SCHAUM'S outlines

PROGRAMMING with C++

Second Edition

JOHN R. HUBBARD, Ph.D.

- Conforms to the new ANSI/ISO Standard for C++
- 470 examples and solved problems step-by-step
- The only C++ book in solved-problem format
- Ideal for independent study
- Solutions to all the examples and problems can be downloaded from the author's World Wide Web page

MORE THAN 30 MILLION SCHAUM'S OUTLINES SOLD

Use with these courses: ☑ Computer Science I and II ☑ Introduction to C++ ☑ C++ I ☑ Fundamentals of C++ ☑ Programming with C++ ☑ Advanced Placement Computer Science ☑ Data Structures ☑ Software Engineering ☑ Computer Architecture ☑ Programming Languages

SCHAUM'S OUTLINE OF

THEORY AND PROBLEMS

of

PROGRAMMING WITH C++

Second Edition

•

JOHN R. HUBBARD, Ph.D.

Professor of Mathematics and Computer Science
University of Richmond

•

SCHAUM'S OUTLINE SERIES

McGRAW-HILL

New York San Francisco Washington, D.C. Auckland Bogotá Caracas
Lisbon London Madrid Mexico City Milan Montreal
New Delhi San Juan Singapore Sydney Tokyo Toronto

McGraw-Hill

A Division of The McGraw-Hill Companies

0-07-136811-6

The material in this eBook also appears in the print version of this title: ISBN 0-07-135346-1.

DOI: 10.1036/0071368116

Preface

Like all Schaum's Outline Series books, this volume is intended to be used primarily for self study, preferably in conjunction with a regular course in C++ programming language or computer science. However, it is also well-suited for use in independent study or as a reference.

The book includes over 200 examples and solved problems. The author firmly believes that the principles of data structures can be learned from a well-constructed collection of examples with complete explanations. This book is designed to provide that support.

C++ was created by Bjarne Stroustrup in the early 1980s. Based upon C and Simula, it is now one of the most popular languages for object-oriented programming. The language was standardized in 1998 by the American National Standards Institute (ANSI) and the International Standards Organization (ISO). This new ANSI/ISO Standard includes the powerful Standard Template Library (STL). This book conforms to these standards.

Although most people who undertake to learn C++ have already had some previous programming experience, this book assumes none. It approaches C++ as one's first programming language. Therefore, those who have had previous experience may need only skim the first few chapters.

C++ is a difficult language for at least two reasons. It inherits from the C language an economy of expression that novices often find cryptic. And as an object-oriented language, its widespread use of classes and templates presents a formidable challenge to those who have not thought in those terms before. It is the intent of this book to provide the assistance necessary for first-time programmers to overcome these obstacles.

Source code for all the examples and problems in this book, including the Supplementary Problems, may be downloaded from these websites **`http://projectEuclid.net/schaums`**, **`http://www.richmond.edu/~hubbard/schaums`**, **`http://hubbards.org/schaums`**, or **`http://jhubbard.net/schaums`**. Any corrections or addenda for the book will also be available at these sites.

I wish to thank all my friends, colleagues, students, and the McGraw-Hill staff who have helped me with the critical review of this manuscript, including John Aliano, Arthur Biderman, Francis Minhthang Bui, Al Dawson, Peter Dailey, Mohammed El-Beltagy, Gary Galvez, Libbie Geiger, Sergei Gorlatch, Chris Hanes, John B. Hubbard, Raana Jeelani, Dick Palas, Blake Puhak, Arni Sigurjonsson, Andrew Somers, Joe Troncale, Maureen Walker, Stefan Wentzig, and Nat Withers. Their editorial advice and debugging skills are gratefully appreciated.

Special thanks to my wife and colleague, Anita H. Hubbard, for her advice, encouragement, and creative ideas for this book. Many of the original problems used here are hers.

JOHN R. HUBBARD
Richmond, Virginia

Contents

Chapter 1

Programming is best regarded as
the process of creating works of literature,
which are meant to be read.
—Donald E. Knuth

Elementary C++ Programming

A *program* is a sequence of instructions that can be executed by a computer. Every program is written in some programming language. C++ (pronounced "see-plus-plus") is one of the most powerful programming languages available. It gives the programmer the power to write efficient, structured, object-oriented programs.

1.1 GETTING STARTED

To write and run C++ programs, you need to have a text editor and a C++ compiler installed on your computer. A *text editor* is a software system that allows you to create and edit text files on your computer. Programmers use text editors to write programs in a programming language such as C++. A *compiler* is a software system that translates programs into the machine language (called *binary code*) that the computer's operating system can then run. That translation process is called *compiling* the program. A *C++ compiler* compiles C++ programs into machine language.

If your computer is running a version of the Microsoft Windows operating system (*e.g.*, Windows 98 or Windows 2000), then it already has two text editors: WordPad and Notepad. These can be started from the Start key. In Windows 98, they are listed under Accessories.

Windows does not come with a built-in C++ compiler. So unless someone has installed a C++ compiler on the machine you are using, you will have to do that yourself. If you are using a Windows computer that is maintained by someone else (*e.g.*, an Information Services department at your school or company), you may find a C++ compiler already installed. Use the Start key to look under Programs for Borland C++Builder, Metrowerks CodeWarrior, Microsoft Visual C++, or any other program with "C++" in its name. If you have to buy your own C++ compiler, browse the Web for inexpensive versions of any of the compilers mentioned above. These are usually referred to as IDEs (*Integrated Development Environment*s) because they include their own specialized text editors and debuggers.

If your computer is running a proprietary version of the UNIX operating system on a workstation (*e.g.*, Sun Solaris on a SPARCstation), it may already have a C++ compiler installed. An easy way to find out is to create the program shown in Example 1.1 on page 2, name it `hello.C`, and then try to compile it with the command

```
CC hello
```

The Free Software Foundation has a suite of UNIX software, named "GNU" software that can be downloaded for free from

```
http://www.gnu.org/software/software.html
```

Use their GCC package which includes a C++ compiler and their Emacs editor. For DOS systems, use their DJGPP which includes a C++ compiler.

1.2 SOME SIMPLE PROGRAMS

Now you have a text editor for writing C++ programs and a C++ compiler for compiling them. If you are using an IDE such as Borland C++Builder on a PC, then you can compile and run your programs by clicking on the appropriate buttons. Other systems may require you to use the command line to run your programs. In that case, you do so by entering the file name as a command. For example, if your source code is in a file named `hello.cpp`, type

```
hello
```

at the command line to run the program after it has been compiled.

When writing C++ programs, remember that C++ is *case-sensitive*. That means that `main()` is different from `Main()`. The safest policy is to type everything in lower-case except when you have a compelling reason to capitalize something.

EXAMPLE 1.1 The "Hello, World" Program

This program simply prints "Hello, World!":

```
#include <iostream>
int main()
{ std::cout << "Hello, World!\n";
}
```

The first line of this source code is a *preprocessor directive* that tells the C++ compiler where to find the definition of the `std::cout` object that is used on the third line. The identifier `iostream` is the name of a file in the *Standard C++ Library*. Every C++ program that has standard input and output must include this preprocessor directive. Note the required punctuation: the pound sign `#` is required to indicate that the word "`include`" is a preprocessor directive; the angle brackets `< >` are required to indicate that the word "`iostream`" (which stands for "input/output stream") is the name of a Standard C++ Library file. The expression `<iostream>` is called a *standard header*.

The second line is also required in every C++ program. It tells where the program begins. The identifier `main` is the name of a function, called *the main function* of the program. Every C++ program must have one and only one `main()` function. The required parentheses that follow the word "main" indicate that it is a function. The keyword `int` is the name of a data type in C++. It stands for "integer". It is used here to indicate the *return type* for the `main()` function. When the program has finished running, it can return an integer value to the operating system to signal some resulting status.

The last two lines constitute the actual body of the program. A *program body* is a sequence of program statements enclosed in braces `{ }`. In this example there is only one statement:

```
std::cout << "Hello, World!\n";
```

It says to send the string `"Hello, World!\n"` to the *standard output stream* object `std::cout`. The single symbol `<<` represents the *C++ output operator*. When this statement executes, the characters enclosed in quotation marks `" "` are sent to the *standard output device* which is usually the computer screen. The last two characters `\n` represent the *newline character*. When the output device encounters that character, it advances to the beginning of the next line of text on the screen. Finally, note that every program statement must end with a semicolon (`;`).

Notice how the program in Example 1.1 is formatted in four lines of source code. That formatting makes the code easier for humans to read. The C++ compiler ignores such formatting. It

reads the program the same as if it were written all on one line, like this:

```
#include <iostream>
int main(){std::cout<<"Hello, World!\n";}
```

Blank spaces are ignored by the compiler except where needed to separate identifiers, as in

```
int main
```

Note that the preprocessor directive must precede the program on a separate line.

EXAMPLE 1.2 Another "Hello, World" Program

This program has the same output as that in Example 1.1:

```
#include <iostream>
using namespace std;
int main()
{ // prints "Hello, World!":
  cout << "Hello, World!\n";
  return 0;
}
```

The second line

```
using namespace std;
```

tells the C++ compiler to apply the prefix `std::` to resolve names that need prefixes. It allows us to use `cout` in place of `std::cout`. This makes larger programs easier to read.

The fourth line

```
{ // prints "Hello, World!"
```

includes the comment "`prints "Hello, World!"`". A *comment* in a program is a string of characters that the preprocessor removes before the compiler compiles the programs. It is included to add explanations for human readers. In C++, any text that follows the double slash symbol `//`, up to the end of the line, is a comment. You can also use C style comments, like this:

```
{ /*  prints "Hello, World!"  */
```

A *C style comment* (introduced by the programming language named "C") is any string of characters between the symbol `/*` and the symbol `*/`. These comments can run over several lines.

The sixth line

```
return 0;
```

is optional for the `main()` function in Standard C++. We include it here only because some compilers expect it to be included as the last line of the `main()` function.

A *namespace* is a named group of definitions. When objects that are defined within a namespace are used outside of that namespace, either their names must be prefixed with the name of the namespace or they must be in a block that is preceded by a `using namespace` statement. Namespaces make it possible for a program to use different objects with the same name, just as different people can have the same name. The `cout` object is defined within a namespace named `std` (for "standard") in the `<iostream>` header file.

Throughout the rest of this book, every program is assumed to begin with the two lines

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

These two required lines will be omitted in the examples. We will also omit the line

```
  return 0;
```

from the `main()` function. Be sure also to include this line if you are using a compiler (such as Microsoft Visual C++) that expects it.

1.3 THE OUTPUT OPERATOR

The symbol `<<` is called the *output operator* in C++. (It is also called the *put operator* or the *stream insertion operator*.) It inserts values into the output stream that is named on its left. We usually use the `cout` output stream, which ordinarily refers to the computer screen. So the statement

```
cout << 66;
```

would display the number 66 on the screen.

An *operator* is something that performs an action on one or more objects. The output operator `<<` performs the action of sending the value of the expression listed on its right to the output stream listed on its left. Since the direction of this action appears to be from right to left, the symbol `<<` was chosen to represent it. It should remind you of an arrow pointing to the left.

The `cout` object is called a "stream" because output sent to it flows like a stream. If several things are inserted into the `cout` stream, they fall in line, one after the other as they are dropped into the stream, like leaves falling from a tree into a natural stream of water. The values that are inserted into the `cout` stream are displayed on the screen in that order.

EXAMPLE 1.3 Yet Another "Hello, World" Program

This program has the same output as that in Example 1.1:

```
int main()
{ // prints "Hello, World!":
  cout << "Hel" << "lo, Wo" << "rld!" << endl;
}
```

The output operator is used four times here, dropping the four objects `"Hel"`, `"lo, Wo"`, `"rld!"`, and `endl` into the output stream. The first three are strings that are concatenated together (*i.e.*, strung end-to-end) to form the single string `"Hello, World!"`. The fourth object is the *stream manipulator* object `endl` (meaning "end of line"). It does the same as appending the *endline character* `'\n'` to the string itself: it sends the print cursor to the beginning of the next line. It also "flushes" the output buffer.

1.4 CHARACTERS AND LITERALS

The three objects `"Hel"`, `"lo, Wo"`, and `"rld!"` in Example 1.3 are called *string literals*. Each literal consists of a sequence of characters delimited by quotation marks.

A *character* is an elementary symbol used collectively to form meaningful writing. English writers use the standard Latin alphabet of 26 lower case letters and 26 upper case letters along with the 10 Hindu-Arabic numerals and a collection of punctuation marks. Characters are stored in computers as integers. A *character set code* is a table that lists the integer value for each character in the set. The most common character set code in use at the end of the millennium is the *ASCII Code*, shown in Appendix A. The acronym (pronounced "as-key") stands for American Standard Code for Information Interchange.

The *newline character* `'\n'` is one of the nonprinting characters. It is a single character formed using the backslash `\` and the letter `n`. There are several other characters formed this way, including the *horizontal tab* character `'\t'` and the *alert character* `'\a'`. The backslash is also used to denote the two printing characters that could not otherwise be used within a string literal: the quote character `\"` and the backslash character itself `\\`.

Characters can be used in a program statement as part of a string literal, or as individual objects. When used individually, they must appear as character constants. A character constant is a character enclosed in single quotes. As individual objects, character constants can be output the same way string literals are.

EXAMPLE 1.4 A Fourth Version of the "Hello, World" Program

This program has the same output as that in Example 1.1:

```
int main()
{ // prints "Hello, World!":
  cout << "Hello, W" << 'o' << "rld" << '!' << '\n';
}
```

This shows that the output operator can process characters as well as string literals. The three individual characters `'o'`, `'!'`, and `'\n'` are concatenated into the output the same was as the two string literals `"Hello, W"` and `"rld"`.

EXAMPLE 1.5 Inserting Numeric Literals into the Standard Output Stream

```
int main()
{ // prints "The Millennium ends Dec 31 2000.":
  cout << "The Millennium ends Dec " << 3 << 1 << ' ' << 2000 << endl;
}
```

When numeric literals like `3` and `2000` are passed to the output stream they are automatically converted to string literals and concatenated the same way as characters. Note that the *blank character* (`' '`) must be passed explicitly to avoid having the digits run together.

1.5 VARIABLES AND THEIR DECLARATIONS

A *variable* is a symbol that represents a storage location in the computer's memory. The information that is stored in that location is called the *value* of the variable. One common way for a variable to obtain a value is by an *assignment*. This has the syntax

variable **=** *expression*;

First the *expression* is evaluated and then the resulting value is assigned to the *variable*. The equals sign "=" is the *assignment operator* in C++.

EXAMPLE 1.6 Using Integer Variables

In this example, the integer `44` is assigned to the variable `m`, and the value of the expression `m + 33` is assigned to the variable `n`:

```
int main()
{ // prints "m = 44 and n = 77":
  int m, n;
  m = 44;  // assigns the value 44 to the variable m
  cout << "m = " << m;
  n = m + 33;  // assigns the value 77 to the variable n
  cout << " and n = " << n << endl;
}
```

The output from the program is shown in the shaded panel at the top of the next page.

```
m = 44 and n = 77
```

We can view the variables `m` and `n` like this:

The variable named `m` is like a mailbox. Its name `m` is like the address on a mailbox, its value `44` is like the contents of a mailbox, and its type `int` is like a legal classification of mailboxes that stipulates what may be placed inside it. The type `int` means that the variable holds only integer values.

Note in this example that both `m` and `n` are declared on the same line. Any number of variables can be declared together this way if they have the same type.

Every variable in a C++ program must be declared before it is used. The syntax is

`specifier type name initializer;`

where *`specifier`* is an optional keyword such as `const` (see Section 1.8), *`type`* is one of the C++ data types such as `int`, *`name`* is the name of the variable, and *`initializer`* is an optional initialization clause such as `=44` (see Section 1.7).

The purpose of a declaration is to introduce a name to the program; *i.e.*, to explain to the compiler what the name means. The *`type`* tells the compiler what range of values the variable may have and what operations can be performed on the variable.

The location of the declaration within the program determines the *scope* of the variable: the part of the program where the variable may be used. In general, the scope of a variable extends from its point of declaration to the end of the immediate block in which it is declared or which it controls.

1.6 PROGRAM TOKENS

A computer program is a sequence of elements called *tokens*. These tokens include keywords such as `int`, identifiers such as `main`, punctuation symbols such as `{`, and operators such as `<<`. When you compile your program, the compiler scans the text in your source code, parsing it into tokens. If it finds something unexpected or doesn't find something that was expected, then it aborts the compilation and issues error messages. For example, if you forget to append the semicolon that is required at the end of each statement, then the message will report the missing semicolon. Some syntax errors such as a missing second quotation mark or a missing closing brace may not be described explicitly; instead, the compiler will indicate only that it found something wrong near that location in your program.

EXAMPLE 1.7 A Program's Tokens

```
int main()
{ // prints "n = 44":
  int n=44;
  cout << "n = " << n << endl;
}
```

The output is

```
n = 44
```

This source code has 19 tokens: "`int`", "`main`", "`(`", "`)`", "`{`", "`int`", "`n`", "`=`", "`44`", "`;`", "`cout`", "`<<`", "`"n = "`", "`<<`", "`n`", "`<<`", "`endl`", "`;`", and "`}`". Note that the compiler ignores the comment symbol `//` and the text that follows it on the second line.

EXAMPLE 1.8 An Erroneous Program

This is the same program as above except that the required semicolon on the third line is missing:

```
int main()
{ // THIS SOURCE CODE HAS AN ERROR:
  int n=44
  cout << "n = " << n << endl;
}
```

One compiler issued the following error message:

```
Error   : ';' expected
Testing.cpp line 4     cout << "n = " << n << endl;
```

This compiler underlines the token where it finds the error. In this case, that is the "`cout`" token at the beginning of the fourth line. The missing token was not detected until the next token was encountered.

1.7 INITIALIZING VARIABLES

In most cases it is wise to initialize variables where they are declared.

EXAMPLE 1.9 Initializing Variables

This program contains one variable that is not initialized and one that is initialized.

```
int main()
{ // prints "m = ?? and n = 44":
  int m;  // BAD: m is not initialized
  int n=44;
  cout << "m = " << m << " and n = " << n << endl;
}
```

```
m = ?? and n = 44
```

The output is shown in the shaded box.

This compiler handles uninitialized variables in a special way. It gives them a special value that appears as `??` when printed. Other compilers may simply leave "garbage" in the variable, producing output like this:

```
m = -2107339024 and n = 44
```

In larger programs, uninitialized variables can cause troublesome errors.

1.8 OBJECTS, VARIABLES, AND CONSTANTS

An *object* is a contiguous region of memory that has an address, a size, a type, and a value. The *address* of an object is the memory address of its first byte. The *size* of an object is simply the number of bytes that it occupies in memory. The *value* of an object is the constant determined by the actual bits stored in its memory location and by the object's type which prescribes how those bits are to be interpreted.

For example, with GNU C++ on a UNIX workstation, the object `n` defined by

```
int n = 22;
```

has the memory address `0x3fffcd6`, the size 4, the type **`int`**, and the value 22. (The memory address is a hexadecimal number. See Appendix G.)

The type of an object is determined by the programmer. The value of an object may also be determined by the programmer at compile time, or it may be determined at run-time. The size of an object is determined by the compiler. For example, in GNU C++ an `int` has size 4, while in Borland C++ its size is 2. The address of an object is determined by the computer's operating system at run-time.

Some objects do not have names. A *variable* is an object that has a name. The object defined above is a variable with name '`n`'.

The word "variable" is used to suggest that the object's value can be changed. An object whose value cannot be changed is called a *constant*. Constants are declared by preceding its type specifier with the keyword `const`, like this:

```
const int N = 22;
```

Constants must be initialized when they are declared.

EXAMPLE 1.10 The `const` Specifier

This program illustrates constant definitions:

```
int main()
{ // defines constants; has no output:
  const char BEEP = '\b';
  const int MAXINT = 2147483647;
  const int N = MAXINT/2;
  const float KM_PER_MI = 1.60934;
  const double PI = 3.14159265358979323846;
}
```

Constants are usually defined for values like π that will be used more than once in a program but not changed.

It is customary to use all capital letters in constant identifiers to distinguish them from other kinds of identifiers. A good compiler will replace each constant symbol with its numeric value.

1.9 THE INPUT OPERATOR

In C++, input is almost as simple as output. The *input operator* `>>` (also called the *get operator* or the *extraction operator*) works like the output operator `<<`.

EXAMPLE 1.11 Using the Input Operator

```
int main()
{ // tests the input of integers, floats, and characters:
  int m, n;
  cout << "Enter two integers: ";
  cin >> m >> n;
  cout << "m =  " << m << ", n = " << n << endl;
  double x, y, z;
  cout << "Enter three decimal numbers: ";
  cin >> x >> y >> z;
  cout << "x =  " << x << ", y = " << y << ", z = " << z << endl;
  char c1, c2, c3, c4;
  cout << "Enter four characters: ";
```

```
    cin >> c1 >> c2 >> c3 >> c4;
    cout << "c1 =  " << c1 << ", c2 = " << c2 << ", c3 = " << c3
         << ", c4 = " << c4 << endl;
}
Enter two integers: 22 44
m =  22, n = 44
Enter three decimal numbers: 2.2 4.4 6.6
x =  2.2, y = 4.4, z = 6.6
Enter four characters: ABCD
c1 =  A, c2 = B, c3 = C, c4 = D
```

The input is shown in boldface in the output panel.

Review Questions

1.1 Describe the two ways to include comments in a C++ program.

1.2 What is wrong with this program?

```
#include <iostream>
int main()
{ //  prints "Hello, World!":
  cout << "Hello, World!\n"
}
```

1.3 What is wrong with the following C-style comment?

```
cout << "Hello, /* change? */ World.\n";
```

1.4 What's wrong with this program:

```
#include <iostream>;
int main
{ // prints "n = 22":
  n = 22;
  cout << "n = << n << endl;
}
```

1.5 What does a declaration do?

1.6 What is the purpose of the preprocessing directive:

```
#include <iostream>
```

1.7 What is the shortest possible C++ program?

1.8 Where does the name "C++" come from?

1.9 What's wrong with these declarations:

```
int first = 22, last = 99, new = 44, old = 66;
```

1.10 In each of the following, assume that `m` has the value 5 and `n` has the value 2 before the statement executes. Tell what the values of `m` and `n` will be after each of the following statements executes:

a. `m *= n++;`

b. `m += --n;`

1.11 Evaluate each of the following expressions, assuming in each case that `m` has the value 25 and `n` has the value 7:

a. `m - 8 - n`

b. `m = n = 3`

c. `m%n`

d. `m%n++`

e. `m%++n`

f. `++m - n--`

1.12 Parse the following program, identifying all the keywords, identifiers, operators, literals, punctuation, and comments:

```
int main()
{ int n;
  cin >> n;
  n *= 3;  // multiply n by 3
  cout << "n=" << n << endl;
}
```

1.13 Identify and correct the error in each of the following:

a. `cout >> count;`

b. `int double=44;`

1.14 How do the following two statements differ:

```
char ch = 'A';
char ch = 65;
```

1.15 What code could you execute to find the character whose ASCII code is 100?

1.16 What does "floating-point" mean, and why is it called that?

1.17 What is numeric overflow?

1.18 How is integer overflow different from floating-point overflow?

1.19 What is a run-time error? Give examples of two different kinds of run-time errors.

1.20 What is a compile-time error? Give examples of two different kinds of compile-time errors.

Problems

1.1 Write four different C++ statements, each subtracting 1 from the integer variable `n`.

1.2 Write a block of C++ code that has the same effect as the statement

```
n = 100 + m++;
```

without using the post-increment operator.

