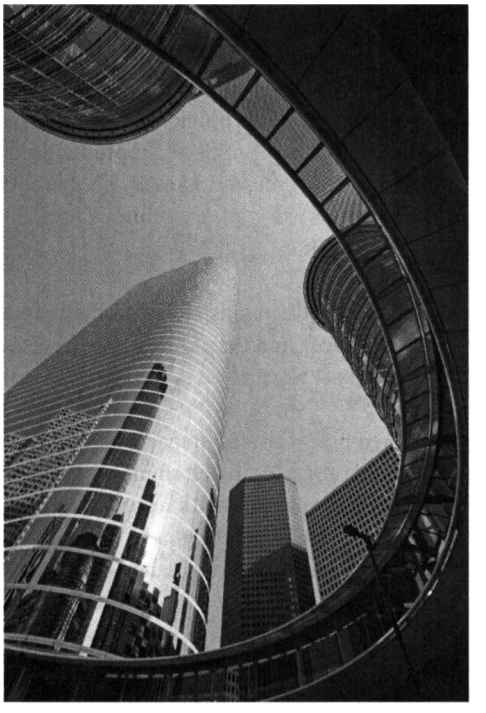
Data Abstraction & Problem Solving with C++

Walls And Mirrors

Seventh Edition



Frank M. Carrano
University of Rhode Island

Timothy M. Henry
New England Institute of Technology

PEARSON

Boston Columbus Indianapolis Hoboken New York San Francisco
Amsterdam Cape Town Dubai London Madrid Milan Munich Paris Montreal Toronto
Delhi Mexico City Sao Paulo Sydney Hong Kong Seoul Singapore Taipei Tokyo

Vice President and Editorial Director, ECS: Marcia J. Horton
Executive Editor: Tracy Johnson (Dunkelberger)
Editorial Assistant: Kristy Alaura
Program Manager: Carole Snyder
Project Manager: Robert Engelhardt
Media Team Lead: Steve Wright
R&P Manager: Rachel Youdelman
R&P Senior Project Manager: William Opaluch
Senior Operations Specialist: Maura Zaldivar-Garcia
Inventory Manager: Meredith Maresca
Marketing Manager: Demetrius Hall
Product Marketing Manager: Bram Van Kempen
Marketing Assistant: Jon Bryant
Cover Designer: Marta Samsel
Cover Art: © Jeremy Woodhouse/Ocean/Corbis
Full-Service Project Management: John Orr, Cenveo Publisher Services / Nesbitt Graphics, Inc.

Pearson Education Ltd., London
Pearson Education Australia Ply. Ltd., Sydney
Pearson Education Singapore, Pte. Ltd.
Pearson Education North Asia Ltd., Hong Kong
Pearson Education Canada, Inc., Toronto
Pearson Education de Mexico, S.A. de C.V.
Pearson Education–Japan, Tokyo
Pearson Education Malaysia, Pte. Ltd.
Pearson Education, Inc., Hoboken, New Jersey

Library of Congress Cataloging-in-Publication Data on file

10 9 8 7 6 5 4 3 2 1

PEARSON www.pearsonhighered.com

ISBN-10: 0-13-446397-8
ISBN 13: 978-0-13-446397-1

Appendix

A

Review of C++ Fundamentals

Contents

Prerequisite

Knowledge of a modern programming language

This book assumes that you already know how to write programs in a modern programming language. If that language is C++, you can probably skip this appendix, returning to it for reference as necessary. If instead you know a language such as Python, Java, or C, this appendix will introduce you to C++. Also, Appendixes K and L compare C++ to Java and Python, respectively.

It isn't possible to cover all of C++ in these pages. Instead this appendix focuses on the parts of the language used in this book. First we look at the basics of data types, variables, expressions, operators, and simple input/output. We continue with functions, decision constructs, looping constructs, arrays, and strings. Various C++ Interludes, which appear throughout the book as needed, will cover classes, pointers, exceptions, iterators, and the Standard Template Library.

A.1 Language Basics

Let's begin with the aspects of the language that allow you to perform simple computations. For example, the C++ program in Figure A-1 computes the volume of a sphere. Running this program produces the following output, where the user's response appears in blue:

```
Enter the radius of the sphere: 19.1
The volume of a sphere of radius 19.1 is 29186.927734
```

A typical C++ program consists of several modules, some of which you write and some of which you use from standard libraries. C++ provides a **source-inclusion facility**, which allows the system to *include* the contents of a file automatically at a specified point in a program before the program is compiled. For example, our sample program uses a standard library to perform input and output operations. The first line of this program is an `include` **directive** that names a **standard header** `iostream`, which enables the program to use input and output statements.

Each C++ program must contain a function `main`

A C++ program is a collection of functions, one of which must be called `main`. Program execution always begins with the function `main`. The following paragraphs provide an overview of the basics of C++ and refer to the simple program in Figure A-1 by line number. Note that the only function this simple program contains is `main`.

A.1.1 Comments

Each comment line begins with two slashes

Each comment line in C++ begins with two slashes `//` and continues until the end of the line. You can also begin a multiple-line comment with the characters `/*` and end it with `*/`. However,

FIGURE A-1 A simple C++ program

```
1. Enables input and output ---------->  #include <iostream>
2. Begins the function main ---------->  int main()
3. Begins body of function ----------->  {
4. A comment ------------------------->     // Computes the volume of a sphere of a given radius
5. Defines a constant ---------------->     const double PI = 3.14159;
6. Declares a variable --------------->     double radius;

7. Displays a prompt to the user ----->     std::cout << "Enter the radius of the sphere: ";
8. Reads radius ---------------------->     std::cin  >> radius;

9. Declares and computes volume ---->       double volume = 4 * PI * radius * radius * radius / 3;
10. Displays results ----------------->     std::cout << "The volume of a sphere of radius "
11. Statement continues -------------->               << radius << " inches is " << volume
12. Statement continues -------------->               << " cubic inches.\n";
13. Normal program termination ------->     return 0;
14. Ends body of function ------------>  }  // end program
```

a comment that begins with `/*` and ends with `*/` cannot contain another comment that begins with `/*` and ends with `*/`.

Appendix I talks about documentation comments that begin with `/**` and end with `*/`. We use this style of comment in this book at the beginning of classes, methods, and functions.

A.1.2 Identifiers and Keywords

A C++ **identifier** is a sequence of letters, digits, and underscores that must begin with either a letter or an underscore. C++ distinguishes between uppercase and lowercase letters, so be careful when typing identifiers.

C++ is case-sensitive

You use identifiers to name various parts of the program. Certain identifiers, however, are reserved by C++ as **keywords**, and you should not use them for other purposes. A list of all C++ reserved keywords appears inside the cover of this book. The keywords that occur within C++ statements in this book appear in color.

A.1.3 Primitive Data Types

The primitive data types in C++ are organized into four categories: boolean, character, integer, and floating point. With the exception of boolean, each category contains several data types. For most applications, you can use

`bool`	for boolean values
`char`	for character values
`int`	for integer values
`double`	for floating-point values

FIGURE A-2 Primitive data types

Category	Available Data Types by Category		
Boolean	`bool`		
Character	`char`	`signed char`	`unsigned char`
Signed integer	`short`	`int`	`long`
Unsigned integer	`unsigned short`	`unsigned`	`unsigned long`
Floating point	`float`	`double`	`long double`

Most of the data types are available in several forms and sizes. Although you will probably not need more than the four types given previously, Figure A-2 lists the available primitive data types for your reference.

A boolean value can be either true or false. Characters are represented by their ASCII integer values, which are listed in Appendix J. Integer values are either signed, such as −5 and +98, or unsigned, such as 5 and 98. The **floating-point types** are used for real numbers that have both an integer portion and a fractional portion. Boolean, character, and integer types are called **integral types**. Integral and floating-point types are called **arithmetic types**.

The size of a data type affects the range of its values. For example, a `long` integer can have a larger magnitude than a `short` integer. The sizes of and therefore the specific ranges for—a data type depend on the particular computer and version of C++ that you use. C++, however, provides a way to determine these ranges, as you will see later in Section A.1.6.

A.1.4 Variables

A variable, whose name is a C++ identifier, represents a memory location that contains a value of a particular data type. You declare a variable's data type by preceding the variable name with the data type, as in

```
double radius; // Radius of a sphere
```

Note that you can write a comment on the same line to describe the purpose of the variable.

This declaration is also a definition in that it assigns memory for the variable `radius`. The memory, however, has no particular initial value and so is said to be uninitialized. The program in Figure A-1 declares `radius` without an initial value and later reads its value by using `std::cin >> radius`.

When possible, avoid uninitialized variables

When possible, you should avoid uninitialized variables of a primitive data type. That is, you should initialize a variable when you first declare its data type or, alternatively, declare a variable's data type when you first assign it a value. For example, `volume` appears for the first time in line 9 of Figure A-1 in the statement

```
double volume = 4 * PI * radius * radius * radius / 3;
```

Because we did not declare `volume`'s data type earlier in the program—thus avoiding an uninitialized value—we declare its data type *and* assign it a value in the same statement.

A.1.5 Literal Constants

You use **literal constants** to indicate particular values within a program. The 4 and 3 in line 9 of Figure A-1 are examples of literal constants that are used within a computation. You can also use a literal constant to initialize the value of a variable. For example, you use `true` and `false` as the values of a boolean variable, as mentioned previously.

You write decimal integer constants without commas, decimal points, or leading zeros. The default data type of such a constant is either `int`, if small enough, or `long`.

Do not begin a decimal integer constant with zero

You write floating-point constants, which have a default type of `double`, with a decimal point. You can specify an optional power-of-10 multiplier by writing `e` or `E` followed by the power of 10. For example, `1.2e-3` means 1.2×10^{-3}.

Character constants are enclosed in single quotes—for example, `'A'` and `'2'`—and have a default type of `char`. You write a literal character string as a sequence of characters enclosed in double quotes.

Several characters have names that use a backslash notation, as given in Figure A-3. This notation is useful when you want to embed one of these characters within a literal character string. For example, the program in Figure A-1 uses the new-line character `\n` in the string `"cubic inches.\n"` to end the line of output. Any additional output will appear on the next line. You will see this use of `\n` in the discussion of output later in this appendix. You also use the backslash notation to specify either a single quote as a character constant (`'\''`) or a double quote within a character string.

