EE1003
Introduction to Computer I

Chapter 3
Control Statement: Part I

Andy, Yu-Guang Chen
Assistant Professor, Department of EE
National Central University
andyygchen@ee.ncu.edu.tw

Learning 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.

Outline

3.1 Introduction

- Before writing a program to solve a problem, we must have a thorough understanding of the problem and a carefully planned approach to solving it.
- When writing a program, we must also understand the types of building blocks that are available and employ proven program construction techniques.
- In this chapter and in Chapter 4, Control Statements: Part 2, we discuss these issues as we present the theory and principles of structured programming.

3.2 Algorithms

- Any solvable computing problem can be solved by the execution of a series of actions in a specific order.
- An algorithm is procedure for solving a problem in terms of
  - the actions to execute and
  - the order in which the actions execute
- Specifying the order in which statements (actions) execute in a computer program is called program control.
- This chapter investigates program control using C++'s control statements.

3.3 Pseudocode

- Pseudocode (or "fake" code) is an artificial and informal language that helps you develop algorithms.
- Similar to everyday English
- Convenient and user friendly.
- Helps you "think out" a program before attempting to write it.
- Carefully prepared pseudocode can easily be converted to a corresponding C++ program.
- Normally describes only executable statements.
  - Declarations are not executable statements.

3.3 Pseudocode

- An example for the addition program

```
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
```

3.4 Control Structures

- Normally, statements in a program execute one after the other in the order in which they're written.
  - Called sequential execution.
- Various C++ statements enable you to specify the next executing statement that is not the next one in sequence.
  - Called transfer of control.
- All programs could be written in terms of only three control structures (referred as "control statements")
  - the sequence structure
  - the selection structure and
  - the repetition structure

3.4 Control Structures (cont.)

- C++ provides three types of selection statements (discussed in this chapter and Chapter 4).
- The `if` selection statement: (single selection)
  - The condition is true: perform (selects) the following action
  - The condition is false: skip the action
- The `if…else` selection statement: (double selection)
  - The condition is true: perform (selects) the following action
  - The condition is false: perform a different action
- The `switch` selection statement: (multiple selection)
  - Perform one of many different actions, depending on the value of selection expression.
  - Introduced in Chapter 4

3.4 Control Structures (cont.)

- C++ provides three repetition statements (also called looping statements) for performing statements repeatedly.
  - 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.

3.4 Control Structures (cont.)

- Each of the words `if`, `else`, `switch`, `while`, `do` and `for` is a C++ keyword.
- Keywords must not be used as identifiers, such as variable names.

Common Programming Error 3.1

Using a keyword as an identifier is a syntax error.

Common Programming Error 3.2

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

3.4 Control Structures (cont.)

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			

3.4 Control Structures (cont.)

- Each program combines control statements as appropriate for the algorithm the program implements.
- C++ control statements are single-entry/single-exit
- Control statements are attached to one another by connecting the exit point of one to the entry point of the next.
  - Called control-statement stacking
- Another way to connect control statements is containing one control statement inside another one
  - Called control-statement nesting

3.5 `if` Selection Statement

- The following pseudocode determines whether "student's grade is greater than or equal to 60" is `true` or `false`.

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

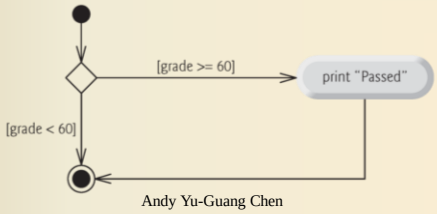
  - If `true`, "Passed" is printed and the next pseudocode statement in order is "performed".
  - If `false`, the print statement is ignored and the next pseudocode statement in order is performed.
  - The indentation of the second line is optional, but it's recommended because it emphasizes the inherent structure of structured programs.

3.5 `if` Selection Statement

- The preceding pseudocode can be written in C++ as

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

- The diamond (decision symbol) indicates that a decision is to be made.
  - The workflow will continue along a path determined by the symbol's associated guard conditions, which can be true or false.
  - If a guard condition is true, the workflow enters the action state to which that transition arrow points.



