Chapter 4, Control Statements: Part 1

C++ How to Program, 8/e

OBJECTIVES

In this chapter you'll 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.

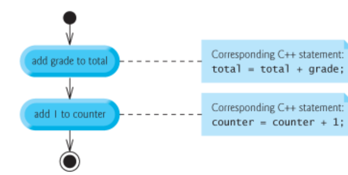

Fig. 4.2 | Sequence-structure activity diagram.

4.4 Control Structures (cont.)

- C++ provides three types of selection statements (discussed in this chapter and Chapter 5).
- The `if` selection statement either performs (selects) an action if a condition (predicate) is true or skips the action if the condition is false.
- The `if…else` selection statement performs an action if a condition is true or performs a different action if the condition is false.
- The `switch` selection statement (Chapter 5) performs one of many different actions, depending on the value of an integer expression.

4.4 Control Structures (cont.)

- The `if` selection statement is a single-selection statement because it selects or ignores a *single action* (or, as we'll soon see, a *single group of actions*).
- The `if…else` statement is called a double-selection statement because it selects between two different actions (or groups of actions).
- The `switch` selection statement is called a multiple-selection statement because it selects among many different actions (or groups of actions).

4.4 Control Structures (cont.)

- C++ provides three types of repetition statements (also called looping statements or loops) for performing statements repeatedly while a condition (called the loop-continuation condition) remains true.
- These are the `while`, `do…while` and `for` statements.
- The `while` and `for` statements perform the action (or group of actions) in their bodies zero or more times.
- The `do…while` statement performs the action (or group of actions) in its body *at least once*.

4.4 Control Structures (cont.)

- Each of the words `if`, `else`, `switch`, `while`, `do` and `for` is a C++ keyword.
- These words are reserved by the C++ programming language to implement various features, such as C++'s control statements.
- Keywords *cannot* be used as identifiers, such as variable names.
- Figure 4.3 provides a complete list of C++ keywords-.

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

Fig. 4.3 | C++ keywords. (Part 1 of 2.)

C++ Keywords

typeid	typename	using	virtual	wchar_t
xor	xor_eq			

C++11 keywords

alignas	alignof	char16_t	char32_t	constexpr
decltype	noexcept	nullptr	static_assert	thread_local

Fig. 4.3 | C++ keywords. (Part 2 of 2.)

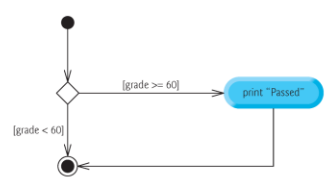

Fig. 4.4 | if single-selection statement activity diagram.

4.5 `if` Selection Statement (cont.)

- A decision can be based on any expression—if the expression evaluates to *zero*, it's treated as *false*; if the expression evaluates to *nonzero*, it's treated as true.
- C++ provides the data type `bool` for variables that can hold only the values `true` and `false`—each of these is a C++ keyword.

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...else` double-selection statement
  - specifies an action to perform when the condition is true and a different action to perform when the condition is `false`.
- The following pseudocode prints "Passed" if the student's grade is greater than or equal to 60, or "Failed" if the student's grade is less than 60.

```
If student's grade is greater than or equal to 60
    Print "Passed"
Else
    Print "Failed"
```


4.6 `if...else` Double-Selection Statement

- In either case, after printing occurs, the next pseudocode statement in sequence is "performed."
- The preceding pseudocode *If…Else* statement can be written in C++ as

```
if ( grade >= 60 )
   cout << "Passed";
else
   cout << "Failed";
```


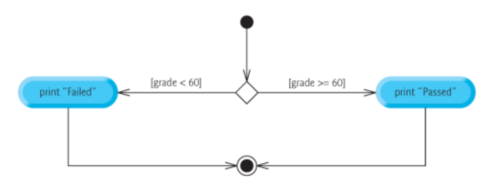

Fig. 4.5 | `if...else` double-selection statement activity diagram.

