4

Control Statements: Part 1

Let's all move one place on.

—Lewis Carroll

The wheel is come full circle.

—William Shakespeare

How many apples fell on Newton's head before he took the hint!

—Robert Frost

All the evolution we know of proceeds from the vague to the definite.

—Charles Sanders Peirce

OBJECTIVES

In this chapter you will learn:

- Basic problem-solving techniques.
- To develop algorithms through the process of top-down, stepwise refinement.
- To use the `if` and `if...else` selection statements to choose among alternative actions.
- To use the `while` repetition statement to execute statements in a program repeatedly.
- Counter-controlled repetition and sentinel-controlled repetition.
- To use the increment, decrement and assignment operators.

Outline

4.1 Introduction

- **Before writing a program**
  - **Have a thorough understanding of problem**
  - **Carefully plan your approach for solving it**
- **While writing a program**
  - **Know what “building blocks” are available**
  - **Use good programming principles**

4.2 Algorithms

- **Algorithms**
  - **The actions to execute**
  - **The order in which these actions execute**
- **Program control**
  - **Specifies the order in which actions execute in a program**
  - **Performed in C++ with control statements**

4.3 Pseudocode

- **Pseudocode**
  - **Artificial, informal language used to develop algorithms**
    - **Used to think out program before coding**
      - **Easy to convert into C++ program**
  - **Similar to everyday English**
    - **Only executable statements**
      - **No need to declare variables**
  - **Not executed on computers**

```
1  Prompt the user to enter the first integer
2  Input the first integer
3
4  Prompt the user to enter the second integer
5  Input the second integer
6
7  Add first integer and second integer, store result
8  Display result
```

Fig. 4.1 | Pseudocode for the addition program of Fig. 2.5.

4.4 Control Structures

- **Sequential execution**
  - **Statements executed in sequential order**
- **Transfer of control**
  - **Next statement executed is *not* the next one in sequence**
- **Structured programming**
  - **Eliminated `goto` statements**

4.4 Control Structures (Cont.)

- **Only three control structures needed**
  - **No `goto` statements**
  - **Demonstrated by Böhm and Jacopini**
  - **Three control structures**
    - **Sequence structure**
      - **Programs executed sequentially by default**
    - **Selection structures**
      - `if`, `if…else`, `switch`
    - **Repetition structures**
      - `while`, `do…while`, `for`

4.4 Control Structures (Cont.)

- **UML activity diagram**
  - **Models the workflow**
    - **Action state symbols**
      - **Rectangles with curved sides**
    - **Small circles**
      - **Solid circle is the initial state**
      - **Solid circle in a hollow circle is the final state**
    - **Transition arrows**
      - **Represent the flow of activity**
    - **Comment notes**
      - **Connected to diagram by dotted lines**

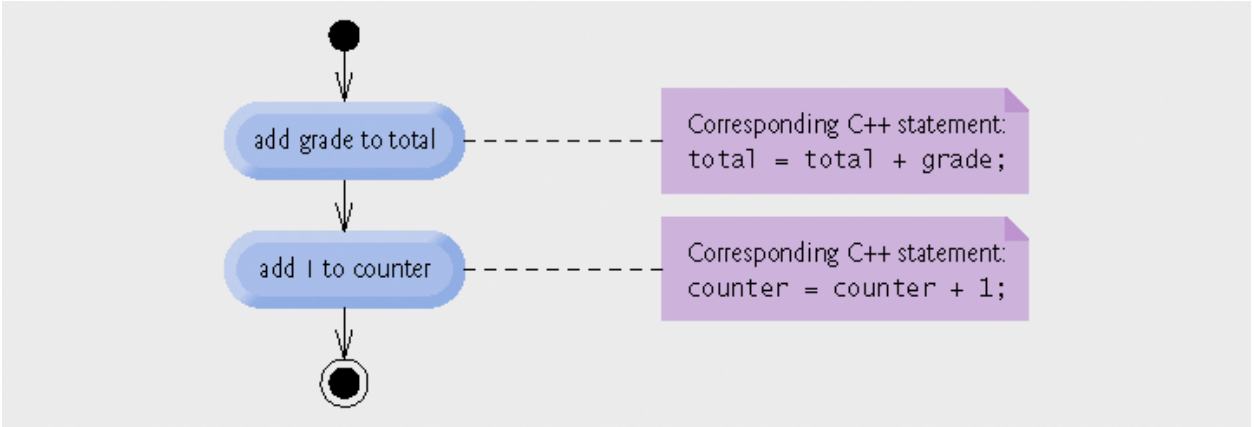


Fig. 4.2 | Sequence-structure activity diagram.

4.4 Control Structures (Cont.)

- **Single-entry/single-exit control statements**
  - **Three types of control statements**
    - **Sequence statement**
    - **Selection statements**
    - **Repetition statements**
  - **Combined in one of two ways**
    - **Control statement stacking**
      - **Connect exit point of one to entry point of the next**
    - **Control statement nesting**

C++ Keywords				
Keywords common to the C and C++ programming languages				
auto	break	case	char	Const
Continue	default	do	double	Else
enum	extern	float	for	Goto
if	int	long	register	Return
short	signed	sizeof	static	Struct
switch	typedef	union	unsigned	Void
volatile	while			
C++-only keywords				
and	and_eq	asm	bitand	Bitor
bool	catch	class	compl	const_cast
delete	dynamic_cast	explicit	export	False
friend	inline	mutable	namespace	New
not	not_eq	operator	or	or_eq
private	protected	public	reinterpret_cast	static_cast
template	this	throw	true	Try
typeid	typename	using	virtual	wchar_t
xor	xor_eq			

Fig. 4.3 | C++ keywords.

Common Programming Error 4.1

Using a keyword as an identifier is a syntax error.

Common Programming Error 4.2

Spelling a keyword with any uppercase letters is a syntax error. All of C++'s keywords contain only lowercase letters.

Software Engineering Observation 4.1

Any C++ program we will ever build can be constructed from only seven different types of control statements (sequence, `if`, `if...else`, `switch`, `while`, `do...while` and `for`) combined in only two ways (control-statement stacking and control-statement nesting).

4.5 `if` Selection Statement

- **Selection statements**
  - **Choose among alternative courses of action**
  - **Pseudocode example**
    - ***If student's grade is greater than or equal to 60***

      Print "Passed"
      - **If the condition is `true`**
        - **Print statement executes, program continues to next statement**
      - **If the condition is `false`**
        - **Print statement ignored, program continues**
      - **Indenting makes programs easier to read**
        - **C++ ignores white-space characters**

4.5 if Selection Statement (Cont.)