3.5 `if` Selection Statement

- 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 3.1

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.*

3.6 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"
```

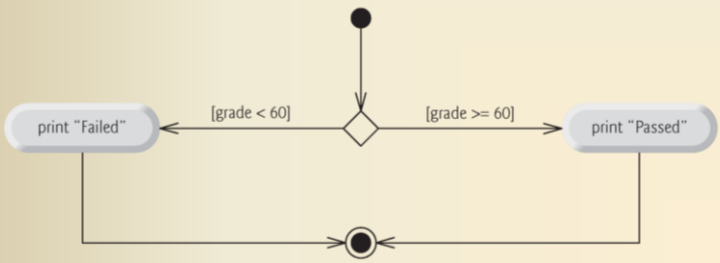
- In either case, after printing occurs, the next pseudocode statement in sequence is "performed."

3.6 if…else Double-Selection Statement

- The preceding pseudocode *If…Else* statement can be written in C++ as

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

- The control flow of `if…else` is shown as follows.



3.6 if…else Double-Selection Statement

Good Programming Practice 3.2

Whatever indentation convention you choose should be applied consistently throughout your programs. It's difficult to read programs that do not obey uniform spacing conventions.

Good Programming Practice 3.4

If there are several levels of indentation, each level should be indented the same additional amount of space to promote readability and maintainability.

3.6 if…else Double-Selection Statement

```
1. (grade>=60) ? cout << "Passed" : cout << "Failed" ;
2. final = (grade>=50) ? 60 : grade ;
```

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

Error-Prevention Tip 4.1

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

3.6 if…else Double-Selection Statement

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

```
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"
```

3.6 if…else Double-Selection Statement

- The pseudo code can rewritten in C++ as follows.

```
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, only the output statement after the first test executes.
  - Skip the other branches

3.6 if…else Double-Selection Statement

- Most write the preceding `if…else` 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.

3.6 if…else Double-Selection Statement

Performance Tip 3.1

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 3.2

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 they would if infrequently occurring cases were tested first.

3.6 if…else Double-Selection Statement

- The C++ compiler always associates an `if` or `else` with the immediately preceding actions.
- This behavior can lead to the dangling-else problem.

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

- What's the difference with a pair of braces ({}) ?

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

3.6 if…else Double-Selection Statement

- 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.

3.6 if…else Double-Selection Statement

Software Engineering Observation 3.2
A block can be placed anywhere in a program that a single statement can be placed.

Common Programming Error 3.3
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 3.5
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.*

3.6 if…else Double-Selection Statement

- 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 statement is represented by placing a semicolon (`;`) where a statement would normally be.

Common Programming Error 3.4
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).*

3.7 while Repetition Statement

- A repetition statement (also called a looping statement) allows you to 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

- 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;
```

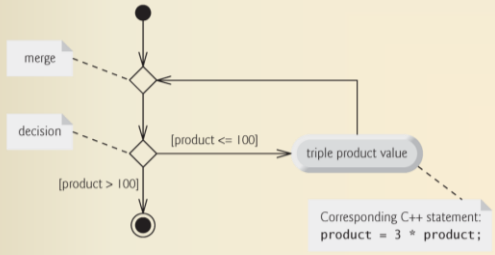
3.7 while Repetition Statement

Common Programming Error 3.5

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.*

3.7 while Repetition Statement

- `while` statement can be represented by a merge symbol
- The merge symbol joins the transitions from the initial state and from the action state
  - Determine whether the loop should begin (or continue) executing
- A merge symbol has two or more input transition arrows and only one output transition arrow
  - Unlike the decision symbol, the merge symbol does not have a counterpart in C++ code



3.8 Formulating Algorithms: Counter-Controlled Repetition

- Consider the following problem statement:
  - A class of 10 students took a quiz.
  - The grades (integers in the range 0 to 100) are available to you.
  - Calculate and display the total of all student grades and the class average on the quiz.
- We use counter-controlled repetition to input the 10 grades one by one.
  - 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.

3.8 Formulating Algorithms: Counter-Controlled Repetition

- A total is a variable used to accumulate the sum of several values.
- A counter is a variable used to count—in this case, the grade counter indicates which of the 10 grades is about to be entered by the user.
- The class average is equal to the sum of the grades (total) divided by the number of students (10).
- Dividing two integers results in integer division—any fractional part of the calculation is lost (i.e., truncated).

3.8 Formulating Algorithms: Counter-Controlled Repetition

