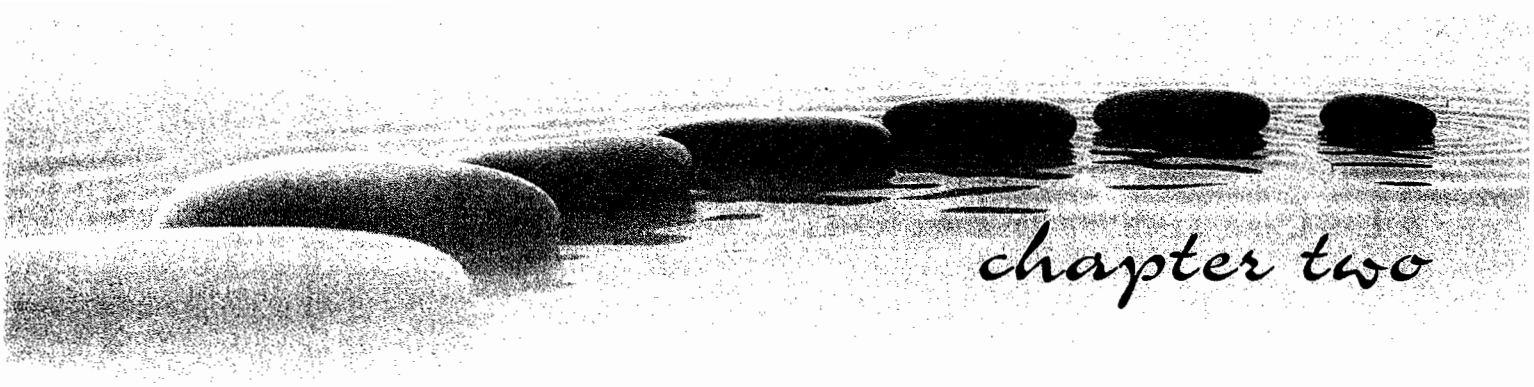


Overview of C++

Chapter Objectives

- To become familiar with the general form of a C++ program and the basic elements in a program
- To learn how to include standard library files in a program
- To appreciate the importance of writing comments in a program
- To understand the use of data types and the differences between the numeric types for real numbers and integers
- To be able to declare variables with and without initial values
- To understand how to write assignment statements to change the values of variables
- To learn how C++ evaluates arithmetic expressions and how to write them
- To learn how to read data values into a program and to display results
- To learn how to use files for input/output
- To understand the differences between syntax errors, run-time errors, and logic errors, and how to avoid them and to correct them.

THIS CHAPTER INTRODUCES C++—a high-level programming language developed in the mid-1980s by Bjarne Stroustrup at AT&T's Bell Laboratories. C++ is widely used in industry because it supports object-oriented programming and incorporates all of the features of the popular C programming language. Throughout this book, we will follow the standard for the C++ language approved in 1998. Our focus in this chapter will be on the C++ statements for entering data, performing simple computations, and displaying results.

2.1 C++ Language Elements

One advantage of C++ is that it allows programmers to write programs that resemble everyday English. Even though you don't yet know how to write your own programs, you can probably read and understand parts of the program in Listing 1.2, which is repeated in Listing 2.1. In this section, we will describe the different language elements in this program and explain their syntax.

Comments

The first two lines in the program shown in Listing 2.1

```
// miles.cpp
// Converts distances from miles to kilometers.
```

comment
Text in a program that makes the code more understandable to the human reader but is ignored—that is, not translated—by the compiler.

start with the symbol pair `//` that indicates that these lines are **comments**—text used to clarify the program to the person who is reading it but ignored by the C++ compiler, which does not attempt to translate comments into machine language. We show all comments in italics; your C++ system may not. The first comment line gives the name of the file containing this program (`miles.cpp`), and the second describes in English what the program does. (You can download all program files from the Web site for this book—see the Preface.) We discuss the use of comments in Section 2.5.

The C++ syntax for comments is described in the syntax display that follows. We use this stylistic form throughout the text to describe C++ language features.

Listing 2.1 Converting miles to kilometers

```
// miles.cpp
// Converts distance in miles to kilometers.

#include <iostream>
using namespace std;

int main()           // start of main function
{

      const float KM_PER_MILE = 1.609;  // 1.609 km in a mile
      float miles,                      // input: distance in miles
            kms;                        // output: distance in kilometers
```

(continued)

Listing 2.1 Converting miles to kilometers (continued)

```
    // Get the distance in miles.
    cout << "Enter the distance in miles: ";
    cin >> miles;

    // Convert the distance to kilometers.

    kms = KM_PER_MILE * miles;

    // Display the distance in kilometers.

    cout << "The distance in kilometers is " << kms << endl;

    return 0;                                  // Exit main function

}
```

Comment

Form:
```
// comment
/* comment */
```

Example:
```
// This is a comment
/* and so is this */
```

Interpretation: A double slash indicates the start of a comment. Alternatively, the symbol pair `/*` may be used to mark the beginning of a comment that is terminated by the symbol pair `*/`. Comments are listed with the program but are otherwise ignored by the C++ compiler.

Compiler Directive `#include`

The first line of Listing 2.1 after the comments—

```
#include <iostream>
```

is a **compiler directive**, a statement that is processed before the program is translated. This directive causes the statements from the header file for `class iostream` to be inserted at this spot in our program. A **header file** describes the data and operations in a class so that the compiler can check that the class is used correctly.

The left and right angle brackets < > indicate that `class iostream` is part of the C++ **standard library**, a component that contains a number of

compiler directive
A C++ program line beginning with # that is processed by the compiler before translation begins.

header file
A file that describes the data and operations in a class.

standard library
A collection of C++ predefined classes provided with a C++ system.

predefined classes such as `iostream` that provide important functionality for the C++ language. For example, class `iostream` has operators that enable a program to read input data typed at the keyboard and to display information on the screen. Libraries enable us to *reuse* C++ code modules that have already been written and tested, saving us from the burden of creating this code ourselves. There may be many `#include` statements, and possibly other compiler directives, in each program that you write.

Compiler Directive `#include`

Form: `#include <filename>`

Example: `#include <iostream>`

Interpretation: Prior to translation, this line is replaced by the C++ standard library file header named inside the angle brackets. Usually compiler directives, such as `#include`, appear at the beginning of a program, but they will work as long as they appear before they are used.

Note: A compiler directive should not end with a semicolon.

Namespace `std`

The line

```
using namespace std;
```

indicates that we will be using objects that are named in a special region of the C++ compiler called `namespace std` (short for standard). Because the C++ standard library is defined in the standard namespace, this line will appear in all our programs. The using statement ends with a semicolon.

`using namespace`

Form: `using namespace` *region*`;`

Example: `using namespace std;`

Interpretation: This line indicates that our program uses objects defined in the namespace specified by *region*. This line should follow the `#include` lines.

Function `main`

The line

```
int main()        // start of function main
```

indicates the start of function `main`. A **function** is a collection of related statements that perform a specific operation. For example, the `main` function in Listing 2.1 is the program for converting from kilometers to miles. The word `int` indicates that the `main` function should return an integer value to the operating system. The empty parentheses after `main` indicate that no special information is passed to this function by the operating system. We added a comment (beginning with `//`) to the end of this line to clarify its meaning to you, the program reader. The body of function `main`, or any other function, is enclosed in curly braces `{ }` and it consists of the rest of the lines in Listing 2.1. Execution of a C++ program always begins with the first line in the body of function `main`.

function
A collection of related statements that perform a specific operation. These statements are executed as a unit.

`main` function definition

Form:
```
int main()
{
    function body
}
```

Example:
```
int main()
{
    cout << "Enjoy C++" << endl;
    return 0;
}
```

Interpretation: A program begins its execution with function `main`. The body of the function is enclosed in curly braces. The function body should end with the line

```
return 0;
```

which causes function `main` to return a value of 0 to the operating system when it finishes execution.

Declaration Statements

A function body consists of two kinds of statements: *declaration statements* and *executable statements*. The declarations tell the compiler what data are needed in the function. For example, the declaration statements

```
const float KM_PER_MILE = 1.609;  // 1.609 km in a mile
float miles,        // input: distance in miles
      kms;          // output: distance in kilometers
```

tell the compiler that the function main needs three data items named `KM_PER_MILE`, `miles`, and `kms`. `KM_PER_MILE` is the conversion constant

1.609. To create this part of the function, the programmer uses the problem data requirements identified during problem analysis. We describe the syntax of declaration statements in Section 2.3.

Executable Statements

The executable statements cause some action to take place when a program is executed. Executable statements that begin with cout (pronounced c-out) display output on the screen. The first such line

```
cout << "Enter the distance in miles: ";
```

displays the prompt (in black type) in the first line of the sample execution repeated below:

```
Enter the distance in miles: 10.0
```

cout (pronounced c-out) refers to the standard output device, the screen. The symbol pair << is the insertion operator, so named because it inserts characters into the text displayed on the screen. In this case, it inserts the characters shown in quotes, which ask the user to type in a distance in miles. The program user types in the number 10.0. The next statement in function main

```
cin >> miles;
```

reads the data value (10.0) into the variable miles. cin (pronounced c-in) refers to the standard input device, the keyboard. The symbol pair >> is the extraction operator, so named because it "extracts" characters typed at the keyboard and stores them in memory. The statement

```
kms = KM_PER_MILE * miles;
```

computes the equivalent distance in kilometers by multiplying miles by the number (1.609) stored in KM_PER_MILE; the result (16.09) is stored in memory cell kms.

Then, the statement

```
cout << "The distance in kilometers is " << kms << endl;
```

displays a ***string*** (the characters enclosed in double quotes) followed by the value contained in kms. The word endl stands for "endline" and ends the output line shown next:

```
The distance in kilometers is 16.09
```

Finally, the line

```
return 0;                    // Exit main function
```

causes the function exit. The function returns a value of `0` to the operating system which indicates that it exited normally. We describe these executable statements in Section 2.4.

EXERCISES FOR SECTION 2.1

Self-Check

1. Explain what is wrong with the following declaration statement and correct it.

```
float miles,     /* input: distance in miles
      kms;          output: distance in kilometers */
```

2. Explain what is wrong with the comments below and correct them.

   a. `/* This is a comment? /*`

   b. `/* How about this one /* it seems like a comment */ doesn't it? */`

3. Indicate which of the following are translated into machine language: compiler directives, comments, `using namespace` statement, variable declarations, executable statements.

4. What is the purpose of the `#include`? Of the using namespace statement? Which does not end with a semicolon?

5. What are the symbols >> and << ?

6. Why do you think C++ uses `cin >> x` instead of `cin << x` ?

7. Correct the errors in the statement sequence below:

```
cout >> "weight";
cin << "Enter weight in pounds: ";
```

Programming

1. Write the declarations for a program that converts a weight in pounds to a weight in kilograms.

2. Write the C++ statements to ask for and read the weight in pounds.

2.2 Reserved Words and Identifiers

Reserved Words

reserved word (keyword)
A word that has a specific meaning in C++.

Each line of the program in Listing 2.1 contains a number of different syntactic elements, such as reserved words, identifiers, and special character symbols (for example, `//`). The **reserved words** (or **keywords**) have a specific meaning in C++, and they cannot be used for other purposes. Table 2.1 lists the reserved words used in Listing 2.1.

Table 2.1 Reserved Words in Listing 2.1

Reserved Words	Meaning
`const`	Constant; indicates a data element whose value cannot change
`float`	Floating point; indicates that a data item is a real number
`include`	Preprocessor directive; used to insert a library file
`int`	Integer; indicates that the main function returns an integer value
`namespace`	Region where program elements are defined
`return`	Causes a return from a function to the unit that activates it
`using`	Indicates that a program is using elements from a particular namespace

Identifiers

Table 2.2 lists the identifiers from Listing 2.1. We use identifiers to name the data elements and objects manipulated by a program. The identifiers `cin`, `cout`, and `std` are predefined in C++, but the others were chosen by us. You have quite a bit of freedom in selecting the identifiers that you use as long as you follow these syntactic rules:

1. An identifier must always begin with a letter or underscore symbol (the latter is not recommended).

Table 2.2 Identifiers in Listing 2.1

Identifier	Use
`cin`	C++ name for standard input stream
`cout`	C++ name for standard output stream
`km`	Data element used for storing the distance in kilometers
`KM_PER_MILE`	Data element used for storing the conversion constant
`miles`	Data element used for storing the distance in miles
`std`	C++ name for the standard namespace

2. An identifier can consist only of letters, digits, and underscores.
3. You cannot use a C++ reserved word as an identifier.

The following are some valid identifiers.

```
letter1, Letter1, letter2, inches, cent, centPerInch,
cent_per_inch, hello
```

Table 2.3 shows some invalid identifiers.

Uppercase and Lowercase Letters

The C++ programmer must use uppercase and lowercase letters with care, because the compiler considers such usage significant. The names `Rate`, `rate`, and `RATE` are viewed by the compiler as *different* identifiers. Adopting a consistent pattern in the way you use uppercase and lowercase letters will be helpful to the readers of your programs. All reserved words in C++ use only lowercase letters.