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

- Conditional operator (`?:`)
  - Closely related to the `if…else` statement.
- C++'s only ternary operator—it takes three operands.
- The operands, together with the conditional operator, form a conditional expression.
  - The first operand is a condition
  - The second operand is the value for the entire conditional expression if the condition is `true`
  - The third operand is the value for the entire conditional expression if the condition is `false`.
- The values in a conditional expression also can be actions to execute.

Conditional operator (?:)

```
if (a > b)
{
      largest = a;
} else
      largest = b;
```

is essentially the same as:

```
largest = ((a > b) ? a : b);
```


Error-Prevention Tip 4.2

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 test for multiple cases by placing `if…else` selection statements inside other `if…else` selection statements.

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.)

- This pseudocode can be written in C++ as

```
if ( studentGrade >= 90 ) // 90 and above gets "A"
   cout << "A";
else
   if ( studentGrade >= 80 ) // 80-89 gets "B"
      cout << "B";
   else
      if ( studentGrade >= 70 ) // 70-79 gets "C"
         cout << "C";
      else
         if ( studentGrade >= 60 ) // 60-69 gets "D"
            cout << "D";
         else // less than 60 gets "F"
            cout << "F";
```

- If `studentGrade` is greater than or equal to 90, the first four conditions are `true`, but only the output statement after the first test executes. Then, the program skips the `else`-part of the "outermost" `if…else` statement.

4.6 if...else Double-Selection Statement (cont.)

- Most programmers write the preceding statement as

```
if ( studentGrade >= 90 ) // 90 and above gets "A"
   cout << "A";
else if ( studentGrade >= 80 ) // 80-89 gets "B"
   cout << "B";
else if ( studentGrade >= 70 ) // 70-79 gets "C"
   cout << "C";
else if ( studentGrade >= 60 ) // 60-69 gets "D"
   cout << "D";
else // less than 60 gets "F"
   cout << "F";
```

- The two forms are identical except for the spacing and indentation, which the compiler ignores.
- The latter form is popular because it avoids deep indentation of the code to the right, which can force lines to wrap.

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 statement. This will enable the nested `if…else` statement to run faster by exiting earlier than if infrequently occurring cases were tested first.

4.6 `if...else` Double-Selection Statement (cont.)

- The C++ compiler always associates an `else` with the immediately preceding `if` unless told to do otherwise by the placement of braces (`{` and `}`).
- This behavior can lead to what's referred to as the dangling-`else` problem.

```
if ( x > 5 )
    if ( y > 5 )
        cout << "x and y are > 5";
else
    cout << "x is <= 5";
```

appears to indicate that if `x` is greater than `5`, the nested `if` statement determines whether `y` is also greater than `5`.

4.6 `if...else` Double-Selection Statement (cont.)

- The compiler actually interprets the statement as

```
if ( x > 5 )
    if ( y > 5 )
        cout << "x and y are > 5";
    else
        cout << "x is <= 5";
```

- To force the nested `if...else` statement to execute as intended, use:

```
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 `if` and that the `else` is associated with the first `if`.

4.6 `if...else` Double-Selection Statement (cont.)

- The `if` selection statement expects only one statement in its body.
- Similarly, the `if` and `else` parts of an `if...else` statement each expect only one body statement.
- To include several statements in the body of an `if` or in either part of an `if...else`, enclose the statements in braces (`{` and `}`).
- A set of statements contained within a pair of braces is called a compound statement or a block.

Software Engineering Observation 4.2

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

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

- Just as a block can be placed anywhere a single statement can be placed, it's also possible to have no statement at all—called a null statement (or an empty statement).
- The null state-ment is represented by placing a semicolon (`;`) where a statement would normally be.

Common Programming Error 4.1

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

- A repetition statement (also called a looping statement or a loop) allows you to specify that a program should repeat an action while some condition remains true.

  While there are more items on my shopping list
  Purchase next item and cross it off my list

- "There are more items on my shopping list" is true or false.
  - If true, "Purchase next item and cross it off my list" is performed.
    - Performed repeatedly while the condition remains true.
  - The statement contained in the *While* repetition statement constitutes the body of the *While*, which can be a single statement or a block.
  - Eventually, the condition will become false, the repetition will terminate, and the first pseudocode statement after the repetition statement will execute.

4.7 `while` Repetition Statement (cont.)

- Consider a program segment designed to find the first power of 3 larger than 100. Suppose the integer variable `product` has been initialized to `3`.
- When the following `while` repetition statement finishes executing, `product` contains the result:

