25% in increment of 5%)	`for (double rate = .1;`
	`rate <= .25; rate = rate + .05)`
	`{`

calculate the commission by multiplying the sales amount by the rate	`commission = sales * rate;`

display the commission	`cout << rate * 100 <<`
	`"% commission: $" <<`
	`commission << endl;`

end repeat	`} //end for`

Figure 7-40 IPO chart information and C++ instructions for the Colfax Sales program

The Colfax Sales Program (cont'd.)

Processing steps

1. The computer creates the `sales` and `commission` variables and initializes them to 0.0.
2. The computer prompts the user to enter a sales amount and then stores the user's response (in this case, 25000) in the `sales` variable.
3. The computer processes the `for` clause's *initialization* argument (`double rate = .1`), which creates the `rate` variable and initializes it to .1.
4. The computer processes the `for` clause's *condition* argument (`rate <= .25`), which checks whether the `rate` variable's value is less than or equal to .25. It is, so the computer processes the statements in the loop body. Those statements calculate a 10% commission and display the result (2500) on the screen.
5. The computer processes the `for` clause's *update* argument (`rate = rate + .05`), which adds the number .05 to the value stored in the `rate` variable; the result is .15.
6. The computer processes the `for` clause's *condition* argument, which checks whether the `rate` variable's value is less than or equal to .25. It is, so the computer processes the statements in the loop body. Those statements calculate a 15% commission and display the result (3750) on the screen.
7. The computer processes the `for` clause's *update* argument, which adds the number .05 to the value stored in the `rate` variable; the result is .2.

Figure 7-41 Processing steps for the code in Figure 7-40

The Colfax Sales Program (cont'd.)

8. The computer processes the `for` clause's *condition* argument, which checks whether the `rate` variable's value is less than or equal to .25. It is, so the computer processes the statements in the loop body. Those statements calculate a 20% commission and display the result (5000) on the screen.
9. The computer processes the `for` clause's *update* argument, which adds the number .05 to the value stored in the `rate` variable; the result is .25.
10. The computer processes the `for` clause's *condition* argument, which checks whether the `rate` variable's value is less than or equal to .25. It is, so the computer processes the statements in the loop body. Those statements calculate a 25% commission and display the result (6250) on the screen.
11. The computer processes the `for` clause's *update* argument, which adds the number .05 to the value stored in the `rate` variable; the result is .3.
12. The computer processes the `for` clause's *condition* argument, which checks whether the `rate` variable's value is less than or equal to .25. It's not, so the computer stops processing the `for` loop and removes its local `rate` variable from internal memory. Processing continues with the statement following the end of the loop.

Figure 7-41 Processing steps for the code in Figure 7-40 (cont'd.)

The Colfax Sales Program (cont'd.)

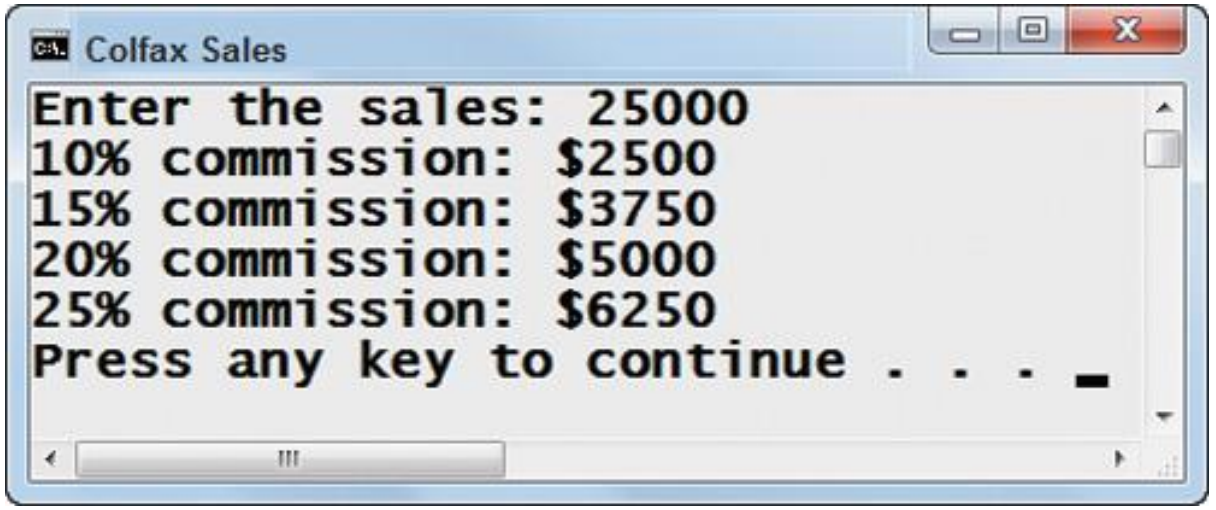

Figure 7-42 A sample run of the Colfax Sales program

The Colfax Sales Program (cont'd.)