PROGRAM STYLE Choosing Identifier Names

We discuss program style throughout the text in displays like this one. A program that "looks good" is easier to read and understand than one that is sloppy. Most programs will be examined or studied by someone other than the original programmers. In industry, programmers spend considerably more time on program maintenance (that is, updating and modifying the program) than they do on its original design or coding. A program that is neatly stated and whose meaning is clear makes everyone's job simpler.

Pick a meaningful name for a user-defined identifier, so its use is immediately clear. For example, the identifier `salary` would be a good name for a memory cell used to store a person's salary, whereas the identifier `s` or `bagel` would be a bad choice.

If an identifier consists of two or more words, C programmers place the underscore character (_) between words to improve the readability of the

Table 2.3 Invalid Identifiers

Invalid Identifier	Reason Invalid
`1Letter`	Begins with a number
`float`	Reserved word
`const`	Reserved word
`Two*Four`	Character * not allowed
`Joe's`	Character ' not allowed
`two-dimensional`	Underscore allowed but not hyphen

name (`dollars_per_hour` rather than `dollarsperhour`). C programmers write constants in all uppercase letters (for example, `KM_PER_MILE`).

In this book, we will follow the C style for constants, but we will use a different naming convention for multiple-word variables called "Hungarian notation." Instead of using underscore characters, we will capitalize the first letter of each word—except for the first word. Therefore, we prefer the identifier `dollarsPerHour` to `dollars_per_hour`. Your instructor may recommend a different convention. Whichever convention you choose, make sure you use it consistently.

Choose identifiers long enough to convey your meaning, but avoid excessively long names because you are more likely to make a typing error in a longer name. For example, use the shorter identifier `lbsPerSqIn` instead of the longer identifier `poundsPerSquareInch`.

If you mistype an identifier, the compiler may detect this as a syntax error and display an *undefined identifier* error message during program translation. But if you mistype a name so that the identifier looks like another, the compiler may not detect your error. For this reason and to avoid confusion, do not choose names that are similar to each other. Especially avoid selecting two names that are different only in their use of uppercase and lowercase letters (such as `LARGE` and `large`), or in the presence or absence of an underscore (`xcoord` and `x_coord`).

EXERCISES FOR SECTION 2.2

Self-Check

1. Can reserved words be used as identifiers?
2. Why is it important to use consistent programming style?
3. Indicate which of the symbols below are C++ reserved words, or valid or invalid identifiers.

```
float       cout        Bill        "hello"  rate    start
var         xyz123      123xyz      'a'      include
return
x=y*z       Prog#2      thisIsLong
so_is_this  hyphen-ate  under_score
```

4. Explain the difference between Hungarian notation and C-style notation.
5. Why should e (2.7182818) be defined as a constant instead of a variable?

Programming

1. Write a multiline comment with your name, your instructor's name, and the course name. Use both methods for multiline comments.
2. (Continuation of programming Exercises 1 and 2 from Section 2.1) Write the C++ statement to multiply the weight in pounds by the conversion constant and store the resulting weight in kilograms. Write the C++ statement(s) to display the weight in kilograms.

2.3 Data Types and Declarations

Data Types

A **data type** is a set of values and operations that can be performed on those values. (A value that appears in a C++ program line is called a **literal**. In C++, some data types are predefined in the language and some are defined in class libraries. In this section, we describe four predefined data types and one that is defined in the standard library.

data type
A set of values and operations that can be performed on those values.
literal
A value that appears in a C++ program line.

Data Type `int`

In mathematics, integers are positive or negative whole numbers. Examples are 5, –52, and 343,222 (written in C++ as 343222). A number without a sign is assumed to be positive. C++ has three data types to represent integers—`short`, `int`, and `long`. Each data type is considered to be an abstraction (or model) for the integers because it represents a subset of all the integers. Each integer is represented as a binary number. The number of bytes allocated for storing an integer value limits the range of integers that can be represented. The difference between these three types is that some compilers use more bytes to represent `int` values than `short` values and more bytes to represent `long` values than `int` values. In these compilers, you can represent larger integers with type `int` than you can with type `short`, and larger integers with type `long` that you can with type `int`.

We will use the data type `int` throughout this text. The predefined constants `INT_MIN` and `INT_MAX` represent the smallest and largest values of type int, respectively. Use the line

```
cout << INT_MIN << "through" << INT_MAX << endl;
```

in a program to show the range of integer values for your computer.

In C++, you write integers (and all other numbers) without commas. Some valid integers are

```
-10500      435      15      -25
```

We can read and display integer values and can perform the common arithmetic operations (add, subtract, multiply, and divide) on type `int` values.

Data Type `float`

A real number has an integral part and a fractional part that are separated by a decimal point. Examples are 2.5, 3.66666666, –.000034, and 5.0. C++ has three data types to represent real numbers—`float`, `double`, and `long double`. Just as it is for integers, the number of bytes used to represent each of these types is different, so the range and precision of real numbers that can be represented varies. We will use type `float` in this text.

We can use scientific notation to represent very large and very small values. In normal scientific notation, the real number 1.23×10^5 is equivalent to 123000.0, where the exponent `5` means "move the decimal point `5` places to the right." In C++ scientific notation, we write this number as `1.23e5` or `1.23e+5`. If the exponent has a minus sign, the decimal point is moved to the left (for example, `0.34e-4` is equivalent to `0.000034`). Table 2.4 shows examples of both valid and invalid real numbers in C++.

fixed-point format
The internal storage form for an integer.

floating-point format
The internal storage form for a real number.

mantissa
A binary fraction between 0.5 and 1.0 used in floating-point format.

C++ uses different formats (see Figure 2.1) to represent real numbers (floating-point format) and integers (fixed-point format). In **fixed-point format**, the binary number corresponding to an integer value is stored directly in memory. In **floating point-format**, a positive real number is represented as a **mantissa** (a binary fraction between 0.5 and 1.0) and an integer exponent (power of 2) analogous to scientific notation. Therefore, the

Table 2.4 Valid and Invalid Real Numbers

Valid	Invalid
`3.14159`	`-15e-0.3` (`0.3` invalid exponent)
`0.005`	`12.5e.3` (`.3` invalid exponent)
`.12345`	`.123E3` (needs lowercase *e*)
`12345.0`	`e32` (doesn't start with a digit)
`16`	`a34e03` (doesn't start with a digit)
`15.0e-04` (value is `0.0015`)	
`2.345e2` (value is `234.5`)	
`1.15e-3` (value is `0.00115`)	
`12e+5` (value is `1200000.0`)	

Figure 2.1 Internal formats for integers and real numbers

integer 10 and the real number 10.0 are stored differently in memory, so they can't be used interchangeably. As with integers, we can read and display real numbers and perform the common arithmetic operations (add, subtract, multiply, and divide).

Data Type `bool`

The `bool` data type (named after mathematician George Boole who invented a two-valued algebra) has just two possible values: false and true. We can use this data type to represent conditional values so that a program can make decisions. For example, "if the first number is bigger than the second number, then switch their values." We discuss this data type in more detail in Chapter 4.

Data Type `char`

Data type `char` represents an individual character value—a letter, digit, or special symbol that can be typed at the keyboard. Each character is unique. For example, the character for the digit one (`'1'`) is distinct from the character for the letter "el" (`'l'`), even though they might look identical on a computer screen. Each type `char` literal is enclosed in apostrophes (single quotes) as shown below.

```
'A' 'z' '2' '9' '*' ':' '"' ' '
```

The next-to-last literal above represents the character "(a quotation mark); the last literal represents the blank character, which is typed by pressing the apostrophe key, the space bar, and the apostrophe key.

Each type `char` value is stored in a byte of memory. The digit character `'1'` has a different storage representation than the integer `1`.

Although a type `char` literal in a program requires apostrophes, you don't need them for a type `char` data value. When entering the letter `z` as a character data item to be read by a program, press just the `z` key.

Characters are used primarily in strings, a topic we will introduce shortly. A special set of symbols, called an **escape sequence**, allows you to do simple textual control. Each escape sequence is written as a type `char` literal consisting of a backslash `\` followed by another character. Table 2.5 shows some

escape sequence
A two-character sequence beginning with a backslash \ that represents a special character.

Table 2.5 Escape Sequences

Escape Sequence	Meaning
`'\n'`	linefeed
`'\b'`	backspace
`'\r'`	carriage return
`'\t'`	tab
`'\"'`	double quote
`'\'`	single quote
`'\f'`	formfeed
`'\\'`	backslash

common escape sequences. The first four escape sequences represent characters that control the appearance of text that is displayed. The last row shows that C++ uses two backslash characters to represent the backslash character.

`string` Class

Besides these built-in data types, C++ provides definitions for a number of other data types in its standard library. The ability to extend C++ by using data types defined in libraries is one of the primary advantages of C++ and object-oriented languages in general. We will use two of these classes, `string` and `iostream`, in much of our programming. We discuss the `string` class next and some features of class `iostream` in Section 2.4.

string literal
A sequence of characters enclosed in quotation marks displayed as prompts or as labels for a program's results.

A **string literal** (or string) is a sequence of characters enclosed in quotation marks. String literals are often displayed as prompts or as labels for a program's results ("the value of x is"). The next line contains five string literals.

```
"A"   "1234"   "true"   "the value of x is "
"Enter speed in mph: "
```

Note that the string `"A"` is stored differently from the character `'A'`. Similarly, the string `"1234"` is not stored the same way as the integer `1234` and cannot be used in a numerical computation. The string `"true"` is also stored differently from the `bool` value `true`. A string can contain any number of characters, but all must be on the same line. Programmers often use string literals in lines that begin with `cout`. C++ displays the characters in the string literal exactly as they appear without the enclosing quotes.

In C++, strings can be read, stored, compared, joined together, and taken apart. A program that performs any of these operations on string data must contain the compiler directive

```
#include <string>
```

before the main function definition. This directive is not needed if you only use strings as literals in lines that begin with `cout`.

Purpose of Data Types

Why have different data types? The reason is that they allow the compiler to know what operations are valid and how to represent a particular value in memory. For example, a whole number is stored differently in a type `float` variable than in a type `int` variable. Also, a type `float` value usually requires more memory space than an `int` value. A string will also take up more memory space than a `float` or `char` under most circumstances.

If you try to manipulate a value in memory in an incorrect way (for example, adding two `bool` values), the C++ compiler displays an error message telling you that this is an incorrect operation. Or if you try to store the wrong kind of value (such as a string in a data field that is type `int`), you get an error message. Detecting these errors keeps the computer from performing operations that don't make sense.

Declarations

The memory cells used for storing a program's input data and its computational results are called **variables** because the values stored in variables can change (and usually do) as the program executes. The **variable declarations** in a C++ program communicate to the compiler the names of all variables used in a program. They also tell the compiler what kind of information will be stored in each variable and how that information will be represented in memory. You must declare each variable before its first use in a function.

variable
A name associated with a memory cell whose contents can change during program execution.

variable declarations
Statements that communicate to the compiler the names of variables and the kind of information stored in each variable.

A variable declaration begins with an identifier (for example, `float`) that tells the C++ compiler the type of data (for example, a real number) stored in a particular variable. The declaration statement

```
float miles,        // input: distance in miles
      kms;          // output: distance in kilometers
```

gives the names of two variables (`miles, kms`) used to store real numbers. Note that C++ ignores the comments on the right of each line describing the use of each variable. The two lines above make up a single C++ declaration statement.

The statement

```
string lastName;
```

declares an identifier, `lastName`, for storing a string. Since `string` is a class, it is more accurate to refer to `lastName` as an object (or class instance) instead of a variable.

Variable and Object Declarations

Form: *type identifier-list;*

Examples:
```
float x, y;
int me, you;
float week = 40.0;
string flower = "rose";
```

Interpretation: Storage is allocated for each identifier in the *identifier-list*. The *type* of data (`float`, `int`, etc.) to be stored in each variable or object is specified. Commas are used to separate the identifiers in the *identifier-list*. The last two lines show how to store an initial value (`40.0`, `"rose"`) in a variable or object. This value can be changed during program execution. A semicolon appears at the end of each declaration.

Constant Declarations

constant
A memory cell whose contents cannot change.

constant declaration
A statement that communicates to the compiler the name of a constant and the value associated with it.

We often want to associate names with special program **constants**—memory cells whose values cannot change. This association may be done using a **constant declaration**. The constant declaration

```
const float KM_PER_MILE = 1.609; // 1.609 km in a mile
```

specifies that the identifier `KM_PER_MILE` will be associated with the program constant 1.609. Because the identifier `KM_PER_MILE` is a constant, C++ will not allow you to change its value. You should consistently use `KM_PER_MILE` instead of the value 1.609 in the program. This makes it easier to read and understand the program. Make sure you only associate constant identifiers with data values that should never change (for example, the

Constant Declaration

Form: `const` *type constant-identifier = value;*

Example:
```
const float PI = 3.14159;
```

Interpretation: The specified *value* is associated with the *constant-identifier*. This *value* cannot be changed at any time by the program. A semicolon appears at the end of each declaration.