```
int product = 3;

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

4.7 `while` Repetition Statement (cont.)

- The UML activity diagram of Fig. 4.6 illustrates the flow of control that corresponds to the preceding `while` statement.
- This diagram introduces the UML's merge symbol, which joins two flows of activity into one flow of activity.
- The UML represents both the merge symbol and the decision symbol as diamonds.
- The merge symbol joins the transitions from the initial state and from the action state, so they both flow into the decision that determines whether the loop should begin (or continue) executing.

4.7 `while` Repetition Statement (cont.)

- The decision and merge symbols can be distinguished by the number of "incoming" and "outgoing" transition arrows.
  - A decision symbol has one transition arrow pointing to the diamond and two or more transition arrows pointing out from the diamond to indicate possible transitions from that point.
  - Each transition arrow has a guard condition next to it.
- A merge symbol has two or more transition arrows pointing to the diamond and only one transition arrow pointing from the diamond, to indicate multiple activity flows merging to continue the activity.
  - Unlike the decision symbol, the merge symbol does not have a counterpart in C++ code.

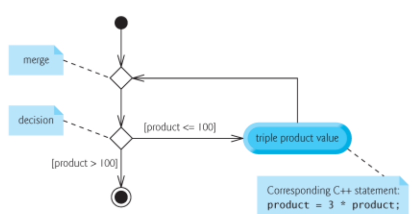

Fig. 4.6 | `while` repetition statement UML activity diagram.

4.8 Formulating Algorithms: Counter-Controlled Repetition

- Consider the following problem statement:
  - A class of ten students took a quiz. The grades (0 to 100) for this quiz are available to you. Calculate and display the total of the grades and the class average.
- The class average is equal to the sum of the grades divided by the number of students.
- The algorithm for solving this problem on a computer must input each of the grades, calculate the average and print the result.

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

- We use counter-controlled repetition to input the grades one at a time.
  - This technique uses a variable called a counter to control the number of times a group of statements will execute (also known as the number of iterations of the loop).
  - Often called definite repetition because the number of repetitions is known *before* the loop begins executing.


```cpp
// Fig. 4.8: GradeBook.h
// Definition of class GradeBook that determines a class average.
// Member functions are defined in GradeBook.cpp
#include <string> // program uses C++ standard string class

// GradeBook class definition
class GradeBook
{
public:
   explicit GradeBook( std::string ); // initializes course name
   void setCourseName( std::string ); // set the course name
   std::string getCourseName() const; // retrieve the course name
   void displayMessage() const; // display a welcome message
   void determineClassAverage() const; // averages user-entered grades
private:
   std::string courseName; // course name for this GradeBook
}; // end class GradeBook
```

Fig. 4.8 | Class average problem using counter-controlled repetition: GradeBook header.


```cpp
// display a welcome message to the GradeBook user
void GradeBook::displayMessage() const
{
   cout << "Welcome to the grade book for\n" << getCourseName() << "!\n"
      << endl;
} // end function displayMessage

// determine class average based on 10 grades entered by user
void GradeBook::determineClassAverage() const
{
   // initialization phase
   int total = 0; // sum of grades entered by user
   unsigned int gradeCounter = 1; // number of grade to be entered next

   // processing phase
   while ( gradeCounter <= 10 ) // loop 10 times
   {
      cout << "Enter grade: "; // prompt for input
      int grade = 0; // grade value entered by user
      cin >> grade; // input next grade
      total = total + grade; // add grade to total
      gradeCounter = gradeCounter + 1; // increment counter by 1
   } // end while
```