Programming Tip: Do not begin a decimal integer constant with zero. A constant that begins with zero is either an octal constant or a hexadecimal constant.[1]

FIGURE A-3 Some special character constants

Constant	Name
`\n`	New line
`\t`	Tab
`\'`	Single quote
`\"`	Double quote
`\0`	Zero

A.1.6 Named Constants

Unlike variables, whose values can change during program execution, **named constants** have values that do not change. The declaration of a named constant is like that of an initialized variable, but the keyword `const` precedes the data type. For example, the statement

The value of a named constant does not change

```
const double PI = 3.14159;
```

declares `PI` as a named floating constant, as is the case in the sample program in Figure A-1. Once a named constant such as `PI` is declared, you can use it, but you cannot assign it another value. By using named constants, you make your program both easier to read and easier to modify.

Named constants make a program easier to read and modify

The standard header file `climits` contains named constants such as `INT_MIN` and `LONG_MAX` that specify installation-dependent maximum and minimum values for the integral data types. Likewise, the standard header file `cfloat` contains named constants that specify installation-dependent maximum and minimum values for the floating data types. You use the `include` directive to gain access to these header files.

[1] Octal and hexadecimal constants are also available, but they are not used in this book. An octal constant begins with `0`, a hex constant with `0x` or `0X`.

A.1.7 Enumerations

Enumeration provides another way to name constants

Enumeration provides another way to name integer constants. For example, the statement

```
enum {SUN, MON, TUE, WED, THU, FRI, SAT};
```

is equivalent to the statements

```
const int SUN = 0;
const int MON = 1;
. . .
const int SAT = 6;
```

By default, the values assigned to the constants---called **enumerators**—begin with zero and are consecutive. You can, however, assign explicit values to any or all of the enumerators, as in

```
enum {PLUS = '+', MINUS = '-'};
```

You can create an integral data type by naming an enumeration

By naming an enumeration, you create a distinct integral type. For example,

```
enum Season {WINTER, SPRING, SUMMER, FALL};
```

creates a type `Season`. The variable `whichSeason`, declared as

```
Season whichSeason;
```

can have values `WINTER`, `SPRING`, `SUMMER`, or `FALL`. This use of named enumerations instead of `int` can make your program easier to understand.

A.1.8 The `typedef` Statement

You use the `typedef` statement to give another name to an existing data type. In this way, you can make your program easier to modify and to read. For example, the statement

The `typedef` statement gives another name to an existing data type, making your program easier to change

```
typedef double Real;
```

declares `Real` as a synonym for `double` and allows you to use `Real` and `double` interchangeably.

Suppose that you revise the program in Figure A-1 by using `Real` as follows:

```
int main()
{
   typedef double Real;
   const Real PI = 3.14159;
   Real radius = 0.0;

   std::cout << "Enter the radius of the sphere: ";
   std::cin >> radius;
   Real volume = 4 * PI * radius * radius * radius / 3;
   . . .
```

At first glance, this program does not seem to be more advantageous than the original version, but suppose that you decide to increase the precision of your computation by declaring `PI`, `radius`, and `volume` as `long double` instead of `double`. In the original version of the program (Figure A-1), you would have to locate and change each occurrence of `double` to `long double`. In the revised program, you simply change the `typedef` statement to

```
typedef long double Real;
```

The `typedef` statement does not create a new data type

Realize that `typedef` does not create a new data type, but simply declares a new name for an existing data type. A new data type requires more than a name; it requires a set of operations. C++, however, does provide a way to create your own data types, as described in C++ Interlude 1.

A.1.9 Assignments and Expressions

You form an **expression** by combining variables, constants, operators, and parentheses. The assignment statement

```
volume = 4 * PI * radius * radius * radius / 3;
```

assigns to the previously declared variable `volume` the value of the arithmetic expression on the right-hand side of the **assignment operator** =, assuming that `PI` and `radius` have values. The assignment statement

```
double volume = 4 * PI * radius * radius * radius / 3;
```

which appears in line 9 of Figure A-1, also declares `volume`'s data type, since it was not declared previously.

The various kinds of expressions that you can use in an assignment statement are discussed next.

Arithmetic expressions. You can combine variables and constants with arithmetic operators and parentheses to form arithmetic expressions. The arithmetic operators are

- `+` Binary add or unary plus
- `-` Binary subtract or unary minus
- `*` Multiply
- `/` Divide
- `%` Modulo (remainder after division)

Operators have a set precedence

The operators `*`, `/`, and `%` have the same **precedence**,[2] which is higher than that of `+` and `-`; unary operators[3] have a higher precedence than binary operators. The following examples demonstrate operator precedence:

`a - b / c`	means $a - (b/c)$	(precedence of / over −)
`-5 / a`	means $(-5)/a$	(precedence of unary operator −)
`a / -5`	means $a/(-5)$	(precedence of unary operator −)

Operators are either left- or right-associative

Arithmetic operators and most other operators are left-associative. That is, operators of the same precedence execute from left to right within an expression. Thus,

```
a / b * c
```

means

```
(a / b) * c
```

The assignment operator and all unary operators are right-associative, as you will see later. You can use parentheses to override operator precedence and associativity.

Relational and logical expressions. You can combine variables and constants with parentheses; with the relational, or comparison, operators `<`, `<=`, `>=`, and `>`; and with the equality operators `==` (equal to) and `!=` (not equal to) to form a relational expression. Such an expression is true or false according to the validity of the specified relation. For example, the expression `5 == 4` is false because 5 is not equal to 4. Note that equality operators have a lower precedence than relational operators.

Logical expressions are evaluated from left to right

You can combine variables and constants of the arithmetic types, relational expressions, and the logical operators `&&` (and) and `||` (or) to form logical expressions, which are either true or false. C++ evaluates logical expressions from left to right and stops as soon as the value of

[2] A list of all C++ operators and their precedences appears inside the cover of this book.

[3] A unary operator requires only one operand; for example, the - in -5. A binary operator requires two operands; for example, the + in 2 + 3.

Sometimes the value of a logical expression is apparent before it is completely examined

the entire expression is apparent; that is, C++ uses **short-circuit evaluation**. For example, C++ determines the value of each of the following expressions without evaluating `(a < b)`:

```
(5 -- 4) && (a < b) // False since (5 == 4) is false
(5 == 5) || (a < b) // True since (5 == 5) is true
```

Programming Tip: Remember that = is the assignment operator; == is the equality operator.

Conditional expressions. The expression

expression$_1$ `?` *expression*$_2$ `:` *expression*$_3$

has the value of either *expression*$_2$ or *expression*$_3$ according to whether *expression*$_1$ is true or false, respectively. For example, the statement

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

assigns the larger of `a` and `b` to `larger`, because the expression `a > b` is true if `a` is larger than `b` and false if not.

Conversions from one data type to another occur during both assignment and expression evaluation

Implicit type conversions. Automatic conversions from one data type to another can occur during assignment and during expression evaluation. For assignments, the data type of the expression on the right-hand side of the assignment operator is converted to the data type of the item on the left-hand side just before the assignment occurs. Floating-point values are truncated—not rounded—when they are converted to integral values.

During the evaluation of an expression, any values of type `char` or `short` are converted to `int`. Similarly, any enumerator value is converted to `int` if `int` can represent all the values of that particular `enum`; otherwise, it is converted to `unsigned`. These conversions are called **integral promotions**. After these conversions, if the operands of an operator differ in data type, the data type that is lower in the following hierarchy is converted to one that is higher (`int` is lowest):

`int` → `unsigned` → `long` → `unsigned long` → `float` → `double` → `long double`

For example, if `a` is `long` and `b` is `float`, `a + b` is `float`. Only a copy of `a`'s `long` value is converted to `float` prior to the addition, so that the value stored at `a` is unchanged.

Use a static cast to convert explicitly from one data type to another

Explicit type conversions. You can explicitly convert from one data type to another by using a **static cast**, with the following notation:

`static_cast<`*type*`>(`*expression*`)`

which converts *expression* to the data type *type*. For example, `static_cast<int>(14.9)` converts the `double` value 14.9 to the `int` value 14. Thus, the sequence

```
double volume = 14.9;
std::cout << static_cast<int>(volume);
```

displays 14 but does not change the value of `volume`.

Other assignment operators. In addition to the assignment operator =, C++ provides several two-character assignment operators that perform another operation before assignment. For example,

`a += b` means `a = a + b`

Other operators, such as `-=`, `*=`, `/=`, and `%=`, have analogous meanings.

Two more operators, `++` and `--`, provide convenient incrementing and decrementing operations:

The operators ++ and -- are useful for incrementing and decrementing a variable

`a++` means `a += 1`, which means `a = a + 1`

Similarly,

`a--` means `a -= 1`, which means `a = a - 1`

The operators `++` and `--` can either follow their operands, as you just saw, or precede them. Although `++a`, for instance, has the same effect as `a++`, the results differ when the operations are combined with assignment. For example,

`b = ++a` means `a = a + 1; b = a`

Here, the `++` operator acts on `a` *before* assigning `a`'s new value to `b`. In contrast,

`b = a++` means `b = a; a = a + 1`

The assignment operator assigns `a`'s old value to `b` before the `++` operator acts on `a`. That is, the `++` operator acts on `a` *after* the assignment. The operators `++` and `--` are often used within loops and with array indices, as you will see later in this appendix. When we use these operators with arithmetic variables, we write the operator after the variable.

In addition to the operators described here, C++ provides several other operators. A summary of all C++ operators and their precedences appears inside the cover of this book.

A.2 Input and Output Using `iostream`

A typical C++ program reads its input from a keyboard and writes its output to a display. Such input and output consist of **streams**, which are simply sequences of characters that either come from or go to an input or output (I/O) device.

The data type of an input stream is `istream`, and the data type of an output stream is `ostream`. The `iostream` library provides these data types and three default stream variables: `cin` for the standard input stream, `cout` for the standard output stream, and `cerr` for the standard error stream, which also is an output stream. Your program gains access to the `iostream` library by including the `iostream` header file. This section provides a brief introduction to simple input and output.