1.3 Write a block of C++ code that has the same effect as the statement

```
n = 100 + ++m;
```

without using the pre-increment operator.

1.4 Write a single C++ statement that subtracts the sum of `x` and `y` from `z` and then increments `y`.

1.5 Write a single C++ statement that decrements the variable `n` and then adds it to `total`.

1.6 Write a program that prints the first sentence of the Gettysburg Address (or your favorite quotation).

1.7 Write a program that prints the block letter "B" in a 7×6 grid of stars like this:

```
*****
*    *
*    *
*****
*    *
*    *
*****
```

1.8 Write and run a program that prints the first letter of your last name as a block letter in a 7×7 grid of stars.

1.9 Write and run a program that shows what happens when each of the following ten "escape sequences" is printed: `\a`, `\b`, `\n`, `\r`, `\t`, `\v`, `\'`, `\"`, `\\`, `\?`.

1.10 Write and run a program that prints the sum, difference, product, quotient, and remainder of two integers. Initialize the integers with the values 60 and 7.

1.11 Write and run a program that prints the sum, difference, product, quotient, and remainder of two integers that are input interactively.

1.12 Write and run a test program that shows how your system handles uninitialized variables.

1.13 Write and run a program that causes negative overflow of a variable of type `short`.

1.14 Write and run a program that demonstrates round-off error by executing the following steps: (1) initialize a variable `a` of type `float` with the value 666666; (2) initialize a variable `b` of type `float` with the value `1-1/a`; (3) initialize a variable `c` of type `float` with the value `1/b - 1`; (4) initialize a variable `d` of type `float` with the value `1/c + 1`; (5) print all four variables. Show algebraically that `d` = `a` even though the computed value of `d` ≠ `a`. This is caused by round-off error.

Answers to Review Questions

1.1 One way is to use the standard C style comment

```
/* like this */
```

The other way is to use the standard C++ style comment

```
// like this
```

The first begins with a slash-star and ends with a star-slash. The second begins with a double-slash and ends at the end of the line.

1.2 The semicolon is missing from the last statement.

1.3 Everything between the double quotes will be printed, including the intended comment.

1.4 There are four errors: the precompiler directive on the first line should not end with a semicolon, the parentheses are missing from `main()`, `n` is not declared, and the quotation mark on the last line has no closing quotation mark.

1.5 A declaration tells the compiler the name and type of the variable being declared. It also may be initialized in the declaration.

1.6 It includes contents of the header file `iostream` into the source code. This includes declarations needed for input and output; *e.g.*, the output operator `<<`.

1.7 `int main() { }`

1.8 The name refers to the C language and its increment operator `++`. The name suggests that C++ is an advance over C.

1.9 The only thing wrong with these declarations is that `new` is a keyword. Keywords are reserved and cannot be used for names of variables. See Appendix B for a list of the 62 keywords in C++.

1.10 ***a.*** `m` will be 10 and `n` will be 3.

b. `m` will be 6 and `n` will be 1.

1.11 ***a.*** `m - 8 - n` evaluates to (25 - 8) - 7 = 17 - 7 = 10

b. `m = n = 3` evaluates to 3

1.12 ***a.*** `m - 8 - n` evaluates to (25 - 8) - 7 = 17 - 7 = 10

b. `m = n = 3` evaluates to 3

c. `m%n` evaluates to 25%7 = 4

d. `m%n++` evaluates to 25%(7++) = 25%7 = 4

e. `m%++n` evaluates to 25%(++7) = 25%8 = 1

f. `++m - n--` evaluates to (++25) - (7--) = 26 - 7 = 19

1.13 The keyword is `int`. The identifiers are `main`, `n`, `cin`, `cout`, and `endl`. The operators are `()`, `>>`, `*=`, and `<<`. The literals are `3` and `"n="`. The punctuation symbols are `{`, `;`, and `}`. The comment is "`// multiply n by 3`".

1.14 ***a.*** The output object `cout` requires the output operator `<<`. It should be `cout << count;`

b. The word `double` is a keyword in C++; it cannot be used as a variable name. Use: `int d=44;`

1.15 Both statements have the same effect: they declare `ch` to be a `char` and initialize it with the value 65. Since this is the ASCII code for `'A'`, that character constant can also be used to initialize `ch` to 65.

1.16 `cout << "char(100) = " << char(100) << endl;`

1.17 The term "floating-point" is used to describe the way decimal numbers (rational numbers) are stored in a computer. The name refers to the way that a rational number like 386501.294 can be represented in the form 3.86501294×10^5 by letting the decimal point "float" to the left 5 places.

1.18 Numeric overflow occurs in a computer program when the size of a numeric variable gets too big for its type. For example, on most computers values variables of type `short` cannot exceed 32,767, so if a variable of that type has the value 32,767 and is then incremented (or increased by any arithmetic operation), overflow will occur.

1.19 When integer overflow occurs the value of the offending variable will "wrap around" to negative values, producing erroneous results. When floating-point overflow occurs, the value of the offending variable will be set to the constant `inf` representing infinity.

1.20 A run-time error is an error that occurs when a program is running. Numeric overflow and division by zero are examples of run-time errors.

1.21 A compile-time error is an error that occurs when a program is being compiled. Examples: syntax errors such as omitting a required semicolon, using an undeclared variable, using a keyword for the name of a variable.

Solutions to Problems

1.1 Four different statements, each subtracting 1 from the integer variable `n`:

a. `n = n - 1;`

b. `n -= 1;`

c. `--n;`

d. `n--;`

1.2
```
n = 100 + m;
++m;
```

1.3
```
++m;
n = 100 + m;
```

1.4
```
z -= (x + y++);
```

1.5
```
total += --n;
```

1.6
```
int main()
{ // prints the first sentence of the Gettysburg Address
  cout << "\tFourscore and seven years ago our fathers\n";
  cout << "brought forth upon this continent a new nation,\n";
  cout << "conceived in liberty, and dedicated to the\n";
  cout << "proposition that all men are created equal.\n";
}
```
```
        Fourscore and seven years ago our fathers
brought forth upon this continent a new nation,
conceived in liberty, and dedicated to the
proposition that all men are created equal.
```

1.7
```
int main()
{ // prints "B" as a block letter
  cout << "*****"  << endl;
  cout << "*    *" << endl;
  cout << "*    *" << endl;
  cout << "*****"  << endl;
  cout << "*    *" << endl;
  cout << "*    *" << endl;
```

```
  cout << "*****"  << endl;
}
```

```
*****
*    *
*    *
*****
*    *
*    *
*****
```

1.8

```
int main()
{ // prints "W" as a block letter
  cout << "*                  *"  << endl;
  cout << " *                *" << endl;
  cout << "  *              *" << endl;
  cout << "   *     *     *"  << endl;
  cout << "    *   * *   *" << endl;
  cout << "     * *   * *" << endl;
  cout << "      *     *"  << endl;
}
```

```
*                  *
 *                *
  *              *
   *     *     *
    *   * *   *
     * *   * *
      *     *
```

1.9

```
int main()
{ // prints escape sequences
  cout << "Prints \"\\nXXYY\": " << "\nXXYY" << endl;
  cout << "-----------------------------------------" << endl;
  cout << "Prints \"\\nXX\\bYY\": " << "\nXX\bYY" << endl;
  cout << "-----------------------------------------" << endl;
  cout << "Prints \"\\n\\tXX\\tYY\": " << "\n\tXX\tYY" << endl;
  cout << "-----------------------------------------" << endl;
  cout << "Prints the \'\\a\' character: " << '\a' << endl;
  cout << "-----------------------------------------" << endl;
  cout << "Prints the \'\\r\' character: " << '\r' << endl;
  cout << "-----------------------------------------" << endl;
  cout << "Prints the \'\\v\' character: " << '\v' << endl;
  cout << "-----------------------------------------" << endl;
  cout << "Prints the \'\\?\' character: " << '\?' << endl;
  cout << "-----------------------------------------" << endl;
}
```

```
Prints the '\v' character:
-----------------------------------------
Prints the '\?' character: ?
-----------------------------------------
Prints "\nXXYY":
XXYY
-----------------------------------------
Prints "\nXX\bYY":
XYY
-----------------------------------------
Prints "\n\tXX\tYY":
```

```
        XX      YY
-----------------------------------------
Prints the '\a' character:
-----------------------------------------
Prints the '\r' character:
-----------------------------------------
```

1.10

```
int main()
{ // prints the results of arithmetic operators
  int m = 60, n = 7;
  cout << "The integers are " << m << " and " << n << endl;
  cout << "Their sum is        " << (m + n) << endl;
  cout << "Their difference is " << (m - n) << endl;
  cout << "Their product is    " << (m * n) << endl;
  cout << "Their quotient is   " << (m / n) << endl;
  cout << "Their remainder is  " << (m % n) << endl;
}
```

```
The integers are 60 and 7
Their sum is        67
Their difference is 53
Their product is    420
Their quotient is   8
Their remainder is  4
```

1.11

```
int main()
{ // prints the results of arithmetic operators
  int m, n;
  cout << "Enter two integers: ";
  cin >> m >> n;
  cout << "The integers are " << m << " and " << n << endl;
  cout << "Their sum is        " << (m + n) << endl;
  cout << "Their difference is " << (m - n) << endl;
  cout << "Their product is    " << (m * n) << endl;
  cout << "Their quotient is   " << (m / n) << endl;
  cout << "Their remainder is  " << (m % n) << endl;
}
```

```
Enter two integers: 60 7
The integers are 60 and 7
Their sum is        67
Their difference is 53
Their product is    420
Their quotient is   8
Their remainder is  4
```

1.12

```
int main()
{ // prints the values of uninitialized variables
  bool b;  // not initialized
  cout << "b = " << b << endl;
  char c;  // not initialized
  cout << "c = [" << c << "]" << endl;
  int m;  // not initialized
  cout << "m = " << m << endl;
  int n;  // not initialized
  cout << "n = " << n << endl;
  long nn;  // not initialized
```

```
  cout << "nn = " << nn << endl;
  float x;  // not initialized
  cout << "x = " << x << endl;
  double y;  // not initialized
  cout << "y = " << y << endl;
}
```

```
b = 0
c =
m = 4296913
n = 4296716
nn = 4296794
x = 6.02438e-39
y = 9.7869e-307
```

1.13

```
int main()
{ // prints the values an overflowing negative short int
  short m=0;
  cout << "m = " << m << endl;
  m -= 10000;  // m should be -10,000
  cout << "m = " << m << endl;
  m -= 10000;  // m should be -20,000
  cout << "m = " << m << endl;
  m -= 10000;  // m should be -30,000
  cout << "m = " << m << endl;
  m -= 10000;  // m should be -40,000
  cout << "m = " << m << endl;
}
```

```
m = 0
m = -10000
m = -20000
m = -30000
m = 25536
```

1.14

```
int main()
{ float a = 666666;              // = a = 666666
  float b = 1 - 1/a;             // = (a-1)/a = 666665/666666
  float c = 1/b - 1;             // = 1/(a-1) = 1/666665
  float d = 1/c + 1;             // = a = 666666 != 671089
  cout << "a = " << a << endl;
  cout << "b = " << b << endl;
  cout << "c = " << c << endl;
  cout << "d = " << d << endl;
}
```

```
a = 666666
b = 0.999999
c = 1.49012e-06
d = 671089
```

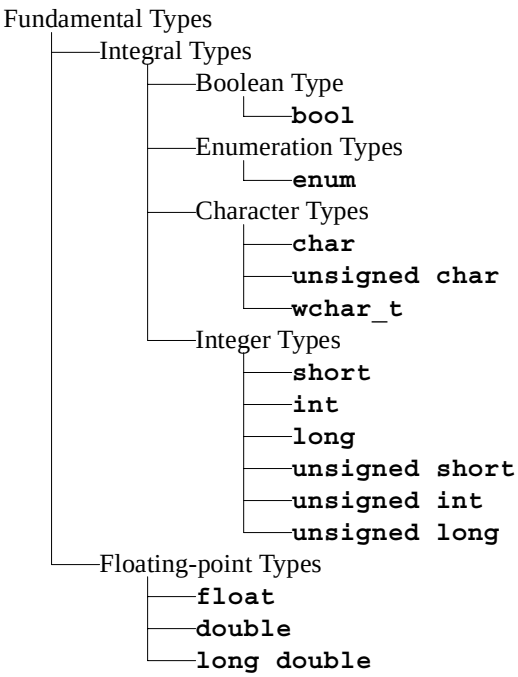
Chapter 2

Fundamental Types

2.1 NUMERIC DATA TYPES

In science there are two kinds of numbers: whole numbers (*e.g.*, 666) and decimal numbers (*e.g.*, 3.14159). Whole numbers, including 0 and negative whole numbers, are called *integers*. Decimal numbers, including negative decimal numbers and all integers, are called *rational numbers* because they can always be expressed as ratios of whole numbers (*i.e.*, fractions). Mathematics also uses irrational real numbers (*e.g.*, $\sqrt{2}$ and π), but these must be approximated with rational numbers to be used in computers.

Integers are used for counting; rational numbers are used for measuring. Integers are meant to be exact; rational numbers are meant to be approximate. When we say there are 12 people on the jury, we mean exactly 12, and anyone can count them to verify the statement. But when we say the tree is 12 meters high, we mean approximately 12.0 meters, and someone else may be just as accurate in saying that it is 12.01385 meters high.



This philosophical dichotomy is reflected in computers by the different ways in which these two fundamentally different kinds of numbers are stored and manipulated. Those differences are embodied in the two kinds of numeric types common to all programming languages: integral types and floating-point types. The term "floating-point" refers to the scientific notation that is used for rational numbers. For example, 1234.56789 can also be represented as 1.23456789×10^{3}, and 0.00098765 as 9.8765×10^{-4}. These alternatives are obtained by letting the decimal point "float" among the digits and using the exponent on 10 to count how many places it has floated to the left or right.

Standard C++ has 14 different *fundamental types*: 11 integral types and 3 floating-point types. These are outlined in the diagram shown above. The *integral types* include the boolean type **`bool`**, enumeration types defined with the **`enum`** keyword, three character types, and six explicit integer types. The three *floating-point types* are **`float`**, **`double`**, and **`long double`**. The most frequently used fundamental types are **`bool`**, **`char`**, **`int`**, and **`double`**.

2.2 THE BOOLEAN TYPE

A *boolean* type is an integral type whose variables can have only two values: **false** and **true**. These values are stored as the integers 0 and 1. The boolean type in Standard C++ is named **bool**.

EXAMPLE 2.1 Boolean Variables

```
int main()
{ // prints the value of a boolean variable:
  bool flag=false;
  cout << "flag = " << flag << endl;
  flag = true;
  cout << "flag = " << flag << endl;
}
```

```
flag = 0
flag = 1
```

Note that the value `false` is printed as the integer `0` and the value `true` is printed as the integer `1`.

2.3 ENUMERATION TYPES

In addition to the predefined types such as **int** and **char**, C++ allows you to define your own special data types. This can be done in several ways, the most powerful of which use classes as described in Chapter 11. We consider here a much simpler kind of user-defined type.

An *enumeration type* is an integral type that is defined by the user with the syntax

enum *typename* **{** *enumerator-list* **};**

Here **enum** is a C++ keyword, *typename* stands for an identifier that names the type being defined, and *enumerator-list* stands for a list of names for integer constants. For example, the following defines the enumeration type `Semester`, specifying the three possible values that a variable of that type can have

```
enum Semester {FALL, SPRING, SUMMER};
```

We can then declare variables of this type:

```
Semester s1, s2;
```

and we can use those variables and those type values as we would with predefined types:

```
s1 = SPRING;
s2 = FALL;
if (s1 == s2) cout << "Same semester." << endl;
```

The actual values defined in the enumerator-list are called *enumerators*. In fact, they are ordinary integer constants. For example, the enumerators `FALL`, `SPRING`, and `SUMMER` that are defined for the `Semester` type above could have been defined like this:

```
const int FALL=0;
const int WINTER=1;
const int SUMMER=2;
```

The values 0, 1, … are assigned automatically when the type is defined. These default values can be overridden in the *enumerator-list*:

```
enum Coin {PENNY=1, NICKEL=5, DIME=10, QUARTER=25};
```

If integer values are assigned to only some of the enumerators, then the ones that follow are given consecutive values. For example,

```
enum Month {JAN=1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV
            DEC};
```

will assign the numbers 1 through 12 to the twelve months.

Since enumerators are simply integer constants, it is legal to have several different enumerators with the same value:

```
enum Answer {NO = 0, FALSE=0, YES = 1, TRUE=1, OK = 1};
```

This would allow the code

```
int answer;
cin >> answer;
  ⋮
  ⋮
if (answer == YES) cout << "You said it was o.k." << endl;
```

to work as expected. If the value of the variable `answer` is 1, then the condition will be true and the output will occur. Note that since the integer value 1 always means "true" in a condition, this selection statement could also be written

```
if (answer) cout << "You said it was o.k." << endl;
```

Notice the conspicuous use of capitalization here. Most programmers usually follow these conventions for capitalizing their identifiers:

1. Use only upper-case letters in names of constants.
2. Capitalize the first letter of each name in user-defined types.
3. Use all lower-case letters everywhere else.

These rules make it easier to distinguish the names of constants, types, and variables, especially in large programs. Rule 2 also helps distinguish standard C++ types like `float` and `string` from user-defined types like `Coin` and `Month`.

Enumeration types are usually defined to make code more *self-documenting*; *i.e.*, easier for humans to understand. Here are a few more typical examples:

```
enum Sex {FEMALE, MALE};
enum Day {SUN, MON, TUE, WED, THU, FRI, SAT};
enum Radix {BIN=2, OCT=8, DEC=10, HEX=16};
enum Color {RED, ORANGE, YELLOW, GREEN, BLUE, VIOLET};
enum Rank {TWO=2, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN,
           JACK, QUEEN, KING, ACE};
enum Suit {CLUBS, DIAMONDS, HEARTS, SPADES};
enum Roman {I=1, V=5, X=10, L=50, C=100, D=500, M=1000};
```

Definitions like these can help make your code more readable. But enumerations should not be overused. Each enumerator in an enumerator list defines a new identifier. For example, the definition of `Roman` above defines the seven identifiers `I`, `V`, `X`, `L`, `C`, `D`, and `M` as specific integer constants, so these letters could not be used for any other purpose within the scope of their definition.

Note that enumerators must be valid identifiers. So for example, this definition would not be valid

```
enum Grade {F, D, C-, C, C+, B-, B, B+, A-, A};          // ERRONEOUS
```

because the characters `'+'` and `'-'` cannot be used in identifiers. Also, the definitions for `Month` and `Radix` shown above could not both be in the same scope because they both define the symbol `OCT`.

Enumerations can also be anonymous in C++:

```
enum {I=1, V=5, X=10, L=50, C=100, D=500, M=1000};
```

This is just a convenient way to define integer constants.

2.4 CHARACTER TYPES

A *character type* is an integral type whose variables represent characters like the letter `'A'` or the digit `'8'`. Character literals are delimited by the apostrophe (`'`). Like all integral type values, character values are stored as integers.

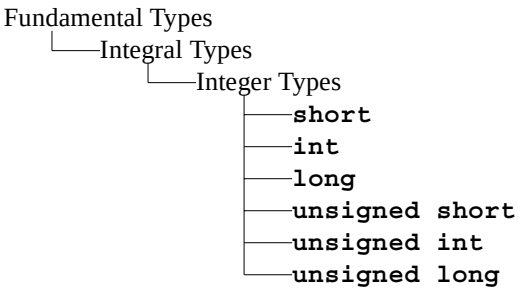
EXAMPLE 2.2 Character Variables

```
int main()
{ // prints the character and its internally stored integer value:
  char c='A';
  cout << "c = " << c << ", int(c) = " << int(c) << endl;
  c='t';
  cout << "c = " << c << ", int(c) = " << int(c) << endl;
  c='\t';  // the tab character
  cout << "c = " << c << ", int(c) = " << int(c) << endl;
  c='!';
  cout << "c = " << c << ", int(c) = " << int(c) << endl;
}
```

```
c = A, int(c) = 65
c = t, int(c) = 116
c =     , int(c) = 9
c = !, int(c) = 33
```

Since character values are used for input and output, they appear in their character form instead of their integral form: the character `'A'` is printed as the letter "A", not as the integer 65 which is its internal representation. The *type cast operator* `int()` is used here to reveal the corresponding integral value. These are the characters' ASCII codes. (See Appendix A.)

2.5 INTEGER TYPES



There are 6 integer types in Standard C++: These types actually have several names. For example, **`short`** is also named **`short int`**, and **`int`** is also named **`signed int`**.

You can determine the numerical ranges of the integer types on your system by running the program in the following example.

EXAMPLE 2.3 Integer Type Ranges

This program prints the numeric ranges of the 6 integer types in C++:

```
#include <iostream>
#include <climits>    // defines the constants SHRT_MIN, etc.
using namespace std;
int main()
{ // prints some of the constants stored in the <climits> header:
  cout << "minimum short = " << SHRT_MIN << endl;
  cout << "maximum short = " << SHRT_MAX << endl;
```

```
    cout << "maximum unsigned short = 0" << endl;
    cout << "maximum unsigned short = " << USHRT_MAX << endl;
    cout << "minimum int = " << INT_MIN << endl;
    cout << "maximum int = " << INT_MAX << endl;
    cout << "minimum unsigned int = 0" << endl;
    cout << "maximum unsigned int = " << UINT_MAX << endl;
    cout << "minimum long= " << LONG_MIN << endl;
    cout << "maximum long= " << LONG_MAX << endl;
    cout << "minimum unsigned long = 0" << endl;
    cout << "maximum unsigned long = " << ULONG_MAX << endl;
}
```

```
minimum short = -32768
maximum short = 32767
maximum unsigned short = 0
maximum unsigned short = 65535
minimum int = -2147483648
maximum int = 2147483647
minimum unsigned int= 0
maximum unsigned int= 4294967295
minimum long = -2147483648
maximum long = 2147483647
minimum unsigned long = 0
maximum unsigned long = 4294967295
```

The header file `<climits>` defines the constants `SHRT_MIN`, `SHRT_MAX`, `USHRT_MIN`, *etc.* These are the limits on the range of values that a variable of the indicated type can have. For example, the output shows that variables of type **`int`** can have values in the range –2,147,483,648 to 2,147,483,647 on this computer.

On this computer, the three **`signed`** integer types have the same range as their corresponding unqualified integer type. For example, **`signed short int`** is the same as **`short int`**. This tells us that the **`signed`** integer types are redundant on this computer.

The output also reveals that the range of the **`int`** type (–2,147,483,648 to 2,147,483,647) is the same as that of the **`long int`** type, and that the range of the **`unsigned int`** type (0 to 4,294,967,295) is the same as that of the **`unsigned long int`** type. This tells us that the **`long`** integer types are redundant on this computer.

The output from Example 2.3 shows that on this computer (a Pentium II PC running the Windows 98 operating system and the CodeWarrior 3.2 C++ compiler), the six integer types have the following ranges:

`short`:	–32,768 to 32,767;	(2^8 values ⇒ 1 byte)
`int`:	–2,147,483,648 to 2,147,483,647;	(2^{32} values ⇒ 4 bytes)
`long`:	–2,147,483,648 to 2,147,483,647;	(2^{32} values ⇒ 4 bytes)
`unsigned short`:	0 to 65,535;	(2^8 values ⇒ 1 byte)
`unsigned int`:	0 to 4,294,967,295;	(2^{32} values ⇒ 4 bytes)
`unsigned long`:	0 to 4,294,967,295;	(2^{32} values ⇒ 4 bytes)

Note that **`long`** is the same as **`int`** and **`unsigned long`** is the same as **`unsigned int`**.