Fig. 4.9 | Class average problem using counter-controlled repetition: GradeBook source code file. (Part 3 of 4.)


```cpp

   // termination phase
   int average = total / 10; // ok to mix declaration and calculation

   // display total and average of grades
   cout << "\nTotal of all 10 grades is " << total << endl;
   cout << "Class average is " << average << endl;
} // end function determineClassAverage
```

Fig. 4.9 | Class average problem using counter-controlled repetition: GradeBook source code file. (Part 4 of 4.)

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

- You'll normally initialize counter variables to zero or one, depending on how they are used in an algorithm.
- An uninitialized variable contains a "garbage" value (also called an undefined value)—the value last stored in the memory location reserved for that variable.

Good Programming Practice 4.2

Declare each variable on a separate line with its own comment for readability.

Error-Prevention Tip 4.3

Always initialize variables when they're declared. This helps you avoid logic errors that occur when you perform calculations with uninitialized variables.

Error-Prevention Tip 4.4

In some cases, compilers issue a warning if you attempt to use an uninitialized variable's value. You should always get a clean compile by resolving all errors and warnings.

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

- The averaging calculation performed in response to the function call in line 12 of Fig. 4.10 produces an integer result.
- Dividing two integers results in integer division—any fractional part of the calculation is truncated (discarded).

Common Programming Error 4.3

Assuming that integer division rounds (rather than truncates) can lead to incorrect results. For example, 7 ÷ 4, yields 1.75 in conventional arithmetic, but truncates the floating-point part (.75) in integer arithmetic. So the result is 1. Similarly, –7 ÷ 4, yields –1.

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

A Note About Arithmetic Overflow

- In Fig. 4.9, line 54

```
total = total + grade; // add grade to total
```

added each grade entered by the user to the total.

- Even this simple statement has a *potential* problem—adding the integers could result in a value that's *too large* to store in an `int` variable.
- This is known as arithmetic overflow and causes *undefined behavior*, which can lead to unintended results (`en.wikipedia.org/wiki/Integer_overflow#Security_ ramifications`).

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

- Figure 2.5's addition program had the same issue in line 19, which calculated the sum of two `int` values entered by the user:

```
// add the numbers; store result in sum
sum = number1 + number2;
```

- The maximum and minimum values that can be stored in an `int` variable are represented by the constants `INT_MAX` and `INT_MIN`, respectively, which are defined in the header `<climits>`.

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

- There are similar constants for the other integral types and for floating-point types.
- You can see your platform's values for these constants by opening the headers `<climits>` and `<cfloat>` in a text editor (you can search your file system for these files).
- It's considered a good practice to ensure that *before* you perform arithmetic calculations like the ones in line 54 of Fig. 4.9 and line 19 of Fig. 2.5, they will *not* overflow.
- The code for doing this is shown on the CERT website `www.securecoding.cert.org`—just search for guideline "INT32-CPP."

4.9 Formulating Algorithms: Sentinel-Controlled Repetition

- Let's generalize the class average problem.
  - Develop a class average program that processes grades for an arbitrary number of students each time it's run.
- The program must process an arbitrary number of grades.
  - How can the program determine when to stop the input of grades?
- Can use a special value called a sentinel value (also called a signal value, a dummy value or a flag value) to indicate "end of data entry."
- Sentinel-controlled repetition is often called indefinite repetition
  - the number of repetitions is not known in advance.
- The sentinel value must not be an acceptable input value.

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

- We don't know in advance how many grades are to be processed, so we'll use sentinel-controlled repetition.
- The user enters legitimate grades one at a time.
- After entering the last legitimate grade, the user enters the sentinel value.
- The program tests for the sentinel value after each grade is input and terminates the loop when the user enters the sentinel value.