- **Selection statements (Cont.)**
  - **Translation into C++**
    - `if ( grade >= 60 )`
      `cout << "Passed";`
  - **Any expression can be used as the condition**
    - **If it evaluates to false, it is treated as false**
- **Diamond symbol in UML modeling**
  - **Indicates decision is to be made**
  - **Contains guard conditions**
    - **Test condition**
    - **Follow correct path**

Good Programming Practice 4.1

Consistently applying reasonable indentation conventions throughout your programs greatly improves program readability. We suggest three blanks per indent. Some people prefer using tabs but these can vary across editors, causing a program written on one editor to align differently when used with another.

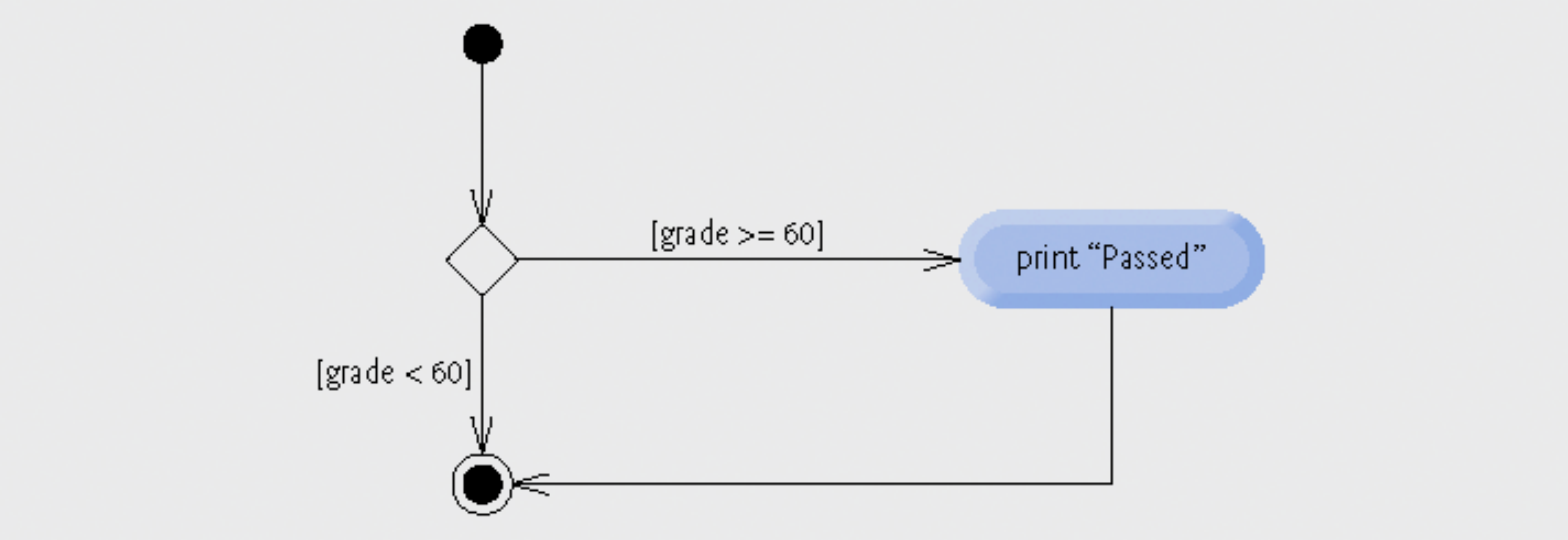


Fig. 4.4 | `if` single-selection statement activity diagram.

Portability Tip 4.1

For compatibility with earlier versions of C, which used integers for Boolean values, the `bool` value `true` also can be represented by any nonzero value (compilers typically use 1) and the `bool` value `false` also can be represented as the value zero.

4.6 if…else Double-Selection Statement

- if
  - Performs action if condition true
- if…else
  - Performs one action if condition is true, a different action if it is false
- Pseudocode
  - *If student's grade is greater than or equal to 60*
    print "Passed"
    Else
    print "Failed"
- C++ code
  - ```
    if ( grade >= 60 )
       cout << "Passed";
    else
       cout << "Failed";
    ```


    ```

Good Programming Practice 4.2

Indent both body statements of an `if...else` statement.

Good Programming Practice 4.3

If there are several levels of indentation, each level should be indented the same additional amount of space.

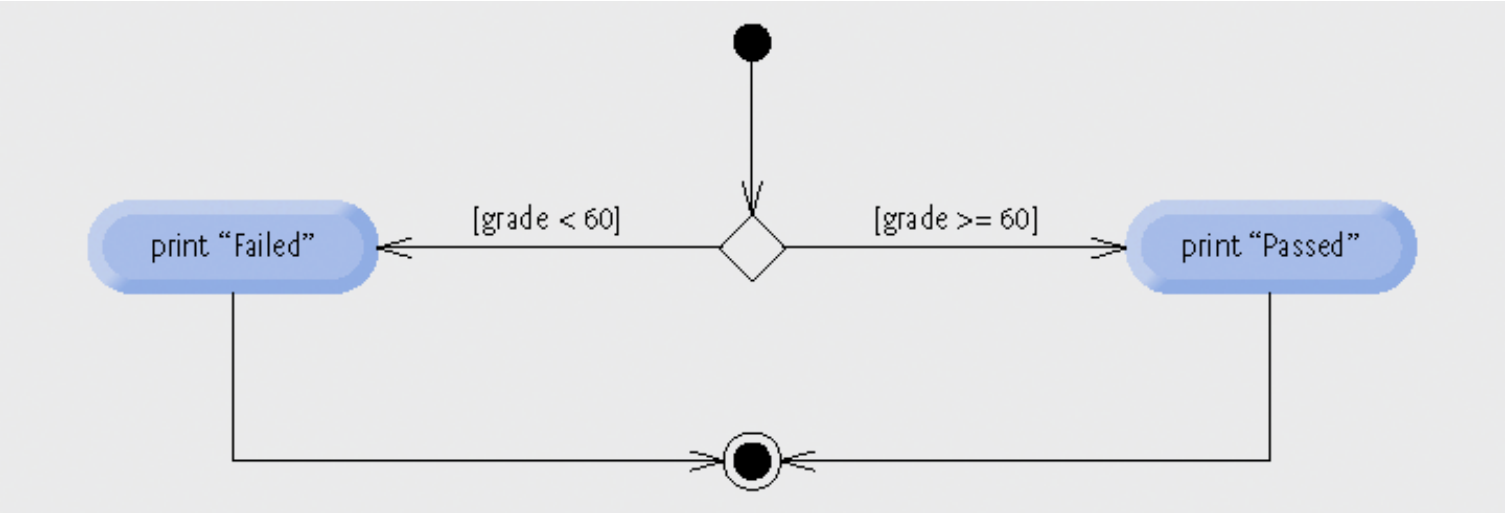
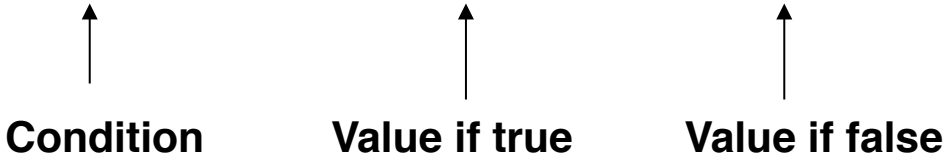


Fig. 4.5 | `if...else` double-selection statement activity diagram.

4.6 `if…else` Double-Selection Statement (Cont.)

- **Ternary conditional operator (`?:`)**
  - **Three arguments (condition, value if `true`, value if `false`)**
- **Code could be written:**
  - `cout << ( grade >= 60 ? “Passed” : “Failed” );`



Error-Prevention Tip 4.1

To avoid precedence problems (and for clarity), place conditional expressions (that appear in larger expressions) in parentheses.

4.6 if…else Double-Selection Statement (Cont.)

- **Nested `if…else` statements**
  - **One inside another, test for multiple cases**
  - **Once a condition met, other statements are skipped**
  - **Example**
    - *If student's grade is greater than or equal to 90*
      Print "A"
      Else
      If student's grade is greater than or equal to 80
      Print "B"
      Else
      If student's grade is greater than or equal to 70
      Print "C"
      Else
      If student's grade is greater than or equal to 60
      Print "D"
      Else
      Print "F"

4.6 if…else Double-Selection Statement (Cont.)

- Nested if…else statements (Cont.)
  - Written In C++