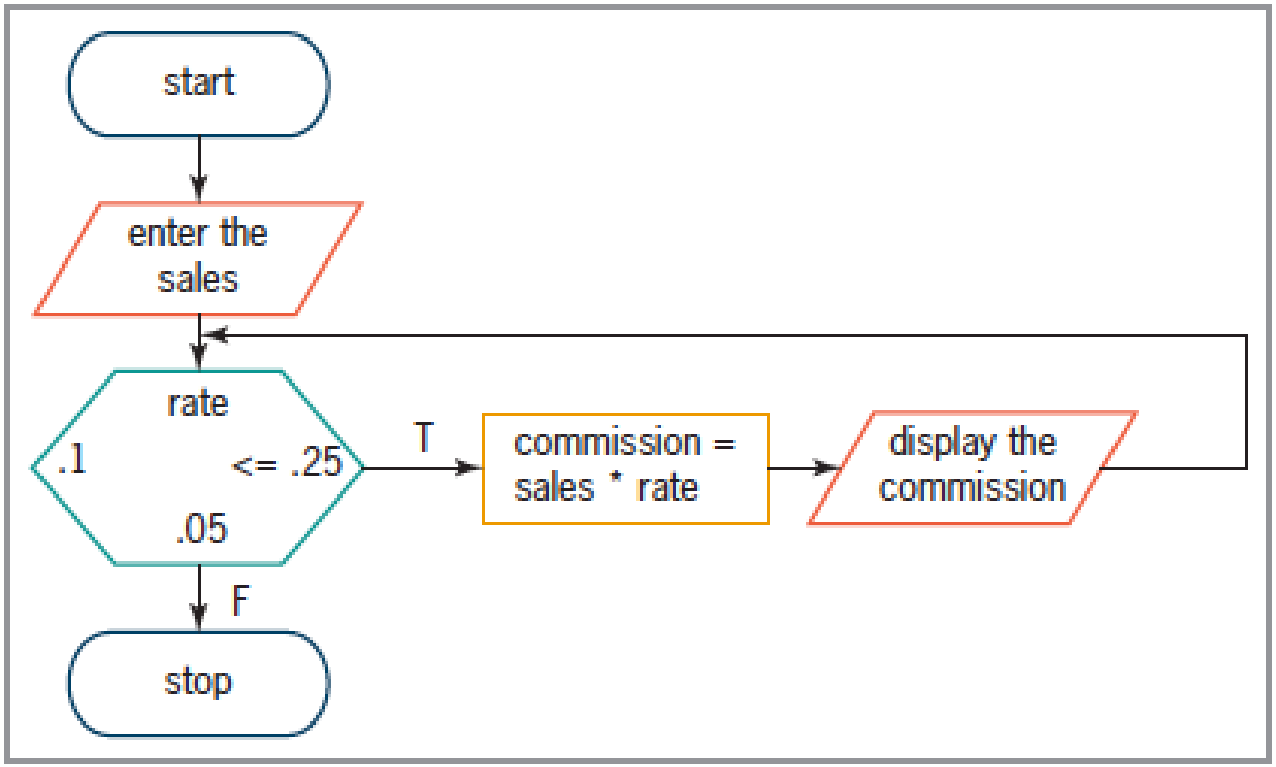

Figure 7-43 Colfax Sales algorithm shown in flowchart form

Another Version of the Wheels & More Program

- Alternative version of the Wheels & More program (following slides)
  - Uses a `for` loop instead of a `while` loop

Another Version of the Wheels & More Program (cont'd.)

Problem specification
The Wheels & More store has several part-time employees; each earns $10 per hour. The store manager wants a program that calculates and displays the weekly gross pay amount for as many employees as needed without having to run the program more than once. Because the number of hours an employee worked can be a positive number only, the store manager will indicate that he is finished with the program by entering a negative number (in this case, –1) as the number of hours.

IPO chart information	C++ instructions
Input	
pay rate ($10 per hour) hours worked	`const double RATE = 10.0;` `double hours = 0.0;`
Processing	
none	
Output	
gross pay	`double gross = 0.0;`
Algorithm	
1. enter the hours worked	`cout << "First hours worked (-1 to stop): ";` `cin >> hours;`
2. repeat while (the hours worked are not equal to -1)	`for (; hours != -1;)` `{`
calculate the gross pay by multiplying the hours worked by the pay rate	`gross = hours * RATE;`
display the gross pay	`cout << "Gross pay: $" << gross;` `cout << endl << endl;`
enter the hours worked	`cout << "Next hours worked (-1 to stop): ";` `cin >> hours;`
end repeat	`}` `//end for`

semicolon after the *initialization* argument

condition argument

semicolon after the *condition* argument

Figure 7-44 IPO chart information and modified C++ instructions for the Wheels & More program

Another Version of the Wheels & More Program (cont'd.)