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

- An averaging calculation is likely to produce a number with a decimal point—a real number or floating-point number (e.g., 7.33, 0.0975 or 1000.12345).
- Type `int` cannot represent such a number.
- C++ provides several data types for storing floating-point numbers in memory, including `float` and `double`.
- Compared to `float` variables, `double` variables can typically store numbers with larger magnitude and finer detail
  - more digits to the right of the decimal point—also known as the number's precision.
- Cast operator can be used to force the averaging calculation to produce a floating-point numeric result.

```cpp
// Fig. 4.12: GradeBook.h
// Definition of class GradeBook that determines a class average.
// Member functions are defined in GradeBook.cpp
#include <string> // program uses C++ standard string class

// GradeBook class definition
class GradeBook
{
public:
   explicit GradeBook( std::string ); // initializes course name
   void setCourseName( std::string ); // set the course name
   std::string getCourseName() const; // retrieve the course name
   void displayMessage() const; // display a welcome message
   void determineClassAverage() const; // averages user-entered grades
private:
   std::string courseName; // course name for this GradeBook
}; // end class GradeBook
```

Fig. 4.12 | Class average problem using sentinel-controlled repetition: GradeBook header.


```cpp
// Fig. 4.13: GradeBook.cpp
// Member-function definitions for class GradeBook that solves the
// class average program with sentinel-controlled repetition.
#include <iostream>
#include <iomanip> // parameterized stream manipulators
#include "GradeBook.h" // include definition of class GradeBook
using namespace std;

// constructor initializes courseName with string supplied as argument
GradeBook::GradeBook( string name )
{
   setCourseName( name ); // validate and store courseName
} // end GradeBook constructor

```

Fig. 4.13 | Class average problem using sentinel-controlled repetition: GradeBook source code file. (Part 1 of 5.)


```cpp
// function to set the course name;
// ensures that the course name has at most 25 characters
void GradeBook::setCourseName( string name )
{
   if ( name.size() <= 25 ) // if name has 25 or fewer characters
      courseName = name; // store the course name in the object
   else // if name is longer than 25 characters
   { // set courseName to first 25 characters of parameter name
      courseName = name.substr( 0, 25 ); // select first 25 characters
      cerr << "Name \"" << name << "\" exceeds maximum length (25).\n"
         << "Limiting courseName to first 25 characters.\n" << endl;
   } // end if...else
} // end function setCourseName

// function to retrieve the course name
string GradeBook::getCourseName() const
{
   return courseName;
} // end function getCourseName

```

Fig. 4.13 | Class average problem using sentinel-controlled repetition: GradeBook source code file. (Part 2 of 5.)


```cpp
// display a welcome message to the GradeBook user
void GradeBook::displayMessage() const
{
   cout << "Welcome to the grade book for\n" << getCourseName() << "!\n"
      << endl;
} // end function displayMessage

// determine class average based on 10 grades entered by user
void GradeBook::determineClassAverage() const
{
   // initialization phase
   int total = 0; // sum of grades entered by user
   unsigned int gradeCounter = 0; // number of grades entered

   // processing phase
   // prompt for input and read grade from user
   cout << "Enter grade or -1 to quit: ";
   int grade = 0; // grade value
   cin >> grade; // input grade or sentinel value

```

Fig. 4.13 | Class average problem using sentinel-controlled repetition: GradeBook source code file. (Part 3 of 5.)

```
      // loop until sentinel value read from user
      while ( grade != -1 ) // while grade is not -1
      {
         total = total + grade; // add grade to total
         gradeCounter = gradeCounter + 1; // increment counter

         // prompt for input and read next grade from user
         cout << "Enter grade or -1 to quit: ";
         cin >> grade; // input grade or sentinel value
      } // end while

      // termination phase
      if ( gradeCounter != 0 ) // if user entered at least one grade...
      {
         // calculate average of all grades entered
         double average = static_cast< double >( total ) / gradeCounter;

         // display total and average (with two digits of precision)
         cout << "\nTotal of all " << gradeCounter << " grades entered is "
            << total << endl;
         cout << setprecision( 2 ) << fixed;
         cout << "Class average is " << average << endl;
      } // end if
```

