Control Statements: Part 1

C++ How to Program, 7/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.

4.4 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 that the next statement to execute may be other than the next one in sequence.
  - Called transfer of control.
- All programs could be written in terms of only three control structures
  - the sequence structure
  - the selection structure and
  - the repetition structure
- When we introduce C++'s implementations of control structures, we'll refer to them in the terminology of the C++ standard document as "control statements."

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 must not 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
typeid	typename	using	virtual	wchar_t
xor	xor_eq			

Fig. 4.3 | C++ keywords.

4.5 `if` Selection Statement (cont.)

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

  -
    ```
    if ( grade >= 60)
       cout << "Passed";
    ```
- Figure 4.4 illustrates the single-selection `if` statement.

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"
```

- 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";
```

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.

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 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 of the code to the right, which can force lines to wrap.

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.)

- 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.

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;
    ```

4.8 Formulating Algorithms: Counter-Controlled Repetition

- Consider the following 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.
- 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.

```
#include<iostream>
using namespace std;
int main()
{
    cout << “welcome to the grade book for \nCS101 C++ Programming” << endl << endl;
    int total = 0;
    int gradeCounter = 1;
    int grade;
    while(gradeCounter++ <= 10)
    {
        cout << "Enter grade:    ";
        cin >> grade;
        total += grade;
    }
    double avg = (double)total / gradeCounter;
    cout << “Total of all 10 grades is “ << total << endl << “Class average is " << avg;
    return 0;
}
```

```
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
```

Fig. 4.10 | Class average problem using counter-controlled repetition: Creating an object of class `GradeBook` (Fig. 4.8–Fig. 4.9) and invoking its `determineClassAverage` member function. (Part 2 of 2.)

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

- In the second refinement, we commit to specific variables.
- Only the variables *total* and *counter* need to be initialized before they're used.
- The variables *average* and *grade (*for the calculated average and the user input, respectively) need not be initialized, because their values will be replaced as they're calculated or input.

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    cout << "Welcome to the grade book for \nCS101 C++ Programming" << endl
         << endl;
    int total = 0;
    int gradeCounter = 0;
    int grade = 0;
    cout << "Enter grade (-1 for quit) :    ";
    cin >> grade;
    while (grade != -1)
    {
        total += grade;
        gradeCounter++;
        cout << "Enter grade (-1 for quit) :    ";
        cin >> grade;
    }
    double avg = (double)total / gradeCounter;
    cout << "Total of all " << gradeCounter << " grades is " << total << endl
         << "Class average is " << fixed << setw(10) << setprecision(2) << avg;
    return 0;
}
```

```
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 32-bit 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 32-bit 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 constants.
- Floating-point numbers often arise as a result of division.

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

- The call to setprecision (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`), 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>`.

4.10 Formulating Algorithms: Nested Control Statements

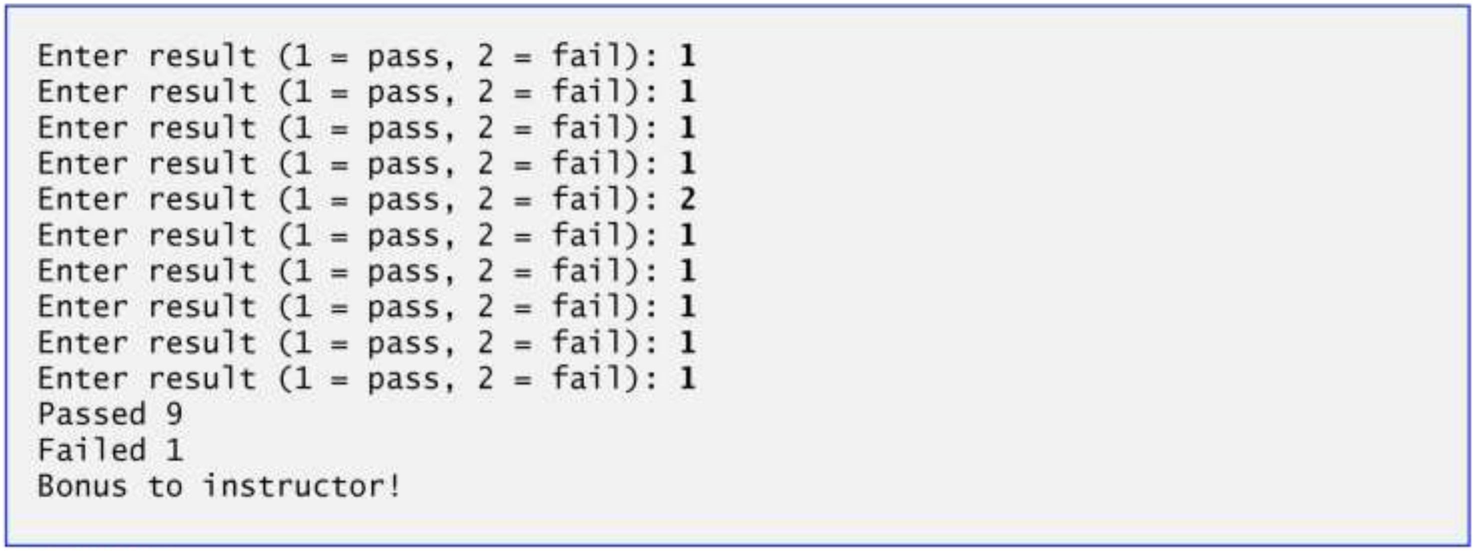
- Consider the following problem statement:
  - A college offers a course that prepares students for the state licensing exam for real es-tate brokers. Last year, ten of the students who completed this course took the exam. The college wants to know how well its students did on the exam. You’ve been asked to write a program to summarize the results. 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 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 “Bonus to instructor!”

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

int main()
{
   // initializing variables in declarations
   int passes = 0; // number of passes
   int failures = 0; // number of failures
   int studentCounter = 1; // student counter
   int result; // one exam result (1 = pass, 2 = fail)

   // process 10 students using counter-controlled loop
   while ( studentCounter <= 10 )
   {
      // prompt user for input and obtain value from user
      cout << "Enter result (1 = pass, 2 = fail): ";
      cin >> result; // input result