A.2.1 Input

The input operator >> reads from an input stream

C++ provides the input operator `>>` to read integers, floating-point numbers, and characters into variables whose data types are any of the primitive data types. The input operator has the input stream as its left operand and the variable that will contain the value read as its right operand. Thus,

```
std::cin >> x;
```

reads a value for `x` from the standard input stream. The `>>` operator is left-associative. Thus,

```
std::cin >> x >> y
```

means

```
(std::cin >> x) >> y
```

That is, both of these expressions read characters for `x` from the input stream and then read subsequent characters for `y`.

The input operator >> skips whitespace

The input operator >> skips whitespace, such as blanks, tabs, and new-line characters, that might occur between values in the input data line. For example, after the program segment

```
int ia = 0;
int ib = 0;
double da = 0;
double db = 0;
std::cin >> ia >> da >> ib;
std::cin >> db;
```

reads the data line

```
21  -3.45  -6 475.1e-2 <eol>
```

the variable `ia` contains 21, `da` contains −3.45, `ib` contains −6, and `db` contains 4.751. A subsequent attempt to read from `cin` will look beyond the end of the line (*<eol>*) and read from the next data line, if one exists. An error occurs if no data exists for a corresponding variable processed by >> or if the variable's data type does not match the type of the data available. For example, after the previous program segment reads the data line

```
-1.23  456.1e-2  -7 8 <eol>
```

the variable `ia` contains −1, `da` contains 0.23, `ib` contains 456, and `db` contains 0.001. The rest of the data line is left for a subsequent read, if any. As another example, if the segment attempts to read a data line that begins with `.21`, the read would terminate because `ia` is `int` and `.21` is not.

An expression such as `std::cin >> x` has a value after the read operation takes place. If the operation is successful, this value is true; otherwise the value is false. You can examine this value by using the selection and iteration statements that are described later in this appendix.

You can also use the >> operator to read individual characters from the input stream into character variables. Again, any whitespace is skipped. For example, after the program segment

```
char ch1 = '';
char ch2 = '';
char ch3 = '';
std::cin >> ch1 >> ch2 >> ch3;
```

reads the data line

```
xy  z
```

`ch1` contains `'x'`, `ch2` contains `'y'`, and `ch3` contains `'z'`.

You can read whitespace when reading individual characters into character variables by using the C++ method `get`. Either of the statements

Use `get` to read whitespace

```
std::cin.get(ch1);
```

or

```
ch1 = std::cin.get();
```

reads the next character, even if it is a blank, a tab, or a new-line character, from the input stream into the `char` variable `ch1`.

Section A.7, later in this appendix, describes how to read character strings.

A.2.2 Output

The output operator << writes to an output stream

C++ provides the output operator << to write character strings and the contents of variables whose data types are any of the primitive ones. For example, the program segment

```
int count = 5;
double average = 20.3;
std::cout << "The average of the " << count
          << " distances read is " << average
          << " miles.\n";
```

produces the following output:

```
The average of the 5 distances read is 20.3 miles.
```

Subsequent output will appear on the next line. Like the input operator, the output operator is left-associative. Thus, the previous statements append the string `"The average of the "` to the output stream, then append the characters that represent the value of `count`, and so on.

You need to explicitly introduce new-line characters and whitespace where desired in a program's output

Note the use of the new-line character `\n`, which you can conveniently embed within a character string. Observe also that the output operator does not automatically introduce whitespace between values that are written; you must do so explicitly. The following statements provide another example of this:

```
int x = 2;
int y = 3;
char ch = 'A';
std::cout << x << y << ch << "\n"; // Displays 23A
```

Although you can use the output operator to display individual characters, you can also use the `put` method for this task. Further, you can specify a character either as a `char` variable or in ASCII. Thus, the statements

```
char ch = 'a';
std::cout.put(ch);    // Displays a
std::cout.put('b');   // Displays b
std::cout.put(99);    // Displays c, which is 99 in ASCII
std::cout.put(ch+3);  // Displays d
std::cout.put('\n');  // Carriage return
```

display `abcd` followed by a carriage return.

Section A.7, later in this appendix, provides further information about writing character strings.

A.2.3 Manipulators

C++ enables you to gain more control over the format of your output and the treatment of whitespace during input than the previous discussion has indicated. Most of these techniques apply to the format of output.

Suppose, for example, that you have computed your grade point average and you want to display it with one digit to the right of the decimal point. If the floating variable `gpa` contains 4.0, the statement

```
std::cout << "My GPA is " << gpa << "\n";
```

writes 4 without a decimal point. A number of **manipulators** affect the appearance of your output. You can use these with `cout`:

Use manipulators to specify the appearance of a program's output

```
std::cout << std::manipulator;
```

where *manipulator* has any of the values listed in Figure A-4. A manipulator is a predefined value or function that you use with the input and output operator. For example,

```
std::cout << std::showpoint;
```

uses the `showpoint` manipulator and causes all floating-point output to appear with a decimal point.

FIGURE A-4 Stream manipulators

Manipulator	Meaning
`endl`	Insert new line and flush stream
`fixed`	Use fixed decimal point in floating-point output
`left`	Left-align output
`right`	Right-align output
`scientific`	Use exponential (e) notation in floating-point output
`setfill(f)`	Set fill character to `f`
`setprecision(n)`	Set floating-point precision to integer `n`
`setw(n)`	Set field width to integer `n`
`showpoint`	Show decimal point in floating-point output
`showpos`	Show + with positive integers
`ws`	Extract whitespace characters (input only)

Even if you use the `showpoint` manipulator, `gpa` will likely appear as 4.00000 instead of 4.0. You can specify the number of digits that appear to the right of the decimal point by using the manipulator function `setprecision`, and you can insert a new-line character and flush the output stream by using the manipulator `endl`. Thus,

```
std::cout << std::showpoint;
std::cout << std::setprecision(1) << gpa << std::endl;
```

displays 4.0 followed by a carriage return.

The effect of `setprecision` on the output stream remains until another `setprecision` is encountered. Except for `setprecision`, however, a manipulator affects the appearance of only the next characters on which << (or >>) operates. For example,

```
std::cout << std::right; // Right-align output
std::cout << "abc" << std::setw(6) << "def" << "ghi";
```

displays

```
abc  defghi
```

Although manipulator values, such as `endl`, are available when you include `iostream` in your program, you must also include `iomanip` to use any of the manipulator functions `setfill`, `setprecision`, and `setw`. Note that all of the manipulators are in the C++ standard namespace.

A.3 Functions

A C++ program is a collection of functions

As was mentioned earlier in this appendix, a C++ program is a collection of functions. Usually, each function should perform one well-defined task. For example, the following function returns the larger of two integers:

A function definition implements a function's task

```
int maxOf(int x, int y)
{
   if (x > y)
      return x;
   else
      return y;
   // end maxOf
```

A function definition, like the one just given, has the following form:

```
type name(parameter-declaration-list)
{
   body
}
```

The portion of the definition before the left brace specifies a return type, the function name, and a list of **parameters**. The part of the definition that appears between the braces is the function's body.

The return type of a **valued function**—one that returns a value—is the data type of the value that the function will return. The body of a valued function must contain a statement of the form

A valued function must use `return` to return a value

```
return expression;
```

where *expression* has the value to be returned.

Each parameter represents either an input to or an output from the function. You declare a parameter by writing a data type and a parameter name, separating it from other parameter declarations with a comma, as in

```
int x, int y
```

When you **call**, or **invoke**, the function `maxOf`, you pass it **arguments** that correspond to the parameters with respect to number, order, and data type. For example, the following statements contain two calls to `maxOf`:

When you call a function, you pass it arguments that correspond to the parameters in number, order, and data type

```
int a = 0;
int b = 0;
int c = 0;
std::cin >> a >> b >> c;

int largerAB = maxOf(a, b);
std::cout << "The largest of " << a << ", " << b << ", "
          << " and " << c << " is " << maxOf(largerAB, c) << ".\n";
```

As written, the definition of `maxOf` indicates that its arguments are **passed by value**. That is, the function makes local copies of the values of the arguments—`a` and `b`, for example—and uses these copies wherever `x` and `y` appear in the function definition. Thus, the function cannot alter the arguments that you pass to it. This restriction is desirable in this example because `x` and `y` are input parameters, which `maxOf` does not change.

An argument passed by value is copied within the function

Alternatively, arguments can be **passed by reference**. The function does not copy such arguments; rather, it references the argument locations whenever the parameters appear in the function's definition. This allows a function to change the values of the arguments, thus implementing output parameters.

An argument passed by reference is not copied but is accessed directly within the function

For example, consider the following function `computeMax`:

```
void computeMax(int x, int y, int& larger)
{
   larger = ((x > y) ? x : y);
}  // end computeMax
```

`computeMax` is a **void function** instead of a valued function. That is, its return type is `void`, and it does not return a value by using a `return` statement.[4] Instead, `computeMax` returns the larger of `x` and `y` in the output parameter `larger`. The `&` that follows `larger`'s data type `int` indicates that `larger`

A void function does not use `return` to return a value

[4] Whereas valued functions must contain a statement of the form `return` *expression*, void functions cannot contain such a statement. A void function can, however, contain `return` without an expression. Such a statement causes the function to return to the statement that follows its call. This book does not use `return` with void functions.

An output argument should be a reference argument

is a **reference parameter**. Thus, `computeMax` will access and alter the argument that corresponds to `larger`, whereas the function will make and use copies of the values of the arguments that correspond to the **value parameters** `x` and `y`.

The following statements demonstrate how to invoke `computeMax`:

```
int a = 0;
int b = 0;
int largerAB = 0;
std::cin >> a >> b;
computeMax(a, b, largerAB);
std::cout << "The larger of " << a << " and " << b
          << " is " << largerAB << ".\n";
```

An input argument should be either a value argument or a constant reference argument

If a function's input argument is a large object, like the objects you will encounter in this book, you might not want the function to copy it. Thus, you would not pass the argument by value. Because it is an input argument, however, you do not want the function to be able to alter it. **A constant reference parameter** is a reference parameter that is tagged as `const`. The function uses the actual argument that is passed to such a parameter, not a copy of it, yet cannot modify it.