Fig. 4.13 | Class average problem using sentinel-controlled repetition: GradeBook source code file. (Part 4 of 5.)


```
      else // no grades were entered, so output appropriate message
         cout << "No grades were entered" << endl;
   } // end function determineClassAverage
```

Fig. 4.13 | Class average problem using sentinel-controlled repetition: GradeBook source code file. (Part 5 of 5.)


```
// Fig. 4.14: fig04_14.cpp
// Create GradeBook object and invoke its determineClassAverage function.
#include "GradeBook.h" // include definition of class GradeBook

int main()
{
   // create GradeBook object myGradeBook and
   // pass course name to constructor
   GradeBook myGradeBook( "CS101 C++ Programming" );

   myGradeBook.displayMessage(); // display welcome message
   myGradeBook.determineClassAverage(); // find average of 10 grades
} // end main
```

Fig. 4.14 | Class average problem using sentinel-controlled repetition: Creating a GradeBook object and invoking its determineClassAverage member function. (Part 1 of 2.)


```
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
```

Fig. 4.14 | Class average problem using sentinel-controlled repetition: Creating a GradeBook object and invoking its determineClassAverage member function. (Part 2 of 2.)

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

- Variables of type `float` represent single-precision floating-point numbers and have seven significant digits on most of today's systems.
- Variables of type `double` represent double-precision floating-point numbers.
  - These require twice as much memory as `float` variables and provide 15 significant digits on most of today's systems
  - Approximately double the precision of `float` variables
- C++ treats all floating-point numbers in a program's source code as `double` values by default.
  - Known as floating-point literals.
- Floating-point numbers often arise as a result of division.

Common Programming Error 4.7

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

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

- The variable `average` is declared to be of type `double` to capture the fractional result of our calculation.
- `total` and `gradeCounter` are both integer variables.
- Recall that dividing two integers results in integer division, in which any fractional part of the calculation is lost (i.e., truncated).
- In the following statement the division occurs *first*—the result's fractional part is lost before it's assigned to `average`:
  - `average = total / gradeCounter;`

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

- To perform a floating-point calculation with integers, create *temporary* floating-point values.
- `static_cast` operator accomplishes this task.
- The cast operator `static_cast<double>(total)` creates a *temporary* floating-point copy of its operand in parentheses.
  - Known as explicit conversion.
  - The value stored in `total` is still an integer.

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

- The calculation now consists of a floating-point value divided by the integer `gradeCounter`.
  - The compiler knows how to evaluate only expressions in which the operand types of are *identical*.
  - Compiler performs promotion (also called implicit conversion) on selected operands.
  - In an expression containing values of data types `int` and `double`, C++ promotes `int` operands to `double` values.
- Cast operators are available for use with every data type and with class types as well.

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

- The call to `setprecision` in line 75 (with an argument of 2) indicates that `double` values should be printed with *two* digits of precision to the right of the decimal point (e.g., 92.37).
  - Parameterized stream manipulator (argument in parentheses).
  - Programs that use these must include the header `<iomanip>`.
- `endl` is a nonparameterized stream manipulator and does not require the `<iomanip>` header file.
- If the precision is not specified, floating-point values are normally output with six digits of precision.

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

- Stream manipulator `fixed` indicates that floating-point values should be output in fixed-point format, as opposed to scientific notation.
- Fixed-point formatting is used to force a floating-point number to display a specific number of digits.
- Specifying fixed-point formatting also forces the decimal point and trailing zeros to print, even if the value is a whole number amount, such as 88.00.
  - Without the fixed-point formatting option, such a value prints in C++ as 88 without the trailing zeros and decimal point.