Note: By convention, we place constant declarations before any variable or object declarations in a C++ function.

number of kilometers in a mile is always 1.609). You should never declare an input data item or output result as a constant.

EXAMPLE 2.1

Listing 2.2 contains a C++ program that reads character and string data and displays a personalized message to the program user.

The line starting with `char` declares two variables (`letter1, letter2`) used to store the first two initials of the name of the program user. The line beginning with `string` declares an object `lastName` used to store the last name of the user. The instruction

```
cin >> letter1 >> letter2 >> lastName;
```

reads the two letters, `E` and `B`, and the last name, `Koffman`, all typed by the program user, and stores them in the three variables listed (`E` in `letter1`, `B` in `letter2`, and `Koffman` in `lastName`). The next line

```
cout << "Hello " << letter1 << ". " << letter2 << ". "
     << lastName << "!";
```

displays `Hello E. B. Koffman!`. The two string literals `". "` cause a period and one blank space to be displayed after each initial. Finally, the last `cout` line displays the rest of the second line shown in the program output.

Listing 2.2 Printing a welcoming message

```
// File: hello.cpp
// Displays a user's name

#include <iostream>
#include <string>
using namespace std;

int main()
{
   char letter1, letter2;  // input and output: first two initials
   string lastName;        // input and output: last name

   // Enter letters and print message.
   cout << "Enter 2 initials and a last name:";
   cin >> letter1 >> letter2 >> lastName;
   cout << "Hello " << letter1 << "." << letter2 << "."
        << lastName << "!";
   cout << "We hope you enjoy studying C++." << endl;
   return 0;
}
```

(continued)

Listing 2.2 Printing a welcoming message (continued)

```
Enter 2 initials and a last name: EB Koffman
Hello E. B. Koffman! We hope you enjoy studying C++.
```

EXERCISES FOR SECTION 2.3

Self-Check

1. What is the difference between a predefined type and a class type? Which has data stores that are variables and which has data stores that are objects? Is `string` a predefined type or a class type?
2. List the four types of predefined data.
3. a. Write the following C++ scientific notation numbers in normal decimal notation:

   ```
   345E-4   3.456E+6   345.678E+3
   ```

   b. Write the following numbers in C++ scientific notation with one digit before the decimal point:

   ```
   5678   567.89   0.00567
   ```

4. Indicate which of the following literal values are legal in C++ and which are not. Identify the data type of each valid literal value.

   ```
   15    'XYZ'   '*'    $    25.123    15.    -999    .123    'x'
   "X"   '9'     '-5'     True     'True'
   ```

5. What would be the best variable type for the area of a circle in square inches? How about the number of cars passing through an intersection in an hour? Your name? The first letter of your last name?
6. Distinguish between character and integer types.
7. What is the difference between:

   ```
   char color = 'r';
   string colorS = "r";
   ```

8. What is the difference between these three declarations?

   ```
   string aPresident;
   string pres = "Obama";
   const string LAST_PRES = "Bush";
   ```

9. What is the difference between these three declarations?

```
string name;
string pres = "Obama";
const string FIRST_PRES = "Washington";
```

Programming

1. Write the variable declarations and a constant declaration for a program that calculates the area and circumference of a circle given its radius.
2. Write declarations for variables to store the number of seconds it takes to run the 50-yard dash, the number of siblings a person has, a person's street address, a person's grade point average (G.P.A.), and a person's gender (M or F).
3. Write the declarations for a program with variables for `firstName`, `lastName`, and `middleInitial`.

2.4 Executable Statements

The **executable statements** follow the declarations in a function. They are the C++ statements used to write or code the algorithm and its refinements. The C++ compiler translates the executable statements into machine language; the computer executes the machine language version of these statements when we run the program.

executable statement
A C++ statement that is translated into machine language instructions that are executed.

Programs in Memory

Before examining the executable statements in the miles-to-kilometers conversion program (Listing 2.1), let's see what computer memory looks like before and after that program executes. Figure 2.2(a) shows the program loaded into memory and the program memory area before the program executes. The question marks in memory cells `miles` and `kms` indicate that the values of these cells are undefined before program execution begins. During program execution, the data value 10.0 is copied from the input device into the variable `miles`. After the program executes, the variables are defined as shown in Figure 2.2(b). We will see why next.

Assignment Statements

An **assignment statement** stores a value or a computed result in a variable, and is used to perform most arithmetic operations in a program. The assignment statement

assignment statement
A statement used to store a value or a computed result in a variable.

```
kms = KM_PER_MILE * miles;
```

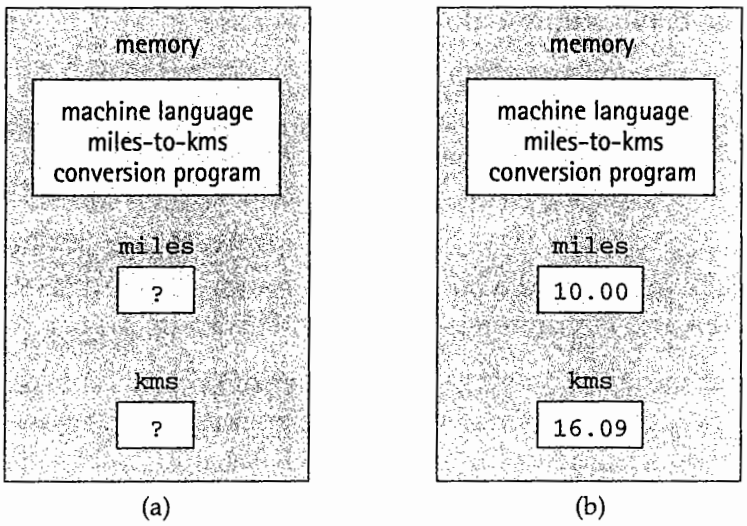


Figure 2.2 Memory (a) before and (b) after execution of a program

assigns a value to the variable `kms`. The value assigned is the result of the multiplication (* means multiply) of the constant `KM_PER_MILE (1.609)` by the variable `miles`. The memory cell for `miles` must contain valid information (in this case, a real number) before the assignment statement is executed. Figure 2.3 shows the contents of memory before and after the assignment statement executes; only the value of `kms` is changed.

In C++ the symbol = is the assignment operator. Read it as "becomes," "gets," or "takes the value of" rather than "equals" because it is *not* equivalent to the "equal sign" of mathematics. In mathematics, this symbol states a relationship between two values, but in C++ it represents an action to be carried out by the computer.

Section 2.6 continues the discussion of type `int` and `float` expressions and operators.

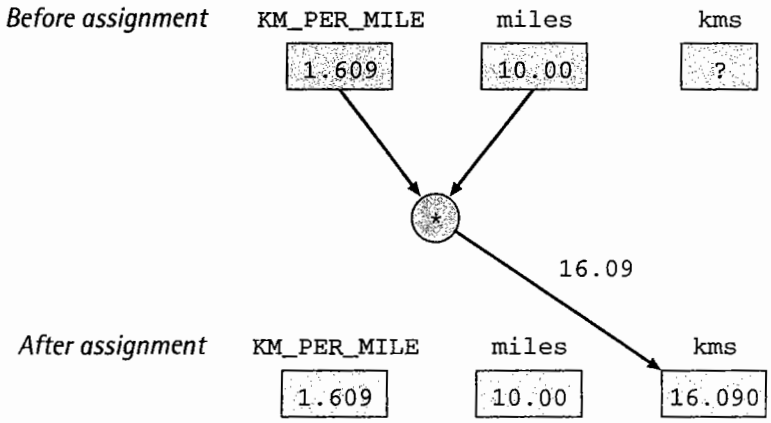


Figure 2.3 Effect of `kms = KM_PER_MILE * miles;`

EXAMPLE 2.2

In C++ you can write assignment statements of the form

```
sum = sum + item;
```

where the variable sum appears on both sides of the assignment operator. This is obviously not an algebraic equation, but it illustrates a common programming practice. This statement instructs the computer to add the current value of `sum` to the value of `item;` the result is then stored back into `sum`. The previous value of `sum` is destroyed in the process, as illustrated in Figure 2.4. The value of `item`, however, is unchanged.

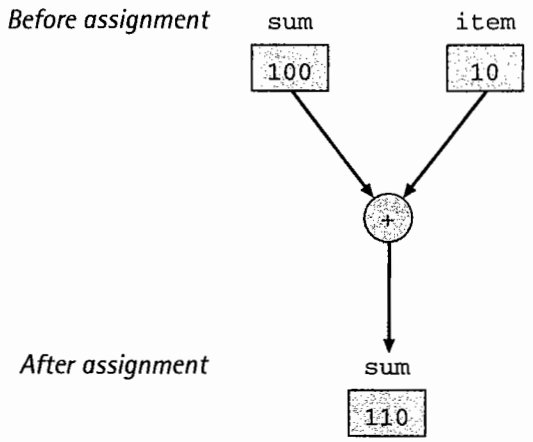


Figure 2.4 Effect of `sum = sum + item;`

EXAMPLE 2.3

You can also write assignment statements that assign the value of a single variable or constant to a variable. If `x` and `newX` are type `float` variables, the statement

```
newX = x;
```

copies the value of variable `x` into variable `newX`. The statement

```
newX = -x;
```

instructs the computer to get the value of `x`, negate that value, and store the result in `newX`. For example, if `x` is `3.5`, `newX` is `-3.5`. Neither of the assignment statements above changes the value of `x`.

Input/Output Operations

input operation
An instruction that reads data from an input device into memory.

Data can be stored in memory in two different ways: through assignment to a variable or by reading the data from an input device through an **input operation**. Use an input operation if you want the program to manipulate different data each time it executes.

As it executes, a program performs computations and assigns new values to variables and objects. These program results can be displayed on the screen through an **output operation**.

output operation
An instruction that displays information stored in memory.

Several C++ class libraries provide instructions for performing input and output. In this section, we will discuss how to use the input/output operations defined in the C++ class `iostream`.

stream
A sequence of characters associated with an input device, an output device, or a disk file.

In C++, a **stream** is a sequence of characters associated with an input device, an output device, or a disk file. Class `iostream` defines object `cin` as the stream associated with the standard input device (the keyboard). Class `iostream` defines object `cout` as the stream associated with the standard output device (the screen). Class `iostream` also defines the input operator >> and output operator <<. If you insert the compiler directive

```
#include <iostream>
```

before the main function, you can use `cin` and `cout` and the operators >> and << in your program. You also need to insert the statement

```
using namespace std;
```

Input Statements

The statement

```
cin >> miles;
```

indicates that the input operator >> (also called the *extraction operator*) should read data into the variable `miles`. Where does the extraction operator get the data it stores in variable `miles`? It gets the data from `cin`, the standard input device. In most cases the standard input device is the keyboard; consequently, the computer will attempt to store in `miles` whatever data the program user types at the keyboard.

If `miles` is declared as type `float`, the input operation will proceed correctly only if the program user types in a number. Figure 2.5 shows the effect of this input statement.

When finished entering data, the program user should press the RETURN or ENTER key. Any space characters preceding the data item will be ignored during the input operation. This is true for the input of type `int`, `float`, `char`, `bool`, and `string` data.

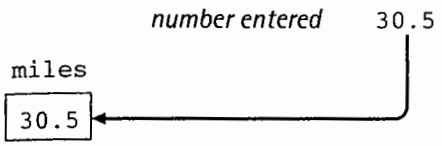


Figure 2.5 Effect of `cin >> miles;`

The Input (Extraction) Operator >>

Form: `cin >>` *data-store*`;`

Example: `cin >> age >> firstInitial;`

Interpretation: The extraction operator >> causes data typed at the keyboard to be read into the indicated *data-store* (variable or object) during program execution. The program extracts one data item for each data-store. The symbol pair >> precedes each data-store.

The order of the data you type in must correspond to the order of the data-stores in the input statement. You must insert one or more blank characters between numeric and string data items, and you may insert blanks between consecutive character data items and strings. The input operator >> skips any blanks that precede the data value to be read. This operator stops reading when a character (normally a space) that cannot legally be a part of the value being read in is encountered. Press the ENTER or RETURN key after entering all data items.

The program in Listing 2.2 reads a person's first two initials and last name. The input statement

```
cin >> letter1 >> letter2 >> lastName;
```

causes data to be extracted from the input stream object `cin` and stored in the variables `letter1` and `letter2` and object `lastName`. One character will be stored in `letter1` and one in `letter2` (type `char`). Figure 2.6 shows the effect of this statement when the characters `EBKoffman` are typed. Note that the result would be the same if one or more blanks appear before `E`, `B`, or `Koffman`.