For example, for the function `f` that begins

```
void f(const int& x, int y, int& z)
```

An argument that is both an input to and an output from a function is passed by reference

`x` is a constant reference parameter, `y` is a value parameter, and `z` is a reference parameter. Here `x` and `y` are suitable as input parameters because `f` cannot change them, while `z` is an output parameter. Note that `z` can also be an input parameter. That is, the argument corresponding to `z` can both provide a value to the function and return a value from the function. Such arguments must be passed by reference.

> **Note:** Use reference parameters with caution, as you might inadvertently change an argument. On the other hand, constant reference parameters are safe to use.

If you write another function `f` that calls `computeMax`, you must either place the definition of `f` after the definition of `computeMax` or precede `f`'s definition with a **function declaration** for `computeMax`. For example, you can use either of the following statements to declare the function `computeMax`:

A function declaration ends with a semicolon

```
void computeMax(int x, int y, int& max);
```

or

```
void computeMax(int, int, int&);
```

A function declaration provides the data types of the function's parameters and its return type. Parameter names are optional in a function declaration, although they are helpful stylistically. However, parameter names are required in the function's definition. Although a function declaration ends with a semicolon, a semicolon does not appear in a function definition.

Declarations for each function usually appear at the beginning of a program

A typical C++ program contains a function declaration for every function used in the program. These declarations appear first in the program, usually with comments that describe each function's purpose, parameters, and assumptions. The program in Listing A-1 demonstrates the placement of a function declaration, function definition, and main function:

LISTING A-1 A program that contains a function declaration

```
#include <iostream>

/** Returns the larger of two given integers.
 @param x  An integer.
 @param y  An integer.
 @return  The larger of x and y. */
int maxOf(int x, int y);          // A function declaration

int main()
{
   int a = 0;
   int b = 0;
   std::cout << "Please enter two integers: ";
   std::cin >> a >> b;

   int largerAB = maxOf(a, b);
   std::cout << "The larger of " << a << " and " << b
             << " is " << largerAB << ".\n";
} // end main

// A function definition
int maxOf(int x, int y)
{
   return (x > y) ? x : y;
} // end maxOf
```

A.3.1 Standard Functions

Standard functions provide many common operations and require a specific header file

C++ provides many standard functions, such as the square root function `sqrt` and the input function `get`. Appendix H provides a summary of the standard functions and indicates which header file you need to include in your program to gain access to them. For example, the standard functions listed in Figure A-5 facilitate character processing and require the header file `cctype`. Thus, you need to include the statement

```
#include <cctype>
```

in your program when you want to use functions such as `isupper` and `toupper`. For the character variable `ch`, `isupper(ch)` is `true` if `ch` is an uppercase letter, and `toupper(ch)` returns the uppercase version of the letter `ch` without actually changing `ch`.

FIGURE A-5 A selection of standard functions

(a) Standard classification functions

Function	Returns true if ch is
`isalnum(ch)`	A letter or digit
`isalpha(ch)`	A letter
`isdigit(ch)`	A digit
`islower(ch)`	A lowercase letter
`isupper(ch)`	An uppercase letter

(b) Standard conversion functions

Function	Returns
`tolower(ch)`	Lowercase version of **ch**
`toupper(ch)`	Uppercase version of **ch**
`toascii(ch)`	**int** ASCII code for **ch**

A.4 Selection Statements

Selection statements allow you to choose among several courses of action according to the value of an expression. In this category of statements, C++ provides the `if` statement and the `switch` statement.

A.4.1 The `if` Statement

You can write an `if` statement in one of two ways:

An `if` statement has two basic forms

```
if (expression)
   statement1
```

or

Parentheses around the expression in an `if` statement are required

```
if (expression)
   statement1
else
   statement2
```

where *statement1* and *statement2* represent any C++ statement except a declaration. Such statements can be compound; a **compound statement**, or **block**, is a sequence of statements enclosed in braces. If the value of *expression* is true, *statement1* is executed. Otherwise, the first form of the `if` statement does nothing, whereas the second form executes *statement2*. Note that the parentheses around *expression* are required.

For example, the following `if` statements each compare the values of two integer variables `a` and `b`, and copy the larger value to the integer variable `largerAB`:

```
if (a > b)
   std::cout << a << " is larger than " << b << ".\n";
std::cout << "This statement is always executed.\n";
```

```
if (a > b)
{
   largerAB = a;
   std::cout << a << " is larger than " << b << ".\n";

else
{
   largerAB = b;
   std::cout << b << " is larger than " << a << ".\n";
   // end if

std::cout << largerAB << " is the larger value.\n";
```

You can nest `if` statements

You can nest `if` statements in several ways, since either *statement1* or *statement2* can itself be an `if` statement. The following example, which finds the largest of three integer variables `a`, `b`, and `c`, shows a common way to nest `if` statements:

```
if ((a >= b) && (a >= c))
   largest = a;
else if (b >= c) // a is not largest at this point
   largest = b;
else
   largest = c;
```

Note: An arithmetic expression whose value is not zero is treated as true; one having a value of zero is false.

A.4.2 The switch Statement

A switch statement provides a choice of several actions according to the value of an integral expression

When you must choose among more than two courses of action, the `if` statement can become unwieldy. If your choice is to be made according to the value of an integral expression, you can use a `switch` statement.

For example, the following statement assigns the number of days in a month to the previously defined integer variable `daysInMonth`. The `int` variable `month` designates the month as an integer from 1 to 12, and the boolean variable `leapYear` is true if the year is a leap year.

```
switch (month)
{
   // 30 days hath Sept., Apr., June, and Nov.
   case 9: case 4: case 6: case 11:
      daysInMonth = 30;
      break;

   // All the rest have 31
   case 1: case 3: case 5: case 7:
   case 8: case 10: case 12:
      daysInMonth = 31;
      break;

   // Except February
   case 2: // Assume leapYear is true if a leap year, else is false
      if (leapYear)
         daysInMonth = 29;
      else
         daysInMonth = 28;
      break;

   default:
      std::cout << "Incorrect value for month.\n";
}  // end switch
```

Parentheses must enclose the integral `switch` expression—`month`, in this example. The `case` labels have the form

`case` *expression*:

where *expression* is a constant integral expression. After the `switch` expression is evaluated, execution continues at the `case` label whose expression has the same value as the `switch` expression. Subsequent statements execute until either a `break` statement is encountered or the `switch` statement ends.

Without a break statement, execution of a case will continue into the next case

Unless you terminate a `case` with a `break` statement, execution of the `switch` statement continues. Although this action can be useful, omitting the `break` statements in the previous example would be incorrect.

If no `case` label matches the current value of the `switch` expression, the statements that follow the `default` label, if one exists, are executed. If no `default` exists, the `switch` statement exits.

A.5 Iteration Statements

C++ has three statements that provide for repetition by iteration, that is, loops: the `while`, `for`, and `do` statements. Each statement controls the number of times that another C++ statement—the body—is executed. The body cannot be a declaration and is often a compound statement.

A.5.1 The `while` Statement

A `while` statement executes as long as the expression is true

The general form of the `while` statement is

```
while (expression)
   statement
```

As long as the value of *expression* is true, *statement* is executed. Because *expression* is evaluated before *statement* is executed, it is possible that *statement* will not execute at all. Note that the parentheses around *expression* are required.

Suppose that you wanted to compute the sum of positive integers that you enter at the keyboard. Since the integers are positive, you can use a negative value or zero to indicate the end of the input. The following `while` statement accomplishes this task:

```
int nextValue = 0;
int sum = 0;

std::cin >> nextValue;
while (nextValue > 0)
{
   sum += nextValue;
   std::cin >> nextValue;
} // end while
```

If 0 was the first value read, the body of the `while` statement would not execute.

Recall that the expression `std::cin >> nextValue` is true if the input operation was successful and false otherwise. Thus, you could revise the previous statements as

```
int nextValue = 0;
int sum = 0;
while ( (std::cin >> nextValue) && (nextValue > 0) )
   sum += nextValue;
```

This loop control is difficult to maintain, and so we do not recommend it.

A.5.2 The `for` Statement

A `for` statement lists the initialization, testing, and updating steps in one location

The `for` statement provides for counted loops and has the general form

```
for (initialize; test; update)
   statement
```

where *initialize, test*, and *update* are expressions. Typically, *initialize* is an assignment expression that initializes a counter to control the loop. This initialization occurs only once. Then if *test*, which is usually a logical expression, is true, *statement* executes. The expression *update* executes next, usually incrementing or decrementing the counter. This sequence of events repeats, beginning with the evaluation of *test*, until the value of *test* is false.

For example, the following `for` statement displays the integers from 1 to `n`:

```
for (int counter = 1; counter <= n; counter++)
   std::cout << counter << " ";
std::cout << std::endl; // This statement is always executed
```

If `n` is less than 1, the `for` statement does not execute at all. Thus, the previous statements are equivalent to the following `while` loop:

```
int counter = 1;
while (counter <= n)
{
```

```
   std::cout << counter << " ";
   counter++;
}  // end while
std::cout << std::endl; // This statement is always executed
```

In general, the logic of a `for` statement is equivalent to

A `for` statement is equivalent to a `while` statement

```
initialize;
while (test)
{
   statement;
   update;
}
```

Note that in a `for` statement the first expression *initialize* must have either an arithmetic type or a pointer type.[5] Note that `char` in the following example is considered an arithmetic type:

```
for (char ch = 'z'; ch >= 'a'; ch--)
// ch ranges from 'z' to 'a'
```

The *initialize* and *update* portions of a `for` statement each can contain several expressions separated by commas, thus performing more than one action. For example, the following loop raises a floating-point value to an integer power by using multiplication:

```
// Floating-point power equals floating-point x raised to int n;
// assumes int expon
for (power = 1.0, expon = 1; expon <= n; expon++)
   power *= x;
```

Both `power` and `expon` are assigned values before the body of the loop executes for the first time. The comma here is an example of the **comma operator**, which evaluates its operand expressions from left to right.