```
1  // Fig. 3.8: fig03_08.cpp
2  // Class average program with counter-controlled repetition.
3  #include <iostream>
4  using namespace std;
5
6  int main ()
7  {
8     int total; // sum of grades entered by user
9     int gradeCounter; // number of the grade to be entered next
10    int grade; // grade value entered by user
11    int average; // average of grades
12
13    // initialization phase
14    total = 0; // initialize total
15    gradeCounter = 1; // initialize loop counter
16
17    // processing phase
18    while ( gradeCounter <= 10 ) // loop 10 times
19    {
20       cout << "Enter grade: "; // prompt for input
21       cin >> grade; // input next grade
22       total = total + grade; // add grade to total
23       gradeCounter = gradeCounter + 1; // increment counter by 1
24    } // end while
```

Fig. 3.8 | Class average problem using counter-controlled repetition. (Part 1 of 2.)

3.8 Formulating Algorithms: Counter-Controlled Repetition

```
25
26      // termination phase
27      average = total / 10; // integer division yields integer result
28
29      // display total and average of grades
30      cout << "\nTotal of all 10 grades is " << total << endl;
31      cout << "Class average is " << average << endl;
32   } // end main
```

```
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
```

Fig. 3.8 | Class average problem using counter-controlled repetition. (Part 2 of 2.)

3.8 Formulating Algorithms: Counter-Controlled Repetition

Common Programming Error 3.7

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 3.8

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).

3.8 Formulating Algorithms: Counter-Controlled Repetition

- An uninitialized variable contains a "garbage" value (also called an undefined value)—the value last stored in the memory location reserved for that variable.
- Variables used to store totals are normally initialized to zero before being used in a program
  - Prevent the previous value stored in the total's memory location.
- Counter variables are normally initialized to zero or one, depending on their use.
- The variables `grade` and `average` need not be initialized before they're used—their values will be assigned from input or later calculation.

3.8 Formulating Algorithms: Counter-Controlled Repetition

Common Programming Error 3.6
Not initializing counters and totals can lead to logic errors.

Good Programming Practice 3.7
Declare each variable on a separate line with its own comment for readability.

```
1  // Fig. 3.8: fig03_08.cpp
2  // Class average program with counter-controlled repetition.
3  #include <iostream>
4  using namespace std;
5
6  int main ()
7  {
8     int total; // sum of grades entered by user
9     int gradeCounter; // number of the grade to be entered next
10    int grade; // grade value entered by user
11    int average; // average of grades
12
```

3.9 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.
- 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, or a flag value) to indicate "end of data entry."
  - The sentinel value must not be an acceptable input value.
- Sentinel-controlled repetition is often called indefinite repetition
  - The number of repetitions is not known in advance.

3.9 Formulating Algorithms: Sentinel-Controlled Repetition

- We approach the class average program with a technique called top-down, stepwise refinement
  - Essential to the development of well-structured programs.
- We begin with a pseudocode of the overall function:
  - Determine the class average for the quiz for an arbitrary number of students
- We divide the top into a series of smaller tasks and list these in the order in which they need to be performed.
  - *Initialize variables*
  - *Input, sum and count the quiz grades*
  - *Calculate and print the total of all student grades and the class average*
- This refinement uses only the sequence structure—these steps execute in order.

3.9 Formulating Algorithms: Sentinel-Controlled Repetition

Software Engineering Observation 3.5

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.

3.9 Formulating Algorithms: Sentinel-Controlled Repetition

- 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.
- Enter the sentinel value after the last legitimate grade.
- Test for the sentinel value after each grade is input.

Prompt the user to enter the first grade
Input the first grade (possibly the sentinel)

While the user has not yet entered the sentinel
Add this grade into the running total
Add one to the grade counter
Prompt the user to enter the next grade
Input the next grade (possibly the sentinel)

3.9 Formulating Algorithms: Sentinel-Controlled Repetition

- The pseudocode statement

  Calculate and print the total of all student grades and the class average

- can be refined as follows:

  If the counter is not equal to zero
  Set the average to the total divided by the counter
  Print the total of the grades for all students in the class
  Print the class average
  Else
  Print "No grades were entered"

- Test for the possibility of division by zero
  - Normally a fatal logic error that, if undetected, would cause the program to fail (often called "crashing").

3.9 Formulating Algorithms: Sentinel-Controlled Repetition