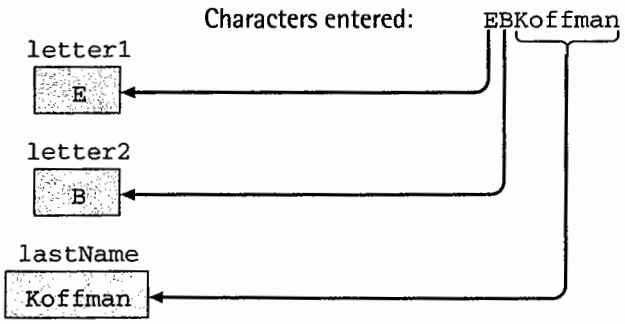


Figure 2.6 Effect of `cin >> letter1 >> letter2 >> lastName;`

The number of characters extracted from `cin` depends on the type of the variable or object in which the data will be stored. The next nonblank character is stored for a type `char` variable. For a type `float`, `int`, or `string` variable, the program continues to extract characters until it reaches a character that cannot be part of the data item (usually the result of pressing the space bar or the RETURN or ENTER key).

How do we know when to enter the input data and what data to enter? Your program should display a prompt as a signal that informs the program user what data to enter and when. (Prompts are discussed in the next section.)

Program Output

In order to see the results of a program execution, we must have some way of displaying data stored in variables. In Listing 2.1, the output statement

```
cout << "The distance in kilometers is "
     << kms << endl;
```

displays a line of program output containing two data elements: the string literal `"The distance . . . is "` and the value of `kms` (type `float`). The characters inside the quotes are displayed, but the quotes are not. The output operator << (also called the *insertion operator*) causes the data on the right of each occurrence of the operator to be inserted into the output stream object `cout`. Thus the previous output statement displays the line

```
The distance in kilometers is 16.09
```

In Listing 2.2, the line

```
cout << "Hello" << letter1 << ". " << letter2 << ". "
     << lastName << "! ";
```

displays

```
Hello E. B. Koffman!
```

In this case, the values of `letter1`, `letter2`, and `lastName` are displayed after the string `"Hello"`. The punctuation and space characters come from the other strings in the output statement.

Finally, the lines

```
cout << "Enter the distance in miles: ";
cout << "Enter 2 initials and a last name: ";
```

The Output (Insertion) Operator <<

Form: `cout << data-element;`

Example:
```
cout << "My height in inches is" << height
     << endl;
```

Interpretation: The *data-element* can be a variable, object, constant, or literal (such as a `string` or `float` literal). Each data-element must be preceded by the output operator. The output operator causes the value of the data-element that follows it to be displayed. Each value is displayed in the order in which it appears. An `endl` at the end of the output statement advances the screen cursor to the next line. A string is displayed without the quotes.

If no `endl` appears, the screen cursor will not advance to the next line. The next output statement will display characters on the same line as the previous one.

in Listings 2.1 and 2.2, respectively, display **prompts** or **prompting messages**. You should always display a prompt just before an input statement executes to inform the program user to enter data. The prompt should provide a description of the data to be entered—for example, `distance in miles`. If you don't display a prompt, the user will not know that the program is waiting for data or what data to enter.

prompt (prompting message)
A message displayed by the computer indicating its readiness to receive data or a command from the user.

The **screen cursor** is a moving place marker that indicates the position on the screen where the next character will be displayed. After executing an output statement, the cursor does not automatically advance to the next line of the screen. For this reason, the data you type in after a prompt will normally appear on the same line as the prompt. This is also the reason the two output statements in Listing 2.2

screen cursor
A moving place marker that indicates the position on the screen where the next character will be displayed.

```
cout << "Hello" << letter1 << ". " << letter2 << ". "
     << lastName << "! ";
cout << "We hope you enjoy studying C++." << endl;
```

display the single output line

```
Hello E. B. Koffman! We hope you enjoy studying C++.
```

But there are also many times when we would like to end one line and start a new one. We can do this by displaying the predefined constant `endl` (pronounced as *end-line*) when we wish to end a line.

To help you remember which operator follows cin and which follows cout, visualize the operator as an arrow indicating the direction of information flow. Data flows from `cin` to a variable (`cin >>`) and from a variable to `cout` (`cout <<`).

EXAMPLE 2.4

INSERTING BLANK LINES

The statements

```
cout << "The distance in kilometers is "
     << kms << endl << endl;
cout << "Conversion completed." << endl;
```

display the lines

```
The distance in kilometers is 16.09

Conversion completed.
```

A blank line occurs because we inserted `endl` twice in succession (`<< endl << endl`). Inserting `endl` always causes the display to be advanced to the next line. If there is nothing between the two `endl`s, a blank line appears in the program output. Most of the time, you will want to place `<< endl` at the end of an output statement.

The `return` Statement

The last line in the `main` function (Listing 2.1)

```
return 0;
```

transfers control from your program to the operating system. The value `0` is considered the result of function `main`'s execution. By convention, returning `0` from function `main` indicates to the operating system that your program executed without error.

Return Statement

Form: `return` *expression*`;`

Example: `return 0;`

Interpretation: The `return` statement transfers control from a function back to the activator of the function. For function `main`, control is transferred back to the operating system. The value of *expression* is returned as the result of function execution.

EXERCISES FOR SECTION 2.4

Self-Check

1. Show the output displayed by the program lines below when the data entered are 3.0 and 5.0.

```
cout << "Enter two numbers: ";
cin >> a >> b;
a = a - 5.0;
b = a * b;
cout << "a = " << a << endl;
cout << "b = " << b << endl;
```

2. Show the contents of memory before and after the execution of the program lines shown in Self-Check Exercise 1.

3. Show the output displayed by the lines below.

```
cout << "My name is: ";
cout << "Doe, Jane" << endl;
cout << "I live in ";
cout << "Ann Arbor, MI ";
cout << "and my zip code is " << 48109 << endl;
```

4. How would you modify the code in Exercise 3 so that a blank line would be displayed between the two sentences?

5. Which library contains the definition of `endl`?

6. Explain the purpose of the statement

```
using namespace std;
```

Programming

1. Write the C++ instructions that first ask a user to type three integers and then read the three user responses into the variables `first`, `second`, and `third`. Do this in two different ways.

2. **a.** Write a C++ statement that displays the value of `x` as indicated in the line below.

```
The value of x is _______
```

b. Assuming radius and area are type `float` variables, write a statement that displays this information in the following form:

```
The area of a circle with radius _______ is _______
```

3. Write a program that asks the user to enter the list price of an item and the discount as a percentage, and then computes and displays the sale price and the savings to the buyer. For example, if the list price is $40 and the discount percentage is 25%, the sale price would be $30 and the savings to the buyer would be $10. Use the formulas

   percentAsFraction = percent / 100.0

   discount = listPrice* percentAsFraction

   salePrice = listPrice discount

2.5 General Form of a C++ Program

Now that we have discussed the individual statements that can appear in C++ programs, we'll review the rules for combining them into programs. We'll also discuss the use of punctuation, spacing, and comments in a program.

As shown in Listing 2.3, each program begins with compiler directives that provide information about classes to be included from libraries. Examples of such directives are `#include <iostream>` and `#include <string>`. Unlike the `using` statement and the statements of the `main` function body, the compiler directives do not end in semicolons.

A C++ program defines the `main` function after the `using` statement. An open curly brace `{` signals the beginning of the `main` function body. Within this body, we first see the declarations of all the variables and objects to be used by the `main` function. Next come the executable statements that are translated into machine language and eventually executed. The executable statements we have looked at so far perform computations or input/output operations. The end of the `main` function body is marked by a closing curly brace `}`.

Listing 2.3 General form of a C++ program

```
compiler directives
using namespace std;

int main()
{
   declaration statements
   executable statements
}
```

C++ treats most line breaks like spaces, so a C++ statement can extend over more than one line. (For example, the variable declaration in Listing 2.1 extends over two lines.) You should not split a statement that extends over more than one line in the middle of an identifier, a reserved word, or a numeric or string literal.

You can write more than one statement on a line. For example, the line

```
cout << "Enter the distance in miles: "; cin >> miles;
```

contains an output statement that displays a prompt and an input statement that gets the data requested. We recommend that you place only one statement on a line; that will improve readability and make it easier to maintain a program.

PROGRAM STYLE Spaces in Programs

The consistent and careful use of blank spaces can improve the style of a program. A blank space is required between consecutive words in a program line.

The compiler ignores extra blanks between words and symbols, but you may insert space to improve the readability and style of a program. You should always leave a blank space after a comma and before and after operators such as `*`, `-`, and `=`. You should indent the body of the `main` function and insert blank lines between sections of the `main` function.

Although stylistic issues have no effect whatever on the meaning of the program as far as the computer is concerned, they can make it easier for people to read and understand the program. But take care not to insert blank spaces where they do not belong. You can't place a space between special character pairs (for example, `//`, `/*`, `*/`, `>>`, or `<<`), or write the identifier `MAX_ITEMS` as `MAX ITEMS`.

Comments in Programs

Programmers can make a program easier to understand by using comments to describe the purpose of the program, the use of identifiers, and the purpose of each program step. Comments are part of the **program documentation** because they help others read and understand the program. The compiler, however, ignores comments and they are not translated into machine language.

program documentation
Information (comments) that make it easier to read and understand a program.

A comment can appear by itself on a program line, at the end of a line following a statement, or embedded in a statement. In the following variable

declaration statement, the first comment is embedded in the declaration, while the second one follows the semicolon that terminates the declaration.

```
float miles,     // input - distance in miles
      kms;       // output - distance in kilometers
```

We document most variables in this way.

PROGRAM STYLE Using Comments

Each program should begin with a header section that consists of a series of comments specifying

- The programmer's name
- The date of the current version
- A brief description of what the program does

If you write the program for a class assignment, you should also list the class identification and your instructor's name:

```
/*
 * Programmer: William Bell Date completed: May 9, 2003
 * Instructor: Janet Smith Class: CIS61
 *
 * Calculates and displays the area and circumference of
 * a circle
 */
```

As shown above, programmers often use the symbol pairs `/*`, `*/` to surround multiline comments instead of placing the symbol pair `//` before each line.

Before you implement each step in the initial algorithm, you should write a comment that summarizes the purpose of the algorithm step. This comment should describe what the step does rather than simply restate the step in English. For example, the comment

```
// Convert the distance to kilometers.
kms = KM_PER_MILE * miles;
```

is more descriptive than (so preferable to)

```
// Multiply KM_PER_MILE by miles and store result in kms.
kms = KM_PER_MILE * miles;
```

EXERCISES FOR SECTION 2.5

Self-Check

1. Change the following comments so they are syntactically correct.

```
/* This is a comment? *\
/* This one /* is a comment */ isn't it? */
```

2. Correct the syntax errors in the following program, and rewrite the program so that it follows our style conventions. What does each statement of your corrected program do? What output does it display?

```
/*
* Calculate and display the product of two input values
//
#include [iostream]
void main(float)
{int Y,    /* first input value */
            y,     /* second input value */
      sum; /* product of inputs */
cin << Y << sum;
Y * y = sum;
cout >> sum >> " = " >> Y " * " y;
return 0;}
```

Programming

1. Write a program that performs the implementation step for the case study presented in Section 1.5.

   Problem: Determine the total cost of apples given the number of pounds of apples purchased and the cost per pound of apples.

   Run this using several sets of input data. Try special values such as 0 and 1 for pounds of apples.

2.6 Arithmetic Expressions

To solve most programming problems, you will need to write arithmetic expressions that manipulate numeric data. This section describes the operators used in arithmetic expressions and rules for writing and evaluating the expressions.

Table 2.6 Arithmetic Operators

Arithmetic Operator	Meaning	Examples
+	addition	5 + 2 is 7
		5.0 + 2.0 is 7.0
–	subtraction	5 - 2 is 3
		5.0 - 2.0 is 3.0
*	multiplication	5 * 2 is 10
		5.0 * 2.0 is 10.0
/	division	5.0 / 2.0 is 2.5
		5 / 2 is 2
%	remainder	5 % 2 is 1

Table 2.6 shows all the arithmetic operators. Each operator manipulates two *operands*, which may be constants, variables, or other arithmetic expressions. The operators `+`, `-`, `*`, and `/` may be used with type `int` or `float` operands. As shown in the last column, the data type of the result is the same as the data type of its operands. An additional operator, the remainder operator (`%`), can be used with integer operands to find the remainder of longhand division. We will discuss the division and remainder operators next.

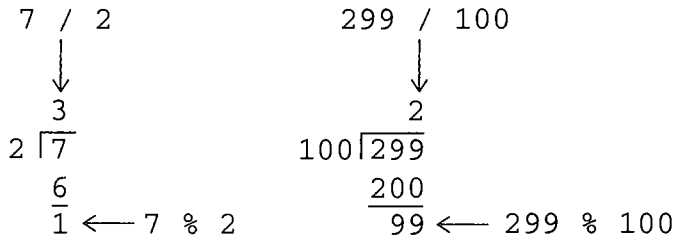
Operators / and %

When applied to two positive integers, the division operator (/) computes the integral part of the result of dividing its first operand by its second. For example, the value of `7.0 / 2.0` is `3.5`, but the value of `7 / 2` is the integral part of this result, or `3`. Similarly, the value of `299.0 / 100.0` is `2.99`, but the value of `299 / 100` is the integral part of this result, or `2`. If the / operator is used with a negative and a positive integer, the result may vary from one C++ implementation to another. For this reason, you should avoid using division with negative integers. The / operation is undefined when the divisor (the second operand) is `0`. Table 2.7 shows some examples of integer division.