The **`unsigned`** integer types are used for bit strings. A *bit string* is a string of 0s and 1s as is stored in the computer's random access memory (RAM) or on disk. Of course, everything stored in a computer, in RAM or on disk, is stored as 0s and 1s. But all other types of data are formatted; *i.e.*, interpreted as something such as a signed integer or a string of characters.

2.6 ARITHMETIC OPERATORS

Computers were invented to perform numerical calculations. Like most programming languages, C++ performs its numerical calculations by means of the five *arithmetic operators* `+`, `-`, `*`, `/`, and `%`.

EXAMPLE 2.4 Integer Arithmetic

This example illustrates how the arithmetic operators work.

```
int main()
{ // tests operators +, -, *, /, and %:
  int m=54;
  int n=20;
  cout << "m = " << m << " and n = " << n << endl;
  cout << "m+n = " << m+n << endl;  // 54+20 = 74
  cout << "m-n = " << m-n << endl;  // 54-20 = 34
  cout << "m*n = " << m*n << endl;  // 54*20 = 1080
  cout << "m/n = " << m/n << endl;  // 54/20 = 2
  cout << "m%n = " << m%n << endl;  // 54%20 = 14
}
```

```
m = 54 and n = 20
m+n = 74
m-n = 34
m*n = 1080
m/n = 2
m%n = 14
```

Note that integer division results in another integer: 54/20 = 2, not 2.7.

The last two operators used in Example 2.4 are the *division operator* `/` and the *modulus operator* `%` (also called the *remainder operator*). The modulus operator results in the remainder from the division. Thus, 54%20 = 14 because 14 is the remainder after 54 is divided by 20.

2.7 THE INCREMENT AND DECREMENT OPERATORS

The values of integral objects can be incremented and decremented with the `++` and `--` operators, respectively. Each of these operators has two versions: a "pre" version and a "post" version. The "pre" version performs the operation (either adding 1 or subtracting 1) on the object before the resulting value is used in its surrounding context. The "post" version performs the operation after the object's current value has been used.

EXAMPLE 2.5 Applying the Pre-increment and Post-increment Operators

```
int main()
{ // shows the difference between m++ and ++m:
  int m, n;
  m = 44;
  n = ++m;  // the pre-increment operator is applied to m
  cout << "m = " << m << ", n = " << n << endl;
```

```
  m = 44;
  n = m++;  // the post-increment operator is applied to m
  cout << "m = " << m << ", n = " << n << endl;
}
m = 45, n = 45
m = 45, n = 44
```

The line

```
n = ++m;  // the pre-increment operator is applied to m
```

increments `m` to 45 and then assigns that value to `n`. So both variables have the same value 45 when the next output line executes.

The line

```
n = m++;  // the post-increment operator is applied to m
```

increments `m` to 45 only after it has assigned the value of `m` to `n`. So `n` has the value 44 when the next output line executes.

2.8 COMPOSITE ASSIGNMENT OPERATORS

The standard assignment operator in C++ is the equals sign `=`. In addition to this operator, C++ also includes the following *composite assignment operators*: `+=`, `-=`, `*=`, `/=`, and `%=`. When applied to a variable on the left, each applies the indicated arithmetic operation to it using the value of the expression on the right.

EXAMPLE 2.6 Applying Composite Arithmetic Assignment Operators

```
int main()
{ // tests arithmetic assignment operators:
  int n=22;
  cout << "n = " << n << endl;
  n += 9;  // adds 9 to n
  cout << "After n += 9, n = " << n << endl;
  n -= 5;  // subtracts 5 from n
  cout << "After n -= 5, n = " << n << endl;
  n *= 2;  // multiplies n by 3
  cout << "After n *= 2, n = " << n << endl;
  n /= 3;  // divides n by 9
  cout << "After n /= 3, n = " << n << endl;
  n %= 7;  // reduces n to the remainder from dividing by 4
  cout << "After n %= 7, n = " << n << endl;
}
n = 22
After n += 9, n = 31
After n -= 5, n = 26
After n *= 2, n = 52
After n /= 3, n = 17
After n %= 7, n = 3
```

2.9 FLOATING-POINT TYPES

C++ supports three real number types: **float**, **double**, and **long double**. On most systems, **double** uses twice as many bytes as **float**. Typically, **float** uses 4 bytes, **double** uses 8 bytes, and **long double** uses 8, 10, 12, or 16 bytes.

Types that are used for real numbers are called "floating-point" types because of the way they are stored internally in the computer. On most systems, a number like 123.45 is first converted to binary form:

$$123.45 = 1111011.01110011_2 \times 2^7$$

Then the point is "floated" so that all the bits are on its right. In this example, the floating-point form is obtained by floating the point 7 bits to the left, producing a mantissa 2^7 times smaller. So the original number is

$$123.45 = 0.111101101110011_2 \times 2^7$$

This number would be represented internally by storing the mantissa `111101101110011` and the exponent 7 separately. For a 32-bit **float** type, the *mantissa* is stored in a 23-bit segment and the exponent in an 8-bit segment, leaving 1 bit for the sign of the number. For a 64-bit **double** type, the mantissa is stored in a 52-bit segment and the exponent in an 11-bit segment.

EXAMPLE 2.7 Floating-Point Arithmetic

This program is nearly the same as the one in Example 2.4. The important difference is that these variables are declared to have the floating-point type `double` instead of the integer type `int`.

```
int main()
{ // tests the floating-point operators +, -, *, and /:
  double x=54.0;
  double y=20.0;
  cout << "x = " << x << " and y = " << y << endl;
  cout << "x+y = " << x+y << endl;  // 54.0+20.0 = 74.0
  cout << "x-y = " << x-y << endl;  // 54.0-20.0 = 34.0
  cout << "x*y = " << x*y << endl;  // 54.0*20.0 = 1080.0
  cout << "x/y = " << x/y << endl;  // 54.0/20.0 = 2.7
}
```

```
x = 55 and y = 20
x+y = 75
x-y = 35
x*y = 1100
x/y = 2.7
```

Unlike integer division, floating-point division does not truncate the result: 54.0/20.0 = 2.7.

The next example can be used on any computer to determine how many bytes it uses for each type. The program uses the **sizeof** operator which returns the size in bytes of the type specified.

EXAMPLE 2.8 Using the `sizeof` Operator

This program tells you how much space each of the 12 fundamental types uses:

```
int main()
{ // prints the storage sizes of the fundamental types:
  cout << "Number of bytes used:\n";
```

```
  cout << "\t          char: " <<           sizeof(char) << endl;
  cout << "\t         short: " <<          sizeof(short) << endl;
  cout << "\t           int: " <<            sizeof(int) << endl;
  cout << "\t          long: " <<           sizeof(long) << endl;
  cout << "\t unsigned char: " << sizeof(unsigned char) << endl;
  cout << "\tunsigned short: " << sizeof(unsigned short) << endl;
  cout << "\t  unsigned int: " <<  sizeof(unsigned int) << endl;
  cout << "\t unsigned long: " << sizeof(unsigned long) << endl;
  cout << "\t   signed char: " <<   sizeof(signed char) << endl;
  cout << "\t         float: " <<         sizeof(float) << endl;
  cout << "\t        double: " <<        sizeof(double) << endl;
  cout << "\t   long double: " <<   sizeof(long double) << endl;
}
```

```
Number of bytes used:
                  char: 1
                 short: 2
                   int: 4
                  long: 4
         unsigned char: 1
        unsigned short: 2
          unsigned int: 4
         unsigned long: 4
           signed char: 1
                 float: 4
                double: 8
           long double: 8
```

The output below shows the sizes for a typical UNIX workstation. On this machine, **`int`** and **`long`** are equivalent, **`unsigned int`** and **`unsigned long`** are equivalent, and **`double`** and **`long double`** are equivalent. In other words, 'long' is no different from 'regular' on this computer.

The next program can be used to investigate floating-point types on any computer system. It reads the values of various constants from the **`<cfloat>`** header file. To access it, the program must include the preprocessor directive:

```
#include <cfloat>
```

This is like the **`#include <iostream>`** directive that we always include in order to use the **`cin`** and **`cout`** objects.

EXAMPLE 2.9 Reading from the `<cfloat>` Header File

This program tells you the precision and magnitude range that the **`float`** type has on your system:

```
#include <cfloat>  // defines the FLT constants
#include <iostream>  // defines the FLT constants
using namespace std;
int main()
{ // prints the storage sizes of the fundamental types:
  int fbits = 8*sizeof(float);        // each byte contains 8 bits
  cout << "float uses " << fbits << " bits:\n\t"
       << FLT_MANT_DIG - 1 << " bits for its mantissa,\n\t "
       << fbits - FLT_MANT_DIG << " bits for its exponent,\n\t "
       << 1 << " bit for its sign\n"
       << "          to obtain: " << FLT_DIG << " sig. digits\n"
```

```
        << " with minimum value: " << FLT_MIN << endl
        << "  and maximum value: " << FLT_MAX << endl;
}
```

```
float uses 32 bits:
        23 bits for its mantissa,
         8 bits for its exponent,
         1 bit for its sign
          to obtain: 6 sig. digits
 with minimum value: 1.17549e-38
  and maximum value: 3.40282e+38
```

The constants `FLT_MANT_DIG`, `FLT_DIG`, `FLT_MIN`, and `FLT_MAX` are defined in the `<cfloat>` header file.

This output is from a UNIX workstation. It shows that the 32 bits it uses to store a `float` are partitioned into 3 parts: 23 bits for the mantissa, 8 bits for the exponent, and 1 bit for the sign. The 23-bit mantissa produces a floating-point value with 6 significant digits, and the 8-bit exponent yields a range in magnitude from about 10^{-37} to about 3×10^{38}. *i.e.*,

$$0.00000000000000000000000000000000000001 < |x| < 300{,}000{,}000{,}000{,}000{,}000{,}000{,}000{,}000{,}000{,}000{,}000{,}000$$

for any variable *x* declared to have type **`float`**.

All floating-point arithmetic is done in **`double`** precision. So the only time you should use **`float`** instead of **`double`** is when you are storing large quantities of real numbers and are concerned about storage space or access time.

2.10 TYPE CONVERSIONS

We saw in Chapter 1 how one integer type can be converted automatically to another. C++ also converts integral types into floating point types when they are expected. For example,

```
int n = 22;
float x = 3.14159;
x += n;          // the value 22 is automatically converted to 22.0
cout << x - 2 << endl; // value 2 is automatically converted to 2.0
```

Converting from integer to float like this is what one would expect and is usually taken for granted. But converting from a floating point type to an integral type is not automatic.

In general, if `T` is one type and `v` is a value of another type, then the expression

```
T(v)
```

converts `v` to type `T`. This is called *type casting*. For example, if *`expr`* is a floating-point expression and `n` is a variable of type **`int`**, then

```
n = int(expr);
```

converts the value of *`expr`* to type **`int`** and assigns it to `n`. The effect is to remove the real number's fractional part, leaving only its whole number part to be assigned to `n`. For example, 2.71828 would be converted to 2. Note that this is *truncating*, not *rounding*.

EXAMPLE 2.10 Simple Type Casting

This program casts a `double` value into `int` value:

```
int main()
{ // casts a double value as an int:
  double v = 1234.56789;
  int n = int(v);
```

```
  cout << "v = " << v << ", n = " << n << endl;
}
v = 1234.57, n = 1234
```

The `double` value 1234.56789 is converted to the **`int`** value 1234.

When one type is to be converted to a "higher" type, the type case operator is not needed. This is called *type promotion*. Here's a simple example of *promotion* from **`char`** all the way up to **`double`**:

EXAMPLE 2.11 Promotion of Types

This program promotes a `char` to a `short` to an `int` to a `float` to a `double`:

```
int main()
{ // prints promoted vales of 65 from char to double:
  char c='A';  cout << "  char c = " << c << endl;
  short k=c;   cout << " short k = " << k << endl;
  int m=k;     cout << "   int m = " << m << endl;
  long n=m;    cout << "  long n = " << n << endl;
  float x=m;   cout << " float x = " << x << endl;
  double y=x;  cout << "double y = " << y << endl;
}
  char c = A
 short k = 65
   int m = 65
  long n = 65
 float x = 65
double y = 65
```

The integer value of the character `'A'` is its ASCII code 65. This value is converted as a **`char`** in `c`, a **`short`** in `k`, an **`int`** in `m`, and a **`long`** in `n`. The value is then converted to the floating point value 65.0 and stored as a **`float`** in `x` and as a **`double`** in `y`. Notice that **`cout`** prints the integer `c` as a character, and that it prints the real numbers `x` and `y` as integers because their fractional parts are 0.

Because it is so easy to convert between integer types and real types in C++, it is easy to forget the distinction between them. In general, integers are used for counting discrete things, while reals are used for measuring on a continuous scale. This means that integer values are exact, while real values are approximate.

Note that type casting and promotion convert the type of the value of a variable or expression, but it does not change the type of the variable itself.

In the C programming language, the syntax for casting `v` as type `T` is `(T) v`. C++ inherits this form also, so we could have done `n = int(v)` as `n = (int) v`.

2.11 NUMERIC OVERFLOW

On most computers the `long int` type allows 4,294,967,296 different values. That's a lot of values, but it's still finite. Computers are finite, so the range of any type must also be finite. But in mathematics there are infinitely many integers. Consequently, computers are manifestly prone to error when their numeric values become too large. That kind of error is called *numeric overflow.*

EXAMPLE 2.12 Integer Overflow

This program repeatedly multiplies `n` by 1000 until it overflows.

```
int main()
{ // prints n until it overflows:
  int n=1000;
  cout << "n = " << n << endl;
  n *= 1000;  // multiplies n by 1000
  cout << "n = " << n << endl;
  n *= 1000;  // multiplies n by 1000
  cout << "n = " << n << endl;
  n *= 1000;  // multiplies n by 1000
  cout << "n = " << n << endl;
}
```

```
n = 1000
n = 1000000
n = 1000000000
n = -727379968
```

This shows that the computer that ran this program cannot multiply 1,000,000,000 by 1000 correctly.

EXAMPLE 2.13 Floating-point Overflow

This program is similar to the one in Example 2.12. It repeatedly squares `x` until it overflows.

```
int main()
{ // prints x until it overflows:
  float x=1000.0;
  cout << "x = " << x << endl;
  x *= x;  // multiplies n by itself; i.e., it squares x
  cout << "x = " << x << endl;
  x *= x;  // multiplies n by itself; i.e., it squares x
  cout << "x = " << x << endl;
  x *= x;  // multiplies n by itself; i.e., it squares x
  cout << "x = " << x << endl;
  x *= x;  // multiplies n by itself; i.e., it squares x
  cout << "x = " << x << endl;
}
```

```
x = 1000
x = 1e+06
x = 1e+12
x = 1e+24
x = inf
```

This shows that, starting with `x` = 1000, this computer cannot square `x` correctly more than three times. The last output is the special symbol `inf` which stands for “infinity.”

Note the difference between integer overflow and floating-point overflow. The last output in Example 2.12 is the negative integer –727,379,968 instead of the correct value of 1,000,000,000,000 = 10^{12}. The last output in Example 2.13 is the infinity symbol `inf` instead of the correct value of 10^{48}. Integer overflow “wraps around” to negative integers. Floating-point overflow “sinks” into the abstract notion of infinity.

2.12 ROUND-OFF ERROR

Round-off error is another kind of error that often occurs when computers do arithmetic on rational numbers. For example, the number 1/3 might be stored as 0.333333, which is not exactly equal to 1/3. The difference is called *round-off error.* In some cases, these errors can cause serious problems.

EXAMPLE 2.14 Round-off Error

This program does some simple arithmetic to illustrate roundoff error:

```
int main()
{ // illustrates round-off error::
  double x = 1000/3.0;cout << "x = " << x << endl;  // x = 1000/3
  double y = x - 333.0;cout << "y = " << y << endl; // y = 1/3
  double z = 3*y - 1.0;cout << "z = " << z << endl; // z = 3(1/3) - 1
  if (z == 0) cout << "z == 0.\n";
  else cout << "z does not equal 0.\n";             // z != 0
}
```

```
x = 333.333
y = 0.333333
z = -5.68434e-14
z does not equal 0.
```

In exact arithmetic, the variables would have the values $x = 333\ 1/3$, $y = 1/3$, and $z = 0$. But 1/3 cannot be represented exactly as a floating-point value. The inaccuracy is reflected in the residue value for `z`.

Example 2.14 illustrates an inherent problem with using floating-point types within conditional tests of equality. The test `(z == 0)` will fail even if `z` is very nearly zero, which is likely to happen when `z` should algebraically be zero. So it is better to avoid tests for equality with floating-point types.

The next example shows that round-off error can be difficult to recognize.

EXAMPLE 2.15 Hidden Round-off Error

This program implements the *quadratic formula* to solve quadratic equations.

```
#include <cmath>  // defines the sqrt() function
#include <iostream>
using namespace std;
int main()
{ // implements the quadratic formula
  float a, b, c;
  cout << "Enter the coefficients of a quadratic equation:" << endl;
  cout << "\ta: ";
  cin >> a;
  cout << "\tb: ";
  cin >> b;
  cout << "\tc: ";
  cin >> c;
  cout << "The equation is: " << a << "*x*x + " << b
       << "*x + " << c << " = 0" << endl;
```

```
    float d = b*b - 4*a*c;  // discriminant
    float sqrtd = sqrt(d);
    float x1 = (-b + sqrtd)/(2*a);
    float x2 = (-b - sqrtd)/(2*a);
    cout << "The solutions are:" << endl;
    cout << "\tx1 = " << x1 << endl;
    cout << "\tx2 = " << x2 << endl;
    cout << "Check:" << endl;
    cout << "\ta*x1*x1 + b*x1 + c = " <<  a*x1*x1 + b*x1 + c << endl;
    cout << "\ta*x2*x2 + b*x2 + c = " <<  a*x2*x2 + b*x2 + c << endl;
}
```

The quadratic formula requires computing the square root $\sqrt{b^2 - 4ac}$. This is done on the line

```
float sqrtd = sqrt(d);
```

which calls the square root function `sqrt()` defined in the header file `<cmath>`. The last two lines of the program check the solutions by substituting them back into the original quadratic equation. If the resulting expression on the left evaluates to 0 then the solutions are correct.

This run solves the equation $2x^2 + 1x - 3 = 0$ correctly:

```
Enter the coefficients of a quadratic equation:
        a: 2
        b: 1
        c: -3
The equation is: 2*x*x + 1*x + -3 = 0
The solutions are:
        x1 = 1
        x2 = -1.5
Check:
        a*x1*x1 + b*x1 + c = 0
        a*x2*x2 + b*x2 + c = 0
```

But this run attempts to solve the equation $x^2 + 10000000000x + 1 = 0$ and fails:

```
Enter the coefficients of a quadratic equation:
        a: 1
        b: 1e10
        c: 1
The equation is: 1*x*x + 1e10*x + 1 = 0
The solutions are:
        x1 = 0
        x2 = -1e10
Check:
        a*x1*x1 + b*x1 + c = 1
        a*x2*x2 + b*x2 + c = 1
```

The first solution, $x_1 = 0$, is obviously incorrect: the resulting quadratic expression $ax_1^2 + bx_1 + c$ evaluates to 1 instread of 0. The second solution, $x_2 = -1e10 = -10{,}000{,}000{,}000$ is even worse. The correct solutions are $x_1 = -0.00000\ 00000\ 99999\ 99999\ 99999\ 99519$ and $x_2 = 9{,}999{,}999{,}999.99999\ 99999$.

Numeric overflow and round-off errors are examples of *run-time errors*, which are errors that occur while the program is running. Such errors are more serious than compile-time errors such as neglecting to declare a variable or forgetting a semicolon because they are usually harder to detect and locate. Compile-time errors are caught by the compiler, which usually gives a pretty good report on where they are. But run-time errors are detected only when the user notices that the results are incorrect. Even if the program crashes, it still may be difficult to find where the problem is in the program.

EXAMPLE 2.16 Other Kinds of Run-Time Errors

Here are two more runs of the quadratic formula program in Example 2.15:

```
Enter the coefficients of a quadratic equation:
        a: 1
        b: 2
        c: 3
The equation is: 1*x*x + 2*x + 3 = 0
The solutions are:
        x1 = nan
        x2 = nan
Check:
        a*x1*x1 + b*x1 + c = nan
        a*x2*x2 + b*x2 + c = nan
```

The quadratic equation $1x^2 + 2x + 3 = 0$ has no real solution because the discriminant $b^2 - 4ac$ is negative. When the program runs, the square root function `sqrt(d)` fails because `d` < 0. It returns the symbolic constant `nan` which stands for "not a number." Then every subsequent numeric operation that uses this constant results in the same value. That's why the check values come out as `nan` at the end of the run.

This run attempts to solve the equation $0x^2 + 2x + 5 = 0$. That equation has the solution $x = 2.5$. But the quadratic formula fails because $a = 0$:

```
Enter the coefficients of a quadratic equation:
        a: 0
        b: 2
        c: 5
The equation is: 0*x*x + 2*x + 5 = 0
The solutions are:
        x1 = nan
        x2 = -inf
Check:
        a*x1*x1 + b*x1 + c = nan
        a*x2*x2 + b*x2 + c = nan
```

Notice that x_1 comes out as `nan`, but x_2 comes out as `-inf`. The symbol `inf` stands for "infinity." That's what you get when you divide a nonzero number by zero. The quadratic formula computes x_2 as

$$x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a} = \frac{-(2) - \sqrt{(2)^2 - 4(0)(5)}}{2(0)} = \frac{-2-2}{0} = \frac{-4}{0}$$

which becomes `-inf`. But it computes x_1 as

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} = \frac{-(2) + \sqrt{(2)^2 - 4(0)(5)}}{2(0)} = \frac{-2+2}{0} = \frac{0}{0}$$

which becomes `nan`.

The three symbols `inf`, `-inf`, and `nan` are numeric constants. The usual numeric operators can be applied to them, although the results are usually useless. For example, you can multiply `nan` by any number, but the result will still be `nan`.

2.13 THE E-FORMAT FOR FLOATING-POINT VALUES

When input or output, floating-point values may be specified in either of two formats: *fixed-point* and *scientific*. The output in Example 2.16 illustrates both: `333.333` has fixed-point format, and `-5.68434e-14` has scientific format.

In scientific format, the letter `e` stands for "exponent on 10." So `e-14` means 10^{-14}, and thus `-5.68434e-14` means $-5.68434 \times 10^{-14} = -0.000000000000568434$. Obviously, the scientific format is more efficient for very small or very large numbers.

Floating-point values with magnitude in the range 0.1 to 999,999 will normally be printed in fixed-point format; all others will be printed in scientific format.

EXAMPLE 2.17 Scientific Format

This program shows how floating-point values may be input in scientific format:

```
int main()
{ // prints double values in scientific e-format:
  double x;
  cout << "Enter float: ";  cin >> x;
  cout << "Its reciprocal is: " << 1/x << endl;
}
```

```
Enter float: 234.567e89
Its reciprocal is: 4.26317e-92
```

You can use either `e` or `E` in the scientific format.