```

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

```
         // if...else nested in while
         if ( result == 1 )          // if result is 1,
            passes = passes + 1;     // increment passes;
         else                        // else result is not 1, so
            failures = failures + 1; // increment failures

         // increment studentCounter so loop eventually terminates
         studentCounter = studentCounter + 1;
      } // end while

      // termination phase; display number of passes and failures
      cout << "Passed " << passes << "\nFailed " << failures << endl;

      // determine whether more than eight students passed
      if ( passes > 8 )
         cout << "Bonus to instructor!" << endl;
   } // end main
```

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



```
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!
```

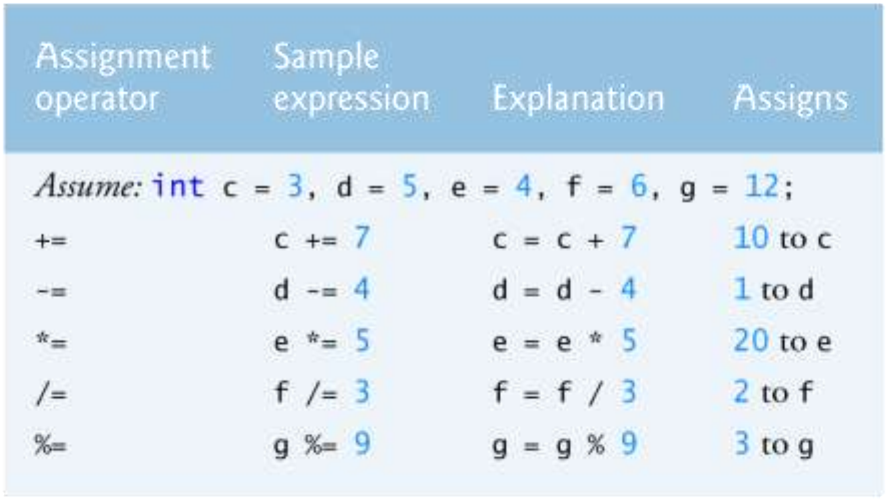
Fig. 4.16 | Examination-results problem: Nested control statements. (Part 3 of 4.)