Table 2.7 Results of Integer Division

`3 / 15 = 0`	`18 / 3 = 6`
`15 / 3 = 5`	`16 / -3` system dependent
`16 / 3 = 5`	`0 / 4 = 0`
`17 / 3 = 5`	`4 / 0` is undefined

The remainder operator (%) returns the *integer remainder* of the result of dividing its first operand by its second. For example, the value of `7 % 2` is `1` because the integer remainder is `1`.



You can use longhand division to determine the result of a / or % operation with integers. The calculation on the left shows the effect of dividing `7` by `2` using longhand division: we get a quotient of `3` `(7 / 2)` and a remainder of `1` `(7 % 2)`. The calculation on the right shows that `299 % 100` is `99` because we get a remainder of `99` when we divide `299` by `100`.

The magnitude of *m* % *n* must always be less than the divisor *n*, so if *m* is positive, the value of *m* % `100` must be between 0 and 99. The % operation is undefined when *n* is zero and varies from one implementation to another if *n* is negative. Table 2.8 shows some examples of the % operator.

The formula

m equals (*m* /*n*) * *n* + (*m* % *n*)

defines the relationship between the operators / and % for an integer dividend of *m* and an integer divisor of *n*. We can see that this formula holds for the two problems discussed earlier by plugging in values for *m*, *n*, *m* / *n*, and *m* % *n*. In the first example that follows, *m* is 7 and *n* is 2; in the second, *m* is 299 and *n* is 100.

```
7    equals (7 / 2)        *2 + (7 % 2)
     equals     3          *2 + 1
290  equals (299 / 100)    *100 + (299 % 100)
     equals       2        *100 + 99
```

Table 2.8 Results of % Operation

`3 % 5 = 3`	`5 % 3 = 2`
`4 % 5 = 4`	`5 % 4 = 1`
`5 % 5 = 0`	`15 % 5 = 0`
`6 % 5 = 1`	`15 % 6 = 3`
`7 % 5 = 2`	`15 % -7` system dependent
`8 % 5 = 3`	`15 % 0` is undefined

EXAMPLE 2.5

If you have `p` people and `b` identical boats, the expression

```
p / b
```

tells you how many people to put in each boat. For example, if `p` is `18` and `b` is `4`, then four people would go in each boat. The formula

```
p % b
```

tells you how many people would be left over (`18 % 4` is 2).

Data Type of a Mixed-Type Expression

The data type of each variable must be specified in its declaration, but how does C++ determine the type of an expression? The data type of an expression depends on the type of its operands. For example, the expression

```
m + n
```

is type `int` if both `m` and `n` are type `int`; otherwise, it is type `float`. A C++ expression is type `int` only if all its operands are type `int`, and a C++ expression is type `float` if any of its operands are type `float`. For example, `5 / 2` is type `int`, but `5 / 2.5` is type `float`. The latter expression, containing an integer and a floating-point operand, is called a **mixed-type expression**. The type of a mixed-type expression involving integer and floating-point data must be `float`.

mixed-type expression
An expression involving operands of type `int` and type `float`

In evaluating a mixed-type expression such as `5 / 2.5`, C++ cannot perform the division directly because the internal representations of the numbers `5` and `2.5` are in different formats (fixed-point for `5`, floating-point for `2.5`). Consequently, C++ must first compute and store the floating-point equivalent of `5` in a new memory cell and then divide the contents of that cell (`5.0`) by the contents of the cell containing `2.5`. C++ "converts" the type `int` value to type `float` instead of the other way around because there is no type `int` value that is equivalent to the real number `2.5`.

Mixed-Type Assignment Statement

When an assignment statement is executed, the expression is first evaluated and then the result is assigned to the variable listed to the left of the assignment operator (=). Either a type `float` or type `int` expression may be

assigned to a type `float` variable. If the expression is type `int`, a real number with a fractional part of zero will be stored in a type `float` variable. For example, if a is type `float`, the statement

```
a = 10;
```

stores the floating-point value `10.0` in `a`. In a mixed-type assignment such as

```
a = 10 / 3;
```

the expression is type `int` and the variable `a` is type `float`. A common error is to assume that since `a` is type `float`, the expression should be evaluated as if its operands were type `float` as well. If you make this error, you will calculate the expression value incorrectly as `3.3333...`. Remember, the expression is evaluated before the assignment is made, and the type of the variable being assigned has no effect on the expression value. The expression `10 / 3` evaluates to `3`, so `3.0` is stored in `a`.

In a similar vein, consider the assignment statement below when `n` is type `int`.

```
n = 10.5 + 3.7;
```

The expression is type `float`, and it is evaluated before the assignment. Its value is `14.2`, so the integral part of `14.2`, or `14`, is stored in `n`. Again, be careful not to evaluate the expression as if its operands were the same type as the variable being assigned. If you make this error, you might calculate the expression result incorrectly as `13 (10 + 3)`.

Expressions with Multiple Operators

In our programs so far, most expressions have involved a single arithmetic operator; however, expressions with multiple operators are common in C++. Expressions can include both unary and binary operators. **Unary operators** take only one operand. In these expressions, we see the unary negation (-) and plus (+) operators.

unary operator
An operator that has one operand.

```
x = -y;
p = +x * y;
```

Binary operators require two operands. When + and - are used to represent addition and subtraction, they are binary operators.

binary operator
An operator that has two operands.

```
x = y + z;
z = y - x;
```

To understand and write expressions with multiple operators, we must know the C++ rules for evaluating expressions. For example, in the expression `x + y / z`, is + performed before / or is + performed after /? Is the expression `x / y * z` evaluated as `(x / y) * z` or as `x / (y * z)`? Verify for yourself that the order of evaluation does make a difference by substituting some simple values for `x`, `y`, and `z`. In both expressions, the / operator is evaluated first; the reasons are explained in the C++ rules for evaluation of arithmetic expressions that follow. These rules are based on familiar algebraic rules.

Rules for Evaluating Expressions

1. *Parentheses rule:* All expressions in parentheses must be evaluated separately. Nested parenthesized expressions must be evaluated from the inside out, with the innermost expression evaluated first.
2. *Operator precedence rule:* Operators in the same expression are evaluated in the following order:

```
unary +, -    first
*, /, %       next
binary +, -   last
```

3. *Associativity rule:* Binary operators in the same subexpression and at the same precedence level (such as + and -) are evaluated left to right (*left associativity*). Unary operators in the same subexpression and at the same precedence level (such as + and -) are evaluated right to left (*right associativity*).

These rules will help you understand how C++ evaluates expressions. Use parentheses as needed to specify the order of evaluation. Often it is a good idea in complicated expressions to use extra parentheses to document clearly the order of operator evaluation. For example, the expression

```
x * y * z + a / b - c * d
```

can be written in a more readable form using parentheses:

```
(x * y * z ) + (a / b) - (c * d)
```

In Figure 2.8, we see a step-by-step evaluation of the same expression for a `radius` value of `2.0`. You may want to use a similar notation when computing by hand the value of an expression with multiple operators.

EXAMPLE 2.6

The formula for the area of a circle

```
a = πr2
```

can be written in C++ as

```
area = PI * radius * radius;
```

where the value of the constant `PI` is `3.14159`. Figure 2.7 shows the *evaluation tree* for this formula. In this tree, which you read from top to bottom, arrows connect each operand with its operator. The order of operator evaluation is shown by the number to the left of each operator; the letter to the right of the operator indicates which evaluation rule applies.

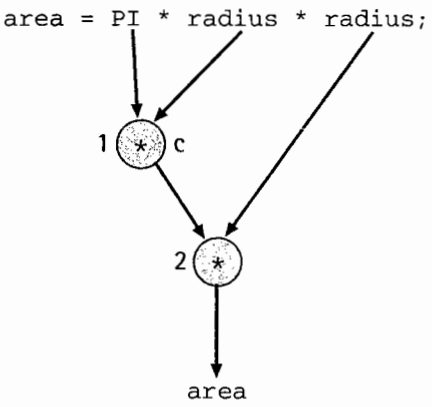


Figure 2.7 Evaluation tree for `area = PI * radius * radius;`

```
area   =   PI    *   radius    *   radius
         3.14159 *     2.0             2.0
               6.28318         *       2.0
                          12.56636
```

Figure 2.8 Step-by-step evaluation

EXAMPLE 2.7

The formula for the average velocity, v, of a particle traveling on a line between points p_1 and p_2 in time t_1 to t_2 is

$$v = \frac{p_2 - p_1}{t_2 - t_1}$$

This formula can be written and evaluated in C++ as shown in Figure 2.9.

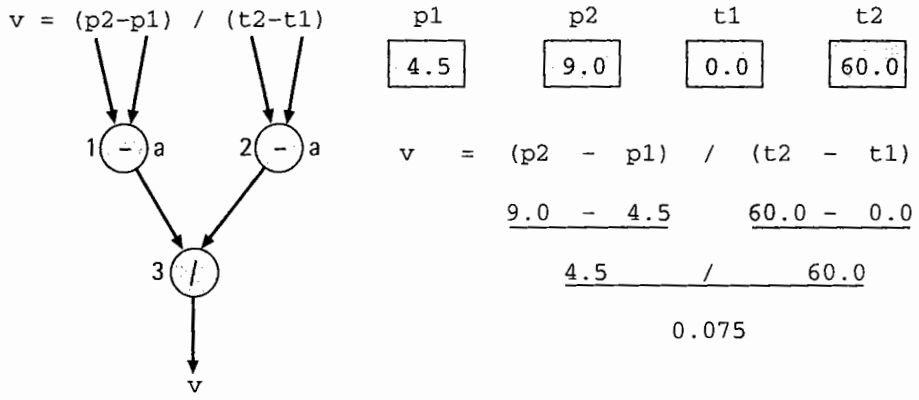


Figure 2.9 Evaluation for `v = (p2 - p1) / (t2 - t1);`

EXAMPLE 2.8

Consider the expression

```
z - (a + b / 2) + w * -y
```

containing type `int` variables only. The parenthesized expression

```
(a + b / 2)
```

is evaluated first (rule 1) beginning with `b / 2` (rule 2). Once the value of `b / 2` is determined, it can be added to `a` to obtain the value of `(a + b / 2)`. Next, `y` is negated (rule 2). The multiplication operation can now be performed (rule 2) and the value for `w * -y` is determined. Then, the value of `(a + b / 2)` is subtracted from `z` (rule 3). Finally, this result is added to `w * -y`. The evaluation tree and step-by-step evaluation for this expression are shown in Figure 2.10.

EXAMPLE 2.9

The evaluation of multiple operator expressions containing both type `int` and `float` values can be quite tricky to follow. Don't make the mistake of just converting all operands to type `float`, but evaluate each operator and its operands separately. Consider the statement

```
m = x + k / 2;
```

shown in Figure 2.11, with `x = 5.5` (type `float`), `k = 5` (type `int`), and `m` (type `int`).

First, `k / 2` is evaluated (rule 2). Because both `k` and `2` are type `int`, the result is type int `(k / 2 = 2)`. Next, `x + 2` is evaluated. Because `x` is type `float`, `2` is first converted to `2.0` so the expression `5.5 + 2.0` is evaluated. The result `7.5` is then truncated to `7` before it is assigned to the type `int` variable `m`.