2.14 SCOPE

The *scope* of an identifier is that part of the program where it can be used. For example, variables cannot be used before they are declared, so their scopes begin where they are declared. This is illustrated by the next example.

EXAMPLE 2.18 Scope of Variables

```
int main()
{ // illustrates the scope of variables:
  x = 11;    // ERROR: this is not in the scope of x
  int x;
  { x = 22;  // OK: this is in the scope of x
    y = 33;  // ERROR: this is not in the scope of y
    int y;
    x = 44;  // OK: this is in the scope of x
    y = 55;  // OK: this is in the scope of y
  }
  x = 66;    // OK: this is in the scope of x
  y = 77;    // ERROR: this is not in the scope of y
}
```

The scope of `x` extends from the point where it is declared to the end of `main()`. The scope of `y` extends from the point where it is declared to the end of the internal block within which it is declared.

A program may have several objects with the same name as long as their scopes are nested or disjoint. This is illustrated by the next example.

EXAMPLE 2.19 Nested and Parallel Scopes

```
int x = 11;                                          // this x is global

int main()
{ // illustrates the nested and parallel scopes:
  int x = 22;
  { // begin scope of internal block
    int x = 33;
    cout << "In block inside main(): x = " << x << endl;
  }                                          // end scope of internal block
  cout << "In main(): x = " << x << endl;
  cout << "In main(): ::x = " << ::x << endl;
}                                                    // end scope of main()
In block inside main(): x = 33
In main(): x = 22
In main(): ::x = 11
```

There are three different objects named `x` in this program. The `x` that is initialized with the value 11 is a global variable, so its scope extends throughout the file. The `x` that is initialized with the value 22 has scope limited to `main()`. Since this is nested within the scope of the first `x`, it hides the first `x` within `main()`. The `x` that is initialized with the value 33 has scope limited to the internal block within `main()`, so it hides both the first and the second `x` within that block.

The last line in the program uses the *scope resolution operator* **`::`** to access the global `x` that is otherwise hidden in `main()`.

Review Questions

2.1 Write a single C++ statement that prints `"Too many"` if the variable `count` exceeds 100.

2.2 What is wrong with the following code:

a.
```
cin << count;
```

b.
```
if x < y min = x
  else min = y;
```

2.3 What is wrong with this code:

```
cout << "Enter n: ";
cin >> n;
if (n < 0)
  cout << "That is negative. Try again." << endl;
  cin >> n;
else
  cout << "o.k. n = " << n << endl;
```

2.4 What is the difference between a reserved word and a standard identifier?

2.5 What is wrong with this code:

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

2.6 What is wrong with this code:

```
enum Friends {"Jerry", "Henry", "W.D."};
```

Problems

2.1 Write and run a program like the one in Example 2.2 on page 19 that prints the ASCII codes for only the 10 upper case and lower case vowels. Use Appendix A to check your output.

2.2 Modify the program in Example 2.15 on page 28 so that it uses type `double` instead of `float`. Then see how much better it performs on the input that illustrated round-off error.

2.3 Write and run a program to find which, if any, arithmetic operations can be applied to a variable that will change its value from any of the three numeric constants `inf`, `-inf`, and `nan` to something else.

2.4 Write a program that converts inches to centimeters. For example, if the user enters 16.9 for a length in inches, the output would be `42.926 cm`. (One inch equals 2.54 centimeters.)

Answers to Review Questions

2.1 `if (count > 100) cout << "Too many";`

2.2 ***a.*** Either `cout` should be used in place of `cin`, or the extraction operator `>>` should be used in place of the insertion operator `<<`.

b. Parentheses are required around the condition `x < y`, and a semicolon is required at the end of the `if` clause before the `else`.

2.3 There is more than one statement between the `if` clause and the `else` clause. They need to be made into a compound statement by enclosing them in braces { }.

2.4 A *reserved word* is a keyword in a programming language that serves to mark the structure of a statement. For example, the keywords `if` and `else` are reserved words. A *standard identifier* is a keyword that defines a type. Among the 63 keywords in C++, `if`, `else`, and `while` are some of the reserved words, and `char`, `int`, and `float` are some of the standard identifiers.

2.5 The second `enum` definition attempts to redefine the constants `SPRING`, `SUMMER`, and `FALL`.

2.6 Enumerators must be valid identifiers. String literals like `"Jerry"` and `"Henry"` are not identifiers.

Solutions to Problems

2.1

```
int main()
{ // prints the ASCII codes of the vowels
  cout << "int('A') = " << int('A') << endl;
  cout << "int('E') = " << int('E') << endl;
  cout << "int('I') = " << int('I') << endl;
  cout << "int('O') = " << int('O') << endl;
  cout << "int('U') = " << int('U') << endl;
  cout << "int('a') = " << int('a') << endl;
  cout << "int('e') = " << int('e') << endl;
  cout << "int('i') = " << int('i') << endl;
  cout << "int('o') = " << int('o') << endl;
  cout << "int('u') = " << int('u') << endl;
}
```

```
int('A') = 65
int('E') = 69
int('I') = 73
int('O') = 79
int('U') = 85
```

```
int('a') = 97
int('e') = 101
int('i') = 105
int('o') = 111
int('u') = 117
```

2.2

```
int main()
{ // implements the quadratic formula
  double a, b, c;
  cout << "Enter the coefficients:" << endl;
  cout << "\ta: ";
  cin >> a;
  cout << "\tb: ";
  cin >> b;
  cout << "\tc: ";
  cin >> c;
  cout << "The equation is: " << a << "*x*x + " << b
       << "*x + " << c << " = 0" << endl;
  double d = b*b - 4*a*c;
  double sqrtd = sqrt(d);
  double x1 = (-b + sqrtd)/(2*a);
  double x2 = (-b - sqrtd)/(2*a);
  cout << "The solutions are:" << endl;
  cout << "\tx1 = " << x1 << endl;
  cout << "\tx2 = " << x2 << endl;
  cout << "Check:" << endl;
  cout << "\ta*x1*x1 + b*x1 + c = " <<  a*x1*x1 + b*x1 + c << endl;
  cout << "\ta*x2*x2 + b*x2 + c = " <<  a*x2*x2 + b*x2 + c << endl;
}
```

```
Enter the coefficients of a quadratic equation:
        a: 2
        b: 8.001
        c: 8.002
The equation is: 2*x*x + 8.001*x + 8.002 = 0
The solutions are:
        x1 = -2
        x2 = -2.0005
Check:
        a*x1*x1 + b*x1 + c = 0
        a*x2*x2 + b*x2 + c = 0
```

2.3 The following program changes the value of `x` from `inf` to `-inf` and *vice versa*. But no arithmetic operation will change the value of a variable once it becomes `nan`.

```
int main()
{ // changes the value of x after it becomes inf:
  float x=1e30;
  cout << "x= " << x << endl;
  x *= x;
  cout << "x= " << x << endl;
  x *= -1.0;
  cout << "x= " << x << endl;
  x *= -1.0;
  cout << "x= " << x << endl;
}
```

```
x= 1e+30
x= inf
x= -inf
x= inf
```

2.4 We use two variables of type **`float`**

```
int main()
{ // converts inches to centimeters:
  float inches, cm;
  cout << "Enter length in inches: ";
  cin >> inches;
  cm = 2.54*inches;
  cout << inches << " inches = " << cm << " centimeters.\n";
}
```

```
Enter length in inches: 16.9
16.9 inches = 42.926 centimeters.
```

Chapter 3

Selection

The programs in the first two chapters all have *sequential execution*: each statement in the program executes once, and they are executed in the same order that they are listed. This chapter shows how to use selection statements for more flexible programs. It also describes the various integral types that are available in C++.

3.1 THE `if` STATEMENT

The **`if`** statement allows conditional execution. Its syntax is

```
if (condition) statement;
```

where *`condition`* is an integral expression and *`statement`* is any executable statement. The statement will be executed only if the value of the integral expression is nonzero. Notice the required parentheses around the condition.

EXAMPLE 3.1 Testing for Divisibility

This program tests if one positive integer is not divisible by another:

```
int main()
{ int n, d;
  cout << "Enter two positive integers: ";
  cin >> n >> d;
  if (n%d) cout << n << " is not divisible by " << d << endl;
}
```

On the first run, we enter 66 and 7:

```
Enter two positive integers: 66 7
66 is not divisible by 7
```

The value 66%7 is computed to be 3. Since that integral value is not zero, the expression is interpreted as a true condition and consequently the divisibility message is printed.

On the second run, we enter 56 and 7:

```
Enter two positive integers: 56 7
```

The value 56%7 is computed to be 0, which is interpreted to mean “false,” so the divisibility message is not printed.

In C++, whenever an integral expression is used as a condition, the value 0 means “false” and all other values mean “true.”

The program in Example 3.1 is inadequate because it provides no affirmative information when `n` is divisible by `d`. That fault can be remedied with an **`if..else`** statement.

3.2 THE `if..else` STATEMENT

The **`if..else`** statement causes one of two alternative statements to execute depending upon whether the condition is true. Its syntax is

```
if (condition) statement1;
else statement2;
```

where *condition* is an integral expression and *statement1* and *statement2* are executable statements. If the value of the condition is nonzero then *statement1* will execute; otherwise *statement2* will execute.

EXAMPLE 3.2 Testing for Divisibility Again

This program is the same as the program in Example 3.1 except that the `if` statement has been replaced by an `if..else` statement:

```
int main()
{ int n, d;
  cout << "Enter two positive integers: ";
  cin >> n >> d;
  if (n%d) cout << n << " is not divisible by " << d << endl;
  else cout << n << " is divisible by " << d << endl;
}
```

Now when we enter 56 and 7, we get an affirmative response:

```
Enter two positive integers: 56 7
56 is divisible by 7
```

Since 56%7 is zero, the expression is interpreted as being a false condition and consequently the statement after the **else** is executed.

Note that the **if..else** is only one statement, even though it requires two semicolons.

3.3 KEYWORDS

A *keyword* in a programming language is a word that is already defined and is reserved for a unique purpose in programs written in that language. Standard C++ now has 74 keywords:

and	**and_eq**	**asm**	**auto**	**bitand**
bitor	**bool**	**break**	**case**	**catch**
char	**class**	**compl**	**const**	**const_cast**
continue	**default**	**delete**	**do**	**double**
dynamic_cast	**else**	**enum**	**explicit**	**export**
extern	**dfalse**	**float**	**for**	**friend**
goto	**if**	**inline**	**int**	**long**
mutable	**namespace**	**new**	**not**	**not_eq**
operator	**or**	**or_eq**	**private**	**protected**
public	**register**	**reinterpret_cast**	**return**	**short**
signed	**sizeof**	**static**	**static_cast**	**struct**
switch	**template**	**this**	**throw**	**true**
try	**typedef**	**typeid**	**typename**	**using**
union	**unsigned**	**virtual**	**void**	**volatile**
wchar_t	**while**	**xor**	**xor_eq**	

Keywords like **if** and **else** are found in nearly every programming language. Other keywords such as **dynamic_cast** are unique to C++. The 74 keywords of C++ include all 32 of the keywords of the C language.

There are two kinds of keywords: reserved words and standard identifiers. A *reserved word* is a keyword that serves as a structure marker, used to define the syntax of the language. The keywords **if** and **else** are reserved words. A *standard identifier* is a keyword that names a specific element of the language. The keywords **bool** and **int** are standard identifiers because they are names of standard types in C++.

See Appendix B for more information on the C++ keywords.

3.4 COMPARISON OPERATORS

The six *comparison operators* are

```
x < y     // x is less than y
x > y     // x is greater than y
x <= y    // x is less than or equal to y
x >= y    // x is greater than or equal to y
x == y    // x is equal to y
x != y    // x is not equal to y
```

These can be used to compare the values of expressions of any ordinal type. The resulting integral expression is interpreted as a condition that is either false or true according to whether the value of the expression is zero. For example, the expression `7*8 < 6*9` evaluates to zero, which means that the condition is false.

EXAMPLE 3.3 The Minimum of Two Integers

This program prints the minimum of the two integers entered:

```
int main()
{ int m, n;
  cout << "Enter two integers: ";
  cin >> m >> n;
  if (m < n) cout << m << " is the minimum." << endl;
  else cout << n << " is the minimum." << endl;
}
```

```
Enter two integers: 77 55
55 is the minimum.
```

Note that in C++ the single equal sign "=" is the *assignment operator*, and the double equal sign "==" is the *equality operator*:

```
x = 33;   // assigns the value 33 to x
x == 33;  // evaluates to 0 (for false) unless 33 is the value of x
```

This distinction is critically important.

EXAMPLE 3.4 A Common Programming Error

This program is erroneous:

```
int main()
{ int n;
  cout << "Enter an integer: ";
```

```
  cin >> n;
  if (n = 22) cout << n << " = 22" << endl;  // LOGICAL ERROR!
  else cout << n << " != 22" << endl;
}
Enter an integer: 77
22 = 22
```

The expression `n = 22` assigns the value 22 to `n`, changing it from its previous value of 77. But the expression `n = 22` itself is an integral expression that evaluates to 22 after it executes. Thus the condition `(n = 22)` is interpreted as being true, because only 0 yields false, so the statement before the `else` executes. The line should have been written as

```
if (n == 22) cout << n << " = 22" << endl;  // CORRECT
```

The error illustrated in Example 3.4 is called a *logical error*. This is the worst kind of error. Compile-time errors (*e.g.*, omitting a semicolon) are caught by the compiler. Run-time errors (*e.g.*, dividing by zero) are caught by the operating system. But no such help exists for catching logical errors.

EXAMPLE 3.5 The Minimum of Three Integers

This program is similar to the one in Example 3.3 except that it applies to three integers:

```
int main()
{ int n1, n2, n3;
  cout << "Enter three integers: ";
  cin >> n1 >> n2 >> n3;
  int min=n1;               // now min <= n1
  if (n2 < min) min = n2;  // now min <= n1 and min <= n2
  if (n3 < min) min = n3;  // now min <= n1, min <= n2, and min <= n3
  cout << "Their minimum is " << min << endl;
}
Enter two integers: 77 33 55
Their minimum is 33
```

The three comments track the progress of the program: `min` is initialized to equal `n1`, so it is the minimum of the set {`n1`}. After the first `if` statement executes, `min` is equal to either `n1` or `n2`, whichever is smaller, so it is the minimum of the set {`n1`, `n2`}. The last `if` statement changes the value of `min` to `n3` only if `n3` is less than the current value of `min` which is the minimum of the set {`n1`, `n2`}. So in either case, `min` becomes the minimum of the set {`n1`, `n2`, `n3`}.

3.5 STATEMENT BLOCKS

A *statement block* is a sequence of statements enclosed by braces **{ }**, like this:

```
{ int temp=x; x = y; y = temp; }
```

In C++ programs, a statement block can be used anywhere that a single statement can be used.

EXAMPLE 3.6 A Statement Block within an `if` Statement

This program inputs two integers and then outputs them in increasing order:

```
int main()
{ int x, y;
  cout << "Enter two integers: ";
  cin >> x >> y;
```

```
  if (x > y) { int temp=x; x = y; y = temp; }  // swap x and y
  cout << x << " <= " << y << endl;
}
```

```
Enter two integers: 66 44
44 <= 66
```

The three statements within the statement block sort the values of x and y into increasing order by swapping them if they are out of order. Such an interchange requires three separate steps along with the temporary storage location named `temp` here. The program either should execute all three statements or it should execute none of them. That alternative is accomplished by combining the three statements into the statement block.

Note that the variable `temp` is declared inside the block. That makes it *local* to the block; *i.e.*, it only exists during the execution of the block. If the condition is false (*i.e.*, `x ≤ y`), then `temp` will never exist. This illustrates the recommended practice of localizing objects so that they are created only when needed.

Note that a C++ program itself is a statement block preceded by `int main()`.

Recall (Section 1.5 on page 5) that the *scope* of a variable is that part of a program where the variable can be used. It extends from the point where the variable is declared to the end of the block which that declaration controls. So a block can be used to limit the scope of a variable, thereby allowing the same name to be used for different variables in different parts of a program.

EXAMPLE 3.7 Using Blocks to Limit Scope

This program uses the same name `n` for three different variables:

```
int main()
{ int n=44;
  cout << "n = " << n << endl;
  { int n;                          // scope extends over 4 lines
    cout << "Enter an integer: ";
    cin >> n;
    cout << "n = " << n << endl;
  }
  { cout << "n = " << n << endl;    // the n that was declared first
  }
  { int n;                          // scope extends over 2 lines
    cout << "n = " << n << endl;
  }
  cout << "n = " << n << endl;      // the n that was declared first
}
```

```
n = 44
Enter an integer: 77
n = 77
n = 44
n = 4251897
n = 44
```

This program has three internal blocks. The first block declares a new `n` which exists only within that block and overrides the previous variable `n`. So the original `n` retains its value of 44 when this `n` is given the input value 77. The second block does not redeclare `n`, so the scope of the original `n` includes this block. Thus the third output is the original value 44. The third block is like the first block: it declares a new `n` which overrides the original `n`. But this third block does not initialize its local `n`, so the fourth output is a garbage value (4251897). Finally, since the scope of each redeclared `n` extends only to the block where it is declared, the last line of the program is in the scope of the original `n`, so it prints 44.

3.6 COMPOUND CONDITIONS

Conditions such as `n % d` and `x >= y` can be combined to form compound conditions. This is done using the *logical operators* **`&&`** (and), `||` (or), and **`!`** (not). They are defined by

`p && q`	evaluates to true if and only if both `p` and `q` evaluate to true
`p || q`	evaluates to false if and only if both `p` and `q` evaluate to false
`!p`	evaluates to true if and only if `p` evaluates to false

For example, `(n % d || x >= y)` will be false if and only if `n % d` is zero and `x` is less than `y`.

The definitions of the three logical operators are usually given by the *truth tables* below.

p	q	p && q
T	T	T
T	F	F
F	T	F
F	F	F

p	q	p || q
T	T	T
T	F	T
F	T	T
F	F	F

p	!p
T	F
F	T

These show, for example, that if `p` is true and q is false, then the expression `p && q` will be false and the expression `p || q` will be true.

The next example solves the same problem that Example 3.5 on page 39 solved, except that it uses compound conditions.

EXAMPLE 3.8 Using Compound Conditions

This program has the same effect as the one in Example 3.5 on page 39. This version uses compound conditions to find the minimum of three integers:

```
int main()
{ int n1, n2, n3;
  cout << "Enter three integers: ";
  cin >> n1 >> n2 >> n3;
  if (n1 <= n2 && n1 <= n3) cout << "Their minimum is " << n1 <<endl;
  if (n2 <= n1 && n2 <= n3) cout << "Their minimum is " << n2 <<endl;
  if (n3 <= n1 && n3 <= n2) cout << "Their minimum is " << n3 <<endl;
}
```

```
Enter two integers: 77 33 55
Their minimum is 33
```

Note that Example 3.8 is no improvement over Example 3.5. Its purpose was simply to illustrate the use of compound conditions.

Here is another example using a compound condition:

EXAMPLE 3.9 User-Friendly Input

This program allows the user to input either a “Y” or a “y” for “yes”:

```
int main()
{ char ans;
  cout << "Are you enrolled (y/n): ";
  cin >> ans;
  if (ans == 'Y' || ans == 'y') cout << "You are enrolled.\n";
  else cout << "You are not enrolled.\n";
}
```

```
Are you enrolled (y|n): N
You are not enrolled.
```

It prompts the user for an answer, suggesting a response of either `y` or `n`. But then it accepts any character and concludes that the user meant "no" unless either a `Y` or a `y` is input.

3.7 SHORT-CIRCUITING

Compound conditions that use **`&&`** and `||` will not even evaluate the second operand of the condition unless necessary. This is called *short-circuiting*. As the truth tables show, the condition `p && q` will be false if `p` is false. In that case there is no need to evaluate `q`. Similarly if `p` is true then there is no need to evaluate `q` to determine that `p || q` is true. In both cases the value of the condition is known as soon as the first operand is evaluated.

EXAMPLE 3.10 Short-Circuiting

This program tests integer divisibility:

```
int main()
{ int n, d;
  cout << "Enter two positive integers: ";
  cin >> n >> d;
  if (d != 0 && n%d == 0) cout << d << " divides " << n << endl;
  else cout << d << " does not divide " << n << endl;
}
```

In this run, `d` is positive and `n%d` is zero, so the compound condition is true:

```
Enter two positive integers: 300 6
6 divides 300
```

In this run, `d` is positive but `n%d` is not zero, so the compound condition is false:

```
Enter two positive integers: 300 7
7 does not divide 300
```

In this run,`d` is zero, so the compound condition is immediately determined to be false without evaluating the second expression "`n%d == 0`":

```
Enter two positive integers: 300 0
0 does not divide 300
```

This short-circuiting prevents the program from crashing because when `d` is zero the expression `n%d` cannot be evaluated.

3.8 BOOLEAN EXPRESSIONS

A *boolean expression* is a condition that is either true or false. In the previous example the expressions `d > 0`, `n%d == 0`, and `(d > 0 && n%d == 0)` are boolean expressions. As we have seen, boolean expressions evaluate to integer values. The value 0 means "false" and every nonzero value means "true."

Since all nonzero integer values are interpreted as meaning "true," boolean expressions are often disguised. For example, the statement

```
if (n) cout << "n is not zero";
```

will print `n is not zero` precisely when `n` is not zero because that is when the boolean expression `(n)` is interpreted as "true". Here is a more realistic example:

```
if (n%d) cout << "n is not a multiple of d";
```

The output statement will execute precisely when `n%d` is not zero, and that happens precisely when `d` does not divide `n` evenly, because n%d is the remainder from the integer division.

The fact that boolean expressions have integer values can lead to some surprising anomalies in C++.

EXAMPLE 3.11 Another Logical Error

This program is erroneous:

```
int main()
{ int n1, n2, n3;
  cout << "Enter three integers: ";
  cin >> n1 >> n2 >> n3;
  if (n1 >= n2 >= n3) cout << "max = x";          // LOGICAL ERROR!
}
```

```
Enter an integer: 0 0 1
max = 0
```

The source of this error is the fact that boolean expressions have numeric values. Since the expression `(n1 >= n2 >= n3)` is evaluated from left to right, the first part `n1 >= n2` evaluates to "true" since 0 >= 0. But "true" is stored as the numeric value 1. That value is then compared to the value of `n3` which is also 1, so the complete expression evaluates to "true" even though it is really false! (0 is not the maximum of 0, 0, and 1.)

The problem here is that the erroneous line is syntactically correct, so the compiler cannot catch the error. Nor can the operating system. This is another logical error, comparable to that in the program in Example 3.4 on page 38.

The moral from Example 3.11 is to remember that boolean expressions have numeric values, so compound conditions can be tricky.

3.9 NESTED SELECTION STATEMENTS

Like compound statements, selection statements can be used wherever any other statement can be used. So a selection statement can be used within another selection statement. This is called *nesting* statements.

EXAMPLE 3.12 Nesting Selection Statements

This program has the same effect as the one in Example 3.10 on page 42:

```
int main()
{ int n, d;
  cout << "Enter two positive integers: ";
  cin >> n >> d;
  if (d != 0)
    if (n%d == 0) cout << d << " divides " << n << endl;
    else cout << d << " does not divide " << n << endl;
  else cout << d << " does not divide " << n << endl;
}
```

The second `if..else` statement is nested within the `if` clause of the first `if..else` statement. So the second `if..else` statement will execute only when `d` is not zero.