When compared to a `while` statement, the `for` statement can make it easier to understand how the loop is controlled because the initialization, testing, and updating steps of the loop are consolidated into one statement. C++ programmers use `for` statements for loops that process collections or sequences of data.

For counted loops, a `for` statement is usually favored over the `while` statement

A.5.3 The do Statement

Use the `do` statement when you want to execute a loop's body at least once. Its general form is

A `do` statement loops at least once

```
do
   statement
while (expression);
```

Here, *statement* executes until the value of *expression* is false.

For example, suppose that you execute a sequence of statements and then ask the user whether to execute them again. The `do` statement is appropriate, because you execute the statements before you decide whether to repeat them:

```
char response;
do
{
   // A sequence of statements
```

[5] C++ Interlude 2 introduces pointer types.

```
   . . .
   std::cout << "Do it again?";
   std::cin >> response;
} while ( (response == 'Y') || (response == 'y') );
```

A.6 Arrays

An array contains data that has the same type

An array contains data items, or **entries**, that have the same data type. An array's memory locations, or **elements**, have an order: An array has a first element, a second element, and so on, as well as a last element. That is, an array has a finite, limited number of elements. Therefore, you must know the maximum number of elements needed for a particular array when you write your program and *before* you execute it. Because you can access the array elements directly and in any order, an array is a **direct access**, or **random access**, data structure.

You can access array elements directly and in any order

A.6.1 One-Dimensional Arrays

When you decide to use an array in your program, you must declare it and, in doing so, indicate the data type of its entries as well as its **size**, or **capacity**. The following statements declare a one-dimensional array, `maxTemps`, which contains the daily maximum temperatures for a given week:

```
const int DAYS_PER_WEEK = 7;
double maxTemps[DAYS_PER_WEEK];
```

The bracket notation `[]` declares `maxTemps` as an array. This array can contain at most seven floating-point values.

Use an index to specify a particular element in an array

An array index has an integer value greater than or equal to 0

You can refer to any of the floating-point entries in `maxTemps` directly by using an expression, which is called the **index**, or **subscript**, enclosed in square brackets. In C++, array indices must have integer values in the range 0 to *size* − 1, where *size* is the number of elements in the array. The indices for `maxTemps` range from 0 to `DAYS_PER_WEEK` − 1. For example, the fifth element in this array is `maxTemps[4]`. If `k` is an integer variable whose value is 4, `maxTemps[k]` is the fifth element in the array, and `maxTemps[k+1]` is the sixth element. Also, `maxTemps[k++]` accesses `maxTemps[k]` before adding 1 to `k`. Note that you use one index to refer to an element in a one-dimensional array.

Figure A-6 illustrates the array `maxTemps`, which at present contains only five temperatures. The last value in the array is in `maxTemp[4]`; the elements `maxTemps[5]` and `maxTemps[6]` are not initialized and therefore contain unknown values.

You can use enumerators as indices because they have integer values. For example, consider the following definition:

You can use an enumerator as an array index

```
enum Day {SUN, MON, TUE, WED, THU, FRI, SAT};
```

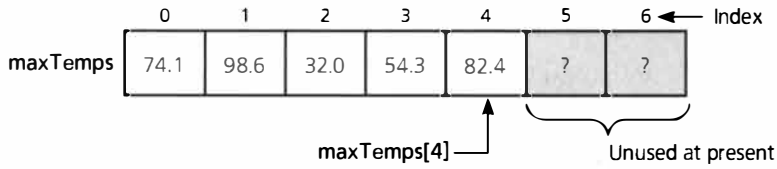
Given this definition, `maxTemps[THU]` has the same meaning as `maxTemps[4]`. You can also use the enumerators within a loop that processes an array, as in the following `for` statement:

```
for (Day dayIndex = SUN; dayIndex <= SAT; dayIndex++)
   std::cout << maxTemps[dayIndex] << std::endl;
```

Clearly, before you access an element of an array, you must assign it a value. You must assign values to array elements one at a time by using the previously described index notation. Note that, if `a` and `b` are arrays of the same type, the assignment `a = b` is illegal.[6]

[6] C++ enables you to define your own array data type and array operators so that this assignment would be valid. To do so, you need to use classes (C++ Interlude 1) and overloaded operators (C++ Interlude 6).

FIGURE A-6 A one-dimensional array of at most seven elements



The data type of `maxTemps` is a **derived type**, which is a type that you derive from the primitive types by using a declaration operator such as `[]`. Naming a derived type by using a `typedef` is often useful. Thus, you can write

```
const int DAYS_PER_WEEK = 7;
typedef double ArrayType[DAYS_PER_WEEK];
ArrayType maxTemps;
```

and make `ArrayType` available for use throughout your program.

Initialization. You can initialize the elements of an array when you declare it for the first time. For example,

You can initialize an array when you declare it

```
double maxTemps[DAYS_PER_WEEK] = {82.0, 71.5, 61.8, 75.0, 88.3};
```

initializes the first five elements of `maxTemps` to the values listed and the last two elements to zero.

Passing an array to a function. If you wanted a function that computed the average of the first *n* entries in a one-dimensional array, you could declare the function as

```
double getAverageTemp(double temperatures[], int n);
```

Because the compiler does not know the number of entries that the array can hold, you must also pass the function either the size of the array or the number of array entries to process. Traditionally, the array is listed as the first parameter and the number of entries as the second. You can invoke the function by writing, for example,

```
double avg = getAverageTemp(maxTemps, 5);
```

where `maxTemps` is the previously defined array.

An array is never passed to a function by value, regardless of how you write its parameter. *An array is always passed by reference.* This restriction avoids the copying of perhaps many array entries. Thus, the function `getAverageTemp` could modify the entries of the array `maxTemps`, even though it is an input to the function. To prevent such alteration, you can specify the array parameter as a constant reference parameter by preceding its type with `const`, as follows:

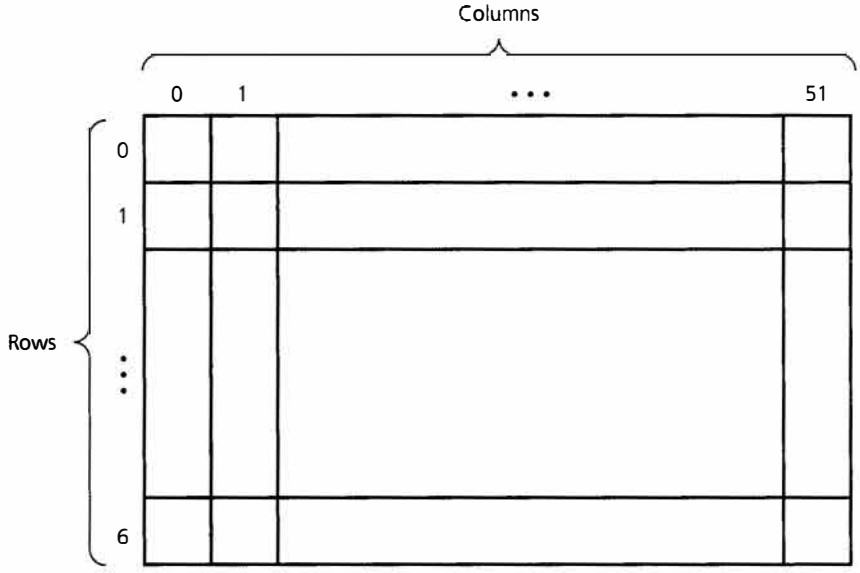
Arrays are always passed by reference to a function

```
double getAverageTemp(const double temperatures[], int n);
```

A.6.2 Multidimensional Arrays

You can use a one-dimensional array, which has one index, for a simple collection of data. For example, you can organize 52 temperatures linearly, one after another. A one-dimensional array of these temperatures can represent this organization.

FIGURE A-7 A two-dimensional array



An array can have more than one dimension

You can also declare multidimensional arrays. You use more than one index to designate an element in a multidimensional array. Suppose that you wanted to represent the minimum temperature for each day during 52 weeks. The following statements declare a two-dimensional array, `minTemps`:

```
const int DAYS_PER_WEEK = 7;
const int WEEKS_PER_YEAR = 52;
double minTemps[DAYS_PER_WEEK][WEEKS_PER_YEAR];
```

These statements specify the ranges for two indices: The first index can range from 0 to 6, while the second index can range from 0 to 51. Most people picture a two-dimensional array as a rectangular arrangement, or matrix, of elements that form rows and columns, as Figure A-7 indicates. The first dimension given in the declaration of `minTemps` is the number of rows. Thus, `minTemps` has 7 rows and 52 columns. Each column in this matrix represents the seven daily minimum temperatures for a particular week.

In a two-dimensional array, the first index represents the row, the second index represents the column

To reference an element in a two-dimensional array, you must indicate both the row and the column that contain the element. You make these indications of row and column by writing two indices, each enclosed in brackets. For example, `minTemps[1][51]` is the element in the 2nd row and the 52nd column. In the context of the temperature example, this element contains the minimum temperature recorded for the 2nd day (Monday) of the 52nd week. The rules for the indices of a one-dimensional array also apply to the indices of multidimensional arrays.

As an example of how to use a two-dimensional array in a program, consider the following program segment, which determines the smallest value in the previously described array `minTemps`. We use enumerators to reference the days of the week.

```
enum Day {SUN, MON, TUES, WED, THURS, FRI, SAT};

// Initially, assume the lowest temperature is first in the array
double lowestTemp = minTemps[0][0];
```

```
Day dayOfWeek = SUN;
int weekOfYear = 1;

// Search array for lowest temperature
for (int weekIndex = 0; weekIndex < WEEKS_PER_YEAR; weekIndex++)
{
   for (Day dayIndex = SUN; dayIndex <= SAT; dayIndex++)
   {
      if (lowestTemp > minTemps[dayIndex][weekIndex])
      {
         lowestTemp = minTemps[dayIndex][weekIndex];
         dayOfWeek = dayIndex;
         weekOfYear = weekIndex + 1;
      } // end if
   } // end for
} // end for

// At this point, lowestTemp is the smallest value in minTemps and
// occurs on the day and week given by dayOfWeek and weekOfYear,
// that is, lowestTemp == minTemps[dayOfWeek][weekOfYear - 1].
```

Although you can declare arrays with more than two dimensions, generally more than three dimensions is unusual. The techniques for working with such arrays, however, are analogous to those for two-dimensional arrays.