```
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. 3.9 | Class average problem pseudocode algorithm with sentinel-controlled repetition.

3.9 Formulating Algorithms: Sentinel-Controlled Repetition

```
1  // Fig. 3.10: fig03_10.cpp
2  // Class average program with sentinel-controlled repetition.
3  #include <iostream>
4  #include <iomanip> // parameterized stream manipulators
5  using namespace std;
6
7  // determine class average based on 10 grades entered by user
8  int main()
9  {
10    int total; // sum of grades entered by user
11    int gradeCounter; // number of grades entered
12    int grade; // grade value
13    double average; // number with decimal point for average
14
15    // initialization phase
16    total = 0; // initialize total
17    gradeCounter = 0; // initialize loop counter
18
19    // processing phase
20    // prompt for input and read grade from user
21    cout << "Enter grade or -1 to quit: ";
22    cin >> grade; // input grade or sentinel value
23
```

Fig. 3.10 | Class average problem using sentinel-controlled repetition: `GradeBook` source code file. (Part 1 of 3.)

3.9 Formulating Algorithms: Sentinel-Controlled Repetition

```
24    // loop until sentinel value read from user
25    while ( grade != -1 ) // while grade is not -1
26    {
27       total = total + grade; // add grade to total
28       gradeCounter = gradeCounter + 1; // increment counter
29
30       // prompt for input and read next grade from user
31       cout << "Enter grade or -1 to quit: ";
32       cin >> grade; // input grade or sentinel value
33    } // end while
34
```

Fig. 3.10 | Class average problem using sentinel-controlled repetition: `GradeBook` source code file. (Part 2 of 3.)

3.9 Formulating Algorithms: Sentinel-Controlled Repetition

```
35     // termination phase
36     if ( gradeCounter != 0 ) // if user entered at least one grade...
37     {
38        // calculate average of all grades entered
39        average = static_cast< double >( total ) / gradeCounter;
40
41        // display total and average (with two digits of precision)
42        cout << "\nTotal of all " << gradeCounter << " grades entered is "
43           << total << endl;
44        cout << "Class average is " << setprecision( 2 ) << fixed << average
45           << endl;
46     } // end if
47     else // no grades were entered, so output appropriate message
48        cout << "No grades were entered" << endl;
49  } // end main
```

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

3.9 Formulating Algorithms: Sentinel-Controlled Repetition

Good Programming Practice 3.8

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.

3.9 Formulating Algorithms: Sentinel-Controlled Repetition

- Notice the block in the `while` loop in Fig. 3.10.
- Without the braces, the last three statements in the body of the loop would fall outside the loop, causing the computer to interpret this code incorrectly, as follows:

```
// loop until sentinel value read from user
while ( grade != -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;
```

- This would cause an infinite loop in the program if the user did not input `-1` for the first grade (in line 57).

3.9 Formulating Algorithms: Sentinel-Controlled Repetition

- An averaging calculation is likely to produce 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 store numbers with larger magnitude and finer precision
  - Approximately double the precision of `float` variables.
- Recall that dividing two integers results in integer division, in which any fractional part is lost (i.e., truncated).
  - Even though the variable `average` is declared as the type `double` to capture the fractional result of our calculation.