Note that the `" does not divide "` statement has to be used twice here. The first one, nested within the `if` clause of the first `if..else` statement, executes when `d` is not zero and `n%d` is zero. The second one executes when `d` is zero.

When **`if..else`** statements are nested, the compiler uses the following rule to parse the compound statement:

Match each **`else`** *with the last unmatched* **`if`**.

Using this rule, the compiler can easily decipher code as inscrutable as this:

```
if (a > 0) if (b > 0) ++a; else if (c > 0)        // BAD CODING STYLE
if (a < 4) ++b; else if (b < 4) ++c; else --a;    // BAD CODING STYLE
else if (c < 4) --b; else --c; else a = 0;        // BAD CODING STYLE
```

To make this readable for humans it should be written either like this:

```
if (a > 0)
  if (b > 0) ++a;
  else
    if (c > 0)
      if (a < 4) ++b;
      else
        if (b < 4) ++c;
        else --a;
    else
      if (c < 4) --b;
      else --c;
else a = 0;
```

or like this:

```
if (a > 0)
  if (b > 0) ++a;
  else if (c > 0)
    if (a < 4) ++b;
    else if (b < 4) ++c;
    else --a;
  else if (c < 4) --b;
  else --c;
else a = 0;
```

This second rendering aligns the `else if` pairs when they form parallel alternatives. (See Section 3.10 on page 46.)

EXAMPLE 3.13 Using Nested Selection Statements

This program has the same effect as those in Example 3.5 on page 39 and Example 3.8 on page 41. This version uses nested `if..else` statements to find the minimum of three integers:

```
int main()
{ int n1, n2, n3;
  cout << "Enter three integers: ";
  cin >> n1 >> n2 >> n3;
  if (n1 < n2)
    if (n1 < n3) cout << "Their minimum is " << n1 << endl;
    else cout << "Their minimum is " << n3 << endl;
  else  // n1 >= n2
    if (n2 < n3) cout << "Their minimum is " << n2 << endl;
```

```
    else cout << "Their minimum is " << n3 << endl;
}
Enter three integers: 77 33 55
Their minimum is 33
```

In this run, the first condition `(n1 < n2)` is false, and the third condition `(n2 < n3)` is true, so it reports that `n2` is the minimum.

This program is more efficient than the one in Example 3.8 on page 41 because on any run it will evaluate only two simple conditions instead of three compound conditions. Nevertheless, it should be considered inferior because its logic is more complicated. In the trade-off between efficiency and simplicity, it is usually best to choose simplicity.

EXAMPLE 3.14 A Guessing Game

This program finds a number that the user selects from 1 to 8:

```
int main()
{ cout << "Pick a number from 1 to 8." << endl;
  char answer;
  cout << "Is it less than 5? (y|n): ";  cin >> answer;
  if (answer == 'y')  // 1 <= n <= 4
  { cout << "Is it less than 3? (y|n): ";  cin >> answer;
    if (answer == 'y')  // 1 <= n <= 2
    { cout << "Is it less than 2? (y|n): ";  cin >> answer;
      if (answer == 'y') cout << "Your number is 1." << endl;
      else cout << "Your number is 2." << endl;
    }
    else  // 3 <= n <= 4
    { cout << "Is it less than 4? (y|n): ";  cin >> answer;
      if (answer == 'y') cout << "Your number is 3." << endl;
      else cout << "Your number is 4." << endl;
    }
  }
  else  // 5 <= n <= 8
  { cout << "Is it less than 7? (y|n): ";  cin >> answer;
    if (answer == 'y')  // 5 <= n <= 6
    { cout << "Is it less than 6? (y|n): ";  cin >> answer;
      if (answer == 'y') cout << "Your number is 5." << endl;
      else cout << "Your number is 6." << endl;
    }
    else  // 7 <= n <= 8
    { cout << "Is it less than 8? (y|n): ";  cin >> answer;
      if (answer == 'y') cout << "Your number is 7." << endl;
      else cout << "Your number is 8." << endl;
    }
  }
}
```

By repeatedly subdividing the problem, it can discover any one of the 8 numbers by asking only three questions. In this run, the user's number is 6.

```
Pick a number from 1 to 8.
Is it less than 5? (y|n): n
Is it less than 7? (y|n): y
Is it less than 6? (y|n): n
Your number is 6.
```

The algorithm used in Example 3.14 is called the *binary search*. It can be implemented more simply. (See Example 6.14 on page 135.)

3.10 THE `else if` CONSTRUCT

Nested **`if..else`** statements are often used to test a sequence of parallel alternatives, where only the **`else`** clauses contain further nesting. In that case, the resulting compound statement is usually formatted by lining up the **`else if`** phrases to emphasize the parallel nature of the logic.

EXAMPLE 3.15 Using the `else if` Construct for Parallel Alternatives

This program requests the user's language and then prints a greeting in that language:

```
int main()
{ char language;
  cout << "Engl., Fren., Ger., Ital., or Rus.? (e|f|g|i|r): ";
  cin >> language;
  if (language == 'e') cout << "Welcome to ProjectEuclid.";
  else if (language == 'f') cout << "Bon jour, ProjectEuclid.";
  else if (language == 'g') cout << "Guten tag, ProjectEuclid.";
  else if (language == 'i') cout << "Bon giorno, ProjectEuclid.";
  else if (language == 'r') cout << "Dobre utre, ProjectEuclid.";
  else cout << "Sorry; we don't speak your language.";
}
```

```
Engl., Fren., Ger., Ital., or Rus.? (e|f|g|i|r): i
Bon giorno, ProjectEuclid.
```

This program uses nested `if..else` statements to select from the five given alternatives.

As ordinary nested `if..else` statements, the code could also be formatted as

```
if (language == 'e') cout << "Welcome to ProjectEuclid.";
else
  if (language == 'f') cout << "Bon jour, ProjectEuclid.";
  else
    if (language == 'g') cout << "Guten tag, ProjectEuclid.";
    else
      if (language == 'i') cout << "Bon giorno, ProjectEuclid.";
       else
         if (language == 'r') cout << "Dobre utre, ProjectEuclid.";
         else cout << "Sorry; we don't speak your language.";
```

But the given format is preferred because it displays the parallel nature of the logic more clearly. It also requires less indenting.

EXAMPLE 3.16 Using the `else if` Construct to Select a Range of Scores

This program converts a test score into its equivalent letter grade:

```
int main()
{ int score;
  cout << "Enter your test score: ";  cin >> score;
  if (score > 100) cout << "Error: that score is out of range.";
  else if (score >= 90) cout << "Your grade is an A." << endl;
  else if (score >= 80) cout << "Your grade is a B."  << endl;
  else if (score >= 70) cout << "Your grade is a C."  << endl;
  else if (score >= 60) cout << "Your grade is a D."  << endl;
  else if (score >=  0) cout << "Your grade is an F." << endl;
  else cout << "Error: that score is out of range.";
}
```

```
Enter your test score: 83
Your grade is a B.
```

The variable `score` is tested through a cascade of selection statements, continuing until either one of the conditions is found to be true, or the last `else` is reached.

3.11 THE `switch` STATEMENT

The `switch` statement can be used instead of the `else if` construct to implement a sequence of parallel alternatives. Its syntax is

```
switch (expression)
{ case constant1: statementList1;
  case constant2: statementList2;
  case constant3: statementList3;
    :
    :
  case constantN: statementListN;
  default: statementList0;
}
```

This evaluates the *expression* and then looks for its value among the **case** constants. If the value is found among the constants listed, then the statements in the corresponding *statementList* are executed. Otherwise if there is a **default** (which is optional), then the program branches to its *statementList*. The *expression* must evaluate to an integral type (see Section 2.1 on page 16) and the *constant*s must be integral constants.

EXAMPLE 3.17 Using a `switch` Statement to Select a Range of Scores

This program has the same effect as the one in Example 3.16:

```
int main()
{ int score;
  cout << "Enter your test score: ";  cin >> score;
  switch (score/10)
  { case 10:
    case  9: cout << "Your grade is an A." << endl;  break;
    case  8: cout << "Your grade is a B."  << endl;  break;
    case  7: cout << "Your grade is a C."  << endl;  break;
```

```
    case  6: cout << "Your grade is a D."  << endl;  break;
    case  5:
    case  4:
    case  3:
    case  2:
    case  1:
    case  0: cout << "Your grade is an F." << endl;  break;
    default: cout << "Error: score is out of range.\n";
  }
  cout << "Goodbye." << endl;
}
```

```
Enter your test score: 83
Your grade is a B.
Goodbye.
```

First the program divides the `score` by 10 to reduce the range of values to 0–10. So in the test run, the score 83 reduces to the value 8, the program execution branches to `case 8`, and prints the output shown. Then the break statement causes the program execution to branch to the first statement after the `switch` block. That statement prints "`Goodbye.`".

Note that scores in the ranges 101 to 109 and -9 to -1 produce incorrect results. (See Problem 3.14.)

It is normal to put a `break` statement at the end of each case clause in a `switch` statement. Without it, the program execution will not branch directly out of the `switch` block after it finishes executing its `case` statement sequence. Instead, it will continue within the `switch` block, executing the statements in the next `case` sequence. This (usually) unintended consequence is called a *fall through*.

EXAMPLE 3.18 An Erroneous Fall-through in a `switch` Statement

This program was intended to have the same effect as the one in Example 3.17. But with no break statements, the program execution falls through all the case statements it encounters:

```
int main()
{ int score;
  cout << "Enter your test score: ";  cin >> score;
  switch (score/10)
  { case 10:
    case  9: cout << "Your grade is an A." << endl;  // LOGICAL ERROR
    case  8: cout << "Your grade is a B."  << endl;  // LOGICAL ERROR
    case  7: cout << "Your grade is a C."  << endl;  // LOGICAL ERROR
    case  6: cout << "Your grade is a D."  << endl;  // LOGICAL ERROR
    case  5:
    case  4:
    case  3:
    case  2:
    case  1:
    case  0: cout << "Your grade is an F." << endl;  // LOGICAL ERROR
    default: cout << "Error: score is out of range.\n";
  }
  cout << "Goodbye." << endl;
}
```

```
Enter your test score: 83
Your grade is a B.
Your grade is a C.
```

```
Your grade is a D.
Your grade is an F.
Error: score is out of range.
Goodbye.
```

After branching to `case 8`, and printing "`Your grade is a B.`", the program execution goes right on to `case 7` and prints "`Your grade is a C.`" Since the `break` statements have been removed, it keeps falling through, all the way down to the `default` clause, executing each of the `cout` statements along the way.

3.12 THE CONDITIONAL EXPRESSION OPERATOR

C++ provides a special operator that often can be used in place of the **`if...else`** statement. It is called the *conditional expression operator*. It uses the **`?`** and the **`:`** symbols in this syntax:

`condition` **`?`** *`expression1`* **`:`** *`expression2`*

It is a *ternary operator*; *i.e.*, it combines three operands to produce a value. That resulting value is either the value of *`expression1`* or the value of *`expression2`*, depending upon the boolean value of the *`condition`*. For example, the assignment

```
min = ( x<y ? x : y );
```

would assign the minimum of `x` and `y` to `min`, because if the condition **`x<y`** is true, the expression **`( x<y ? x : y )`** evaluates to `x`; otherwise it evaluates to `y`.

Conditional expression statements should be used sparingly: only when the condition and both expressions are very simple.

EXAMPLE 3.19 Finding the Minimum Again

This program has the same effect as the program in Example 3.3 on page 38:

```
int main()
{ int m, n;
  cout << "Enter two integers: ";
  cin >> m >> n;
  cout << ( m<n ? m : n ) << " is the minimum." << endl;
}
```

The conditional expression **`( m<n ? m : n )`** evaluates to `m` if `m<n`, and to `n` otherwise.

Review Questions

3.1 Write a single C++ statement that prints `"Too many"` if the variable `count` exceeds 100.

3.2 What is wrong with the following code:

a. `cin << count;`

b.
```
if x < y min = x
  else min = y;
```

3.3 What is wrong with this code:

```
cout << "Enter n: ";
cin >> n;
if (n < 0)
  cout << "That is negative. Try again." << endl;
  cin >> n;
```

```
else
  cout << "o.k. n = " << n << endl;
```

3.4 What is the difference between a reserved word and a standard identifier?

3.5 State whether each of the following is true or false. If false, tell why.

a. `!(p || q)` is the same as `!p || !q`

b. `!!!p` is the same as `!p`

c. `p && q || r` is the same as `p && (q || r)`

3.6 Construct a truth table for each of the following boolean expressions, showing its truth value (0 or 1) for all 4 combinations of truth values of its operands p and q.

a. `!p || q`

b. `p&&q || !p&&!q`

c. `(p||q) && !(p&&q)`

3.7 Use truth tables to determine whether the two boolean expressions in each of the following are equivalent.

a. `!(p && q)` and `!p && !q`

b. `!!p` and `p`

c. `!p || q` and `p || !q`

d. `p && (q && r)` and `(p && q) && r`

e. `p || (q && r)` and `(p || q) && r`

3.8 What is short-circuiting and how is it helpful?

3.9 What is wrong with this code:

```
if (x = 0) cout << x << " = 0\n";
else cout << x << " != 0\n";
```

3.10 What is wrong with this code:

```
if (x < y < z) cout << x << " < " << y << " < " << z << endl;
```

3.11 Construct a logical expression to represent each of the following conditions:

a. `score` is greater than or equal to 80 but less than 90;

b. `answer` is either `'N'` or `'n'`;

c. `n` is even but not 8;

d. `ch` is a capital letter.

3.12 Construct a logical expression to represent each of the following conditions:

a. `n` is between 0 and 7 but not equal to 3;

b. `n` is between 0 and 7 but not even;

c. `n` is divisible by 3 but not by 30;

d. `ch` is a lowercase or uppercase letter.

3.13 What is wrong with this code:

```
if (x == 0)
  if (y == 0) cout << "x and y are both zero." << endl;
else cout << "x is not zero." << endl;
```

3.14 What is the difference between the following two statements:

```
if (n > 2) { if (n < 6) cout << "OK"; } else cout << "NG";
if (n > 2) { if (n < 6) cout << "OK"; else cout << "NG"; }
```

3.15 What is a “fall-through?

3.16 How is the following expression evaluated?

```
(x < y ? -1 : (x == y ? 0 : 1) );
```

3.17 Write a single C++ statement that uses the conditional expression operator to assign the absolute value of `x` to `absx`.

3.18 Write a single C++ statement that prints "too many" if the variable `count` exceeds 100, using
a. an `if` statement;
b. the conditional expression operator.

Problems

3.1 Modify the program in Example 3.1 on page 36 so that it prints a response only if `n` is divisible by `d`.

3.2 Modify the program in Example 3.5 on page 39 so that it prints the minimum of four input integers.

3.3 Modify the program in Example 3.5 on page 39 so that it prints the median of three input integers.

3.4 Modify the program in Example 3.6 on page 39 so that it has the same effect without using a statement block.

3.5 Predict the output from the program in Example 3.7 on page 40 after removing the declaration on the fifth line of the program. Then run that program to check your prediction.

3.6 Write and run a program that reads the user's age and then prints "You are a child." if the age < 18, "You are an adult." if 18 ≤ age < 65, and "You are a senior citizen." if age ≥ 65.

3.7 Write and run a program that reads two integers and then uses the conditional expression operator to print either "multiple" or "not" according to whether one of the integers is a multiple of the other.

3.8 Write and run a program that simulates a simple calculator. It reads two integers and a character. If the character is a `+`, the sum is printed; if it is a `-`, the difference is printed; if it is a `*`, the product is printed; if it is a `/`, the quotient is printed; and if it is a `%`, the remainder is printed. Use a `switch` statement.

3.9 Write and run a program that plays the game of "Rock, paper, scissors." In this game, two players simultaneously say (or display a hand symbol representing) either "rock," "paper," or "scissors." The winner is the one whose choice dominates the other. The rules are: paper dominates (wraps) rock, rock dominates (breaks) scissors, and scissors dominate (cut) paper. Use enumerated types for the choices and for the results.

3.10 Modify the solution to Problem 3.9 by using a `switch` statement.

3.11 Modify the solution to Problem 3.10 by using conditional expressions where appropriate.

3.12 Write and test a program that solves quadratic equations. A *quadratic equation* is an equation of the form $ax^2 + bx + c = 0$, where a, b, and c are given coefficients and x is the unknown. The coefficients are real number inputs, so they should be declared of type `float` or `double`. Since quadratic equations typically have two solutions, use `x1` and `x2` for the solutions to be output. These should be declared of type `double` to avoid inaccuracies from round-off error. (See Example 2.15 on page 28.)

3.13 Write and run a program that reads a six-digit integer and prints the sum of its six digits. Use the *quotient operator* `/` and the *remainder operator* `%` to extract the digits from the integer. For example, if `n` is the integer 876,543, then `n/1000%10` is its thousands digit 6.

3.14 Correct Example 3.17 on page 47 so that it produces the correct response for all inputs.

Answers to Review Questions

3.1
```
if (count > 100) cout << "Too many";
```

3.2 ***a.*** Either `cout` should be used in place of `cin`, or the extraction operator `>>` should be used in place of the insertion operator `<<`.

b. Parentheses are required around the condition `x < y`, and a semicolon is required at the end of the `if` clause before the `else`.

3.3 There is more than one statement between the `if` clause and the `else` clause. They need to be made into a compound statement by enclosing them in braces `{ }`.

3.4 A *reserved word* is a keyword in a programming language that serves to mark the structure of a statement. For example, the keywords `if` and `else` are reserved words. A *standard identifier* is a keyword that defines a type. Among the 63 keywords in C++, `if`, `else`, and `while` are some of the reserved words, and `char`, `int`, and `float` are some of the standard identifiers.

3.5 ***a.*** `!(p || q)` is not the same as `!p || !q`; for example, if `p` is true and `q` is false, the first expression will be false but the second expression will be true. The correct equivalent to the expression `!(p || q)` is the expression `!p && !q`.

b. `!!!p` is the same as `!p`.

c. `p && q || r` is not the same as `p && (q || r)`; for example, if `p` is false and `r` is true, the first expression will be true, but the second expression will be false: `p && q || r` is the same as `(p && q) || r`.

3.6 Truth tables for boolean expressions:

p	q	!p || q
T	T	T
T	F	F
F	T	T
F	F	T

p	q	p&&q || !p&&!q
T	T	T
T	F	F
F	T	F
F	F	T

p	q	(p||q) && !(p&&q)
T	T	F
T	F	T
F	T	T
F	F	F

3.7 ***a.*** These two boolean expressions are not equivalent:

p	q	!(p&&q)
T	T	F
T	F	T
F	T	T
F	F	T

p	q	!p && !q
T	T	T
T	F	T
F	T	T
F	F	F

b. These two boolean expressions are equivalent:

p	!pp
T	T
F	F

p	p
T	T
F	F

c. These two boolean expressions are not equivalent:

p	q	!p || q
T	T	T
T	F	F
F	T	T
F	F	T

p	q	p || !q
T	T	T
T	F	T
F	T	F
F	F	T

d. These two boolean expressions are equivalent:

p	q	r	p && (q&&r)
T	T	T	T
T	T	F	F
T	F	T	F
T	F	F	F
F	T	T	F
F	T	F	F
F	F	T	F
F	F	F	F

p	q	r	(p&&q) && r
T	T	T	T
T	T	F	F
T	F	T	F
T	F	F	F
F	T	T	F
F	T	F	F
F	F	T	F
F	F	F	F

e. These two boolean expressions are not equivalent:

p	q	r	p || (q&&r)
T	T	T	T
T	T	F	T
T	F	T	T
T	F	F	T
F	T	T	T
F	T	F	F
F	F	T	F
F	F	F	F

p	q	r	(p||q) && r
T	T	T	T
T	T	F	F
T	F	T	T
T	F	F	F
F	T	T	T
F	T	F	F
F	F	T	F
F	F	F	F

3.8 The term "short-circuiting" is used to describe the way C++ evaluates compound logical expressions like `(x > 2 || y > 5)` and `(x > 2 && y > 5)`. If `x` is greater than 2 in the first expression, then `y` will not be evaluated. If `x` is less than or equal to 2 in the second expression, then `y` will not be evaluated. In these cases only the first part of the compound expression is evaluated because that value alone determines the truth value of the compound expression.

3.9 The programmer probably intended to test the condition `(x == 0)`. But by using assignment operator "=" instead of the equality operator "==" the result will be radically different from what was intended. For example, if `x` has the value 22 prior to the `if` statement, then the `if` statement will change the value of `x` to 0. Moreover, the assignment expression `(x = 0)` will be evaluated to 0 which means "false," so the `else` part of the selection statement will execute, reporting that `x` is not zero!

3.10 The programmer probably intended to test the condition `(x < y && y < z)`. The code as written will compile and run, but not as intended. For example, if the prior values of `x`, `y`, and `z` are 44, 66, and 22, respectively, then the algebraic condition "$x < y < z$" is false. But as written, the code will be evaluated from left to right, as `(x < y) < z`. First the condition `x < y` will be evaluated as "true." But this has the numeric value 1, so the expression `(x < y)` is evaluated to 1. Then the combined expression `(x < y) < z` is evaluated as (1) < 66 which is also true. So the output statement will execute, erroneously reporting that 44 < 66 < 22.

3.11 ***a.*** `(score >= 80 && score < 90)`

b. `(answer == 'N' || answer == 'n')`

c. `(n%2 == 0 && n != 8)`

d. `(ch >= 'A' && ch <= 'Z')`

3.12 ***a.*** `(n > 0 && n < 7 && n != 3)`

b. `(n > 0 && n < 7 && n%2 != 0)`

c. `((ch >= 'A' && ch <= 'Z') || (ch >= 'a' && ch <= 'z'))`

3.13 The programmer clearly intended for the second output `"x is not zero."` to be printed if the first condition `(x == 0)` is false, regardless of the second condition `(y == 0)`. That is, the `else` was intended to be matched with the first `if`. But the "`else` matching" rule causes it to be matched with the second condition, which means that the output `"x is not zero."` will be printed only when `x` is zero and `y` is not zero. The "`else` matching" rule can be overridden with braces:

```
if (x == 0)
{ if (y == 0) cout << "x and y are both zero." << endl;
}
else cout << "x is not zero." << endl;
```

Now the `else` will be matched with the first `if`, the way the programmer had intended it to be.

3.14 In the first statement, the `else` is matched with the first `if`. In the second statement, the `else` is matched with the second `if`. If $n \le 2$, the first statement will print `NG` while the second statement will do nothing. If $2 < n < 6$, both statements will print `OK`. If $n \ge 6$, the first statement will do nothing while the second statement will print NG. Note that this code is difficult to read because it does not follow standard indentation conventions. The first statement should be written

```
if (n > 2)
{ if (n < 6) cout << "OK";
}
else cout << "NG";
```

The braces are needed here to override the "`else` matching" rule. This `else` is intended to match the first `if`. The second statement should be written

```
if (n > 2)
  if (n < 6) cout << "OK";
  else cout << "NG";
```

Here the braces are not needed because the `else` is intended to be matched with the second `if`.

3.15 A "fall through" in a `switch` statement is a case that does not include a `break` statement, thereby causing control to continue right on to the next case statement.