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

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.

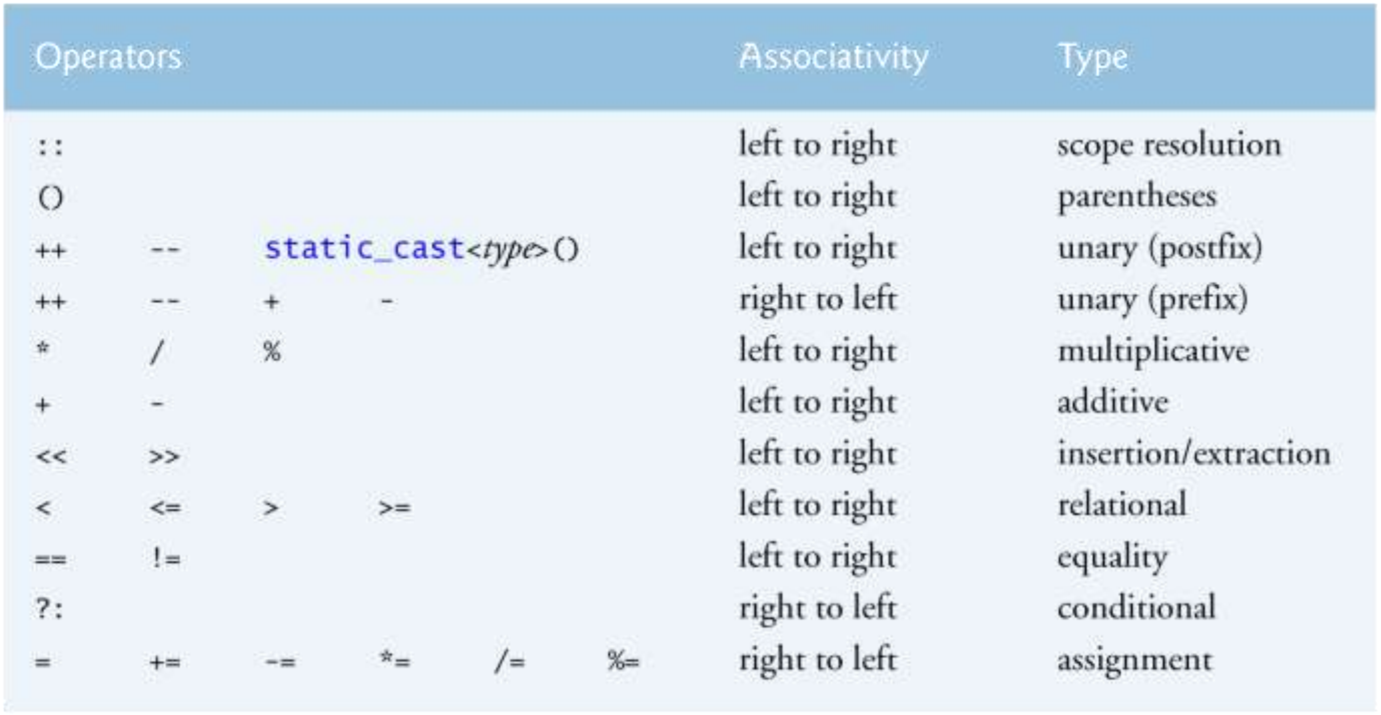
Fig. 4.18 | Increment and decrement operators.

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

int main()
{
   int c;

   // demonstrate postincrement
   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.)



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