```
if ( studentGrade >= 90 )
   cout << "A";
else
   if (studentGrade >= 80 )
      cout << "B";
   else
      if (studentGrade >= 70 )
         cout << "C";
      else
         if ( studentGrade >= 60 )
            cout << "D";
         else
            cout << "F";
```

4.6 if…else Double-Selection Statement (Cont.)

- Nested if…else statements (Cont.)
  - Written In C++ (indented differently)

```
if ( studentGrade >= 90 )
   cout << "A";
else if (studentGrade >= 80 )
   cout << "B";
else if (studentGrade >= 70 )
   cout << "C";
else if ( studentGrade >= 60 )
   cout << "D";
else
   cout << "F";
```

Performance Tip 4.1

A nested `if...else` statement can perform much faster than a series of single-selection `if` statements because of the possibility of early exit after one of the conditions is satisfied.

Performance Tip 4.2

In a nested `if...else` statement, test the conditions that are more likely to be `true` at the beginning of the nested `if...else` statement. This will enable the nested `if...else` statement to run faster and exit earlier than testing infrequently occurring cases first.

4.6 if…else Double-Selection Statement (Cont.)

- **Dangling-else problem**
  - **Compiler associates else with the immediately preceding if**
  - **Example**
    - ```
      if ( x > 5 )
         if ( y > 5 )
            cout << "x and y are > 5";
      else
         cout << "x is <= 5";
      ```
  - **Compiler interprets as**
    - ```
      if ( x > 5 )
         if ( y > 5 )
            cout << "x and y are > 5";
         else
            cout << "x is <= 5";
      ```

4.6 `if…else` Double-Selection Statement (Cont.)

- **Dangling-`else` problem (Cont.)**
  - **Rewrite with braces (`{}`)**
    -
```
if ( x > 5 )
{
   if ( y > 5 )
      cout << "x and y are > 5";
}
else
   cout << "x is <= 5";
```
      - **Braces indicate that the second `if` statement is in the body of the first and the `else` is associated with the first `if` statement**

4.6 if…else Double-Selection Statement (Cont.)

- **Compound statement**
  - **Also called a block**
    - **Set of statements within a pair of braces**
    - **Used to include multiple statements in an `if` body**
  - **Example**
    - ```
      if ( studentGrade >= 60 )
         cout << "Passed.\n";
      else
      {
         cout << "Failed.\n";
         cout << "You must take this course again.\n";
      }
      ```
  - **Without braces,**

    ```
    cout << "You must take this course again.\n";
    ```

    always executes


    ```

Software Engineering Observation 4.2

A block can be placed anywhere in a program that a single statement can be placed.

Common Programming Error 4.3

Forgetting one or both of the braces that delimit a block can lead to syntax errors or logic errors in a program.

Good Programming Practice 4.4

Always putting the braces in an `if...else` statement (or any control statement) helps prevent their accidental omission, especially when adding statements to an `if` or `else` clause at a later time. To avoid omitting one or both of the braces, some programmers prefer to type the beginning and ending braces of blocks even before typing the individual statements within the braces.

4.6 `if…else` Double-Selection Statement (Cont.)

- **Empty statement**
  - **A semicolon (`;`) where a statement would normally be**
  - **Performs no action**
  - **Also called a null statement**

Common Programming Error 4.4

Placing a semicolon after the condition in an `if` statement leads to a logic error in single-selection `if` statements and a syntax error in double-selection `if...else` statements (when the `if` part contains an actual body statement).

4.7 `while` Repetition Statement

- **Repetition statement**
  - **Action repeated while some condition remains true**
  - **Pseudocode**
    - *While there are more items on my shopping list*

      Purchase next item and cross it off my list
  - **`while` loop repeats until condition becomes `false`**
  - **Example**
    - ```
      int product = 3;

      while ( product <= 100 )
         product = 3 * product;
      ```


      ```

Common Programming Error 4.5

Not providing, in the body of a `while` statement, an action that eventually causes the condition in the `while` to become false normally results in a logic error called an infinite loop, in which the repetition statement never terminates. This can make a program appear to "hang" or "freeze" if the loop body does not contain statements that interact with the user.

4.7 while Repetition Statement (Cont.)

- **UML merge symbol**
  - **Joins two or more flows of activity into one flow of activity**
  - **Represented as a diamond**
    - **Unlike the decision symbol a merge symbol has**
      - **Multiple incoming transition arrows**
      - **Only one outgoing transition arrows**
        - **No guard conditions on outgoing transition arrows**
  - **Has no counterpart in C++ code**

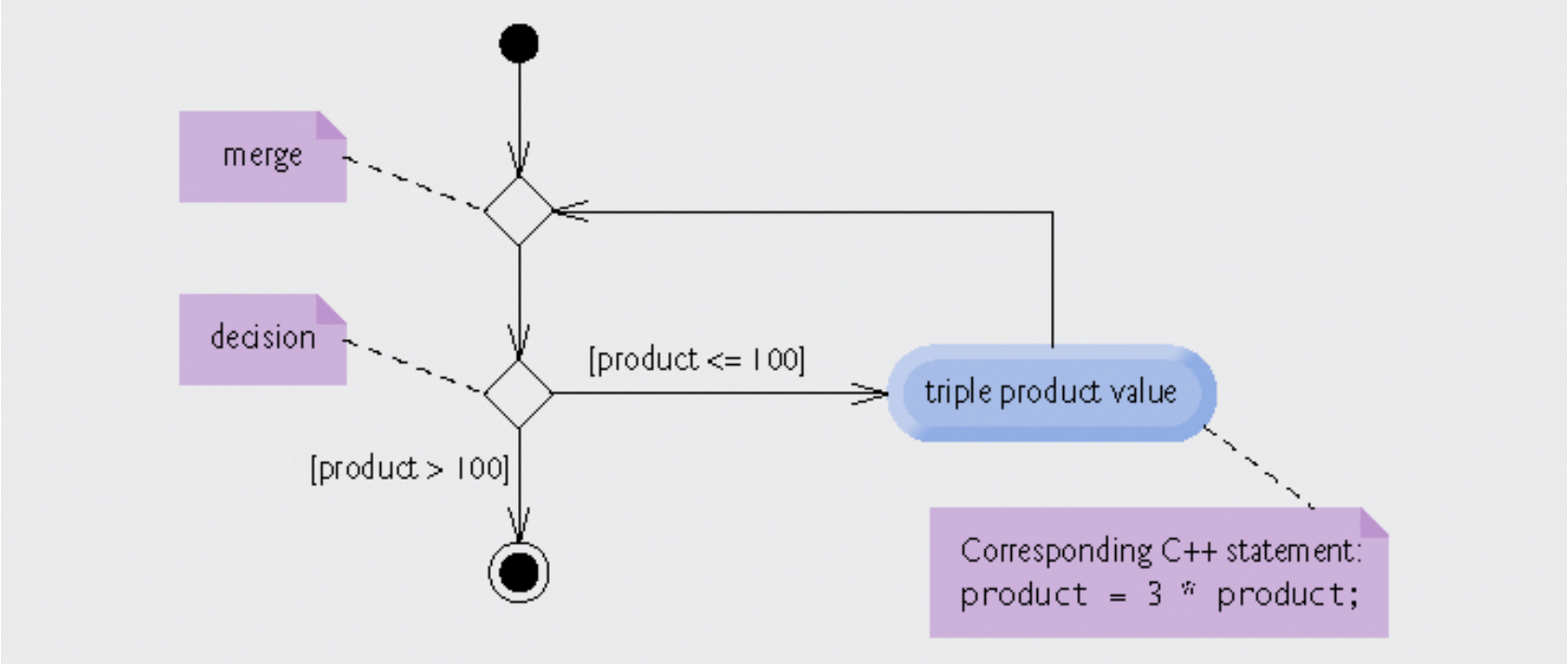


Fig. 4.6 | `while` repetition statement UML activity diagram.

Performance Tip 4.3

Many of the performance tips we mention in this text result in only small improvements, so the reader might be tempted to ignore them. However, a small performance improvement for code that executes many times in a loop can result in substantial overall performance improvement.

4.8 Formulating Algorithms: Counter-Controlled Repetition

- **Problem statement**

  A class of ten students took a quiz. The grades (integers in the range 0 to 100) for this quiz are available to you. Calculate and display the total of all student grades and the class average on the quiz.

- **Counter-controlled repetition**
  - **Loop repeated until counter reaches certain value**
  - **Also known as definite repetition**
    - **Number of repetitions known beforehand**

4.8 Formulating Algorithms: Counter-Controlled Repetition (Cont.)

- **Counter-controlled repetition (Cont.)**
  - **Counter variable**
    - **Used to count**
      - **In example, indicates which of the 10 grades is being entered**
  - **Total variable**
    - **Used to accumulate the sum of several values**
    - **Normally initialized to zero beforehand**
      - **Otherwise it would include the previous value stored in that memory location**

Software Engineering Observation 4.3

Experience has shown that the most difficult part of solving a problem on a computer is developing the algorithm for the solution. Once a correct algorithm has been specified, the process of producing a working C++ program from the algorithm is normally straightforward.