3.9 Formulating Algorithms: Sentinel-Controlled Repetition

- In the following statement the division occurs *first*—the fractional part is lost before it's assigned to `average`:

```
average = total / gradeCounter;
```

- To perform a floating-point calculation with integers, create temporary floating-point values.
- The unary cast operation `static_cast<double>(total)` creates a *temporary* floating-point copy of `total`.
  - Known as explicit conversion.
  - The value stored in `total` is still an integer.
- Cast operators are available for use with every data type and with class types as well.

3.9 Formulating Algorithms: Sentinel-Controlled Repetition

- The call to `setprecision(2)`prints the floating-point values with two digits of precision (e.g., 92.37).
  - Programs that use these must include the header `<iomanip>`.
  - The printed value is rounded to the required number of decimal positions, although the value in memory re-mains unaltered.
- If the precision is not specified, floating-point values are normally output with six digits of precision.
- The manipulator `fixed` forces the floating-point values to be printed in fixed-point format, not scientific notation.
  - Ex: 92.37 vs. 9.237E+1
- Fixed-point formatting also forces the decimal point and trailing zeros to print, such as 88.00.

3.9 Formulating Algorithms: Sentinel-Controlled Repetition

- How to show a floating-point number with scientific number? (Discussed in Chap15)

```
1  // Fig. 15.18: Fig15_18.cpp
2  // Displaying floating-point values in system default,
3  // scientific and fixed formats.
4  #include <iostream>
5  using namespace std;
6
7  int main()
8  {
9     double x = 0.001234567;
10    double y = 1.946e9;
11
12    // display x and y in default format
13    cout << "Displayed in default format:" << endl
14       << x << '\t' << y << endl;
15
16    // display x and y in scientific format
17    cout << "\nDisplayed in scientific format:" << endl
18       << scientific << x << '\t' << y << endl;
19
20    // display x and y in fixed format
21    cout << "\nDisplayed in fixed format:" << endl
22       << fixed << x << '\t' << y << endl;
23 } // end main
```

Fig. 15.18 | Floating-point values displayed in default, scientific and fixed formats. (Part 1 of 2.)

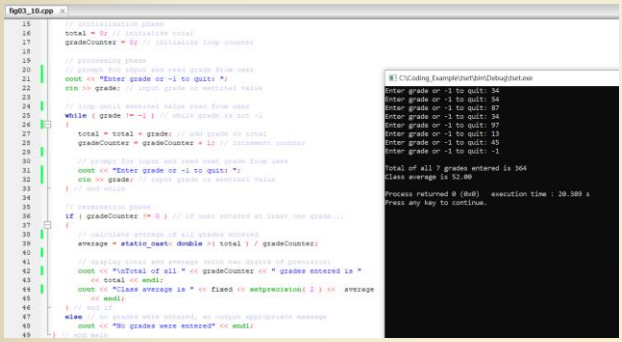
3.9 Formulating Algorithms: Sentinel-Controlled Repetition

```
Displayed in default format:
0.00123457      1.946e+009

Displayed in scientific format:
1.234567e-003   1.946000e+009

Displayed in fixed format:
0.001235        1946000000.000000
```

Fig. 15.18 | Floating-point values displayed in default, scientific and fixed formats. (Part 2 of 2.)

3.9 Formulating Algorithms: Sentinel-Controlled Repetition

- Can we change order of setprecision(2) and fixed?
  - ➔ YES, the program still works!



3.10 Formulating Algorithms: Nested Control Statements

- You've 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 exam results as follows:
  1. Display the prompting message "Enter result" and input each test result (i.e., a 1 or a 2).
  2. Count the number of test results of each type.
  3. Display a summary indicating the numbers of passed students and failed ones.
  4. .If more than eight students passed the exam, print the message "Bonus to instructor!"

3.10 Formulating Algorithms: Nested Control Statements

- 10 exam results, so counter-controlled looping is appropriate.
- Two counters are needed to record the passes and failures
- Another counter is used to control the looping process.
- Inside the loop, an `if…else` statement determines whether each result is a pass or a failure and increment different counter.
  - Nested control
- The refined pseudocode statement is as follows.