3.16 This expression evaluates to –1 if `x < y`, it evaluates to 0 if `x == y`, and it evaluates to 1 if `x > y`.

3.17
```
absx = ( x>0 ? x : -x );
```

3.18 ***a.***
```
if (count > 100) cout << "too many";
```

b.
```
cout << (count > 100 ? "too many" : "" );
```

Solutions to Problems

3.1 This version of Example 3.1 on page 36 prints a response only when `n` is divisible by `d`:

```
int main()
{ int n, d;
  cout << "Enter two positive integers: ";
  cin >> n >> d;
  if (n%d == 0) cout << n << " is divisible by " << d << endl;
}
```

```
Enter two positive integers: 56 7
56 is divisible by 7
```

3.2 This version of Example 3.5 on page 39 prints the minimum of four input integers:

```
int main()
{ int n1, n2, n3, n4;
  cout << "Enter four integers: ";
  cin >> n1 >> n2 >> n3 >> n4;
  int min=n1;               // now min <= n1
  if (n2 < min) min = n2;  // now min <= n1, n2
  if (n3 < min) min = n3;  // now min <= n1, n2, n3
  if (n4 < min) min = n4;  // now min <= n1, n2, n3, n4
  cout << "Their minimum is " << min << endl;
}
```

```
Enter four integers: 44 88 22 66
Their minimum is 22
```

3.3 This program finds the median of three input integers:

```
int main()
{ int n1, n2, n3;
  cout << "Enter three integers: ";
  cin >> n1 >> n2 >> n3;
  cout << "Their median is ";
  if (n1 < n2)
    if (n2 < n3) cout << n2;       // n1 <  n2 <  n3
```

```
    else if (n1 < n3) cout << n3;  // n1 <  n3 <= n2
    else cout << n1;               // n3 <= n1 <  n2
  else if (n1 < n3) cout << n1;    // n2 <= n1 <  n3
  else if (n2 < n3) cout << n2;    // n2 <  n3 <= n1
  else cout << n3;                 // n3 <= n2 <= n1
}
```

```
Enter three integers: 44 88 22
Their median is 44
```

3.4 This program has the same effect as the one in Example 3.6 on page 39:

```
int main()
{ int x, y;
  cout << "Enter two integers: ";
  cin >> x >> y;
  if (x > y) cout << y << " <= " << x << endl;
  else cout << x << " <= " << y << endl;
}
```

```
Enter two integers: 66 44
44 <= 6
```

3.5 Modification of the program in Example 3.7 on page 40:

```
int main()
{ int n=44;
  cout << "n = " << n << endl;
  { cout << "Enter an integer: ";
    cin >> n;
    cout << "n = " << n << endl;
  }
  { cout << "n = " << n << endl;
  }
  { int n;
    cout << "n = " << n << endl;
  }
  cout << "n = " << n << endl;
}
```

```
n = 44
Enter an integer: 77
n = 77
n = 77
n = 4251897
n = 77
```

3.6 Here we used the `else if` construct because the three outcomes depend upon `age` being in one of three disjoint intervals:

```
int main()
{ int age;
  cout << "Enter your age: ";
  cin >> age;
  if (age < 18) cout << "You are a child.\n";
  else if (age < 65) cout << "You are an adult.\n";
  else cout << "you are a senior citizen.\n";
}
```

```
Enter your age: 44
You are an adult.
```

If control reaches the second condition `(age < 65)`, then the first condition must be false so in fact 18 ≤ age < 65. Similarly, if control reaches the second `else`, then both conditions must be false so in fact age ≥ 65.

3.7 An integer `m` is a multiple of an integer `n` if the remainder from the integer division of `m` by `n` is 0. So the compound condition `m % n == 0 || n % m == 0` tests whether either is a multiple of the other:

```
int main()
{ int m, n;
  cin >> m >> n;
  cout << (m % n == 0 || n % m == 0 ? "multiple" : "not") << endl;
}
```

```
30 4
not
```

```
30 5
multiple
```

The value of the conditional expression will be either `"multiple"` or `"not"`, according to whether the compound condition is true. So sending the complete conditional expression to the output stream produces the desired result.

3.8 The character representing the operation should be the control variable for the `switch` statement:

```
int main()
{ int x, y;
  char op;
  cout << "Enter two integers: ";
  cin >> x >> y;
  cout << "Enter an operator: ";
  cin >> op;
  switch (op)
  { case '+': cout << x + y << endl;  break;
    case '-': cout << x - y << endl;  break;
    case '*': cout << x * y << endl;  break;
    case '/': cout << x / y << endl;  break;
    case '%': cout << x % y << endl;  break;
  }
}
```

```
Enter two integers: 30 13
Enter an operator: %
4
```

In each of the five cases, we simply print the value of the corresponding arithmetic operation and then break.

3.9 First define the two `enum` types `Choice` and `Result`. Then declare variables `choice1`, `choice2`, and `result` of these types, and use an integer `n` to get the required input and assign it to them:

```
enum Choice {ROCK, PAPER, SCISSORS};
enum Winner {PLAYER1, PLAYER2, TIE};
int main()
{ int n;
  Choice choice1, choice2;
  Winner winner;
  cout << "Choose rock (0), paper (1), or scissors (2):" << endl;
  cout << "Player #1: ";
  cin >> n;
  choice1 = Choice(n);
```

```
  cout << "Player #2: ";
  cin >> n;
  choice2 = Choice(n);
  if (choice1 == choice2) winner = TIE;
  else if (choice1 == ROCK)
    if (choice2 == PAPER) winner = PLAYER2;
    else winner = PLAYER1;
  else if (choice1 == PAPER)
    if (choice2 == SCISSORS) winner = PLAYER2;
    else winner = PLAYER1;
  else // (choice1 == SCISSORS)
    if (choice2 == ROCK) winner = PLAYER2;
    else winner = PLAYER1;
  if (winner == TIE) cout << "\tYou tied.\n";
  else if (winner == PLAYER1) cout << "\tPlayer #1 wins." <<endl;
  else cout << "\tPlayer #2 wins." << endl;
}
```

```
Choose rock (0), paper (1), or scissors (2):
Player #1: 1
Player #2: 1
    You tied.
```

```
Choose rock (0), paper (1), or scissors (2):
Player #1: 2
Player #2: 1
    Player #1 wins.
```

```
Choose rock (0), paper (1), or scissors (2):
Player #1: 2
Player #2: 0
        Player #2 wins.
```

Through a series of nested `if` statements, we are able to cover all the possibilities.

3.10 Using a `switch` statement:

```
enum Winner {PLAYER1, PLAYER2, TIE};
int main()
{ int choice1, choice2;
  Winner winner;
  cout << "Choose rock (0), paper (1), or scissors (2):" << endl;
  cout << "Player #1: ";
  cin >> choice1;
  cout << "Player #2: ";
  cin >> choice2;
  switch (choice2 - choice1)
  { case  0:
      winner = TIE;
      break;
    case -1:
    case  2:
      winner = PLAYER1;
      break;
    case -2:
    case  1:
      winner = PLAYER2;
  }
```

```
  if (winner == TIE) cout << "\tYou tied.\n";
  else if (winner == PLAYER1) cout << "\tPlayer #1 wins." << endl;
  else cout << "\tPlayer #2 wins." << endl;
}
```

3.11 Using a `switch` statement and conditional expressions:

```
enum Winner {PLAYER1, PLAYER2, TIE};
int main()
{ int choice1, choice2;
  cout << "Choose rock (0), paper (1), or scissors (2):" << endl;
  cout << "Player #1: ";
  cin >> choice1;
  cout << "Player #2: ";
  cin >> choice2;
  int n = (choice1 - choice2 + 3) % 3;
  Winner winner = ( n==0 ? TIE : (n==1?PLAYER1:PLAYER2) );
  if (winner == TIE) cout << "\tYou tied.\n";
  else if (winner == PLAYER1) cout << "\tPlayer #1 wins." << endl;
  else cout << "\tPlayer #2 wins." << endl;
}
```

3.12 The solution(s) to the quadratic equation is given by the *quadratic formula*:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

But this will not apply if a is zero, so that condition must be checked separately. The formula also fails to work (for real numbers) if the expression under the square root is negative. That expression $b^2 + 4ac$ is called the *discriminant* of the quadratic. We define that as the separate variable `d` and check its sign.

```
#include <iostream>
#include <cmath>  // defines the sqrt() function
int main()
{ // solves the equation a*x*x + b*x + c == 0:
  float a, b, c;
  cout << "Enter coefficients of quadratic equation: ";
  cin >> a >> b >> c;
  if (a == 0)
  { cout << "This is not a quadratic equation: a == 0\n";
    return 0;
  }
  cout << "The equation is: " << a << "x^2 + " << b
       << "x + " << c << " = 0\n";
  double d, x1, x2;
  d = b*b - 4*a*c;  // the discriminant
  if (d < 0)
  { cout << "This equation has no real solutions: d < 0\n";
    return 0;
  }
  x1 = (-b + sqrt(d))/(2*a);
  x2 = (-b - sqrt(d))/(2*a);
  cout << "The solutions are: " << x1 << ", " << x2 << endl;
}
```

```
Enter coefficients of quadratic equation: 2 1 -6
The equation is: 2x^2 + 1x + -6 = 0
The solutions are: 1.5, -2
```

```
Enter coefficients of quadratic equation: 1 4 5
The equation is: 1x^2 + 4x + 5 = 0
This equation has no real solutions: d < 0
```

```
Enter coefficients of quadratic equation: 0 4 5
This is not a quadratic equation: a == 0
```

Note how we use the `return` statement inside the selection statements to terminate the program if either `a` is zero or `d` is negative. The alternative would have been to use an `else` clause in each `if` statement.

3.13 This program prints the sum of the digits of the given integer:

```
int main()
{ int n, sum;
  cout << "Enter a six-digit integer: ";
  cin >> n;
  sum = n%10 + n/10%10 + n/100%10 + n/1000%10 + n/10000%10
             + n/100000;
  cout << "The sum of the digits of " << n << " is " << sum <<endl;
}
```

```
Enter a six-digit integer: 876543
The sum of the digits of 876543 is 33
```

3.14 A corrected version of Example 3.17 on page 47:

```
int main()
{ // reports the user's grade for a given test score:
  int score;
  cout << "Enter your test score: ";
  cin >> score;
  if (score > 100 || score < 0)
    cout << "Error: that score is out of range.\n";
  else
    switch (score/10)
    { case 10:
      case  9: cout << "Your grade is an A.\n";  break;
      case  8: cout << "Your grade is a B.\n";   break;
      case  7: cout << "Your grade is a C.\n";   break;
      case  6: cout << "Your grade is a D.\n";   break;
      default: cout << "Your grade is an F.\n";  break;
    }
  cout << "Goodbye." << endl;
}
```

```
Enter your test score: 103
Error: that score is out of range.
Goodbye.
```

```
Enter your test score: 93
Your grade is an A.
Goodbye.
```

```
Enter your test score: -3
Error: that score is out of range.
Goodbye.
```

Chapter 4

Iteration

Iteration is the repetition of a statement or block of statements in a program. C++ has three iteration statements: the `while` statement, the `do..while` statement, and the `for` statement. Iteration statements are also called *loops* because of their cyclic nature.

4.1 THE `while` STATEMENT

The syntax for the `while` statement is

`while` (*`condition`*) *`statement`*;

where *`condition`* is an integral expression and *`statement`* is any executable statement. If the value of the expression is zero (meaning "false") then the *`statement`* is ignored and program execution immediately jumps to the next statement that follows the **`while`** statement. If the value of the *`expression`* is nonzero (meaning "true") then the *`statement`* is executed repeatedly until the *`expression`* evaluates to zero. Note that the *`condition`* must be enclosed by parentheses.

EXAMPLE 4.1 Using a `while` Loop to Compute a Sum of Consecutive Integers

This program computes the sum 1 + 2 + 3 + ··· + *n*, for an input integer *n*:

```
int main()
{ int n, i=1;
  cout << "Enter a positive integer: ";
  cin >> n;
  long sum=0;
  while (i <= n)
    sum += i++;
  cout << "The sum of the first " << n << " integers is " << sum;
}
```

This program uses three local variables: `n`, `i`, and `sum`. Each time the `while` loop iterates, `i` is incremented and then added to `sum`. The loop stops when `i` = `n`, so `n` is the last value added to `sum`. The trace at right shows the values of `i` and `sum` on each iteration after the user input 8 for `n`. The output for this run is

i	sum
0	0
1	1
2	3
3	6
4	10
5	15
6	21
7	28
8	36

```
Enter a positive integer: 8
The sum of the first 8 integers is 36
```

The program computed 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 = 36.

On the second run the user inputs 100 for `n`, so the `while` loop iterated 100 times to compute the sum 1 + 2 + 3 + ··· + 98 + 99 + 100 = 5050:

```
Enter a positive integer: 100
The sum of the first 100 integers is 5050
```

Note that the statement inside the loop is indented. This convention makes the program's logic easier to follow, especially in large programs.

EXAMPLE 4.2 Using a `while` Loop to Compute a Sum of Reciprocals

This program computes the sum of reciprocals $s = 1 + 1/2 + 1/3 + \cdots + 1/n$, where n is the smallest integer for which $n \geq s$:

```
int main()
{ int bound;
  cout << "Enter a positive integer: ";
  cin >> bound;
  double sum=0.0;
  int i=0;
  while (sum < bound)
    sum += 1.0/++i;
  cout << "The sum of the first " << i
       << " reciprocals is " << sum << endl;
}
```

With input 3 for `n`, this run computes $1 + 1/2 + 1/3 + \cdots + 1/11 = 3.01988$:

```
Enter a positive integer: 3
The sum of the first 11 reciprocals is 3.01988
```

The trace of this run is shown at right. The `sum` does not exceed 3 until the 11th iteration.

i	sum
0	0.00000
1	1.00000
2	1.50000
3	1.83333
4	2.08333
5	2.28333
6	2.45000
7	2.59286
8	2.71786
9	2.82897
10	2.92897
11	3.01988

EXAMPLE 4.3 Using a `while` Loop to Repeat a Computation

This program prints the square root of each number input by the user. It uses a `while` loop to allow any number of computations in a single run of the program:

```
int main()
{ double x;
  cout << "Enter a positive number: ";
  cin >> x;
  while (x > 0)
  { cout << "sqrt(" << x << ") = " << sqrt(x) << endl;
    cout << "Enter another positive number (or 0 to quit): ";
    cin >> x;
  }
}
```

```
Enter a positive number: 49
sqrt(49) = 7
Enter another positive number (or 0 to quit): 3.14159
sqrt(3.14159) = 1.77245
Enter another positive number (or 0 to quit): 100000
sqrt(100000) = 316.228
Enter another positive number (or 0 to quit): 0
```

The condition `(x > 0)` in Example 4.3 uses the variable `x` to control the loop. Its value is changed inside the loop by means of an input statement. A variable that is used this way is called a *loop control variable*.

4.2 TERMINATING A LOOP

We have already seen how the `break` statement is used to control the `switch` statement. (See Example 3.17 on page 47.) The `break` statement is also used to control loops.

EXAMPLE 4.4 Using a `break` Statement to Terminate a Loop

This program has the same effect as the one in Example 4.1 on page 60:

```
int main()
{ int n, i=1;
  cout << "Enter a positive integer: ";
  cin >> n;
  long sum=0;
  while (true)
  { if (i > n) break;  // terminates the loop immediately
    sum += i++;
  }
  cout << "The sum of the first " << n << " integers is " << sum;
}
```

```
Enter a positive integer: 100
The sum of the first 100 integers is 5050
```

This runs the same as in Example 4.1: as soon as the value of `i` reaches `n`, the loop terminates and the output statement at the end of the program executes.

Note that the control condition on the `while` loop itself is `true`, which means continue forever. This is the standard way to code a `while` loop when it is being controlled from within.

One advantage of using a `break` statement inside a loop is that it causes the loop to terminate immediately, without having to finish executing the remaining statements in the loop block.

EXAMPLE 4.5 The Fibonacci Numbers

The *Fibonacci numbers* $F_0, F_1, F_2, F_3, \ldots$ are defined recursively by the equations

$$\begin{cases} F_0 = 0 \\ F_1 = 1 \\ F_n = F_{n-1} + F_{n-2} \end{cases}$$

For example, letting $n = 2$ in the third equation yields

$$F_2 = F_{2-1} + F_{2-2} = F_1 + F_0 = 0 + 1 = 1$$

Similarly, with $n = 3$,

$$F_3 = F_{3-1} + F_{3-2} = F_2 + F_1 = 1 + 1 = 2$$

and with $n = 4$,

$$F_4 = F_{4-1} + F_{4-2} = F_3 + F_2 = 2 + 1 = 3$$

The first ten Fibonacci numbers are shown in the table at right.

n	F_n
0	0
1	1
2	1
3	2
4	3
5	5
6	8
7	13
8	21
9	35

This program prints all the Fibonacci numbers up to an input limit:

```
int main()
{ long bound;
  cout << "Enter a positive integer: ";
  cin >> bound;
  cout << "Fibonacci numbers < " << bound << ":\n0, 1";
  long f0=0, f1=1;
```

```
  while (true)
  { long f2 = f0 + f1;
    if (f2 > bound) break;  // terminates the loop immediately
    cout << ", " << f2;
    f0 = f1;
    f1 = f2;
  }
}
```

```
Enter a positive integer: 1000
Fibonacci numbers < 1000:
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987
```

This `while` loop contains a block of five statements. When the condition **`(f2 > bound)`** is evaluated to be true, the `break` statement executes, terminating the loop immediately, without executing the last three statements in that iteration.

Note the use of the *newline character* `\n` in the string `":\n0, 1"`. This prints the colon `:` at the end of the current line, and then prints `0, 1` at the beginning of the next line.

EXAMPLE 4.6 Using the `exit(0)` Function

The `exit()` function provides another way to terminate a loop. When it executes, it terminates the program itself:

```
int main()
{ long bound;
  cout << "Enter a positive integer: ";
  cin >> bound;
  cout << "Fibonacci numbers < " << bound << ":\n0, 1";
  long f0=0, f1=1;
  while (true)
  { long f2 = f0 + f1;
    if (f2 > bound) exit(0);  // terminates the program immediately
    cout << ", " << f2;
    f0 = f1;
    f1 = f2;
  }
}
```

```
Enter a positive integer: 1000
Fibonacci numbers < 1000:
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987
```

Since this program has no statements following its loop, terminating the loop is the same as terminating the program. So this program runs the same as the one in Example 4.5.

The program in Example 4.6 illustrates one way to break out of an infinite loop. The next example shows how to abort an infinite loop. But the preferred method is to use a break statement, as illustrated in Example 4.20 on page 71.

EXAMPLE 4.7 Aborting Infinite Loop

Without some termination mechanism, the loop will run forever. To abort its execution after it starts, press <Ctrl>+C (*i.e.*, hold the Ctrl key down and press the C key on your keyboard):

```
int main()
{ long bound;
  cout << "Enter a positive integer: ";
  cin >> bound;
  cout << "Fibonacci numbers < " << bound << ":\n0, 1";
  long f0=0, f1=1;
  while (true)          // ERROR: INFINITE LOOP!     (Press <Ctrl>+C.)
  { long f2 = f0 + f1;
    cout << ", " << f2;
    f0 = f1;
    f1 = f2;
  }
}
```

```
Enter a positive integer: 1000
Fibonacci numbers < 1000:
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597
81, 6765, 10946, 17711, 28657, 46368, 75025, 121393, 196418, 317811, 5
040, 1346269, 2178309, 3524578, 5702887, 9227465, 14930352, 24157817,
63245986, 102334155, 165580141, 267914296, 433494437, 701408733, 11349
```

Since this program has no statements following its loop, terminating the loop is the same as terminating the program. So this program runs the same as the one in Example 4.5.

4.3 THE `do..while` STATEMENT

The syntax for the `do..while` statement is

`do` *`statement`* **`while (`***`condition`***`);`**

where *`condition`* is an integral expression and *`statement`* is any executable statement. It repeatedly executes the *`statement`* and then evaluates the *`condition`* until that condition evaluates to `false`.

The `do..while` statement works the same as the `while` statement except that its condition is evaluated at the end of the loop instead of at the beginning. This means that any control variables can be defined within the loop instead of before it. It also means that a `do...while` loop will always iterate at least once, regardless of the value of its control condition.

EXAMPLE 4.8 Using a `do..while` Loop to Compute a Sum of Consecutive Integers

This program has the same effect as the one in Example 4.1 on page 60:

```
int main()
{ int n, i=0;
  cout << "Enter a positive integer: ";
  cin >> n;
  long sum=0;
  do
    sum += i++;
  while (i <= n);
  cout << "The sum of the first " << n << " integers is " << sum;
}
```

EXAMPLE 4.9 The Factorial Numbers

The *factorial numbers* 0!, 1!, 2!, 3!, ⋯ are defined recursively by the equations

$$\begin{cases} 0! = 1 \\ n! = n(n-1) \end{cases}$$

For example, letting $n = 1$ in the second equation yields

$$1! = 1((1-1)!) = 1(0!) = 1(1) = 1$$

Similarly, with $n = 2$:

$$2! = 2((2-1)!) = 2(1!) = 2(1) = 2$$

and with $n = 3$:

$$3! = 3((3-1)!) = 3(2!) = 3(2) = 6$$

The first seven factorial numbers are shown in the table at right.

n	*n*!
0	1
1	1
2	2
3	6
4	24
5	120
6	720

This program prints all the factorial numbers up to an input limit:

```
int main()
{ long bound;
  cout << "Enter a positive integer: ";
  cin >> bound;
  cout << "Factorial numbers < " << bound << ":\n1, 1";
  long f=1, i=1;
  do
  { f *= ++i;
    cout << ", " << f;
  }
  while (f < bound);
}
```

```
Enter a positive integer: 1000000
Factorial numbers < 1000000:
1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880
```

The `do..while` loop iterates until its control condition **`(f < bound)`** is false.

4.4 THE `for` STATEMENT

The syntax for the `for` statement is

`for` (*`initialization`*`;` *`condition`*`;` *`update`*`)` *`statement`*`;`

where *`initialization`*, *`condition`*, and *`update`* are optional expressions, and *`statement`* is any executable statement. The three-part (*`initialization`*`;` *`condition`*`;` *`update`*) controls the loop. The *`initialization`* expression is used to declare and/or initialize control variable(s) for the loop; it is evaluated first, before any iteration occurs. The *`condition`* expression is used to determine whether the loop should continue iterating; it is evaluated immediately after the initialization; if it is true, the statement is executed. The *`update`* expression is used to update the control variable(s); it is evaluated after the statement is executed. So the sequence of events that generate the iteration are:

1. evaluate the *`initialization`* expression;
2. if the value of the *`condition`* expression is false, terminate the loop;
3. execute the *`statement`*;
4. evaluate the *`update`* expression;
5. repeat steps 2–4.

EXAMPLE 4.10 Using a `for` Loop to Compute a Sum of Consecutive Integers

This program has the same effect as the one in Example 4.1 on page 60:

```
int main()
{ int n;
  cout << "Enter a positive integer: ";
  cin >> n;
  long sum=0;
  for (int i=1; i <= n; i++)
    sum += i;
  cout << "The sum of the first " << n << " integers is " << sum;
}
```

Here, the initialization expression is **`int i=1`**, the condition expression is **`i <= n`**, and the update expression is **`i++`**. Note that these same expressions are used in the programs in Example 4.1 on page 60, Example 4.4 on page 62, and Example 4.8 on page 64.

In Standard C++, when a loop control variable is declared within a `for` loop, as `i` is in Example 4.10, its scope is limited to that `for` loop. That means that it cannot be used outside that `for` loop. It also means that the same name can be used for different variables outside that `for` loop.