```
1  Set total to zero
2  Set grade counter to one
3
4  While grade counter is less than or equal to ten
5     Prompt the user to enter the next grade
6     Input the next grade
7     Add the grade into the total
8     Add one to the grade counter
9
10 Set the class average to the total divided by ten
11 Print the total of the grades for all students in the class
12 Print the class average
```

Fig. 4.7 | Pseudocode algorithm that uses counter-controlled repetition to solve the class average problem.

Outline

fig04_08.cpp

(1 of 1)

```
1  // Fig. 4.8: GradeBook.h
2  // Definition of class GradeBook that determines a class average.
3  // Member functions are defined in GradeBook.cpp
4  #include <string> // program uses C++ standard string class
5  using std::string;
6
7  // GradeBook class definition
8  class GradeBook
9  {
10 public:
11    GradeBook( string ); // constructor initializes
12    void setCourseName( string ); // function to se
13    string getCourseName(); // function to retrieve
14    void displayMessage(); // display a welcome message
15    void determineClassAverage(); // averages grades entered by the user
16 private:
17    string courseName; // course name for this GradeBook
18 }; // end class GradeBook
```

Function **determineClassAverage** implements the class average algorithm described by the pseudocode

Outline

fig04_09.cpp

(1 of 3)

```
1  // Fig. 4.9: GradeBook.cpp
2  // Member-function definitions for class GradeBook that solves the
3  // class average program with counter-controlled repetition.
4  #include <iostream>
5  using std::cout;
6  using std::cin;
7  using std::endl;
8
9  #include "GradeBook.h" // include definition of class GradeBook
10
11 // constructor initializes courseName with string supplied as argument
12 GradeBook::GradeBook( string name )
13 {
14    setCourseName( name ); // validate and store courseName
15 } // end GradeBook constructor
16
17 // function to set the course name;
18 // ensures that the course name has at most 25 characters
19 void GradeBook::setCourseName( string name )
20 {
21    if ( name.length() <= 25 ) // if name has 25 or fewer characters
22       courseName = name; // store the course name in the object
23    else // if name is longer than 25 characters
24    { // set courseName to first 25 characters of parameter name
25       courseName = name.substr( 0, 25 ); // select first 25 characters
26       cout << "Name \"" << name << "\" exceeds maximum length (25).\n"
27          << "Limiting courseName to first 25 characters.\n" << endl;
28    } // end if...else
29 } // end function setCourseName
30
```

If course name was longer than 25, select first 25 characters

Outline

fig04_09.cpp

(2 of 3)