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

- When the stream manipulators `fixed` and `setprecision` are used in a program, the printed value is rounded to the number of decimal positions indicated by the value passed to `setprecision` (e.g., the value 2 in line 75), although the value in memory re-mains unaltered.
- It's also possible to force a decimal point to appear by using stream manipulator `showpoint`.
  - If `showpoint` is specified without `fixed`, then trailing zeros will not print.
  - Both can be found in header `<iostream>`.

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

A Note About Unsigned Integers

- In Fig. 4.9, line 46 declared the variable `gradeCounter` as an unsigned `int` because it can assume only the values from 1 through 11 (11 terminates the loop), which are all positive values.
- In general, counters that should store only non-negative values should be declared with `unsigned` types.
- Variables of `unsigned` integer types can represent values from 0 to approximately *twice the positive range* of the corresponding signed integer types.
- You can determine your platform's maximum `unsigned int` value with the constant `UINT_MAX` from `<climits>`.

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

- Figure 4.9 could have also declared as `unsigned int` the variables `grade`, `total` and `average`. Grades are normally values from 0 to 100, so the `total` and `average` should each be greater than or equal to 0.
- We declared those variables as `int`s because we can't control what the user actually enters—the user could enter *negative* values.
- Worse yet, the user could enter a value that's not even a number.

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

- Sometimes sentinel-controlled loops use *intentionally* invalid values to terminate a loop.
- For example, in line 56 of Fig. 4.13, we terminate the loop when the user enters the sentinel `-1` (an invalid grade), so it would be improper to declare variable grade as an unsigned int.
- As you'll see, the end-of-file (`EOF`) indicator—which is introduced in the next chapter and is often used to terminate sentinel-controlled loops—is also normally implemented internally in the compiler as a negative number.


```cpp
// Fig. 4.16: fig04_16.cpp
// Examination-results problem: Nested control statements.
#include <iostream>
using namespace std;

int main()
{
   // initializing variables in declarations
   unsigned int passes = 0; // number of passes
   unsigned int failures = 0; // number of failures
   unsigned int studentCounter = 1; // student counter

```

Fig. 4.16 | Examination-results problem: Nested control statements. (Part 1 of 4.)

```
13      // process 10 students using counter-controlled loop
14      while ( studentCounter <= 10 )
15      {
16         // prompt user for input and obtain value from user
17         cout << "Enter result (1 = pass, 2 = fail): ";
18         int result = 0; // one exam result (1 = pass, 2 = fail)
19         cin >> result; // input result
20
21         // if...else nested in while
22         if ( result == 1 )          // if result is 1,
23            passes = passes + 1;     // increment passes;
24         else                        // else result is not 1, so
25            failures = failures + 1; // increment failures
26
27         // increment studentCounter so loop eventually terminates
28         studentCounter = studentCounter + 1;
29      } // end while
30
31      // termination phase; display number of passes and failures
32      cout << "Passed " << passes << "\nFailed " << failures << endl;
33
34      // determine whether more than eight students passed
35      if ( passes > 8 )
36         cout << "Bonus to instructor!" << endl;
37   } // end main
```

Fig. 4.16 | Examination-results problem: Nested control statements. (Part 2 of 4.)

4.10 Formulating Algorithms: Nested Control Statements (cont.)

C++11 List Initialization

- C++11 introduces a new variable initialization syntax. List initialization (also called uniform initialization) enables you to use one syntax to initialize a variable of *any* type.
- Consider line 11 of Fig. 4.16

```
unsigned int studentCounter = 1;
```

- In C++11, you can write this as

```
unsigned int studentCounter = { 1 };
```

- or

```
unsigned int studentCounter{ 1 };
```


4.10 Formulating Algorithms: Nested Control Statements (cont.)

- The braces ({ and }) represent the *list initializer*.
- For a fundamental-type variable, you place only one value in the list initializer.
- For an object, the list initializer can be a comma-separated list of values that are passed to the object's constructor.