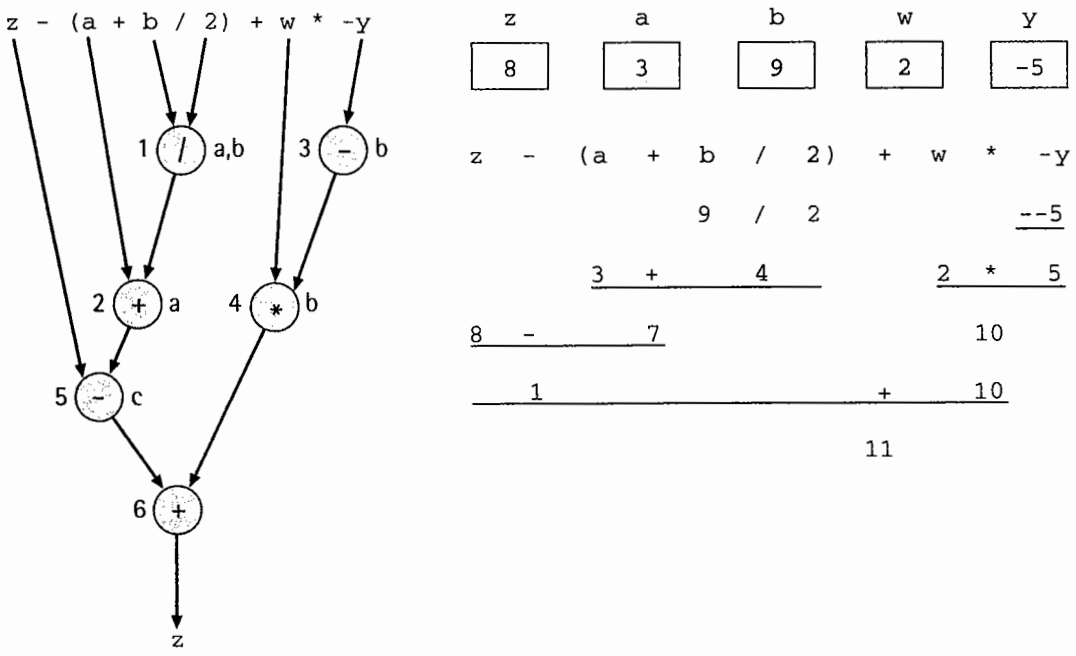


Figure 2.10 Evaluation for `z - (a + b / 2) + w * -y;`

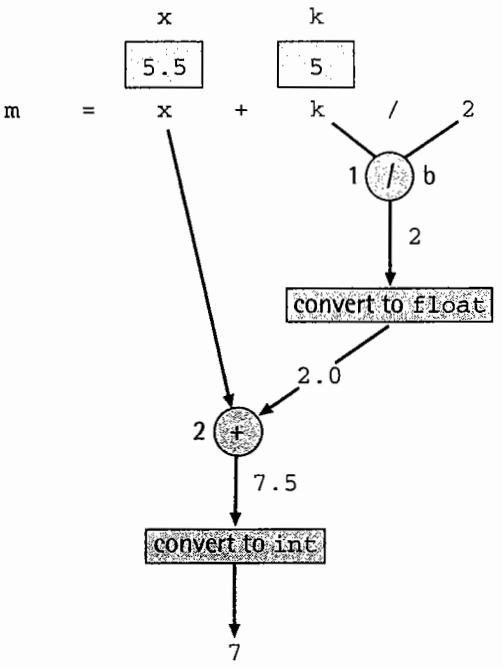


Figure 2.11 Evaluation tree for `m = x + k / 2;`

Writing Mathematical Formulas in C++

You may encounter two problems when writing mathematical formulas in C++: The first concerns multiplication, which is often implied in a mathematical formula by writing the two multiplicands next to each other—for

example, $a = bc$. In C++, however, you must always use the * operator to indicate multiplication, as in

```
a = b * c;
```

The other difficulty arises in formulas using division in which we normally write the numerator and denominator on separate lines:

$$m = \frac{y - b}{x - a}$$

In C++, however, all assignment statements must be written in a linear form. Consequently, parentheses are often needed to separate the numerator from the denominator and to clearly indicate the order of evaluation of the operators in the expression. The formula above would thus be written in C++ as

```
m = (y - b) / (x - a);
```

EXAMPLE 2.10

This example illustrates how several mathematical formulas can be written in C++.

Mathematical Formula	C++ Expression
1. $b^2 - 4ac$	`b * b - 4 * a * c`
2. $a + b - c$	`a + b - c`
3. $\frac{a + b}{c + d}$	`(a + b) / (c + d)`
4. $\frac{1}{1 + y^2}$	`1 / (1 + y * y)`
5. $a \times -(b + c)$	`a * -(b + c)`

The points illustrated are summarized as follows:

- Always specify multiplication explicitly by using the operator * where needed (formulas 1 and 4).
- Use parentheses when required to control the order of operator evaluation (formulas 3 and 4).
- Two arithmetic operators can be written in succession if the second is a unary operator (formula 5).

case study Supermarket Coin Processor

This case study provides an example of manipulating type `int` data (using `/` and `%`).

PROBLEM

You are writing software for the machines placed at the front of supermarkets to convert change to personalized credit slips. In this version, the user will manually enter the number of each kind of coin in the collection, but in the final version, these counts will be provided by code that interfaces with the counting devices in the machine.

ANALYSIS

To solve this problem, you need to get the customer's name to personalize the credit slip. You also need the count of each type of coin (dollars, quarters, dimes, nickels, pennies). From those counts, you determine the total value of coins in cents. Next, you can do an integer division using 100 as the divisor to get the dollar value; the remainder of this division will be the loose change that she should receive. In the data requirements table below, you list the total value in cents (`totalCents`) as a **program variable** because it is needed as part of the computation process but is not a required problem output.

program variable
A variable needed to store a computation in a program.

DATA REQUIREMENTS

Problem Input

```
string name      // user's first name
int dollars      // count of dollars
int quarters     // count of quarters
int dimes        // count of dimes
int nickels      // count of nickels
int pennies      // count of pennies
```

Problem Output

```
int totalDollars  // the number of dollars
int change        // the change
```

Additional Program Variable

```
int totalCents    // the total number of cents
```

FORMULAS

one dollar equals 100 pennies
one nickel equals 5 pennies

DESIGN

The algorithm is straightforward and is presented next.

INITIAL ALGORITHM

1. Read in the user's first name.
2. Read in the count of each kind of coin.
3. Compute the total value in cents.
4. Find the value in dollars and change.
5. Display the value in dollars and change.

Steps 3 and 4 require further refinement.

Step 3 Refinement

3.1. Find the value of each kind of coin in pennies and add these values.

Step 4 Refinement

4.1. `totalDollars` is the integer quotient of `totalCents` and 100.

4.2. `change` is the integer remainder of `totalCents` and 100.

IMPLEMENTATION

Listing 2.4 shows the program. The statement

```
totalCents = 100 * dollars + 25 * quarters + 10 * dimes +
             5 * nickels + pennies;
```

implements algorithm Step 3.1. The statements

```
totalDollars = totalCents / 100;
change = totalCents % 100;
```

use the `/` and `%` operators to implement algorithm Steps 4.1 and 4.2, respectively.

TESTING

To test this program, try running it with a combination of coins that yields an exact dollar amount with no change left over. For example, 1 dollar, 8 quarters, 0 dimes, 35 nickels, and 25 pennies should yield a value of 5 dollars and 0 cents. Then increase and decrease the amount of pennies by 1 (26 and 24 pennies) to make sure that these cases are also handled properly.

Listing 2.4 Value of a coin collection

```
// File: Coins.cpp
// Determines the value of a coin collection

#include <iostream>
using namespace std;

int main ()
{
   string name;            // input: user's first name
   int pennies;            // input: count of pennies
   int nickels;            // input: count of nickels
   int dimes;              // input: count of dimes
   int quarters;           // input: count of quarters
   int dollars;            // input: count of coins in dollars
   int totalDollars;       // output: value of coins in dollars
   int change;             // output: value of coins in cents
   int totalCents;         // total cents represented

   // Read user's first name.
   cout << "Enter your first name: ";
   cin >> name;

   // Read in the count of nickels and pennies.
   cout << "Enter the number of dollars: ";
   cin >> dollars;
   cout << "Enter the number of quarters: ";
   cin >> quarters;
   cout << "Enter the number of dimes: ";
   cin >> dimes;
   cout << "Enter the number of nickels: ";
   cin >> nickels;
   cout << "Enter the number of pennies: ";
   cin >> pennies;

   // Compute the total value in cents.
   totalCents = 100 * dollars + 25 * quarters + 10 * dimes +
                5 * nickels + pennies;

   // Find the value in dollars and change.
   totalDollars = totalCents / 100;
   change = totalCents % 100;
   // Display the value in dollars and change.
   cout << "Coin credit: " << name << endl;
   cout << "Dollars: " << totalDollars << endl;
```

(continued)

Listing 2.4 Value of a coin collection (continued)

```
    cout << "Cents: " << change << endl;

    return 0;
}
```

```
Enter your first name: Richard
Enter the number of dollars: 2
Enter the number of quarters: 3
Enter the number of dimes: 11
Enter the number of nickels: 1
Enter the number of pennies: 5
Coin credit: Richard
Dollars: 3
Cents: 95
```

PROGRAM STYLE Use of Color to Highlight New Constructs

In Listing 2.1, program lines that illustrate key constructs are in color, so that you can find them easily. We will continue to use color for this purpose in listings.

EXERCISES FOR SECTION 2.6

Self-Check

1. For the program fragment below, what values are printed?

```
int x, y, z;
float w;
x = 15;
w = x / 2;
y = 2 * x % 4;
z = 2 + x % 4;
cout << w << " " << x << " " << y << " " << z;
```

2. Evaluate the following expressions with 7 and 22 as operands:

   a. 22 /7 **b.** 7/ 22 **c.** 22 % 7 **d.** 7 % 22

   Repeat this exercise for the following pairs of integers:

   e. 15, 16 **f.** 3, 23 **g.** −3, 16

3. Show that the formula m = (m / n) * n + (m % n) holds when m = 45 and n = 5 (both of type `int`).

4. Show the step-by-step evaluation of the expressions below.

```
(y % z * (5 + w)) + x / y
(w + x / z) - x / (w - 5)
```

5. Given the declarations

```
const float PI = 3.14159;
const int MAX_I = 1000;
float x, y;
int a, b, i;
```

indicate which C++ statements below are valid and find the value of each valid statement. Also, indicate which are invalid and why. Assume that a is 3, b is 4, and y is –1.0.

a. `i = a % b;`
b. `i = (MAX_I – 990) / a;`
c. `i = a % y;`
d. `i = (990 – MAX_I) / a;`
e. `i = PI * a;`
f. `x = PI * y;`
g. `x = PI / y;`
h. `i = (MAX_I – 990) % a;`
i. `x = a % (a / b);`
j. `i = a % 0;`
k. `i = b / 0;`
l. `i = a % (MAX_I – 990);`
m. `x = a / y;`
n. `i = a % (990 – MAX_I);`
o. `x = a / b;`

6. What values are assigned by the legal statements in Exercise 5 above, assuming a is 2, b is 5, and y is 2.0?

7. Assume that you have the following variable declarations:

```
int color, lime, straw, yellow, red, orange;
float black, white, green, blue, purple, crayon;
```

Evaluate each of the statements below given the following values: color is 2, black is 2.5, crayon is –1.3, straw is 1, red is 3, and purple is 0.3e+1.

a. `white = color * 2.5 / purple;`
b. `green = color / purple;`
c. `orange = color / red;`
d. `blue = (color + straw) / (crayon + 0.3);`
e. `lime = red / color + red % color;`
f. `purple = straw / red * color;`

8. Let a, b, c, and x be the names of four type float variables, and let i, j, and k be the names of three type int variables. Each of the statements below contains a violation of the rules for forming arithmetic expressions. Rewrite each statement so that it is consistent with these rules.

a. `x = 4.0 a * c;`
b. `a = ac;`
c. `i = 5j3;`
d. `k = 3(i + j);`
e. `x = 5a / bc;`

Programming

1. Write an assignment statement that might be used to implement the equation below in C++.

$$q = \frac{ka(t_1 - t_2)}{b}$$

2. Write a program that reads two int values into m, n and displays their sum, their differences ($m - n$ and $n - m$), their product, their quotients (m / n and n / m) and both $m \% n$ and $n \% m$. If the numbers are 4 and 5, the line that shows their sum should be displayed as:

```
5 + 4 = 9
```

   Use this format for each output line.

3. Extend the program in Listing 2.4 to handle two-dollar coins and half-dollars.

2.7 Interactive Mode, Batch Mode, and Data Files

interactive mode
A mode of program execution in which the user interacts with a running program, entering data as requested.

batch mode
A noninteractive mode of program execution that requires all program data to be supplied before execution begins.

There are two basic modes of computer operation: interactive and batch. The programs we have written so far are intended to be run in **interactive mode**, a mode of program execution in which the user can interact with the program and enter data while the program is executing. In **batch mode**, all data must be supplied beforehand and the program user cannot interact with the program while it is executing. Batch mode is an option on most computers.

If you use batch mode, you must prepare a batch data file before executing your program. On a time-shared or personal computer, a batch data file is created and saved in the same way as a program or source file.

Input Redirection

input/output redirection
Using operating system commands to associate the standard input device with an input file instead of the keyboard and/or the standard output device with an output file instead of the screen.

Listing 2.5 shows the miles to kilometers conversion program rewritten as a batch program. We assume that the input is associated with a data file instead of the keyboard. In most systems this can be done relatively easily through **input/output redirection** using operating system commands. For example, in the UNIX operating system, you can instruct your program to take its input from file `mydata` instead of the keyboard, by placing the symbols

```
< mydata
```

at the end of the command line that causes your compiled and linked program to execute. If you normally used the UNIX command line

```
metric
```

to execute this program, your new command line would be

```
metric < mydata
```

PROGRAM STYLE Echo Prints versus Prompts

In Listing 2.5, the statement

```
cin >> miles;
```

Listing 2.5 Batch version of miles-to-kilometers conversion program

```
// File: milesBatch.cpp
// Converts distance in miles to kilometers.

#include <iostream>
using namespace std;

int main()
{
      const float KM_PER_MILE = 1.609; // 1.609 km in a mile
      float miles,                     // input: distance in miles
            kms;                       // output: distance in kilometers

      // Get the distance in miles.
      cin >> miles;
      cout << "The distance in miles is " << miles << endl;

      // Convert the distance to kilometers.
      kms = KM_PER_MILE * miles;

      // Display the distance in kilometers.
      cout << "The distance in kilometers is " << kms << endl;

      return 0;
}
```

```
The distance in miles is 10
The distance in kilometers is 16.09
```

reads the value of `miles` from the first (and only) line of the data file. Because the program input comes from a data file, there is no need to precede this statement with a prompt. Instead, we follow the input statement with the output statement

```
cout << "The distance in miles is " << miles << endl;
```

which displays, or *echo prints*, the value just read into `miles`. This statement provides a record of the data to be manipulated by the program; without it, we would have no easy way of knowing what value was read. Whenever you convert an interactive program to a batch program, make sure you replace each prompt with an echo print that follows each input statement.