```
31 // function to retrieve the course name
32 string GradeBook::getCourseName()
33 {
34    return courseName;
35 } // end function getCourseName
36
37 // display a welcome message to the GradeBook user
38 void GradeBook::displayMessage()
39 {
40    cout << "Welcome to the grade book for\n" << getCourseName() << "!\n"
41       << endl;
42 } // end function displayMessage
43
44 // determine class average based on 10 grades entered by user
45 void GradeBook::determineClassAverage()
46 {
47    int total; // sum of grades entered by user
48    int gradeCounter; // number of the grade to be entered next
49    int grade; // grade value entered by user
50    int average; // average of grades
51
```

Function **determineClassAverage** implements the class average algorithm described by the pseudocode

Declare total variable **total**

Declare counter variable **gradeCounter**

Outline

```
52    // initialization phase
53    total = 0; // initialize total
54    gradeCounter = 1; // initialize loop counter
55
56    // processing phase
57    while ( gradeCounter <= 10 ) // loop 10 times
58    {
59       cout << "Enter grade: "; // prompt for input
60       cin >> grade; // input next grade
61       total = total + grade; // add grade to total
62       gradeCounter = gradeCounter + 1; // increment counter by 1
63    } // end while
64
65    // termination phase
66    average = total / 10; // integer division yie
67
68    // display total and average of grades
69    cout << "\nTotal of all 10 grades is " << tota
70    cout << "Class average is " << average << endl
71 } // end function determineClassAverage
```

Initilize total variable **total** to **0**

Initialize counter variable **gradeCounter** to **1**

Continue looping as long as **gradeCounter**'s value is less than or equal to **10**

Add the current grade to **total**

Increment counter by **1**, which causes **gradeCounter** to exceed **10** eventually

Perform the averaging calculation and assign its result to the variable **average**

Outline

fig04_10.cpp

(1 of 1)

```
1  // Fig. 4.10: fig04_10.cpp
2  // Create GradeBook object and invoke its determineClassAverage function.
3  #include "GradeBook.h" // include definition of class GradeBook
4
5  int main()
6  {
7     // create GradeBook object myGradeBook and
8     // pass course name to constructor
9     GradeBook myGradeBook( "CS101 C++ Programming" );
10
11    myGradeBook.displayMessage(); // display welcome message
12    myGradeBook.determineClassAverage(); // find average of 10 grades
13    return 0; // indicate successful termination
14 } // end main
```

```
Welcome to the grade book for
CS101 C++ Programming

Enter grade: 67
Enter grade: 78
Enter grade: 89
Enter grade: 67
Enter grade: 87
Enter grade: 98
Enter grade: 93
Enter grade: 85
Enter grade: 82
Enter grade: 100

Total of all 10 grades is 846
Class average is 84
```

Good Programming Practice 4.5

Separate declarations from other statements in functions with a blank line for readability.

4.8 Formulating Algorithms: Counter-Controlled Repetition (Cont.)

- **Uninitialized variables**
  - **Contain "garbage" (or undefined) values**
- **Notes on integer division and truncation**
  - **Integer division**
    - **When dividing two integers**
    - **Performs truncation**
      - **Fractional part of the resulting quotient is lost**

Common Programming Error 4.6

Not initializing counters and totals can lead to logic errors.

Error-Prevention Tip 4.2

Initialize each counter and total, either in its declaration or in an assignment statement. Totals are normally initialized to 0. Counters are normally initialized to 0 or 1, depending on how they are used (we will show examples of when to use 0 and when to use 1).

Good Programming Practice 4.6

Declare each variable on a separate line with its own comment to make programs more readable.

Common Programming Error 4.7

Assuming that integer division rounds (rather than truncates) can lead to incorrect results. For example, 7 ÷ 4, which yields 1.75 in conventional arithmetic, truncates to 1 in integer arithmetic, rather than rounding to 2.

Common Programming Error 4.8

Using a loop's counter-control variable in a calculation after the loop often causes a common logic error called an off-by-one-error. In a counter-controlled loop that counts up by one each time through the loop, the loop terminates when the counter's value is one higher than its last legitimate value (i.e., 11 in the case of counting from 1 to 10).

4.9 Formulating Algorithms: Sentinel-Controlled Repetition

- **Problem statement**

  Develop a class average program that processes grades for an arbitrary number of students each time it is run.

- **Sentinel-controlled repetition**
  - **Also known as indefinite repetition**
  - **Use a sentinel value**
    - **Indicates "end of data entry"**
    - **A sentinel value cannot also be a valid input value**
    - **Also known as a signal, dummy or flag value**

Common Programming Error 4.9

Choosing a sentinel value that is also a legitimate data value is a logic error.

4.9 Formulating Algorithms: Sentinel-Controlled Repetition (Cont.)

- **Top-down, stepwise refinement**
  - **Development technique for well-structured programs**
  - **Top step**
    - **Single statement conveying overall function of the program**
    - **Example**
      - ***Determine the class average for the quiz***
  - **First refinement**
    - **Multiple statements using only the sequence structure**
    - **Example**
      - ***Initialize variables***
      - ***Input, sum and count the quiz grades***
      - ***Calculate and print the total of all student grades and the class average***

Software Engineering Observation 4.4

Each refinement, as well as the top itself, is a complete specification of the algorithm; only the level of detail varies.

Software Engineering Observation 4.5

Many programs can be divided logically into three phases: an initialization phase that initializes the program variables; a processing phase that inputs data values and adjusts program variables (such as counters and totals) accordingly; and a termination phase that calculates and outputs the final results.

Common Programming Error 4.10

An attempt to divide by zero normally causes a fatal runtime error.

Error-Prevention Tip 4.3

When performing division by an expression whose value could be zero, explicitly test for this possibility and handle it appropriately in your program (such as by printing an error message) rather than allowing the fatal error to occur.

4.9 Formulating Algorithms: Sentinel-Controlled Repetition (Cont.)

- **Top-down, stepwise refinement (Cont.)**
  - **Second refinement**
    - **Commit to specific variables**
    - **Use specific control structures**
    - **Example in Fig. 4.11**
- **Fatal logic error**
  - **Could cause the program the fail**
    - **Often called “bombing” or “crashing”**
  - **Division by zero is normally a fatal logic error**