While student counter is less than or equal to 10
Prompt the user to enter the next exam result
Input the next exam result

If the student passed
Add one to passes
Else
Add one to failures

Add one to student counter

3.10 Formulating Algorithms: Nested Control Statements

```
1  // Fig. 3.12: fig03_12.cpp
2  // Examination-results problem: Nested control statements.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     // initializing variables in declarations
9     int passes = 0; // number of passes
10    int failures = 0; // number of failures
11    int studentCounter = 1; // student counter
12    int result; // one exam result (1 = pass, 2 = fail)
13
14    // process 10 students using counter-controlled loop
15    while ( studentCounter <= 10 )
16    {
17       // prompt user for input and obtain value from user
18       cout << "Enter result (1 = pass, 2 = fail): ";
19       cin >> result; // input result
20
```

Fig. 3.12 | Examination-results problem: Nested control statements. (Part 1 of 4.)

3.10 Formulating Algorithms: Nested Control Statements

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

3.10 Formulating Algorithms: Nested Control Statements

```
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
Bonus to instructor!
```

Fig. 3.12 | Examination-results problem: Nested control statements. (Part 3 of 4.)

3.10 Formulating Algorithms: Nested Control Statements

```
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
```

Fig. 3.12 | Examination-results problem: Nested control statements. (Part 4 of 4.)

3.10 Formulating Algorithms: Nested Control Statements

- C++ allows variable initialization to be incorporated into declarations.
- The `if`…`else` statement (lines 22–25) for processing each result is nested in the `while` statement.
- The `if` statement in lines 35–36 determines whether more than eight students passed the exam and, if so, outputs the message `"Bonus to instructor!"`.

3.9 Formulating Algorithms: Sentinel-Controlled Repetition

- Notice the block in the `while` loop in Fig. 3.10.
- Without the braces, the last three statements in the body of the loop would fall outside the loop, causing the computer to interpret this code incorrectly, as follows:

```
// loop until sentinel value read from user
while ( grade != -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;
```

- This would cause an infinite loop in the program if the user did not input `-1` for the first grade (in line 57).

3.11 Assignment Operators

- C++ provides several assignment operators for abbreviating assignment expressions of the form
  - *variable = variable operator expression;*
- If the same variable appears on both sides of the assignment operator and operator is one of the binary operators `+`, `-`, `*`, `/`, or `%`, it can be written in the form
  - *variable operator= expression;*

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

3.12 Increment and Decrement Operators

- C++ also provides two unary operators for adding 1 to or subtracting 1 from the value of a numeric variable.
  - The increment operator, ++, and the decrement operator, --.
- Unlike binary operators, these unary operators should be placed next to their operands, with no intervening spaces.

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.

3.12 Increment and Decrement Operators

- When you increment (++) a variable in a statement by itself, the preincrement (++c) and postincrement (c++) forms have the same effect.
  - Similarly for predecrementing (--c) and post-decrementing (c--)
- While appearing in a larger expression, preincrementing and postincrementing a variable have different effects.
  - ++c: increment it first **before** using its value for later operations
    → use the new value
  - c++: increment it **after** using its value for later operations
    → use the old value

3.12 Increment and Decrement Operators

```
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     int c;
9
10    // demonstrate postincrement
11    c = 5; // assign 5 to c
12    cout << c << endl; // print 5
13    cout << c++ << endl; // print 5 then postincrement
14    cout << c << endl; // print 6
15
16    cout << endl; // skip a line
17
18    // demonstrate preincrement
19    c = 5; // assign 5 to c
20    cout << c << endl; // print 5
21    cout << ++c << endl; // preincrement then print 6
22    cout << c << endl; // print 6
23 } // end main
```

```
5
5
6

5
6
6
```


3.12 Increment and Decrement Operators

Operators						Associativity	Type
::						left to right	scope resolution
()						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. 3.16 | Operator precedence for the operators encountered so far in the text.

Summary

- Algorithm and Pseudocode
- Control structure
- if / if…else / while statement
- Counter-controlled loop
- Sentinel-controlled loop
- Nested control loop
- Assignment/increment/decrement operators