EXAMPLE 4.11 Reusing `for` Loop Control Variable Names

This program has the same effect as the one in Example 4.1 on page 60:

```
int main()
{ int n;
  cout << "Enter a positive integer: ";
  cin >> n;
  long sum=0;
  for (int i=1; i < n/2; i++)  // the scope of this i is this loop
    sum += i;
  for (int i=n/2; i <= n; i++)  // the scope of this i is this loop
    sum += i;
  cout << "The sum of the first " << n << " integers is "
       << sum << endl;
}
```

The two `for` loops in this program do the same computations as the single `for` loop in the program in Example 4.10. They simply split the job in two, doing the first `n/2` accumulations in the first loop and the rest in the second. Each loop independently declares its own control variable `i`.

Warning: Most pre-Standard C++ compilers extend the scope of a `for` loop's control variable past the end of the loop.

EXAMPLE 4.12 The Factorial Numbers Again

This program has the same effect as the one in Example 4.9 on page 65:

```
int main()
{ long bound;
  cout << "Enter a positive integer: ";
  cin >> bound;
```

```
  cout << "Factorial numbers that are <= " << bound << ":\n1, 1";
  long f=1;
  for (int i=2; f <= bound; i++)
  { f *= i;
    cout << ", " << f;
  }
}
```

```
Enter a positive integer: 1000000
Factorial numbers < 1000000:
1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880
```

This **for** loop program has the same effect as the **do..while** loop program because it executes the same instructions. After initializing `f` to 1, both programs initialize `i` to 2 and then repeat the following five instructions: print `f`, multiply `f` by `i`, increment `i`, check the condition `(f <= bound)`, and terminate the loop if the condition is false.

The **for** statement is quite flexible, as the following examples demonstrate.

EXAMPLE 4.13 Using a Descending `for` Loop

This program prints the first ten positive integers in reverse order:

```
int main()
{ for (int i=10; i > 0; i--)
    cout << " " << i;
}
```

```
 10 9 8 7 6 5 4 3 2 1
```

EXAMPLE 4.14 Using a `for` Loop with a Step Greater than One

This program determines whether an input number is prime:

```
int main()
{ long n;
  cout << "Enter a positive integer: ";
  cin >> n;
  if (n < 2) cout << n << " is not prime." << endl;
  else if (n < 4) cout << n << " is prime." << endl;
  else if (n%2 == 0) cout << n << " = 2*" << n/2 << endl;
  else
  { for (int d=3; d <= n/2; d += 2)
      if (n%d == 0)
      { cout << n << " = " << d << "*" << n/d << endl;
        exit(0);
      }
    cout << n << " is prime." << endl;
  };
}
```

```
Enter a positive integer: 101
101 is prime.
```

```
Enter a positive integer: 975313579
975313579 = 17*57371387
```

Note that this **for** loop uses an increment of 2 on its control variable `i`.

EXAMPLE 4.15 Using a Sentinel to Control a `for` Loop

This program finds the maximum of a sequence of input numbers:

```
int main()
{ int n, max;
  cout << "Enter positive integers (0 to quit): ";
  cin >> n;
  for (max = n; n > 0; )
  { if (n > max) max = n;
    cin >> n;
  }
  cout << "max = " << max << endl;
}
```

```
Enter positive integers (0 to quit): 44 77 55 22 99 33 11 66 88 0
max = 99
```

This **`for`** loop is controlled by the input variable `n`; it continues until $n \leq 0$. When an input variable controls a loop this way, it is called a *sentinel*.

Note the control mechanism **`(max = n; n > 0; )`** in this **`for`** loop. Its update part is missing, and its initialization **`max = n`** has no declaration. The variable `max` has to be declared before the **`for`** loop because it is used outside of its block, in the last output statement in the program.

EXAMPLE 4.16 Using a Loop Invariant to Prove that a `for` Loop is Correct

This program finds the minimum of a sequence of input numbers. It is similar to the program in Example 4.15:

```
int main()
{ int n, min;
  cout << "Enter positive integers (0 to quit): ";
  cin >> n;
  for (min = n; n > 0; )
  { if (n < min) min = n;
    // INVARIANT: min <= n for all n, and min equals one of the n
    cin >> n;
  }
  cout << "min = " << min << endl;
}
```

```
Enter positive integers (0 to quit): 44 77 55 22 99 33 11 66 88 0
min = 11
```

The full-line comment inside the block of the **`for`** loop is called a *loop invariant*. It states a condition that has two characteristic properties: (1) it is true at that point on every iteration of the loop; (2) the fact that it is true when the loop terminates proves that the loop performs correctly. In this case, the condition **`min <= n for all n`** is always true because the preceding **`if`** statement resets the value of `min` if the last input value of `n` was less than the previous value of `min`. And the condition that **`min equals one of the n`** is always true because **`min`** is initialized to the first `n` and the only place where `min` changes its value is when it is assigned to a new input value of `n`. Finally, the fact that the condition is true when the loop terminates means that `min` is the minimum of all the input numbers. And that outcome is precisely the objective of the `for` loop.

EXAMPLE 4.17 More than One Control Variable in a `for` Loop

The **`for`** loop in this program uses two control variables:

```
int main()
{ for (int m=95, n=11; m%n > 0; m -= 3, n++)
    cout << m << "%" << n << " = " << m%n << endl;
}
```

```
95%11 = 7
92%12 = 8
89%13 = 11
86%14 = 2
83%15 = 8
```

The two control variables `m` and `n` are declared and initialized in the control mechanism of this **`for`** loop. Then `m` is decremented by 3 and `n` is incremented on each iteration of the loop, generating the sequence of (`m`,`n`) pairs (95,11), (92,12), (89,13), (86,14), (83,15), (80,16). The loop terminates with the pair (80,16) because 16 divides 80.

EXAMPLE 4.18 Nesting `for` Loops

This program prints a multiplication table:

```
#include <iomanip>   // defines setw()
#include <iostream>  // defines cout
using namespace std;
int main()
{ for (int x=1; x <= 12; x++)
  { for (int y=1; y <= 12; y++)
      cout << setw(4) << x*y;
    cout << endl;
  }
}
```

```
   1   2   3   4   5   6   7   8   9  10  11  12
   2   4   6   8  10  12  14  16  18  20  22  24
   3   6   9  12  15  18  21  24  27  30  33  36
   4   8  12  16  20  24  28  32  36  40  44  48
   5  10  15  20  25  30  35  40  45  50  55  60
   6  12  18  24  30  36  42  48  54  60  66  72
   7  14  21  28  35  42  49  56  63  70  77  84
   8  16  24  32  40  48  56  64  72  80  88  96
   9  18  27  36  45  54  63  72  81  90  99 108
  10  20  30  40  50  60  70  80  90 100 110 120
  11  22  33  44  55  66  77  88  99 110 121 132
  12  24  36  48  60  72  84  96 108 120 132 144
```

Each iteration of the outer `x` loop prints one row of the multiplication table. For example, on the first iteration when `x` = 1, the inner `y` loop iterates 12 times, printing `1*y` for each value of `y` from 1 to 12. And then on the second iteration of the outer `x` loop when `x` = 2, the inner `y` loop iterates 12 times again, this time printing `2*y` for each value of `y` from 1 to 12. Note that the separate `cout << endl` statement must be inside the outer loop and outside the inner loop in order to produce exactly one line for each iteration of the outer loop.

This program uses the *stream manipulator* `setw` to set the width of the output field for each integer printed. The expression `setw(4)` means to "set the output field width to 4 columns" for the next output.

This aligns the outputs into a readable table of 12 columns of right-justified integers. Stream manipulators are defined in the `<iomanip>` header, so this program had to include the directive

```
#include <iomanip>
```

in addition to including the `<iostream>` header.

EXAMPLE 4.19 Testing a Loop Invariant

This program computes and prints the *discrete binary logarithm* of an input number (the greatest integer ≤ the base 2 logarithm of the number). It tests its loop invariant by printing the relevant values on each iteration:

```
#include <cmath>     // defines pow() and log()
#include <iostream>  // defines cin and cout
#include <iomanip>   // defines setw()
using namespace std;

int main()
{ long n;
  cout << "Enter a positive integer: ";
  cin >> n;
  int d=0;  // the discrete binary logarithm of n
  double p2d=1;  // = 2^d
  for (int i=n; i > 1; i /= 2, d++)
  { // INVARIANT: 2^d <= n/i < 2*2^d
    p2d=pow(2,d);  // = 2^d
    cout << setw(2) << p2d << " <= " << setw(2) << n/i
         << " < " << setw(2) << 2*p2d << endl;
  }
  p2d=pow(2,d);  // = 2^d
  cout << setw(2) << p2d << " <= " << setw(2) << n
       << " < " << setw(2) << 2*p2d << endl;
  cout << "  The discrete binary logarithm of " << n
       << " is " << d << endl;
  double lgn = log(n)/log(2);  // base 2 logarithm of n
  cout << "The continuous binary logarithm of " << n
       << " is " << lgn << endl;
}
```

```
Enter a positive integer: 63
 1 <=  1 <  2
 2 <=  2 <  4
 4 <=  4 <  8
 8 <=  9 < 16
16 <= 21 < 32
32 <= 63 < 64
  The discrete binary logarithm of 63 is 5
The continuous binary logarithm of 63 is 5.97728
```

The discrete binary logarithm is computed to be the number of times the input number can be divided by 2 before reaching 1. So the **`for`** loop initializes `i` to `n` and then divides `i` by 2 once on each iteration. The counter `c` counts the number of iterations. So when the loop terminates, `c` contains the value of the discrete binary logarithm of `n`.

In addition to using the `setw()` function that is defined in the `<iomanip>` header, this program also uses the `log()` function that is defined in the `<cmath>` header. That function returns the natural

(base e) logarithm of n: `log(n)` = $\log_e n = \ln n$. It is used in the expression `log(n)/log(2)` to compute the binary (base 2) logarithm of n: $\log_2 n = \lg n = (\ln n)/(\ln 2)$. The printed results compare the discrete binary logarithm with the continuous binary logarithm. The former is equal to the latter truncated downward to its nearest integer (the *floor* of the number).

The loop invariant in this example is the condition **`2^d <= n/i < 2*2^d`** (*i.e.*, $2^d \le n/i < 2 \cdot 2^d$). It is tested by printing the values of the three expressions **`p2d`**, **`n`**, and **`2*p2d`**, where the quantity **`p2d`** is computed with the power function `pow()` that is defined in the `<cmath>` header.

We can prove that this **`for`** loop will always compute the discrete binary logarithm correctly. When it starts, $d = 0$ and $i = n$, so $2^d = 2^0 = 1$, $n/i = n/n = 1$, and $2 \cdot 2^d = 2 \cdot 1 = 2$; thus $2^d \le n/i < 2 \cdot 2^d$. On each iteration, d increments and i is halved, so n/i is doubled. Thus the condition $2^d \le n/i < 2 \cdot 2^d$ remains invariant; *i.e.*, it is true initially and it remains true throughout the life of the loop. When the loop terminates, $i = 1$, so the condition becomes $2^d \le n/1 < 2 \cdot 2^d$, which is equivalent to $2^d \le n < 2^{d+1}$. The logarithm of this expression is $d = \lg(2^d) \le \lg n < \lg(2^{d+1}) = d+1$, so d is greatest integer $\le \lg n$.

4.5 THE `break` STATEMENT

We have already seen the **`break`** statement used in the **`switch`** statement. It is also used in loops. When it executes, it terminates the loop, "breaking out" of the iteration at that point.

EXAMPLE 4.20 Using a `break` Statement to Terminate a Loop

This program has the same effect as the one in Example 4.1 on page 60. It uses a **`break`** statement to control the loop:

```
int main()
{ int n, i=1;
  cout << "Enter a positive integer: ";
  cin >> n;
  long sum=0;
  while (true)
  { if (i > n) break;
    sum += i++;
  }
  cout << "The sum of the first " << n << " integers is " << sum;
}
```

```
Enter a positive integer: 8
The sum of the first 8 integers is 36
```

As long as `(i <= n)`, the loop will continue, just as in Example 4.1. But as soon as `i > n`, the **`break`** statement executes, immediately terminating the loop.

The **`break`** statement provides extra flexibility in the control of loops. Normally a **`while`** loop, a **`do..while`** loop, or a **`for`** loop will terminate only at the beginning or at the end of the complete sequence of statements in the loop's block. But the **`break`** statement can be placed anywhere among the other statements within a loop, so it can be used to terminate a loop anywhere from within the loop's block. This is illustrated by the following example.

EXAMPLE 4.21 Controlling Input with a Sentinel

This program reads a sequence of positive integers, terminated by 0, and prints their average:

```
int main()
{ int n, count=0, sum=0;
  cout << "Enter positive integers (0 to quit):" << endl;
  for (;;)  // "forever"
  { cout << "\t" << count + 1 << ": ";
    cin >> n;
    if (n <= 0) break;
    ++count;
    sum += n;
  }
  cout << "The average of those " << count << " positive numbers is "
       << float(sum)/count << endl;
}
```

```
Enter positive integers (0 to quit):
        1: 4
        2: 7
        3: 1
        4: 5
        5: 2
        6: 0
The average of those 5 positive numbers is 3.8
```

When 0 is input, the **break** executes, immediately terminating the **for** loop and transferring execution to the final output statement. Without the **break** statement, the `++count` statement would have to be put in a conditional, or `count` would have to be decremented outside the loop or initialized to –1.

Note that all three parts of the **for** loop's control mechanism are empty: **for** `(;;)`. This construct is pronounced "forever." Without the **break**, this would be an infinite loop.

When used within nested loops, the **break** statement applies only to the loop to which it directly belongs; outer loops will continue, unaffected by the break. This is illustrated by the following example.

EXAMPLE 4.22 Using a `break` Statement with Nested Loops

Since multiplication is commutative (*e.g.*, 3×4 = 4×3), multiplication tables are often presented with the numbers above the main diagonal omitted. This program modifies that of Example 4.18 on page 69 to print a triangular multiplication table:

```
int main()
{ for (int x=1; x <= 12; x++)
  { for (int y=1; y <= 12; y++)
      if (y > x) break;
      else cout << setw(4) << x*y;
    cout << endl;
  }
}
```

```
 1
 2   4
 3   6   9
 4   8  12  16
 5  10  15  20  25
 6  12  18  24  30  36
 7  14  21  28  35  42  49
 8  16  24  32  40  48  56  64
 9  18  27  36  45  54  63  72  81
10  20  30  40  50  60  70  80  90 100
11  22  33  44  55  66  77  88  99 110 121
12  24  36  48  60  72  84  96 108 120 132 144
```

When `y > x`, the execution of the inner `y` loop terminates and the next iteration of the outer `x` loop begins. For example, when `x` = 3, the `y` loop iterates 3 times (with `y` = 1, 2, 3), printing `3 6 9`. Then on its 4th iteration, the condition `(y > x)` is true, so the **`break`** statement executes, transferring control immediately to the `cout << endl` statement (which is outside of the inner `y` loop). Then the outer `x` loop begins its 4th iteration with `x` = 4.

4.6 THE `continue` STATEMENT

The **`break`** statement skips the rest of the statements in the loop's block, jumping immediately to the next statement outside of the loop. The **`continue`** statement is similar. It also skips the rest of the statements in the loop's block, but instead of terminating the loop, it transfers execution to the next iteration of the loop. It continues the loop after skipping the remaining statements in its current iteration.

EXAMPLE 4.23 Using `continue` and `break` Statements

This little program illustrates the **`continue`** and **`break`** statements:

```
int main()
{ int n;
  for (;;)
  { cout << "Enter int: ";  cin >> n;
    if (n%2 == 0) continue;
    if (n%3 == 0) break;
    cout << "\tBottom of loop.\n";
  }
  cout << "\tOutside of loop.\n";
}
```

```
Enter int: 7
        Bottom of loop.
Enter int: 4
Enter int: 9
        Outside of loop.
```

When `n` has the value 7, both **`if`** conditions are false and control reaches the bottom of the loop. When `n` has the value 4, the first **`if`** condition is true (4 is a multiple of 2), so control skips over the rest of the statements in the loop and jumps immediately to the top of the loop again to continue with its next iteration. When `n` has the value 9, the first **`if`** condition is false (9 is not a multiple of 2) but the second **`if`** condition is true (9 is a multiple of 3), so control breaks out of the loop and jumps immediately to the first statement that follows the loop.

4.7 THE goto STATEMENT

The **break** statement, the **continue** statement, and the **switch** statement each cause the program control to branch to a location other than where it normally would go. The destination of the branch is determined by the context: **break** goes to the next statement outside the loop, **continue** goes to the loop's continue condition, and **switch** goes to the correct case constant. All three of these statements are called *jump statements* because they cause the control of the program to "jump over" other statements.

The **goto** statement is another kind of jump statement. Its destination is specified by a label within the statement.

A *label* is simply an identifier followed by a colon placed in front of a statement. Labels work like the **case** statements inside a **switch** statement: they specify the destination of the jump.

Example 4.22 illustrated how a **break** normally behaves within nested loops: execution breaks out of only the innermost loop that contains the **break** statement. Breaking out of several or all of the loops in a nest requires a **goto** statement, as the next example illustrates.

EXAMPLE 4.24 Using a goto Statement to Break Out of a Nest of Loops

```
int main()
{ const int N=5;
  for (int i=0; i<N; i++)
  { for (int j=0; j<N; j++)
    { for (int k=0; k<N; k++)
        if (i+j+k>N) goto esc;
        else cout << i+j+k << " ";
      cout << "* ";
    }
  esc: cout << "." << endl;  // inside the i loop, outside the j loop
  }
}
```

```
0 1 2 3 4 * 1 2 3 4 5 * 2 3 4 5 .
1 2 3 4 5 * 2 3 4 5 .
2 3 4 5 .
3 4 5 .
4 5 .
```

When the **goto** is reached inside the innermost `k` loop, program execution jumps out to the labeled output statement at the bottom of the outermost `i` loop. Since that is the last statement in the `i` loop, the `i` loop will go on to its next iteration after executing that statement.

When `i` and `j` are 0, the `k` loop iterates 5 times, printing `0 1 2 3 4` followed by a star `*`. Then `j` increments to 1 and the `k` loop iterates 5 times again, printing `1 2 3 4 5` followed by a star `*`. Then `j` increments to 2 and the `k` loop iterates 4 times, printing `2 3 4 5`. But then on the next iteration of the `k` loop, `i` = 0, `j` = 2, and `k` = 4, so `i+j+k` = 6, causing the **goto** statement to execute for the first time. So execute jumps immediately to the labeled output statement, printing a dot and advancing to the next line. Note that both the `k` loop and the `j` loop are aborted before finishing all their iterations.

Now `i` = 1 and the middle `j` loop begins iterating again with `j` = 0. The `k` loop iterates 5 times, printing `1 2 3 4 5` followed by a star `*`. Then `j` increments to 1 and the `k` loop iterates 4 times, printing `2 3 4 5`. But then on the next iteration of the `k` loop, `i` = 1, `j` = 2, and `k` = 3, so `i+j+k` = 6, causing the **goto** statement to execute for the second time. Again execution jumps immediately to the labeled output statement, printing a dot and advancing to the next line.

On the subsequent three iterations of the outer `i` loop, the inner `k` loop never completes its iterations because `i+j+4` is always greater than 5 (because `i` is greater than 2). So no more stars are printed.

Note that the labeled output statement could be placed inside any of the loops or even outside of all of them. In the latter case, the **`goto`** statement would terminate all three of the loops in the nest.

Also note how the labeled statement is indented. The convention is to shift it to the left one indentation level to make it more visible. If it were not a labeled statement, it would be indented as

```
        }
        cout << "." << endl;
    }
```

instead of

```
        }
    esc: cout << "." << endl;
    }
```

Example 4.24 illustrates one way to break out of a nest of loops. Another method is to use a flag. A *flag* is a boolean variable that is initialized to `false` and then later set to `true` to signal an exceptional event; normal program execution is interrupted when the flag becomes true. This is illustrated by the following example.

EXAMPLE 4.25 Using a Flag to Break Out of a Nest of Loops

This program has the same output as that in Example 4.24:

```
int main()
{ const int N=5;
  bool done=false;
  for (int i=0; i<N; i++)
  { for (int j=0; j<N && !done; j++)
    { for (int k=0; k<N && !done; k++)
        if (i+j+k>N) done = true;
        else cout << i+j+k << " ";
      cout << "* ";
    }
    cout << "." << endl;  // inside the i loop, outside the j loop
    done = false;
  }
}
```

When the `done` flag becomes true, both the innermost `k` loop and the middle `j` loop will terminate, and the outer `i` loop will finish its current iteration by printing the dot, advancing to the beginning of the next line, and resetting the `done` flag to false. Then it starts its next iteration, the same as in Example 4.24.

4.8 GENERATING PSEUDO-RANDOM NUMBERS

One of the most important applications of computers is the *simulation* of real-world systems. Most high-tech research and development is heavily dependent upon this technique for studying how systems work without actually having to interact with them directly.

Simulation requires the computer generation of *random numbers* to model the uncertainty of the real world. Of course, computers cannot actually generate truly random numbers because computers are *deterministic*: given the same input, the same computer will always produce the

same output. But it is possible to generate numbers that appear to be randomly generated; *i.e.*, numbers that are uniformly distributed within a given interval and for which there is no discernible pattern. Such numbers are called *pseudo-random numbers*.

The Standard C header file `<cstdlib>` defines the function `rand()` which generates pseudo-random integers in the range 0 to `RAND_MAX`, which is a constant that is also defined in `<cstdlib>`. Each time the `rand()` function is called, it generates another **`unsigned`** integer in this range.

EXAMPLE 4.26 Generating Pseudo-Random Numbers

This program uses the `rand()` function to generate pseudo-random numbers:

```
#include <cstdlib> // defines the rand() function and RAND_MAX const
#include <iostream>
using namespace std;

int main()
{ // prints pseudo-random numbers:
  for (int i = 0; i < 8; i++)
    cout << rand() << endl;
  cout << "RAND_MAX = " << RAND_MAX << endl;
}
```

```
1103527590
377401575
662824084
1147902781
2035015474
368800899
1508029952
486256185
RAND_MAX = 2147483647
```

```
1103527590
377401575
662824084
1147902781
2035015474
368800899
1508029952
486256185
RAND_MAX = 2147483647
```

On each run, the computer generates 8 **`unsigned`** integers that are uniformly distributed in the interval 0 to `RAND_MAX`, which is 2,147,483,647 on this computer. Unfortunately each run produces the same sequence of numbers. This is because they are generated from the same "seed."

Each pseudo-random number is generated from the previously generated pseudo-random number by applying a special "number crunching" function that is defined internally. The first pseudo-random number is generated from an internally defined variable, called the *seed* for the sequence. By default, this seed is initialized by the computer to be the same value every time the program is run. To overcome this violation of pseudo-randomness, we can use the `srand()` function to select our own seed.