```
1  Initialize total to zero
2  Initialize counter to zero
3
4  Prompt the user to enter the first grade
5  Input the first grade (possibly the sentinel)
6
7  While the user has not yet entered the sentinel
8      Add this grade into the running total
9      Add one to the grade counter
10     Prompt the user to enter the next grade
11     Input the next grade (possibly the sentinel)
12
13 If the counter is not equal to zero
14     Set the average to the total divided by the counter
15     Print the total of the grades for all students in the class
16     Print the class average
17 else
18     Print "No grades were entered"
```

Fig. 4.11 | Class average problem pseudocode algorithm with sentinel-controlled repetition.

Software Engineering Observation 4.6

Terminate the top-down, stepwise refinement process when the pseudocode algorithm is specified in sufficient detail for you to be able to convert the pseudocode to C++. Normally, implementing the C++ program is then straightforward.

Software Engineering Observation 4.7

Many experienced programmers write programs without ever using program development tools like pseudocode. These programmers feel that their ultimate goal is to solve the problem on a computer and that writing pseudocode merely delays the production of final outputs. Although this method might work for simple and familiar problems, it can lead to serious difficulties in large, complex projects.

4.9 Formulating Algorithms: Sentinel-Controlled Repetition (Cont.)

- **Floating-point numbers**
  - **A real number with a decimal point**
  - **C++ provides data types `float` and `double`**
    - **`double` numbers can have larger magnitude and finer detail**
      - **Called precision**
    - **Floating-point constant values are treated as `double` values by default**
  - **Floating-point values are often only approximations**

Outline

fig04_12.cpp

(1 of 1)

```
1  // Fig. 4.12: GradeBook.h
2  // Definition of class GradeBook that determines a class average.
3  // Member functions are defined in GradeBook.cpp
4  #include <string> // program uses C++ standard string class
5  using std::string;
6
7  // GradeBook class definition
8  class GradeBook
9  {
10 public:
11    GradeBook( string ); // constructor initializes course name
12    void setCourseName( string ); // function to set the course name
13    string getCourseName(); // function to retrieve the course name
14    void displayMessage(); // display a welcome message
15    void determineClassAverage(); // averages grades entered by the user
16 private:
17    string courseName; // course name for this GradeBook
18 }; // end class GradeBook
```

Outline

```
1  // Fig. 4.13: GradeBook.cpp
2  // Member-function definitions for class GradeBook that solves the
3  // class average program with sentin
4  #include <iostream>
5  using std::cout;
6  using std::cin;
7  using std::endl;
8  using std::fixed; // ensures that decimal point is displayed
9
10 #include <iomanip> // parameterized stream manipulators
11 using std::setprecision; // sets numeric output precision
12
13 // include definition of class GradeBook from GradeBook.h
14 #include "GradeBook.h"
15
16 // constructor initializes courseName with string supplied as argument
17 GradeBook::GradeBook( string name )
18 {
19    setCourseName( name ); // validate and store courseName
20 } // end GradeBook constructor
21
```

fig04_13.cpp

fixed forces output to print in fixed point format (not scientific notation) and forces trailing zeros and decimal point to print

setprecision stream manipulator (in header **<iomanip>**) sets numeric output precision

Outline

fig04_13.cpp

(2 of 4)

```
22 // function to set the course name;
23 // ensures that the course name has at most 25 characters
24 void GradeBook::setCourseName( string name )
25 {
26    if ( name.length() <= 25 ) // if name has 25 or fewer characters
27       courseName = name; // store the course name in the object
28    else // if name is longer than 25 characters
29    { // set courseName to first 25 characters of parameter name
30       courseName = name.substr( 0, 25 ); // select first 25 characters
31       cout << "Name \"" << name << "\" exceeds maximum length (25).\n"
32          << "Limiting courseName to first 25 characters.\n" << endl;
33    } // end if...else
34 } // end function setCourseName
35
36 // function to retrieve the course name
37 string GradeBook::getCourseName()
38 {
39    return courseName;
40 } // end function getCourseName
41
42 // display a welcome message to the GradeBook user
43 void GradeBook::displayMessage()
44 {
45    cout << "Welcome to the grade book for\n" << getCourseName() << "!\n"
46       << endl;
47 } // end function displayMessage
48
```

Outline

Fig04_13.cpp

(3 of 4)

```
49 // determine class average based on 10 grades entered by user
50 void GradeBook::determineClassAverage()
51 {
52    int total; // sum of grades entered by user
53    int gradeCounter; // number of grades entered
54    int grade; // grade value
55    double average; // number with decimal point for average
56
57    // initialization phase
58    total = 0; // initialize total
59    gradeCounter = 0; // initialize loop counte
60
61    // processing phase
62    // prompt for input and read grade from use
63    cout << "Enter grade or -1 to quit: ";
64    cin >> grade; // input grade or sentinel value
65
66    // loop until sentinel value read from user
67    while ( grade != -1 ) // while grade is not -1
68    {
69       total = total + grade; // add grade
70       gradeCounter = gradeCounter + 1; //
71
72       // prompt for input and read next grade from user
73       cout << "Enter grade or -1 to quit: ";
74       cin >> grade; // input grade or sentinel value
75    } // end while
76
```

Function **determineClassAverage** implements the class average algorithm described by the pseudocode

Declare local **int** variables **total**, **gradeCounter** and **grade** and **double** variable **average**

while loop iterates as long as **grade** does not equal the sentinel value **-1**

```
77    // termination phase
78    if ( gradeCounter != 0 ) // if user entered at least one grade...
79    {
80       // calculate average of all grades entered
81       average = static_cast< double >( total ) / gradeCounter;
82
83       // display total and average (with two digi
84       cout << "\nTotal of all " << gradeCounter <
85          << total << endl;
86       cout << "Class average is " << setprecision( 2 ) << fixed << average
87          << endl;
88    } // end if
89    else // no grades were entered, so output appropriate message
90       cout << "No grades were entered" << endl;
91 } // end function determineClassAverage
```

Calculate average grade using **`static_cast< double >`** to perform explicit conversion

Outline

fig04_13.cpp

(4 of 4)

Outline

fig04_14.cpp

(1 of 1)

```
1  // Fig. 4.14: fig04_14.cpp
2  // Create GradeBook object and invoke its determineClassAverage function.
3
4  // include definition of class GradeBook from GradeBook.h
5  #include "GradeBook.h"
6
7  int main()
8  {
9     // create GradeBook object myGradeBook and
10    // pass course name to constructor
11    GradeBook myGradeBook( "CS101 C++ Programming" );
12
13    myGradeBook.displayMessage(); // display welcome message
14    myGradeBook.determineClassAverage(); // find average of 10 grades
15    return 0; // indicate successful termination
16 } // end main
```

```
Welcome to the grade book for
CS101 C++ Programming

Enter grade or -1 to quit: 97
Enter grade or -1 to quit: 88
Enter grade or -1 to quit: 72
Enter grade or -1 to quit: -1