Output Redirection

You can also redirect program output to a disk file instead of the screen. Then you could print the output file to obtain a hard-copy version of the program output. In UNIX, you would type

```
> myoutput
```

to redirect output from the screen to file `myoutput`. The command

```
metric > myoutput
```

executes the compiled and linked code for the metric conversion program, reading program input from the keyboard and writing program output to the file `myoutput`. However, it would be difficult to interact with the running program because all program output, including any prompts, are sent to the output file. It would be better to use the command

```
metric < mydata > myoutput
```

which reads program input from the data file `mydata` and sends program output to the output file `myoutput`.

EXERCISES FOR SECTION 2.7

Self-Check

1. Explain the difference in placement of `cout` statements used to display prompts and `cout` statements used to echo data. Which are used in

interactive programs and which are used in batch programs? How are input data provided to an interactive program? How are input data provided to a batch program?

Programming

1. Rewrite the program in Listing 2.4 as a batch program. Assume data are read from the file `mydata`.

2. Change the main function below to an interactive version.

```
int main() {
   int m, n;        // input - 2 numbers
   int sum;         // output - their sum

   cin >> m >> n;
   cout << "m is " << m << ", n is " << n << endl;

   sum = m + n;
   cout << "Their sum is " << sum << endl;

   return 0;
}
```

3. Write a batch version of the solution to Programming Exercise 2 in Section 2.6 (repeated below). Your output should be sent to an output file. Show the UNIX command that reads data from file `inData` and writes results to output file `outData`.

   Write a program that reads two int values into *m*, *n* and displays their sum, their differences (*m* – *n* and *n* – *m*), their product, their quotients (*m* / *n* and *n* / *m*) and both *m* % *n* and *n* % *m*. If the numbers are 4 and 5, the line that shows their sum should be displayed as:

```
5 + 4 = 9
```

   Use this format for each output line.

2.8 Common Programming Errors

As you begin to program, you will soon discover that a program rarely runs correctly on the first try. Murphy's law—"If something can go wrong, it will"—seems to have been written with the computer program in mind. In fact, errors are so common that they have their own special name—*bugs*—and

debugging
The process of removing errors from a program.

the process of correcting them is called **debugging**. (According to computer folklore, computer pioneer Dr. Grace Murray Hopper diagnosed the first hardware error caused by a large insect found inside a computer component.) To alert you to potential problems, we will provide a section on common errors at the end of each chapter.

When the compiler detects an error, it displays an error message that indicates you made a mistake and what the likely cause of the error might be. Unfortunately, error messages are often difficult to interpret and are sometimes misleading. As you gain some experience, you will become more proficient at locating and correcting errors.

Three kinds of errors—syntax errors, run-time errors, and logic errors—can occur, as discussed in the following sections.

Syntax Errors

syntax error
A violation of a C++ grammar rule that prevents a program from being translated.

A **syntax error** occurs when your code violates one or more of the grammar rules of C++ and is detected by the compiler as it attempts to translate your program. If a statement has a syntax error, it cannot be completely translated, and your program will not execute.

Listing 2.6 shows a version of the miles-to-kilometers conversion program with errors introduced.

One C++ system gives the following error messages:

```
[C++ Error] miles.cpp(12): E2141 Declaration syntax error.
[C++ Error] miles.cpp(15): E2380 Unterminated string or
     character constant.
[C++ Error] miles.cpp(16): E2379 Statement missing ;.
[C++ Error] miles.cpp(19): E2451 Undefined symbol 'kms'.
[C++ Warning] miles.cpp(25): W8080 'miles' is declared but
     never used.
[C++ Warning] miles.cpp(25): W8004 'KM_PER_MILE' is
     assigned a value that is never used.
```

On some systems, if you click on an error message, the statement containing the error will be highlighted. The missing comma after `miles` in the variable declaration statement causes error message `E2141` above (`Declaration syntax error`). The missing quote at the end of the prompt string in the first executable statement causes error message `E2380`. Because of the missing quote, C++ considers the rest of the line to be part of the string including the `;` at the end of the line. This causes error message `E2379`. If you click on this message, you will find that the line beginning with `cin` is highlighted; however, the missing `;` is actually in the line just above. This kind of displacement of error messages is fairly common—you sometimes

Listing 2.6 Miles-to-kilometers conversion program with syntax errors

```
// milesError.cpp
// Converts distances from miles to kilometers.

#include <iostream>
using namespace std;

int main()        // start of function main
{

        const float KM_PER_MILE = 1.609; // 1.609 km in a mile
        float miles                      // input: distance in miles
              kms;                       // output: distance in kilometers
        // Get the distance in miles.
        cout << "Enter the distance in miles: ";
        cin >> miles;

        // Convert the distance to kilometers.
        kms = KM_PER_MILE * miles;

        // Display the distance in kilometers.
        cout << "The distance in kilometers is " << kms << endl;
        return 0;
}
```

need to check one or more lines that precede the highlighted line to find an error. The incorrect variable declaration statement also causes error message `E2451`, which is associated with the assignment statement. If you fix the variable declaration, this error message will also go away. Finally, the last two error messages (`W8080` and `W8004`) are considered warning messages. They will not stop the program from being executed, but they do indicate suspicious behavior in the program.

Your strategy for correcting syntax errors should take into account the fact that one error can lead to many error messages. It is often a good idea to concentrate on correcting the errors in the declaration statements first. Then recompile the program before you attempt to fix other errors. Many of the other error messages will disappear once the declarations are correct.

Run-Time Errors

Run-time errors are detected and displayed by the computer during the execution of a program. A run-time error occurs when the program directs the computer to perform an illegal operation, such as dividing a number by

run-time error
An error detected by the computer during program execution.

Listing 2.7 Program with a potential run-time error

```
// File: runtimeError.cpp
// Provides a possible "division by zero" run-time error.

#include <iostream>
using namespace std;

int main()
{
        int first, second;
        float temp, ans;

        cout << "Enter 2 integers: ";
        cin >> first >> second;

        temp = second / first;
        ans = first / temp;

        cout << "The result is " << ans << endl;

        return 0;
}
```

zero. When a run-time error occurs, the computer will stop executing your program and will display a diagnostic message that indicates the line where the error was detected.

The program in Listing 2.7 compiles successfully but cannot run to completion if the first integer entered is greater than the second. In this case, integer division causes the first assignment statement to store zero in `temp`. Using `temp` as a divisor in the next line causes a run-time error such as `"Floating point division by zero"`.

Undetected Errors

Some execution errors will not prevent a C++ program from running to completion, but they may lead to incorrect results. It is essential that you predict the results your program should produce and verify that the actual output is correct.

A very common source of incorrect results is the input of character and numeric data. Listing 2.8 shows the data entry statements for the coin program from Listing 2.4 If your niece happens to type in her entire name instead of just her first name, a curious result will occur. You indicate the

Listing 2.8 Data entry statements for coins program

```
// Read in the user's first name.
cout << "Enter your first name: ";
cin >> name;

// Read in the count of each kind of coin.
cout << "Enter the number of dollars: ";
cin >> dollars;
cout << "Enter the number of quarters: ";
cin >> quarters;
```

end of a string data item by pressing the space bar or the return key, so her first name will be extracted correctly. But the characters that remain in the input stream (the space and last name) will mess up the data entry for variables `nickels` and `pennies`. Because the input stream is not empty, C++ attempts to extract these data from the current input stream instead of waiting for the user to type in these numbers. Since the next data character to be processed is a letter, C++ will simply leave the values of `dollars` and `quarters` unchanged, which will lead to unpredictable results.

The output from one incorrect run of the program is shown below. Notice that the prompts for `dollars` and `quarters` appear are on the second and third lines and the user is unable to type in any numbers. The program runs to completion using whatever "garbage" values were originally in all input variables.

```
Enter your first name: Sally Smith
Enter the number of dollars: Enter the number of
quarters: . . .
Coin Credit: Sally
Dollars: 210060
Change: 0 cents
```

Logic Errors

Logic errors occur when a program follows a faulty algorithm. Because such errors usually do not cause run-time errors and do not display error messages, they are very difficult to detect. You can detect logic errors by testing the program thoroughly, comparing its output to calculated results. You can prevent logic errors by carefully desk checking the algorithm and the program before you type it in.

Because debugging a program can be very time-consuming, plan your program solutions carefully and desk-check them to eliminate bugs early. If you are unsure of the syntax for a particular statement, look it up in the text.

Chapter Review

1. You learned how to use C++ statements to perform some basic operations: to read information into memory, to perform computations, and to display the results of the computation. You can do all of this using symbols (variable names and operators, such as +, -, *, and /) that are familiar and easy to remember.

2. Use the compiler directive `#include` to reuse files containing classes and functions in previously defined libraries. Use the statement `using namespace std;` to indicate that you are referencing C++ elements defined in the standard namespace.

3. Comments make programs easier to read and understand. Use comments to document the use of variables and to introduce the major steps of the algorithm. A comment in a program line is preceded by the symbol pair `//` and it extends to the end of a line. You can also use the symbol pairs `/*, */` to bracket multiline comments.

4. C++ statements contain reserved words (predefined in C++) and identifiers chosen by the programmer. Identifiers should begin with a letter and can consist of letters, digits, and the underscore character only. Use a standard naming convention for identifiers. We suggest using all uppercase for constants and using all lowercase for one-word names of variables and objects. For variables with multiword names, start each word after the first with an uppercase letter (no underscores). Use underscore characters between words of a constant identifier.

5. A data type specifies a set of values and the operations that can be performed on those values. We will use the C++ predefined data types `float`, `int`, `char`, and `bool`. Types `float` and `int` are abstractions for the integers and real numbers. C++ uses floating-point representation for real numbers and fixed-point representation for integers. C++ uses the data type `char` to represent single characters that are enclosed in apostrophes in a program. The apostrophes are not considered part of the data element and are not stored in memory. Data type `bool` represents the two values `false` and `true`.

6. C++ uses the data type `string` (defined in the standard library) to represent sequences of characters that are enclosed in quotes in a program. The quotes are not considered part of the data element and are not stored in memory.

7. We can write arithmetic expressions using type `float` and `int` data. The rules of evaluation are similar to the rules of algebra, but we must be aware of the data type of an expression. If an operator has only type int data for its operands, the result will be type `int`; otherwise, it will be type `float`. Be careful when evaluating expressions with type `int` and type `float`

data. If an operator has both kinds of data for its operands, C++ converts the type `int` value to a type `float` value before performing the operation.

8. C++ uses assignment statements to assign a computational result to a variable. Remember, the expression is evaluated first, following the rules summarized in point 7 above. Afterward, the result is assigned to the indicated variable. For a mixed-type assignment, C++ must perform a type conversion just before storing the expression result.

9. C++ uses the standard library class `iostream` for modeling input and output streams—sequences of characters associated with a file or device. C++ uses `cin` and `cout` to represent the keyboard and screen, respectively. C++ uses << as the stream insertion operator, also called the output operator because it writes data to an output stream. C++ uses >> as the stream extraction operator, also called the input operator because it reads data into a variable.

10. The syntax rules of C++ are precise and allow no exceptions. The compiler will be unable to translate C++ instructions that violate these rules. Remember to declare every identifier used as a constant or variable and to terminate program statements with semicolons.

The table below summarizes the new C++ constructs.

Summary of New C++ Constructs

Construct	Effect
Compiler Directive `#include <iostream>`	A compiler directive that causes the class `iostream` to be placed in the program where the directive appears.
Using Statement `using namespace std;`	Indicates that the program is using names defined in the region `namespace std` (standard).

Constant Declaration
`const float TAX = 25.00;`
`const char STAR = '*';`
	Associates the constant identifier `TAX` with the floating-point constant `25.00` and the constant identifier `STAR` with the character constant `'*'`.

Variable Declaration
`float x, y, z;`
`int me, it;`
	Allocates memory cells named `x`, `y`, and `z` for storage of floating-point numbers and cells named `me` and `it` for storage of integers.

Assignment Statement `distance = speed * time;`	Assigns the product of `speed` and `time` as the value of `distance`.
Input Statement `cin >> hours >> rate;`	Enters data into the variables `hours` and `rate`.
Output Statement `cout << "Net = " << net << endl;`	Displays the string `"Net = "` followed by the value of `net`. `endl` advances the screen cursor to the next line after this information is displayed.