Programming Tip: When referencing an element of a multidimensional array, do not use comma-separated indices. For example, `myArray[3,6]` does not reference the array element `myArray[3][6]`. The expression `3,6` is a **comma expression** whose value is that of the last item listed, namely 6. Thus, although `myArray[3,6]` is legal, its meaning is `myArray[6]`, which references the element `myArray[0][6]`.

Initialization. You can initialize the elements of a two-dimensional array just as you initialize a one-dimensional array. You list the initial values row by row. For example, the statement

```
int x[2][3] = { {1, 2, 3},
                {4, 5, 6} };  // 2 rows, 3 columns
```

initializes the two-dimensional array `x` so that it appears as

```
1   2   3
4   5   6
```

That is, the statements initialize the elements `x[0][0]`, `x[0][1]`, `x[0][2]`, `x[1][0]`, `x[1][1]`, and `x[1][2]` in that order. In general, when you assign initial values to a multidimensional array, it is the last, or rightmost, index that increases the fastest.

A.7 Strings

Earlier, you saw that C++ provides literal character strings such as

```
"This is a string."
```

You can declare and use variables that contain such strings, and then manipulate the strings as naturally as you manipulate integers by using familiar operators. Our presentation includes only some of the possible operations on strings.

The C++ Standard Library provides the data type `string`. To use `string` in your program, you precede it with the statement

```
#include <string>
```

Note that `string` is in the `std` namespace. We will discuss libraries and namespaces later in this appendix.

You can declare a string variable `title` and initialize it to the empty string by writing

```
std::string title; // Initialization is provided by string's default constructor
```

You can initialize a string variable to a string literal when you declare it by writing

```
std::string title = "Walls and Mirrors";
```

You can subsequently assign another string to `title` by using an assignment statement such as

```
title = "J Perfect's Diary";
```

In each of the previous examples, `title` has a length of 17. You use either of the methods `length` or `size` to get the current length of a string. Thus, `title.length()` and `title.size()` are each 17.

You can reference the individual characters in a string by using the same index notation that you use for an array. Thus, in the previous example, `title[0]` contains the character `J` and `title[16]` contains the character `y`.

You can compare strings by using the familiar comparison operators. Not only can you see whether two strings are equal, but you can also discover which of two strings comes before the other. The ordering of two strings is analogous to alphabetic ordering, but you use the ASCII table instead of the alphabet. Thus, the following relationships are all true:

Examples of true expressions

```
"dig" < "dog"
"Star" < "star"   (because 'S' < 's')
"start" > "star"
"d" > "abc"
```

You can concatenate two strings to form another string by using the + operator. That is, you place one string after another to form a third string. For example, if

```
std::string str1 = "Com";
```

the statement

```
std::string str2 = str1 + "puter";
```

forms the string `"Computer"` and assigns it to `str2`. Similarly, you can append a single character to a string, as in

```
str1 += 's';
```

Use `substr` to access part of a string

You can manipulate a portion of a string by using the method

`substr(`*position*`,` *length*`)`

The first argument specifies the position of the beginning of the substring (remember that 0 is the position of the first character in the string). The second argument is the length of the substring. For example,

```
title.substr(2, 7)
```

is the string `"Perfect"`.

To perform input and output with C++ strings, you must include the library `iostream` by beginning your program with the statement

```
#include <iostream>
```

For example, you then can display the contents of a string variable by executing

You can use << to display a string

```
title = "Walls and Mirrors";
std::cout << title << "\n";
```

The result is `Walls and Mirrors`. The operator << writes the entire string, including the blanks.

You can read a string of characters into a string variable. For example, when the statement

```
std::cin >> title;
```

You can use >> to read a string without whitespace

reads the data line

```
Jamie Perfect's Diary
```

it assigns the string `"Jamie"` to `title`. Whitespace in the input line terminates the read operation for a string. To read the entire line of input, including its blank characters, you write

```
getline(std::cin, title);
```

A.8 Vectors

Another way to hold data items of the same type is by using a **vector**. A vector is similar to a one-dimensional array, but vectors provide additional features for the programmer not found in a simple array. A vector is an object of a standard C++ class named `vector`. This class is a part of the **Standard Template Library**, or **STL**. The STL is a library of template classes that provide data types you can use in your programs. These data types are not part of the official C++ language, but they have been added to the built-in data types. Section A.10 discusses libraries such as the STL and their usefulness. C++ Interlude 8 explores the STL in more detail.

To use a vector in your program, you must begin it with the following statement:

```
#include <vector>
```

You can declare a vector in one of three ways:

- If you know how many elements you want in the vector, you can place the type of data it will hold in angle brackets and the number of elements in parentheses:

  ```
  std::vector<double> firstVector(10);     // Vector to hold 10 doubles
  std::vector<std::string> myVector(12); // Vector to hold 12 strings
  ```

  The size you specify when declaring the vector is only its initial size. As you will see, a vector can grow in size when you add entries. Note that both `vector` and `string` are in the `std` namespace. Section A.11 of this appendix discusses namespaces.
- You can place initial values into a vector when you declare it by writing a second argument:

  ```
  std::vector<int> intVector(5, -1); // Vector to hold 5 integers, initially -1
  ```

  When the elements in the vector are allocated, they are given the value of the second argument.
- You can also create an empty vector—a vector with no elements—by omitting its size and the parentheses:

  ```
  std::vector<char> letterVector; // An empty vector of characters
  ```

Access vector entries using [] just as you would an array

You can store or access the entries in a vector by using the [] operator, just as you would when using an array. As for an array, the subscripts that identify elements in a vector start at 0 and go to s - 1, where s is the current size of the vector. The following statements are examples of accessing an existing value in a vector and changing the value of an existing entry:

```
double x = firstVector[5];                  // Gets sixth entry in x
myVector[3] = "This is a sample string."; // Sets fourth entry's value
```

Using [] stores a value in an *existing* element. If you are not sure how many elements the vector has, you can call the method size, as in the following example:

```
std::cout << intVector.size() << std::endl; // Displays the capacity of intVector
```

size returns the number of elements in the vector

By calling the size method, you can find out whether the vector is full. This is an important advantage that a vector has over an array.

If the vector is either full or has no elements—that is, if it was created without elements, as letterVector was previously—you can still add new values by using the method push_back. The push_back method accepts an argument and adds it after the last element of the vector. In other words, it pushes the value onto the back of the vector.

Earlier, we declared a ten-element vector firstVector that could hold data of type double. If that vector was full, and we needed to add the additional values 2.3 and 3.4, we could use the push_back method:

push_back places its argument into a new element at the back of a vector

```
firstVector.push_back(2.3); // Grow vector and store value
firstVector.push_back(3.4); // Grow vector and store value
```

At this point, calling the size method would return 12, since two additional elements have been added to the vector.

pop_back removes the last element from a vector

You also can reduce the size of a vector by removing either its last element or all of its elements. To remove only the last element, you can use the pop_back method. This method shortens the vector but does not give you the entry in the removed element. You must save that entry before calling pop_back. For example, the following statements get the current size of the vector myVector, save the value in the last element, and then remove the last element from the vector:

```
int length = myVector.size();              // Find the current number of elements
std::string last = myVector[length - 1]; // Get the string in the last element
myVector.pop_back();                       // Reduce the number of elements by 1
```

A subsequent call to myVector.size() would return length - 1, since the last element was removed.

clear removes all elements from a vector so it has a size of 0

To remove all elements from a vector and leave the vector empty, you use the method clear:

```
myVector.clear(); // myVector is now an empty vector
```

After a vector has been cleared, you must use the method push_back to add new entries.

C++ Interlude 8 provides more information about vector, including these and other methods.

A.9 Classes

Object-oriented programming, or **OOP**, views a program not as a sequence of actions but as a collection of components called **objects** that interact to produce a certain result. A group of objects of the same kind belong to a **class**, which is a programming construct that defines the

object's data type. Chapter 1 talks more about OOP; here we want to discuss how to write a class in C++.

An object contains data and can perform certain operations on or with that data. The class associated with a particular object describes its data and its operations. That is, a class is like a blueprint for creating certain objects. An object's operations, or **behaviors**, are defined within the class by **methods**, which are simply functions within a class. These methods, together with the class's data are known as the class's **members**.

A C++ class defines a new data type

We could use a ball as an example of an object. Because thinking of a basketball, volleyball, tennis ball, or soccer ball probably suggests images of the game rather than the object itself, let's abstract the notion of a ball by picturing a sphere. A sphere of a given radius has attributes such as volume and surface area. A sphere as an object should be able to report its radius, volume, surface area, and so on. That is, the sphere object has methods that return such values.

In C++, a class has the following form:

```
class Sphere
{
private:
   <Declarations of private data members and methods>
   <Definitions of private methods>
   . . .

public:
   <Definitions of public methods>
   . . .
}; // Note the semicolon
```

By default, all members in a class are **private**—they are not directly accessible by any program that uses the class—unless you designate them as **public**. However, explicitly indicating the private and public portions of a class is a good programming practice and one that we will follow in this book. You should always declare a class's data members as private.

Most methods are public, but private methods—which only the class can call—can be helpful, as you will see. The definition of a class's method can call any of the class's other methods or use any of its data members, regardless of whether they are private or public.

Classes have special methods, called constructors and destructors, for the creation and destruction of its objects. A **constructor** creates and initializes new objects, or **instances**, of a class. A **destructor** destroys an object by freeing the memory assigned to it, when the object's lifetime ends. A typical class has several constructors, but only one destructor. For many classes, you can omit the destructor. In such cases, the compiler will generate a destructor for you. For now, the compiler-generated destructor is sufficient. C++ Interlude 2 discusses how and why you would write your own destructor.

An object is an instance of a class

A constructor creates and initializes an object

In C++, a constructor has the same name as the class. Constructors have no return type—not even `void`—and cannot use `return` to return a value. Constructors can have arguments. We discuss constructors in more detail shortly, after we look at an example of a class definition.