Total of all 3 grades entered is 257
Class average is 85.67
```

Good Programming Practice 4.7

Prompt the user for each keyboard input. The prompt should indicate the form of the input and any special input values. For example, in a sentinel-controlled loop, the prompts requesting data entry should explicitly remind the user what the sentinel value is.

Common Programming Error 4.11

Omitting the braces that delimit a block can lead to logic errors, such as infinite loops. To prevent this problem, some programmers enclose the body of every control statement in braces, even if the body contains only a single statement.

Common Programming Error 4.12

Using floating-point numbers in a manner that assumes they are represented exactly (e.g., using them in comparisons for equality) can lead to incorrect results. Floating-point numbers are represented only approximately by most computers.

4.9 Formulating Algorithms: Sentinel-Controlled Repetition (Cont.)

- **Unary cast operator**
  - **Creates a temporary copy of its operand with a different data type**
    - **Example**
      - `static_cast< double > ( total )`
        - **Creates temporary floating-point copy of `total`**
  - **Explicit conversion**
- **Promotion**
  - **Converting a value (e.g. `int`) to another data type (e.g. `double`) to perform a calculation**
  - **Implicit conversion**

Common Programming Error 4.13

The cast operator can be used to convert between fundamental numeric types, such as `int` and `double`, and between related class types (as we discuss in Chapter 13, Object-Oriented Programming: Polymorphism). Casting to the wrong type may cause compilation errors or runtime errors.

4.9 Formulating Algorithms: Sentinel-Controlled Repetition (Cont.)

- **Formatting floating-point numbers**
  - **Parameterized stream manipulator `setprecision`**
    - **Specifies number of digits of precision to display to the right of the decimal point**
    - **Default precision is six digits**
  - **Nonparameterized stream manipulator `fixed`**
    - **Indicates that floating-point values should be output in fixed-point format**
      - **As opposed to scientific notation (3.1×10^3)**
  - **Stream manipulator `showpoint`**
    - **Forces decimal point to display**

4.10 Formulating Algorithms: Nested Control Statement

- **Problem statement**

A college offers a course that prepares students for the state licensing exam for real estate brokers. Last year, ten of the students who completed this course took the exam. The college wants to know how well its students did on the exam. You have been asked to write a program to summarize the results. You have been given a list of these 10 students. Next to each name is written a 1 if the student passed the exam or a 2 if the student failed.

Your program should analyze the results of the exam as follows:

1. *Input each test result (i.e., a 1 or a 2). Display the prompting message "Enter result" each time the program requests another test result.*
2. *Count the number of test results of each type.*
3. *Display a summary of the test results indicating the number of students who passed and the number who failed.*
4. *If more than eight students passed the exam, print the message "Raise tuition."*

4.10 Formulating Algorithms: Nested Control Statement (Cont.)

- **Notice that**
  - **Program processes 10 results**
    - **Fixed number, use counter-controlled loop**
  - **Each test result is 1 or 2**
    - **If not 1, assume 2**
  - **Two counters can be used**
    - **One counts number that passed**
    - **Another counts number that failed**
  - **Must decide whether more than eight students passed**

4.10 Formulating Algorithms: Nested Control Statement (Cont.)

- **Top level outline**
  - *Analyze exam results and decide whether tuition should be raised*
- **First refinement**
  - *Initialize variables*
    Input the ten exam results and count passes and failures
    Print a summary of the exam results and decide whether tuition should be raised
- **Second Refinement**
  - *Initialize variables*
    to
    Initialize passes to zero
    Initialize failures to zero
    Initialize student counter to one

4.10 Formulating Algorithms: Nested Control Statement (Cont.)

- **Second Refinement (Cont.)**
  - *Input the ten exam results and count passes and failures*

    to

    While student counter is less than or equal to ten
    Prompt the user to enter the next exam result
    If the student passed
    Add one to passes
    Else
    Add one to failures
    Add one to student counter

4.10 Formulating Algorithms: Nested Control Statement (Cont.)

- **Second Refinement (Cont.)**
  - *Print a summary of the exam results and decide whether tuition should be raised*

    to

    Print the number of passes
    Print the number of failures
    If more than eight students passed
      Print "Raise tuition"

```
1  Initialize passes to zero
2  Initialize failures to zero
3  Initialize student counter to one
4
5  While student counter is less than or equal to 10
6     Prompt the user to enter the next exam result
7     Input the next exam result
8
9     If the student passed
10       Add one to passes
11    Else
12       Add one to failures
13
14    Add one to student counter
15
16 Print the number of passes
17 Print the number of failures
18
19 If more than eight students passed
20    Print "Raise tuition"
```

Fig. 4.15 | Pseudocode for examination-results problem.

```
1  // Fig. 4.16: Analysis.h
2  // Definition of class Analysis that analyzes examination results.
3  // Member function is defined in Analysis.cpp
4
5  // Analysis class definition
6  class Analysis
7  {
8  public:
9     void processExamResults(); // process 10 students' examination results
10 }; // end class Analysis
```

Outline

fig04_16.cpp

(1 of 1)

Outline

fig04_17.cpp

(1 of 2)

```
1  // Fig. 4.17: Analysis.cpp
2  // Member-function definitions for class Analysis that
3  // analyzes examination results.
4  #include <iostream>
5  using std::cout;
6  using std::cin;
7  using std::endl;
8
9  // include definition of class Analysis from Analysis.h
10 #include "Analysis.h"
11
12 // process the examination results of 10 students
13 void Analysis::processExamResults()
14 {
15    // initializing variables in declarations
16    int passes = 0; // number of passes
17    int failures = 0; // number of failures
18    int studentCounter = 1; // student counter
19    int result; // one exam result (1 = pass, 2 = fail)
20
```

Declare function **processExamResults**'s local variables

Outline

fig04_17.cpp

(2 of 2)

```
21    // process 10 students using counter-controlled loop
22    while ( studentCounter <= 10 )
23    {
24       // prompt user for input and obtain value from user
25       cout << "Enter result (1 = pass, 2 = fail): ";
26       cin >> result; // input result
27
28       // if...else nested in while
29       if ( result == 1 )          // if result is 1,
30          passes = passes + 1;     // increment passes;
31       else                        // else result is not 1, so
32          failures = failures + 1; // increment failures
33
34       // increment studentCounter so loop eventual
35       studentCounter = studentCounter + 1;
36    } // end while
37
38    // termination phase; display number of passes and failures
39    cout << "Passed " << passes << "\nFailed " << failures << endl;
40
41    // determine whether more than eight students passed
42    if ( passes > 8 )
43       cout << "Raise tuition " <<
44 } // end function processExamResu
```

Determine whether this student passed or failed, and increment the appropriate variable

Determine whether more than eight students passed

Outline

fig04_18.cpp

(1 of 2)