Quick-Check Exercises

1. What value is assigned to x (type `float`) by the following statement?

```
x = 25 % 3 / 3.0;
```

2. What value is assigned to x by the statement below, assuming x is 10.0?

```
x = x / 20;
```

3. Show the form of the output line displayed by the following `cout` lines when `total` is 152.55.

```
cout << "The value of total is: " << endl;
cout << "$" << total << endl;
```

4. Show the form of the output line displayed by the following `cout` line when `total` is 352.74.

```
cout << "The final total is $" << total << endl;
```

5. Indicate which type of data you use to represent the following items: number of cats in your house; each initial of your name; your full name; the average temperature during the last month.

6. In which step of the software development method are the problem input and output data identified? In which step do you develop and refine the algorithm?

7. In reading two integers using the statement

```
cin >> m >> n;
```

what character should be entered following the first value? What should be entered after the second number?

8. When reading two characters using the input operator >>, does it matter how many blanks (if any) appear
   a. before the first character?
   b. between the first and second characters?
   c. after the second character?

9. How does the compiler determine how many and what type of data values are to be entered when an input statement is executed?

10. What is the syntactic purpose of the semicolon in a C++ program?

11. Does the compiler listing show syntax or run-time errors?

Review Questions

1. What type of information should be specified in the program header section comments?

2. Circle those identifiers below that are valid names for variables.

`salary`	`two fold`	`amount*pct`	`myprogram`
`1stTime`	`R2D2`	`firstTime`	`program`
`CONST`	`income#1`	`main`	`MAIN`
`Jane's`	`int`	`variable`	`PI`

3. What is illegal about the following declarations and assignment statement?

```
const float PI = 3.14159;
float c, r;
PI = c % (2 * r * r);
```

4. What do the following statements do? Which identifier cannot have its value changed?

   a. `float x = 3.5;`
   b. `string flower = "rose";`
   c. `const int FEETINMILE = 5280;`

5. List and define the rules of order of evaluation for arithmetic expressions.

6. Write the data requirements, necessary formulas, and algorithm for Programming Project 6 in the next section.

7. If the average size of a family is 2.8 and this value is stored in the variable `familySize`, provide the C++ statement to display this fact in a readable way (leave the display on the same line).

8. List three language-defined data types of C++.

9. Convert the program statements below to read and echo data in batch mode.

```
cout << "Enter three numbers separated by spaces:"
     << endl;
cin >> x >> y >> z;
cout << "Enter two characters: ";
cin >> ch1 >> ch2;
```

10. Write an algorithm that allows for the input of an integer value, triples it, adds your age to it (also an input), and displays the result.

11. Assuming `a` (10) and `b` (6) are type `int` variables, what are the values of the following expressions:

    a. `a / b`
    b. `a % b`
    c. `a * b`
    d. `a + b`

12. Assuming `a` and `b` are type `int` variables, which of the following expressions evaluate to the same value?

    a. `a + b * c`
    b. `(a + b) * c`
    c. `a + (b * c)`

13. Differentiate among syntax errors, run-time errors, and logic errors.

Programming Projects

1. Write a program to convert a temperature in degrees Fahrenheit to degrees Celsius.

   DATA REQUIREMENTS
   Problem Input
   `int fahrenheit` *// temperature in degrees Fahrenheit*
   Problem Output
   `float celsius` *// temperature in degrees Celsius*
   Formula
   celsius = (5/9) * (fahrenheit – 32)

2. Write a program to read two data items and print their sum, difference, product, and quotient.

   DATA REQUIREMENTS
   Problem Input
   `int x, y` *// two items*
   Problem Output
   `int sum` *// sum of* x *and* y
   `int difference` *// difference of* x *and* y
   `int product` *// product of* x *and* y
   `float quotient` *// quotient of* x *divided by* y

3. You have been watching the World's Strongest Man competition on television. The competitors perform amazing feats of strength such as pulling trucks along a course, flipping huge tires, lifting large objects

onto raised platforms, etc. You can tell the items are quite heavy but the British announcer keeps referring to the weights in stones. Write a program that converts the weight in stones to the equivalent weight in pounds where 1 stone is 14 pounds.

4. Write a program that prints your first initial as a block letter. (Hint: Use a 6×6 grid for the letter and print six strings. Each string should consist of asterisks (*) interspersed with blanks.)

5. Write a program that reads in the length and width of a rectangular yard (in meters) and the length and width of a rectangular house (in meters) placed in the yard. Your program should compute the time (in minutes) required to cut the lawn around the house. Assume the mowing rate in square meters per minutes is entered as a data item.

6. Write a program that reads and stores the numerators and denominators of two fractions as integer values. For example, if the numbers 1 and 4 are entered for the first fraction, the fraction is $^1/_4$. The program should print the product of the two fractions as a fraction and as a decimal value. For example, $^1/_4 * ^1/_2 = 1/8$ or 0.125.

7. Write a program that reads the number of years ago that a dinosaur lived and then computes the equivalent number of months, days, and seconds ago. Use 365.25 days per year. Test your program with a triceratops that lived 145 million years ago and a brontosaurus that lived 182 million years ago. (Hint: Use type double for all variables.)

8. Arnie likes to jog in the morning. As he jogs, he counts the number of strides he makes during the first minute and then again during the last minute of his jogging. Arnie then averages these two and calls this average the number of strides he makes in a minute when he jogs. Write a program that accepts this average and the total time Arnie spends jogging in hours and minutes and then displays the distance Arnie has jogged in miles. Assume Arnie's stride is 2.5 feet (also a data item). There are 5280 feet in a mile (a constant).

9. Write a program that reads a number of seconds between 0 and 18,000 (5 hours) and displays the hours, minutes, and seconds equivalent.

10. Redo Programming Project 6—but this time compute the sum of the two fractions.

11. The Pythagorean theorem states that the sum of the squares of the sides of a right triangle is equal to the square of the hypotenuse. For example, if two sides of a right triangle have lengths 3 and 4, then the hypotenuse must have a length of 5. The integers 3, 4, and 5 together form a Pythagorean triple. There is an infinite number of such triples. Given

two positive integers, m and n, where $m > n$, a Pythagorean triple can be generated by the following formulas:

$$side1 = m^2 - n^2$$
$$side2 = 2mn$$
$$hypotenuse = \sqrt{side1^2 + side2^2}$$

Write a program that reads in values for m and n and prints the values of the Pythagorean triple generated by the formulas above.

12. Write a program to compute the rate of growth, expressed as a percentage, of an insect population. Take as input the initial size of the population and its size one week later. Then compute the rate of growth and predict the size of the population in yet another week, assuming that growth continues at the same rate.

13. Write a program that reads in an integer that is greater than 1,000 and less than 1,000,000. Recall that the number must be entered without a comma. Display the number with a comma inserted where it belongs. (Hint: use the `%` operator.)

14. Write a program that asks a user to enter the distance of a trip in miles, the miles per gallon estimate for the user's car, and the average cost of a gallon of gas. Your program should calculate and display the number of gallons of gas needed and the estimated cost of the trip.

15. Write a program that dispenses change. The program should read the amount of the purchase and the amount paid and then display the number of dollars, quarters, dimes, nickels, and pennies given in change. (Hint: Convert the purchase amount and the amount paid to pennies.) Calculate the difference (`changeDue`). Use the `%` operator to determine how many dollars to pay. Subtract that number of pennies from `changeDue` and continue this process for quarters, dimes, etc. Your answer may be off by a penny. Do you have any idea why this might be the case? How could you correct this?

16. Write a program that calculates mileage reimbursement for a salesperson at a rate of $.35 per mile. Your program should interact with the user in this manner:

```
MILEAGE REIMBURSEMENT CALCULATOR
Enter beginning odometer reading=> 13505.2
Enter ending odometer reading=> 13810.6
You traveled 305.4 miles. At $0.35 per mile,
your reimbursement is $106.89.
```

17. Write a program to assist in the design of a hydroelectric dam. Prompt the user for the height of the dam and for the number of cubic meters of water that are projected to flow from the top to the

bottom of the dam each second. Predict how many megawatts (1MW = 10^6W) of power will be produced if 90% of the work done on the water by gravity is converted to electrical energy. Note that the mass of one cubic meter of water is 1000 kg. Use 9.80 meters/second^2 as the gravitational constant g. Be sure to use meaningful names for both the gravitational constant and the 90% efficiency constant. For one run, use a height of 170 m and flow of $1.30 \times\_ 10^3$ m^3/s. The relevant formula (w = work, m = mass, g = gravity, h = height) is $w = mgh$.

18. Hospitals use programmable pumps to deliver medications and fluids to intravenous lines at a set number of milliliters per hour. Write a program to output information for the labels the hospital pharmacy places on bags of I.V. medications indicating the volume of medication to be infused and the rate at which the pump should be set. The program should prompt the user to enter the quantity of fluid in the bag and the number of minutes over which it should be infused. Output the VTBI (volume to be infused) in ml and the infusion rate in ml/hr.

```
Sample run:
Volume to be infused (ml) => 100
Minutes over which to infuse => 20

VTBI: 100 ml
Rate: 300 ml/hr
```

19. Metro City Planners proposes that a community conserve its water supply by replacing all the community's toilets with low-flush models that use only 2 liters per flush. Assume that there is about 1 toilet for every 3 persons, that existing toilets use an average of 15 liters per flush, that a toilet is flushed on average 14 times per day, and that the cost to install each new toilet is $150. Write a program that would estimate the magnitude (liters/day) and cost of the water saved based on the community's population.

Answers to Quick-Check Exercises

1. `0.333333`
2. `0.5`
3. The value of total is:

   `$152.55`
4. The final total is `$352.74`
5. `int, char, string, float`

6. problem analysis, design
7. a blank, RETURN key
8. For (a) and (b), the number of blanks is irrelevant; any number is allowed or there may be no blanks; the input operator will simply skip all leading blanks and stop reading after the first nonblank is read. For (c), there is no need for any blanks; press the return key after entering the second character.
9. The number of values to be entered depends on the number of variables in the input list. The type of data is determined by the type of each variable.
10. It terminates a C++ statement.
11. syntax errors

Josée Lajoie

Josée Lajoie served as chair of the core language working group for the ANSI/ISO C++ Standard Committee and worked as a staff development analyst in the IBM Canada Laboratory C/C++ Compiler group. In addition, she coauthored The C++ Primer *and was a regular columnist on the evolution of the C++ language standard for the* C++ Report.

What is your educational background?
I have a bachelor's degree in electrical engineering, from L'École Polytechnique at the University of Montreal and I am currently pursuing a master's degree in computer graphics at the University of Waterloo. This master's degree includes both computer science and fine arts courses.

What was your first job in the computer industry? What did it entail?
I worked at IBM as a member of a team building a C compiler for IBM's mainframes. My work was to help develop the compiler front end—that is, the portion of the compiler that analyzes C programs for their syntactic and semantic validity. Over the course of a year and half, I developed with two other coworkers a C compiler front end to support the syntax and semantic requirements of the ANSI C standard.

Which person in the computer science field has inspired you?
Grace Hopper, for her pioneering spirit and her determination. Before computers were widely used, Grace Hopper believed that a much wider audience could use computers if tools existed that were both programmer-friendly and application-friendly. She worked on the development of early compilers to facilitate the use of computers in business and other nonscientific applications.

Do you have any advice for students learning C++?
I think people should learn by first using existing C++ libraries and code. They should start by writing small programs that use the features of an existing library. Then, when they become familiar with how to use the library, they should examine the library interface more closely, look at how the library features were designed and implemented, and if they can, read up on the design decisions behind the library implementation. The C++ Standard Library is a good example of a library that can be used in this way.

Do you have any advice for students entering the computer science field?
I would like to encourage students who are interested in studying computer science, but who do not see themselves as fitting the stereotypical description of a computer scientist as portrayed in the

media, to pursue their interests. The field needs people who are innovative, artistic, and good communicators just as much as it needs people who are good in math.

What do you think the future of the C++ language is?

I think the language itself will remain the way it is for very many years to come. The C++ language tool set is quite extensive and, I think, sufficient to implement a great variety of good libraries.

What I hope will happen is that the support offered by the C++ Standard Library will grow importantly over the next few iterations of the C++ standard. Five years after a standard has been published, individuals and companies can request changes to bring the standard closer to widely accepted industry practices. I hope that as more C++ libraries become available, they will be added to the list of C++ Standard Libraries and that users developing new products will be able to choose from a wide variety of domain-specific libraries available with every standard C++ implementation.

You were a key member of the ANSI/ISO C++ Standard committee. Why was it important to standardize C++?

Standardization is important for users, especially for users writing applications that must be ported to a great variety of architectures. A standard gives these users the guarantee that the code they are writing on one platform will compile and run on another platform where standard C++ is supported, provided that their implementation follows the rules prescribed in the standard. Similarly with the library, the standard guarantees that the standard libraries used by their implementation will be available on other platforms where the C++ standard library is supported and that they won't have to rewrite their code to use another, sometimes quite different, system-specific library. Having a standard means that implementers can port their applications faster to other platforms.

What were the main changes to the C++ definition?

The most important change to the original definition of C++ is the addition of the Standard Template Library, which is a good library not only because it provides important basic tools such as containers and algorithms that programmers use very often, but also because it is a very well-designed piece of software that shows better than anything else the power of C++ and how it can be used to write very good libraries.