4.10 Formulating Algorithms: Nested Control Statements (cont.)

- For example, Exercise 3.14 asked you to create an `Employee` class that could represent an employee's first name, last name and salary.
- Assuming the class defines a constructor that receives `string`s for the first and last names and a `double` for the salary, you could initialize `Employee` objects as follows:

```
Employee employee1{ "Bob", "Blue", 1234.56 };
Employee employee2 = { "Sue", "Green", 2143.65 };
```

4.10 Formulating Algorithms: Nested Control Statements (cont.)

- For fundamental-type variables, list-initialization syntax also *prevents* so-called narrowing conversions that could result in *data loss*.
- For example, previously you could write

```
int x = 12.7;
```

- which attempts to assign the `double` value `12.7` to the `int` variable `x`.
- A `double` value is converted to an `int`, by truncating the floating-point part (.7), which results in a loss of information—a *narrowing conversion*.

4.10 Formulating Algorithms: Nested Control Statements (cont.)

- The actual value assigned to `x` is `12`.
- Many compilers generate a warning for this statement, but still allow it to compile.
- However, using list initialization, as in

```
int x = { 12.7 };
```

- or

```
int x{ 12.7 };
```

- yields a *compilation error*, thus helping you avoid a potentially subtle logic error. For example, Apple's Xcode LLVM compiler gives the error

```
Type 'double' cannot be narrowed to 'int' in initializer list
```


4.11 Assignment Operators

- C++ provides several assignment operators for abbreviating assignment expressions.
- The += operator adds the value of the expression on the right of the operator to the value of the variable on the left of the operator and stores the result in the variable on the left of the operator.
- Any statement of the form
  - *`variable = variable operator expression;`*
- in which the same *variable appears on both sides of the assignment operator and operator is one of the binary operators +, -, *, /, or % (or others we'll discuss later in the text), can be written in the form*
  - *`variable operator= expression;`*
- Thus the assignment `c += 3` adds `3` to `c`.
- Figure 4.17 shows the arithmetic assignment operators, sample expressions using these operators and explanations.

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. 4.17 | Arithmetic assignment operators.

4.12 Increment and Decrement Operators

- C++ also provides two unary operators for adding 1 to or subtracting 1 from the value of a numeric variable.
- These are the unary increment operator, ++, and the unary decrement operator, --, which are summarized in Fig. 4.18.

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. 4.18 | Increment and decrement operators.

Good Programming Practice 4.4

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


```cpp
// Fig. 4.19: fig04_19.cpp
// Preincrementing and postincrementing.
#include <iostream>
using namespace std;

int main()
{
   // demonstrate postincrement
   int c = 5; // assign 5 to c
   cout << c << endl; // print 5
   cout << c++ << endl; // print 5 then postincrement
   cout << c << endl; // print 6

   cout << endl; // skip a line

   // demonstrate preincrement
   c = 5; // assign 5 to c
   cout << c << endl; // print 5
   cout << ++c << endl; // preincrement then print 6
   cout << c << endl; // print 6
} // end main
```

Fig. 4.19 | Preincrementing and postincrementing. (Part 1 of 2.)

```
5
5
6

5
6
6
```

Fig. 4.19 | Preincrementing and postincrementing. (Part 2 of 2.)

4.12 Increment and Decrement Operators (cont.)

- When you increment (++) or decrement (--) a variable in a statement by itself, the preincrement and postincrement forms have the same effect, and the predecrement and postdecrement forms have the same effect.
- It's only when a variable appears in the context of a larger expression that preincrementing the variable and postincrementing the variable have different effects (and similarly for predecrementing and post-decrementing).
- Figure 4.20 shows the precedence and associativity of the operators introduced to this point.

Common Programming Error 4.8

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

Operators	Associativity	Type
:: ()	left to right *[See caution in Fig. 2.10 regarding grouping parentheses.]*	primary
++ -- static_cast<*type*>()	left to right	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. 4.20 | Operator precedence for the operators encountered so far in the text.