Processing steps

1. The `const` statement creates the `RATE` named constant and initializes it to 10.0.
2. The declaration statements create the `hours` and `gross` variables and initialize them to 0.0.
3. The first `cout` statement prompts the user to enter the first number of hours worked, and the `cin` statement stores the user's response (in this case, 15) in the `hours` variable.
4. The `for` clause's *condition* argument (`hours != -1`) checks whether the `hours` variable's value is not equal to –1. The condition evaluates to true, so the statements in the loop body calculate and display the gross pay (150). They also prompt the user for the next hours worked entry and store the user's response (in this case, 8) in the `hours` variable.
5. The `for` clause's *condition* argument checks whether the `hours` variable's value is not equal to –1. The condition evaluates to true, so the statements in the loop body calculate and display the gross pay (80). They also prompt the user for the next hours worked entry and store the user's response (in this case, –1) in the `hours` variable.
6. The `for` clause's *condition* argument checks whether the `hours` variable's value is not equal to –1. In this case, the condition evaluates to false, so the `for` loop ends. Processing continues with the statement following the end of the loop.

Figure 7-45 Processing steps for the code shown in Figure 7-44

Summary

- Use the repetition structure (or loop) to repeatedly process one or more instructions
- Loop repeats as long as looping condition is true (or until loop exit condition has been met)
- A repetition structure can be pretest or posttest
- In a pretest loop, the loop condition is evaluated *before* instructions in loop body are processed
- In a posttest loop, the evaluation occurs *after* instructions in loop body are processed

Summary (cont'd.)

- Condition appears at the beginning of a pretest loop – must be a Boolean expression
- If condition evaluates to true, the instructions in the loop body are processed; otherwise, the loop body instructions are skipped
- Some loops require the user to enter a special sentinel value to end the loop
- Sentinel values should be easily distinguishable from valid data recognized by the program
- Other loops are terminated by using a counter

Summary (cont'd.)

- Input instruction that appears above a pretest loop's condition is the priming read
  - Sets up the loop by getting first value from user
- Input instruction that appears within the loop is the update read
  - Gets the remaining values (if any) from user
- In most flowcharts, diamond (decision symbol) is used to represent a repetition structure's condition

Summary (cont'd.)

- Counters and accumulators are used in repetition structures to calculate totals and averages
- All counters and accumulators must be initialized and updated
- Counters are updated by a constant value
- Accumulators are updated by a variable amount
- You can use either the `while` statement or the `for` statement to code a pretest loop in C++

Lab 7-1: Stop and Analyze

- Study the program in Figure 7-46 and answer the questions
- The program calculates the average outside temperature

Lab 7-2: Plan and Create

Problem specification

Professor Chang wants a program that allows him to enter a student's project and test scores, which will always be integers. The professor assigns three projects and two tests. Each project is worth 50 points, and each test is worth 100 points. The program should calculate and display the total points the student earned on the projects and tests. It also should display the student's grade, which is based on the total points earned. Shown below is the grading scale that Professor Chang uses when assigning grades.

Total points earned	Grade
315 – 350	A
280 – 314	B
245 – 279	C
210 – 244	D
below 210	F

Example 1
Project and test scores: 45, 40, 41, 96, 89
Total points earned and grade: 311, B

Example 2
Project and test scores: 40, 35, 37, 73, 68
Total points earned and grade: 253, C

Figure 7-47 Problem specification for Lab 7-2

Lab 7-3: Modify

- Modify the program in Lab 7-2 to display the total number of scores entered
- Test the program using scores 45, 40, 41, 96, 89, and sentinel value -1
- Test the program again using scores 25, 500 (a mistake, instead of 50), 38, -500 (to correct the mistake), 50, 64, 78, and -1
- Does the program display the correct total points earned and grade?
- How many scores does the program say were entered?

Lab 7-4: Desk-Check

- The code in Figure 7-53 should display the squares of the numbers from 1 through 5 (1, 4, 9, 16, and 25)
- Desk-check the code; did your desk-check reveal any errors?
- If so, correct the code and then desk-check it again

```
//declare variables
int squaredNumber = 0;

for (int number = 1; number < 5; number = number + 1)
{
     squaredNumber = number * number;
     cout << squaredNumber << endl;
}    //end for
```

Figure 7-53 Code for Lab 7-4

Lab 7-5: Debug

- Follow the instructions for starting C++ and opening the Lab7-5.cpp file
- Run the program and enter 15.45 when prompted
- The program goes into an infinite loop
- Type Ctrl+c to end the program
- Debug the program