```
1  // Fig. 4.18: fig04_18.cpp
2  // Test program for class Analysis.
3  #include "Analysis.h" // include definition of class Analysis
4
5  int main()
6  {
7     Analysis application; // create Analysis object
8     application.processExamResults(); // call function to process results
9     return 0; // indicate successful termination
10 } // end main
```

```
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Passed 9
Failed 1
Raise tuition
```

More than eight students passed the exam

```
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Passed 6
Failed 4
```

Outline

`fig04_18.cpp`

(2 of 2)

4.11 Assignment Operators

- **Assignment expression abbreviations**
  - **Addition assignment operator**
    - **Example**
      - `c = c + 3;` **abbreviates to** `c += 3;`
- **Statements of the form**

  variable = *variable operator expression*;

  can be rewritten as

  variable operator= *expression*;
- **Other assignment operators**
  - `d -= 4` `(d = d - 4)`
  - `e *= 5` `(e = e * 5)`
  - `f /= 3` `(f = f / 3)`
  - `g %= 9` `(g = g % 9)`

Assignment operator	Sample expression	Explanation	Assigns
Assume: `int c = 3, d = 5, e = 4, f = 6, g = 12;`			
`+=`	`c += 7`	`c = c + 7`	10 to `c`
`-=`	`d -= 4`	`d = d - 4`	1 to `d`
`*=`	`e *= 5`	`e = e * 5`	20 to `e`
`/=`	`f /= 3`	`f = f / 3`	2 to `f`
`%=`	`g %= 9`	`g = g % 9`	3 to `g`

Fig. 4.19 | Arithmetic assignment operators.

4.12 Increment and Decrement Operators

- **Increment operator ++**
  - **Increments variable by one**
    - **Example**
      - `C++`
- **Decrement operator --**
  - **Decrement variable by one**
    - **Example**
      - `C--`

4.12 Increment and Decrement Operators (Cont.)

- **Preincrement**
  - **When the operator is used before the variable (`++c` or `--c`)**
  - **Variable is changed, then the expression it is in is evaluated using the new value**
- **Postincrement**
  - **When the operator is used after the variable (`c++` or `c--`)**
  - **Expression the variable is in executes using the old value, then the variable is changed**

Operator	Called	Sample expression	Explanation
`++`	preincrement	`++a`	Increment `a` by 1, then use the new value of `a` in the expression in which `a` resides.
`++`	postincrement	`a++`	Use the current value of `a` in the expression in which `a` resides, then increment `a` by 1.
`--`	predecrement	`--b`	Decrement `b` by 1, then use the new value of `b` in the expression in which `b` resides.
`--`	postdecrement	`b--`	Use the current value of `b` in the expression in which `b` resides, then decrement `b` by 1.

Fig. 4.20 | Increment and decrement operators.

Good Programming Practice 4.8

Unlike binary operators, the unary increment and decrement operators should be placed next to their operands, with no intervening spaces.

Outline

fig04_21.cpp

(1 of 1)

```
1  // Fig. 4.21: fig04_21.cpp
2  // Preincrementing and postincrementing.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     int c;
10
11    // demonstrate postincrement
12    c = 5; // assign 5 to c
13    cout << c << endl; // print 5
14    cout << c++ << endl; // print 5 then postincrement
15    cout << c << endl; // print 6
16
17    cout << endl; // skip a line
18
19    // demonstrate preincrement
20    c = 5; // assign 5 to c
21    cout << c << endl; // print 5
22    cout << ++c << endl; // preincrement then print 6
23    cout << c << endl; // print 6
24    return 0; // indicate successful termination
25 } // end main
```

Postincrementing the `c` variable

Preincrementing the `c` variable

```
5
5
6

5
6
6
```

4.12 Increment and Decrement Operators (Cont.)

- If `c = 5`, then
  - `cout << ++c;`
    - `c` is changed to `6`
    - Then prints out `6`
  - `cout << c++;`
    - Prints out `5` (`cout` is executed before the increment)
    - `c` then becomes `6`

4.12 Increment and Decrement Operators (Cont.)

- **When variable is not in an expression**
  - **Preincrementing and postincrementing have same effect**
    - **Example**
      - ```
        ++c;
        cout << c;
        ```
        and
        ```
        c++;
        cout << c;
        ```
        are the same


        ```

Common Programming Error 4.14

Attempting to use the increment or decrement operator on an expression other than a modifiable variable name or reference, e.g., writing `++(x + 1)`, is a syntax error.

Operators						Associativity	Type
`()`						left to right	parentheses
`++`	`--`	`static_cast< type >()`				left to right	unary (postfix)
`++`	`--`	`+`	`-`			right to left	unary (prefix)
`*`	`/`	`%`				left to right	multiplicative
`+`	`-`					left to right	additive
`<<`	`>>`					left to right	insertion/extraction
`<`	`<=`	`>`	`>=`			left to right	relational
`==`	`!=`					left to right	equality
`?:`						right to left	conditional
`=`	`+=`	`-=`	`*=`	`/=`	`%=`	right to left	assignment

Fig. 4.22 | Operator precedence for the operators encountered so far in the text.

4.13 (Optional) Software Engineering Case Study: Identifying Class Attributes in the ATM System

- **Identifying and modeling attributes**
  - **Create attributes and assign them to classes**
    - **Look for descriptive words and phrases in the requirements document**
    - **Each attribute is given an attribute type**
    - **Some attributes may have an initial value**
    - **Example**
      - `userAuthenticated : Boolean = false`
        - **Attribute `userAuthenticated` is a `Boolean` value and is initially `false`**

Class	Descriptive words and phrases
ATM	user is authenticated
BalanceInquiry	account number
Withdrawal	account number amount
Deposit	account number amount
BankDatabase	[no descriptive words or phrases]
Account	account number PIN balance
Screen	[no descriptive words or phrases]
Keypad	[no descriptive words or phrases]
CashDispenser	begins each day loaded with 500 $20 bills
DepositSlot	[no descriptive words or phrases]

Fig. 4.23 | Descriptive words and phrases from the ATM requirements.

4.13 (Optional) Software Engineering Case Study: Identifying Class Attributes in the ATM System (Cont.)

- **Identifying and modeling attributes (Cont.)**
  - **Some classes may end up without any attributes**
    - **Additional attributes may be added later on as the design and implementation process continues**
  - **Class-type attributes are modeled more clearly as associations**

Fig. 4.24 | Classes with attributes.

Software Engineering Observation 4.8

At early stages in the design process, classes often lack attributes (and operations). Such classes should not be eliminated, however, because attributes (and operations) may become evident in the later phases of design and implementation.