A destructor destroys an object

A.9.1 The Header File

You should place each class definition in its own **header file** or **specification file**—whose name by convention ends in `.h`. The header file `Sphere.h` shown in Listing A-2 contains a class definition for sphere objects.

LISTING A-2 The header file Sphere.h

```
1  /** @file Sphere.h */
2  const double PI = 3.14159;
3
4  /** Definition of a class of Spheres. */
5  class Sphere
6  {
7  private:
8     double theRadius; // The sphere's radius
9
10 public:
11    /** Default constructor: Creates a sphere and initializes its radius
12     to a default value.
13     Precondition:  None.
14     Postcondition:  A sphere of radius 1 exists. */
15    Sphere();
16
17    /** Constructor: Creates a sphere and initializes its radius.
18     Precondition:  initialRadius is the desired radius.
19     Postcondition:  A sphere of radius initialRadius exists. */
20    Sphere(double initialRadius);
21
22    /** Sets (alters) the radius of this sphere.
23     Precondition:  newRadius is the desired radius.
24     Postcondition:  The sphere's radius is newRadius. */
25    void setRadius(double newRadius);
26
27    /** Gets this sphere's radius.
28     Precondition:  None.
29     Postcondition:  Returns the radius. */
30    double getRadius() const;
31
32    /** Gets this sphere's diameter.
33     Precondition:  None.
34     Postcondition:  Returns the diameter. */
35    double getDiameter() const;
36
37    /** Gets this sphere's circumference.
38     Precondition:  PI is a named constant.
39     Postcondition:  Returns the circumference. */
40    double getCircumference() const;
41
42    /** Gets this sphere's surface area.
43     Precondition:  PI is a named constant.
44     Postcondition:  Returns the surface area. */
45    double getArea() const;
46
47    /** Gets this sphere's volume.
48     Precondition:  PI is a named constant.
49     Postcondition:  Returns the volume. */
50    double getVolume() const;
51
52    // The compiler-generated destructor is sufficient.
53 }; // end Sphere
54 // End of header file.
```

Comments in the header file specify the methods

You should always place a class's data members within its private section. Typically, you provide methods—such as `setRadius` and `getRadius`—to access the data members. In this way, you control how and whether the rest of the program can access the data members. This design principle should lead to programs that not only are easier to debug, but also have fewer logical errors from the beginning.

A class's data members should be private

Some method declarations, such as

```
double getRadius() const;
```

are tagged with `const`. Such methods cannot alter the data members of the class. Making `getRadius` a `const` method is a fail-safe technique that ensures that it will only return the current value of the sphere's radius, without changing it.

`const` methods cannot change a class's data members

A.9.2 The Implementation File

Let's begin implementing the class `Sphere` by examining its constructors.

Constructors. A constructor allocates memory for an object and can initialize the object's data to particular values. A class can have more than one constructor, as is the case for the class `Sphere`.

The first constructor in `Sphere` is the **default constructor**:

A default constructor has no arguments

```
Sphere();
```

A default constructor by definition has no arguments. Typically, a default constructor initializes data members to values that the class implementation chooses. For example, the implementation

```
Sphere::Sphere()
{
   theRadius = 1.0;
}  // end default constructor
```

sets `theRadius` to 1.0. C++ Interlude 1 will show you another way to initialize data members within constructors that is preferable to using assignment statements.

Notice the qualifier `Sphere::` that precedes the constructor's name. When you implement any method, you qualify its name with its class type followed by the **scope resolution operator** `::` to distinguish it from other methods that might have the same name.

When you declare an instance of the class, you implicitly invoke a constructor. For example, the statement

```
Sphere unitSphere;
```

invokes the default constructor, which creates the object `unitSphere` and sets its radius to 1.0. Notice that you do not include parentheses after `unitSphere`.

The next constructor in `Sphere` is

```
Sphere(double initialRadius);
```

It creates a sphere object of radius `initialRadius`. This constructor needs only to initialize the private data member `theRadius` to `initialRadius`. Its implementation is

```
Sphere::Sphere(double initialRadius)
{
   theRadius = initialRadius;
}  // end constructor
```

You implicitly invoke this constructor by writing a declaration such as

```
Sphere mySphere(5.1);
```

In this case, the object `mySphere` has a radius of 5.1.

We can make the previous constructor ensure that the given radius is not negative by writing its definition as follows:

```
Sphere::Sphere(double initialRadius)
{
   if (initialRadius > 0)
      theRadius = initialRadius;
   else
      theRadius = 1.0; // Set to default value, if bad input
}  // end constructor
```

Note: If you omit all constructors from your class, the compiler will generate a default constructor—that is, one with no arguments—for you. A compiler-generated default constructor, however, might not initialize data members to values that you will find suitable. If you define a constructor that has arguments but you omit the default constructor, the compiler will not generate one for you. Thus, you will not be able to write statements such as

```
Sphere defaultSphere;
```

The implementation file contains the definitions of the class's methods

Typically, you place the implementation of a class's constructors and other methods in an **implementation file** whose name ends in `.cpp`. Listing A-3 contains an implementation file for the class `Sphere`. Notice that within the definition of a method, you can reference the class's data member or invoke its other methods without preceding the member names with `Sphere::`. In particular, notice how the constructor calls the method `setRadius` to avoid duplicating the code that ensures a positive radius.

LISTING A-3 The implementation file `Sphere.cpp`

```
1  /** @file Sphere.cpp */
2  #include "Sphere.h" // Include the header file
3
4  Sphere::Sphere()
5  {
6     theRadius = 1.0;
7  }  // end default constructor
8
9  Sphere::Sphere(double initialRadius)
10 {
11    setRadius(initialRadius); // Sphere:: not needed here
12 }  // end constructor
13
14 void Sphere::setRadius(double newRadius)
15 {
16    if (newRadius > 0)
17       theRadius = newRadius;
```

```
18      else
19         theRadius = 1.0;
20   } // end setRadius
21
22   double Sphere::getRadius() const
23   {
24      return theRadius;
25   } // end getRadius
26
27   double Sphere::getDiameter() const
28   {
29      return 2.0 * theRadius;
30   } // end getDiameter
31
32   double Sphere::getCircumference() const
33   {
34      return PI * getDiameter();
35   } // end getCircumference
36
37   double Sphere::getArea() const
38   {
39      return 4.0 * PI * theRadius * theRadius;
40   } // end getArea
41
42   double  Sphere::getVolume() const
43   {
44      double radiusCubed = theRadius * theRadius * theRadius;
45      return (4.0 * PI * radiusCubed) / 3.0;
46   } // end getVolume
47   // End of implementation file
```

A local variable such as `radiusCubed` should not be a data member

Note: Local variables
You should distinguish between a class's data members and any local variables that the implementation of a method requires. It is inappropriate for such local variables to be data members of the class.

A.9.3 Using the Class `Sphere`

The following simple program demonstrates the use of the class `Sphere`:

```
#include <iostream>
#include "Sphere.h"
int main()
{
   Sphere unitSphere;         // Radius is 1.0
   Sphere mySphere(5.1);      // Radius is 5.1
   mySphere.setRadius(4.2);   // Resets radius to 4.2
   std::cout << mySphere.getDiameter() << std::endl;

   return 0;
} // end main
```

An object such as mySphere can, on request, reset the value of its radius; return its radius; and compute its diameter, surface area, circumference, and volume. These requests to an object are called **messages** and are simply calls to methods. Thus, an object responds to a message by acting on its data. To invoke an object's method, you qualify the method's name with the object variable. For example, we wrote mySphere.getDiameter() in the previous program.

Notice that the previous program included the header file Sphere.h, but did not include the implementation file Sphere.cpp. You compile a class's implementation file separately from the program that uses the class. The way you tell the operating system where to locate the compiled implementation depends on the particular system. Section A.10 of this appendix and C++ Interlude 1 provide more information about header and implementation files.

The previous program is an example of a **client** of a class. A client of a particular class is simply a program or module that uses the class. We will reserve the term **user** for the person who uses a program.

A.9.4 Inheritance

A brief discussion of **inheritance** is provided here, because it is a common way to create new classes in C++. Further discussions of inheritance occur in C++ Interludes 1 and 5, and as needed throughout the book.

A class derived from the class Sphere

Suppose we want to give our spheres a color, knowing that we have already developed the class Sphere. Instead of writing an entirely new class of spheres that have a color, we can reuse the Sphere implementation and add color characteristics and operations by using inheritance. Here is a declaration of the class SphereInColor that uses inheritance:

```
#include "Sphere.h"
enum Color {RED, BLUE, GREEN, YELLOW};
class SphereInColor : public Sphere
{
private:
   Color sphereColor;

public:
   SphereInColor(Color initialColor);
   SphereInColor(Color initialColor, double initialRadius);
   void setColor(Color newColor);
   Color getColor() const;
}; // end SphereInColor
```

The class Sphere is called the **base class** or **superclass**, and SphereInColor is called the **derived class** or **subclass** of the class Sphere.

Any instance of the derived class is also considered to be an instance of the base class and can be used in a program anywhere that an instance of the base class can be used. Also, when the keyword public precedes the name of the base class in the new class's header, any of the publicly defined methods or data members that can be used with instances of the base class can be used with instances of the derived class. The derived class instances also have the additional methods and data members that are publicly defined in the derived class definition.

The implementation of the methods for the class SphereInColor is as follows:

```
SphereInColor::SphereInColor(Color initialColor): Sphere()
{
   sphereColor = initialColor;
} // end constructor
```

```
SphereInColor::SphereInColor(Color initialColor, double initialRadius)
                                                  : Sphere(initialRadius)
{
   sphereColor = initialColor;
}  // end constructor

void SphereInColor::setColor(Color newColor)
{
   sphereColor = newColor;
}  // end setColor

Color SphereInColor::getColor() const
{
   return sphereColor;
}  // end getColor
```

Notice how the constructors for the class `SphereInColor` invoke the base-class constructors `Sphere()` and `Sphere(initialRadius)`. The derived class needs the initialization of the data members in the base class that the base-class constructors can provide. The derived-class constructors then add initializations that are specific to the derived class.

Here is a function that uses the class `SphereInColor`:

An instance of a derived class can invoke public methods of the base class

```
void useSphereInColor()
{
   SphereInColor ball(RED);
   ball.setRadius(5.0);
   std::cout << "The ball diameter is " << ball.getDiameter();
   ball.setColor(BLUE);
   ...
}  // end useSphereInColor
```

This function uses the constructor and the method `setColor` from the derived class `SphereInColor`. It also uses the methods `setRadius` and `getDiameter` that are defined in the base class `Sphere`.

A.10 Libraries

One of the advantages of modular programming is that you can implement modules independently of other modules. You might also find it possible for several different programs to use a particular module. As a result, you can build a **library** of modules—that is, classes and functions—that you can include in future programs.

Any library—a C++ standard library or one that you write—has a corresponding header that provides information about the contents of the library. For standard libraries, the header is simply an abstraction that the compiler either maps to a filename or handles in a different manner. Thus, when using the standard libraries, you do not see the `.h` extension that ends the names of our own header files.

You have already seen some standard libraries, such as the one that provides input and output services. To use the modules contained in a library, you use the `include` directive with the name of the header associated with the library. For example, you write

```
#include <iostream>
```

Appendix H provides a list of some of the available headers.

User-defined libraries are typically organized into two files. One file, the header file, contains a definition for each class in the library that is available to your program. This file could

also contain, for example, function declarations, constant definitions, `typedef` statements, enumerations, and other `include` statements. By convention, the name of a header file associated with a user-defined library ends in `.h`. The other file—the implementation file—contains definitions of the class methods that the header file declares. Typically, the name of an implementation file ends in `.cpp`.

The assumption, of course, is that the files are in source form—that is, they need to be compiled. It certainly would be more efficient to compile the method definitions once, independently of any particular program, and then later merge the results of the compilation with any program that you desire. In fact, you should compile the implementation file and then include the header file in source form in your program by using an `include` directive such as

```
#include "MyHeader.h"
```

You use double quotes instead of angle brackets to enclose the name of a header file that you have written. The mechanics of incorporating the compiled implementation file into your program are system dependent.

Thus, your program can use previously compiled C++ statements, which are no longer available to you in source form. Maybe you did not even write these statements, just as you did not write the standard C++ functions such as `sqrt`. That is, you use a library in the same spirit in which you use standard functions. Because the header file indicates what is available to you, you must think of a library in terms of what it can do for you and not how it is implemented. You should think of all of your modules in this way, even if you eventually implement them yourself.

Note: The C++ Standard Library

The C++ Standard Library is a collection of standard classes and functions that you can use in your C++ programs. This library provides us with such features as input and output services, strings, and functions to perform certain mathematical tasks.

Note: The C++ Standard Template Library (STL)

The C++ Standard Template Library, or STL, is a collection of classes and functions that is a part of the C++ Standard Library, and you can use in your C++ programs with any data type—either built-in or user-defined. To achieve this flexibility, this library uses templates, which are a construct that we will discuss in C++ Interlude 1. Note that `vector` is a template class within the STL.

A.11 Namespaces

Since different libraries can use the same names for their classes and functions, C++ organizes these names into namespaces. A **namespace** is a named group, or category, of identifiers that enables you to differentiate among identical identifiers. For example, if the namespaces `x` and `y` each contain the identifier `z`, `x::z` and `y::z` are different identifiers.

Earlier when we used strings in a program, we included the standard class of strings by writing

```
#include <string>
```

The names of the classes and functions in the C++ Standard Library are organized into the namespace `std`. You can write the directive

```
using namespace std;
```

to tell the compiler to look for `string` in the C++ Standard Library if it does not find its definition in our program. Another library might also contain a class named `string`, but it would be associated with a different namespace. Analogous comments apply to `vector`, as described in Section A.8, since it also is in the `std` namespace.

The following program contains a `using` directive for the namespace `std`:

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
   string title = "Walls and Mirrors";
   cout << title << endl;

   return 0;
} // end main
```

The program can reference all of the classes in the `std` namespace, including `iostream` and `string`, without preceding their names with `std::`.

Programming Tip: The previous `using` directive lets the compiler see all of the names in the `std` namespace. You might use a name in your program that is also in the `std` namespace and defined in the C++ Standard Library. The compiler would be faced with two definitions for the same identifier, yours and the library's. To avoid this confusion and the chance for hard-to-detect errors, professional programmers do not use this directive; we will not use it within the chapters of this book.

One way to omit the `using` directive for the `std` namespace is to precede the identifiers in that namespace with the **namespace indicator** `std::`. For example, we can write the previous program as follows:

```
#include <iostream>
#include <string>

int main()
{
   std::string title = "Walls and Mirrors";
   std::cout << title << std::endl;

   return 0;
} // end main
```

Although writing a namespace indicator is often the best way to proceed, you might find it tedious. In such an event, you could write `using` **declarations** as follows:

```
#include <iostream>
#include <string>

int main()
{
```

```
    using std::string; // string is in the std namespace
    using std::cout;   // cout is in the std namespace

    // You can use string and cout without an std:: prefix
    string title = "Walls and Mirrors";
    cout << title << std::endl;
    return 0;
} // end main
```

After the `using` declarations, you can write `string` and `cout` without needing to precede them with `std::`.

Summary

1. A comment in C++ can begin with
   - `//` as a single line or at the end of a C++ statement.
   - `/**`, end with `*/`, and occupy several lines at the beginning of a method, function, or class.
   - `/*`, end with `*/`, and occupy several lines anywhere within a program.
2. A C++ identifier is a sequence of letters, digits, and underscores that must begin with either a letter or an underscore.
3. You can use a `typedef` statement to declare new names for data types. These names are simply synonyms for the data types; they are not new data types.
4. You define named constants by using a statement of the form

   const *type identifier* = *value*;
5. Enumeration provides another way to name integer constants and to define an integral data type, as in

   ```
   enum Day {SUN, MON, TUE, WED, THU, FRI, SAT};
   ```
6. C++ uses short-circuit evaluation for expressions that contain the logical operators `&&` (and) and `||` (or). That is, evaluation proceeds from left to right and stops as soon as the value of the entire expression is apparent.
7. The output operator `<<` places a value into an output stream, and the input operator `>>` extracts a value from an input stream. You can imagine that these operators point in the direction of data flow. Thus, in `cout << myVar`, the operator points away from the variable `myVar`—data flows from `myVar` to the stream—whereas in `cin >> myVar`, the operator points to the variable `myVar`—data flows from the stream into `myVar`.
8. The general form of a function definition is

   type name(*parameter-declaration-list*)
   {
   body
   }

   A valued function returns a value by using the `return` statement. Although a void function does not return a value, it can use `return` to exit.
9. When invoking a function, the actual arguments must correspond to the parameters in number, order, and type.

10. A function makes local copies of the values of any arguments that are passed by value. Thus, the arguments remain unchanged by the function. Such arguments are, therefore, input arguments. A function does not copy arguments that are passed by reference. Rather, it references the actual argument locations whenever the parameters appear in the function's definition. In this way, a function can change the values of the arguments, thus implementing output arguments. However, a function does not copy and cannot change a constant reference argument. If copying an input argument would be expensive, make it a constant reference argument instead of a value argument.

11. The general form of the `if` statement is

```
if (expression)
   statement1
else
   statement2
```

If *expression* is true, $statement_1$ executes; otherwise $statement_2$ executes.

12. The general form of the `switch` statement is

```
switch (expression)
{
  case constant1:
     statement1
     break;
  ...
  case constantn:
     statementn
     break;
  default:
     statement
}
```

The appropriate *statement* executes according to the value of *expression*. Typically, `break` follows the statement or statements after each `case`. Omitting `break` causes execution to continue to the statement(s) in the next `case`.

13. The general form of the `while` statement is

```
while (expression)
   statement
```

As long as *expression* is true, *statement* executes. Thus, it is possible that *statement* never executes.

14. The general form of the `for` statement is

```
for (initialize; test; update)
  statement
```

where *initialize*, *test*, and *update* are expressions. Typically, *initialize* is an assignment expression that occurs only once. Then if *test*, which is usually a logical expression, is true, *statement* executes. The expression *update* executes next, usually incrementing or decrementing a counter. This sequence of events repeats, beginning with the evaluation of *test*, until *test* is false.

15. The general form of the `do` statement is

```
do
  statement
while (expression);
```

Here, *statement* executes until the value of *expression* is false. Note that *statement* always executes at least once. Also note the required semicolon.

16. An array contains items that have the same data type. You can refer to these items by using an index that begins with zero. Arrays are always passed to functions by reference.

17. You must be careful that an array index does not exceed the size of the array. C++ does not check the range of array indices. Similar comments apply to strings.

18. An object encapsulates both data and operations on that data. In C++, objects are instances of a class, which is a programmer-defined data type.

19. A string is an object of the standard C++ class `string`. It represents a sequence of characters. You can manipulate the entire string, a substring, or the individual characters.

20. A vector is an object of the standard C++ class `vector`. It holds items of the same data type. A vector behaves like a high-level array.

21. A C++ class contains at least one constructor, which is an initialization method, and a destructor, which is a cleanup method that destroys an object when its lifetime ends.

22. If you do not define a constructor for a class, the compiler will generate a default constructor that is, one without arguments—for you. If you do not define a destructor, the compiler will generate one for you. C++ Interlude 2 and Chapter 4 describe when you need to write your own destructor.

23. Members of a class are private unless you designate them as public. The client of the class—that is, the program that uses the class—cannot use members that are private. However, the implementations of methods can use them. You should make the data members of a class private and provide public methods to access some or all of the data members.

24. Because certain classes have applications in many programs, you should take steps to facilitate their use. You can define and implement a class within header and implementation files, which a program can include when it needs to use the class.

25. A typical C++ program uses header files that you incorporate by using the `include` directive. A header file contains class definitions, function declarations, constant definitions, `typedef` statements, enumerations, and other `include` statements. The program might also require an implementation file of function definitions that have been compiled previously and placed into a library. The operating system locates the required implementation file and combines it with the program in ways that are system dependent.