EXAMPLE 4.27 Setting the Seed Interactively

This program is the same as the one in Example 4.26 except that it allows the pseudo-random number generator's seed to be set interactively:

```
#include <cstdlib>  // defines the rand() and srand() functions
#include <iostream>
using namespace std;

int main()
{ // prints pseudo-random numbers:
  unsigned seed;
  cout << "Enter seed: ";
  cin >> seed;
  srand(seed);                          // initializes the seed
  for (int i = 0; i < 8; i++)
    cout << rand() << endl;
}
```

```
Enter seed: 0
12345
1406932606
654583775
1449466924
229283573
1109335178
1051550459
1293799192
```

```
Enter seed: 1
1103527590
377401575
662824084
1147902781
2035015474
368800899
1508029952
486256185
```

```
Enter seed: 12345
1406932606
654583775
1449466924
229283573
1109335178
1051550459
1293799192
794471793
```

The line `srand(seed)` assigns the value of the variable `seed` to the internal "seed" used by the `rand()` function to initialize the sequence of pseudo-random numbers that it generates. Different seeds produce different results.

Note that the `seed` value 12345 used in the third run of the program is the first number generated by `rand()` in the first run. Consequently the first through seventh numbers generated in the third run are the same as the second through eighth numbers generated in the first run. Also note that the sequence generated in the second run is the same as the one produced in Example 4.26. This suggests that, on this computer, the default seed value is 1.

The problem of having to enter a `seed` value interactively can be overcome by using the computer's system clock. The *system clock* keeps track of the current time in seconds. The `time()` function defined in the header file `<ctime>` returns the current time as an **`unsigned`** integer. This then can be used as the seed for the `rand()` function.

EXAMPLE 4.28 Setting the Seed from the System Clock

This program is the same as the one in Example 4.27 except that it sets the pseudo-random number generator's seed from the system clock.

Note: if your compiler does not recognize the `<ctime>` header, then use the pre-standard `<time.h>` header instead.

```
#include <cstdlib>  // defines the rand() and srand() functions
#include <ctime>      // defines the time() function
#include <iostream>
//#include <time.h>   // use this if <ctime> is not recognized
using namespace std;
int main()
{ // prints pseudo-random numbers:
  unsigned seed = time(NULL);          // uses the system clock
  cout << "seed = " << seed << endl;
  srand(seed);                         // initializes the seed
  for (int i = 0; i < 8; i++)
    cout << rand() << endl;
}
```

Here are two runs using a UNIX workstation running a Motorola processor:

```
seed = 808148157
1877361330
352899587
1443923328
1857423289
200398846
1379699551
1622702508
715548277
```

```
seed = 808148160
892939769
1559273790
1468644255
952730860
1322627253
1305580362
844657339
440402904
```

On the first run, the `time()` function returns the integer 808,148,157 which is used to "seed" the random number generator. The second run is done 3 seconds later, so the `time()` function returns the integer 808,148,160 which generates a completely different sequence.

Here are two runs using a Windows PC running an Intel processor:

In many simulation programs, one needs to generate random integers that are uniformly distributed in a given range. The next example illustrates how to do that.

```
seed = 943364015
2948
15841
72
25506
30808
29709
13115
2527
```

```
seed = 943364119
17427
20464
13149
5702
12766
1424
16612
31746
```

EXAMPLE 4.29 Generating Pseudo-Random Numbers in Given Range

This program is the same as the one in Example 4.28 except that the pseudo-random numbers that it generates are restricted to given range:

```
#include <cstdlib>
#include <ctime>      // defines the time() function
#include <iostream>
//#include <time.h>   // use this if <ctime> is not recognized
using namespace std;
int main()
{ // prints pseudo-random numbers:
  unsigned seed = time(NULL);          // uses the system clock
  cout << "seed = " << seed << endl;
  srand(seed);                         // initializes the seed
  int min, max;
  cout << "Enter minimum and maximum: ";
  cin >> min >> max;                  // lowest and highest numbers
  int range = max - min + 1;          // number of numbers in range
  for (int i = 0; i < 20; i++)
  { int r = rand()/100%range + min;
    cout << r << " ";
  }
  cout << endl;
}
```

Here are two runs:

```
seed = 808237677
Enter minimum and maximum: 1 100
85 57 1 10 5 73 81 43 46 42 17 44 48 9 3 74 41 4 30 68
```

```
seed = 808238101
Enter minimum and maximum: 22 66
63 29 56 22 53 57 39 56 43 36 62 30 41 57 26 61 59 26 28
```

The first run generates 20 integers uniformly distributed between 1 and 100. The second run generates 20 integers uniformly distributed between 22 and 66.

In the **for** loop, we divide `rand()` by 100 first to strip way the two right-most digits of the random number. This is to compensate for the problem that this particular random number generator has of producing numbers that alternate odd and even. Then `rand()/100%range` produces random numbers in the range `0` to `range-1`, and `rand()/100%range + min` produces random numbers in the range `min` to `max`.

Review Questions

4.1 What happens in a **while** loop if the control condition is false (*i.e.*, zero) initially?

4.2 When should the control variable in a **for** loop be declared before the loop (instead of within its control mechanism)?

4.3 How does the **break** statement provide better control of loops?

4.4 What is the minimum number of iterations that

a. a while loop could make?

b. a **do..while** loop could make?

4.5 What is wrong with the following loop:

```
while (n <= 100)
sum += n*n;
```

4.6 If *s* is a compound statement, and *e1*, *e2*, and *e3* are expressions, then what is the difference between the program fragment:

```
for (e1; e2; e3)
s;
```

and the fragment:

```
e1;
while (e2)
{ s;
  e3;
}
```

4.7 What is wrong with the following program:

```
int main()
{ const double PI;
  int n;
  PI = 3.14159265358979;
  n = 22;
}
```

4.8 What is an "infinite loop," and how can it be useful?

4.9 How can a loop be structured so that it terminates with a statement in the middle of its block?

4.10 Why should tests for equality with floating-point variables be avoided?

Problems

4.1 Trace the following code fragment, showing the value of each variable each time it changes:

```
float x = 4.15;
for (int i=0; i < 3; i++)
  x *= 2;
```

4.2 Assuming that *e* is an expression and *s* is a statement, convert each of the following **for** loops into an equivalent **while** loop:

a. for (; e;) s

b. `for ( ; ; e) s`

4.3 Convert the following **for** loop into a **while** loop:

```
for (int i=1; i <= n; i++)
  cout << i*i << " ";
```

4.4 Describe the output from this program:

```
int main()
{ for (int i = 0; i < 8; i++)
    if (i%2 == 0) cout << i + 1 << "\t";
    else if (i%3 == 0) cout << i*i << "\t";
    else if (i%5 == 0) cout << 2*i - 1 << "\t";
    else cout << i << "\t";
}
```

4.5 Describe the output from this program:

```
int main()
{ for (int i=0; i < 8; i++)
  { if (i%2 == 0) cout << i + 1 << endl;
    else if (i%3 == 0) continue;
    else if (i%5 == 0) break;
    cout << "End of program.\n";
  }
  cout << "End of program.\n";
}
```

4.6 In a 32-bit **float** type, 23 bits are used to store the mantissa and 8 bits are used to store the exponent.

a. How many significant digits of precision does the 32-bit **float** type yield?

b. What is the range of magnitude for the 32-bit **float** type?

4.7 Write and run a program that uses a **while** loop to compute and prints the sum of a given number of squares. For example, if 5 is input, then the program will print 55, which equals $1^2 + 2^2 + 3^2 + 4^2 + 5^2$.

4.8 Write and run a program that uses a **for** loop to compute and prints the sum of a given number of squares.

4.9 Write and run a program that uses a **do..while** loop to compute and prints the sum of a given number of squares.

4.10 Write and run a program that directly implements the quotient operator **/** and the remainder operator **%** for the division of positive integers.

4.11 Write and run a program that reverses the digits of a given positive integer. (See Problem 3.13 on page 51.)

4.12 Apply the *Babylonian Algorithm* to compute the square root of 2. This algorithm (so called because it was used by the ancient Babylonians) computes $\sqrt{2}$ by repeatedly replacing one estimate x with the closer estimate $(x + 2/x)/2$. Note that this is simply the average of x and $2/x$.

4.13 Write a program to find the integer square root of a given number. That is the largest integer whose square is less than or equal to the given number.

4.14 Implement the *Euclidean Algorithm* for finding the greatest common divisor of two given positive integers. This algorithm transforms a pair of positive integers (m, n) into a pair $(d, 0)$ by repeatedly dividing the larger integer by the smaller integer and replacing the larger with

the remainder. When the remainder is 0, the other integer in the pair will be the greatest common divisor of the original pair (and of all the intermediate pairs). For example, if *m* is 532 and *n* is 112, then the Euclidean Algorithm reduces the pair (532,112) to (28,0) by

$$(532,112) \rightarrow (112,84) \rightarrow (84,28) \rightarrow (28,0).$$

So 28 is the greatest common divisor of 532 and 112. This result can be verified from the facts that $532 = 28 \cdot 19$ and $112 = 28 \cdot 8$. The reason that the Euclidean Algorithm works is that each pair in the sequence has the same set of divisors, which are precisely the factors of the greatest common divisor. In the example above, that common set of divisors is {1, 2, 4, 7, 14, 28}. The reason that this set of divisors is invariant under the reduction process is that when $m = n \cdot q + r$, a number is a common divisor of *m* and *n* if and only if it is a common divisor of *n* and *r*.

Answers to Review Questions

4.1 If the control condition of a `while` loop is initially false, then the loop is skipped altogether; the statement(s) inside the loop are not executed at all.

4.2 The control variable in a **`for`** loop has to be declared before the loop (instead of within its control mechanism) if it is used outside of the loop's statement block, as in Example 4.14 on page 67.

4.3 The **`break`** statement provides better control of loops by allowing immediate termination of the loop after any statement within its block. Without a **`break`** statement, the loop can terminate only at the beginning or at the end of the block.

4.4 ***a.*** The minimum number of iterations that a **`while`** loop could make is 0.

b. The minimum number of iterations that a **`do..while`** loop could make is 1.

4.5 That is an infinite loop because the value of its control variable `n` does not change.

4.6 There is no difference between the effects of those two program fragments, unless *s* is a **`break`** statement or *s* is a compound statement (*i.e.*, a block) that contains a **`break`** statement or a **`continue`** statement. For example, this **`for`** statement will iterate 4 times and then terminate normally:

```
for (i = 0; i < 4; i++)
  if (i == 2) continue;
```

but this **`while`** statement will be an infinite loop:

```
i = 0;
while (i < 4)
{ if (i == 2) continue;
  i++;
}
```

4.7 The constant `PI` is not initialized. Every constant must be initialized at its declaration.

4.8 An infinite loop is one that continues without control; it can be stopped only by a branching statement within the loop (such as a **`break`** or **`goto`** statement) or by aborting the program (*e.g.*, with Ctrl+C). Infinite loops are useful if they are stopped with branching statements.

4.9 A loop can be terminated by a statement in the middle of its block by using a **`break`** or a **`goto`** statement.

4.10 Floating-point variables suffer from round-off error. After undergoing arithmetic transformations, exact values may not be what would be expected. So a test such as `(y == x)` may not work correctly.

Solutions to Problems

4.1 First, `x` is initialized to 4.15 and `i` is initialized to 0. Then `x` is doubled three times by the three iterations of the **`for`** loop.

4.2 The equivalent **`while`** loops are:

a. `while (e) s;`

b. `while (true) { s; e; }`, assuming that *s* contains no **`break`** or **`continue`** statements.

4.3 The equivalent **`while`** loop is:

```
int i=1;
while (i <= n)
{ cout << i*i << " ";
  i++;
}
```

4.4 The output is

```
1       1       3       9       5       9       7       7
```

4.5 The output is

```
End of program.
End of program.
3
End of program.
5
End of program.
End of program.
```

4.6 ***a.*** The 23 bits hold the 2nd through 24th bit of the mantissa. The first bit must be a 1, so it is not stored. Thus 24 bits are represented. These 24 bits can hold 2^{24} numbers. And 2^{24} = 16,777,216, which has 7 digits with full range, so 7 complete digits can be represented. But the last digit is in doubt because of rounding. Thus, the 32-bit `float` type yields 6 significant digits of precision.

b. The 8 bits that the 32-bit `float` type uses for its exponent can hold 2^8 = 256 different numbers. Two of these are reserved for indicating underflow and overflow, leaving 254 numbers for exponents. So an exponent can range from –126 to +127, yielding a magnitude range of $2^{-126} = 1.175494 \times 10^{-38}$ to $2^{127} = 1.70141 \times 10^{38}$.

4.7 This program uses a **`while`** loop to compute the sum of the first *n* squares, where *n* is input:

```
int main()
{ int n;
  cout << "Enter a positive integer: ";
  cin >> n;
  int sum=0, i=0;
  while (i++ < n)
    sum += i*i;
  cout << "The sum of the first " << n << " squares is "
       << sum << endl;
}
```

```
Enter a positive integer: 6
The sum of the first 6 squares is 91
```

4.8 This program uses a **`for`** loop to compute the sum of the first *n* squares, where *n* is input:

```
int main()
{ int n;
  cout << "Enter a positive integer: ";
  cin >> n;
  int sum=0;
  for (int i=1; i <= n; i++)
```

```
    sum += i*i;
  cout << "The sum of the first " << n << " squares is "
       << sum << endl;
}
```

```
Enter a positive integer: 6
The sum of the first 6 squares is 91
```

4.9 This program uses a **`do..while`** loop to compute the sum of the first *n* squares, where *n* is input:

```
int main()
{ int n;
  cout << "Enter a positive integer: ";
  cin >> n;
  int sum=0, i=1;
  do
  { sum += i*i;
  }
  while (i++ < n);
  cout << "The sum of the first " << n << " squares is "
       << sum << endl;
}
```

```
Enter a positive integer: 6
The sum of the first 6 squares is 91
```

4.10 This program directly implements the quotient operator `/` and the remainder operator `%` for the division of positive integers. The algorithm used here, applied to the fraction `n/d`, repeatedly subtracts the `d` from the `n` until `n` is less than `d`. At that point, the value of `n` will be the remainder, and the number `q` of iterations required to reach it will be the quotient:

```
int main()
{ int n, d, q, r;
  cout << "Enter numerator: ";
  cin >> n;
  cout << "Enter denominator: ";
  cin >> d;
  for (q = 0, r = n; r >= d; q++)
    r -= d;
  cout << n << " / " << d << " = " << q << endl;
  cout << n << " % " << d << " = " << r << endl;
  cout << "(" << q << ")(" << d << ") + (" << r << ") = "
       << n << endl;
}
```

```
Enter numerator: 30
Enter denominator: 7
30 / 7 = 4
30 % 7 = 2
(4)(7) + (2) = 30
```

This run iterated 4 times: 30 – 7 = 23, 23 – 7 = 16, 16 – 7 = 9, and 9 – 7 = 2. So the quotient is 4, and the remainder is 2. Note that this relationship must always be true for integer division:

```
(quotient)(denominator) + (remainder) = numerator
```

4.11 The trick here is to strip off the digits one at a time from the given integer and "accumulate" them in reverse in another integer:

```
int main()
{ long m, d, n = 0;
  cout << "Enter a positive integer: ";
  cin >> m;
```

```
  while (m > 0)
  { d = m % 10;    // d will be the right-most digit of m
    m /= 10;       // then remove that digit from m
    n = 10*n + d;  // and append that digit to n
  }
  cout << "The reverse is " << n << endl;
}
```

```
Enter a positive integer: 123456
The reverse is 654321
```

In this run, `m` begins with the value 123,456. In the first iteration of the loop, `d` is assigned the digit 6, `m` is reduced to 12,345, and `n` is increased to 6. On the second iteration, `d` is assigned the digit 5, `m` is reduced to 1,234, and `n` is increased to 65. On the third iteration, `d` is assigned the digit 4, `m` is reduced to 123, and `n` is increased to 654. This continues until, on the sixth iteration, `d` is assigned the digit 1, `m` is reduced to 0, and `n` is increased to 654,321.

4.12 This implements the Babylonian Algorithm:

```
#include <cmath>  // defines the fabs() function
#include <iostream>
using namespace std;
int main()
{ const double TOLERANCE = 5e-8;
  double x = 2.0;
  while (fabs(x*x - 2.0) > TOLERANCE)
  { cout << x << endl;
    x = (x + 2.0/x)/2.0;  // average of x and 2/x
  }
  cout << "x = " << x << ", x*x = " << x*x << endl;
}
```

```
2
1.5
1.41667
1.41422
x = 1.41421, x*x = 2
```

We use a "tolerance" of `5e-8` (= 0.00000005) to ensure accuracy to 7 decimal places. The `fabs()` function (for "floating-point absolute value"), defined in the `<cmath>` header file, returns the absolute value of the expression passed to it. So the loop continues until `x*x` is within the given tolerance of 2.

4.13 This program finds the integer square root of a given number. This method uses an "exhaustive" algorithm to find all the positive integers whose square is less than or equal to the given number:

```
int main()
{ float x;
  cout << "Enter a positive number: ";
  cin >> x;
  int n = 1;
  while (n*n <= x)
    ++n;
  cout << "The integer square root of " << x << " is "
       << n-1 << endl;
}
```

```
Enter a positive number: 1234.56
The integer square root of 1234.56 is 35
```

It starts with `n=1` and continues to increment `n` until `n*n > x`. When the **`for`** loop terminates, `n` is the smallest integer whose square is greater than `x`, so `n-1` is the integer square root of `x`. Note the use of the *null statement* in the **`for`** loop. Everything that needs to be done in the loop is done within the control parts of the loop. But the semicolon is still necessary at the end of the loop.

4.14 This implements the Euclidean Algorithm:

```
int main()
{ int m, n, r;
  cout << "Enter two positive integers: ";
  cin >> m >> n;
  if (m < n) { int temp = m; m = n; n = temp; }    // make m >= n
  cout << "The g.c.d. of " << m << " and " << n << " is ";
  while (n > 0)
  { r = m % n;
    m = n;
    n = r;
  }
  cout << m << endl;
}
```

```
Enter two positive integers: 532 112
The g.c.d. of 532 and 112 is 28
```

Chapter 5

Functions

5.1 INTRODUCTION

Most useful programs are much larger than the programs that we have considered so far. To make large programs manageable, programmers modularize them into subprograms. These subprograms are called functions. They can be compiled and tested separately and reused in different programs. This modularization is characteristic of successful object-oriented software.

5.2 STANDARD C++ LIBRARY FUNCTIONS

The *Standard C++ Library* is a collection of pre-defined functions and other program elements which are accessed through *header files*. We have used some of these already: the `INT_MAX` constant defined in `<climits>` (Example 2.3 on page 19), the `sqrt()` function defined in `<cmath>` (Example 2.15 on page 28), the `rand()` function defined in `<cstdlib>` (Example 4.26 on page 76), and the `time()` function defined in `<ctime>` (Example 4.28 on page 78). Our first example illustrates the use of one of these mathematical functions.

EXAMPLE 5.1 The Square Root Function `sqrt()`

The square root of a given positive number is the number whose square is the given number. The square root of 9 is 3 because the square of 3 is 9. We can think of the square root function as a "black box." When you put in a 9, out comes a 3. When the number 2 is input, the number 1.41421 is output. This function has the same input-process-output nature that complete programs have. However, the processing step is hidden: we do not need to know what the function does to 2 to produce 1.41421. All we need to know is that the output 1.41421 does have the square root property: its square is the input 2.

Here is a simple program that uses the predefined square root function:

```
#include <cmath>     // defines the sqrt() function
#include <iostream>  // defines the cout object
using namespace std;
int main()
{ // tests the sqrt() function:
  for (int x=0; x < 6; x++)
    cout << "\t" << x << "\t" << sqrt(x) << endl;
}
```

```
        0       0
        1       1
        2       1.41421
        3       1.73205
        4       2
        5       2.23607
```

This program prints the square roots of the numbers 0 through 5. Each time the expression `sqrt(x)` is evaluated in the `for` loop, the `sqrt()` function is executed. Its actual code is hidden away within the Standard C++ Library. In using it, we may confidently assume that the expression `sqrt(x)` will be

replaced by the actual square root of whatever value `x` has at that moment.

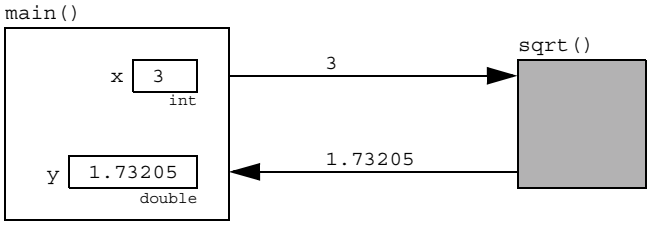
Notice the directive `#include <cmath>` on the first line of the program. This is necessary for the compiler to find the definition of the `sqrt()` function. It tells the compiler that the function is declared in the `<cmath>` header file.

A function like `sqrt()` is executed by using its name as a variable in a statement, like this:

```
y = sqrt(x);
```

This is called *invoking* or *calling* the function. Thus in Example 5.1, the code `sqrt(x)` *calls* the `sqrt()` function. The expression `x` in the parentheses is called the *argument* or *actual parameter* of the function call, and we say that it is *passed by value* to the function. So when `x` is 3, the value 3 is passed to the `sqrt()` function by the call `sqrt(x)`.

This process is illustrated by this diagram. The variables `x` and `y` are declared in `main()`. The value of `x` is passed to the `sqrt()` function which then returns the value 1.73205 back to `main()`. Note that the box representing the `sqrt()` function is shaded, indicating that its internal working mechanism is not visible.



EXAMPLE 5.2 Testing a Trigonometry Identity

Here is another program that uses the `<cmath>` header. Its purpose is to verify empirically the identity $\sin 2x = 2 \sin x \cos x$.

```
int main()
{ // tests the identity sin 2x = 2 sin x cos x:
  for (float x=0; x < 2; x += 0.2)
    cout << x << "\t\t" << sin(2*x) << "\t"
         << 2*sin(x)*cos(x) << endl;
}
```

```
0               0       0
0.2             0.389418        0.389418
0.4             0.717356        0.717356
0.6             0.932039        0.932039
0.8             0.999574        0.999574
1               0.909297        0.909297
1.2             0.675463        0.675463
1.4             0.334988        0.334988
1.6             -0.0583744      -0.0583744
1.8             -0.442521       -0.442521
```

.

The program prints x in the first column, $\sin 2x$ in the second column, and $2 \sin x \cos x$ in the third column. For each value of x tested, $\sin 2x = 2 \sin x \cos x$. Of course, this does not prove the identity, but it does provide convincing empirical evidence of its truth.

Note that `x` has type **`float`** instead of **`int`**. This allows the increment `x += 0.2` to work correctly.

Function values may be used like ordinary variables in an expression. Thus we can write

```
y = sqrt(2);
cout << 2*sin(x)*cos(x);
```

We can even "nest" function calls, like this:

```
y = sqrt(1 + 2*sqrt(3 + 4*sqrt(5)))
```

Most of the mathematical functions that you find on a pocket calculator are declared in the `<cmath>` header file, including all those shown in the table below.

Some Functions Defined in the `<cmath>` Header

Function	Description	Example
`acos(x)`	inverse cosine of `x` (in radians)	`acos(0.2)` returns 1.36944
`asin(x)`	inverse sine of `x` (in radians)	`asin(0.2)` returns 0.201358
`atan(x)`	inverse tangent of `x` (in radians)	`atan(0.2)` returns 0.197396
`ceil(x)`	ceiling of `x` (rounds up)	`ceil(3.141593)` returns 4.0
`cos(x)`	cosine of `x` (in radians)	`cos(2)` returns -0.416147
`exp(x)`	exponential of `x` (base e)	`exp(2)` returns 7.38906
`fabs(x)`	absolute value of `x`	`fabs(-2)` returns 2.0
`floor(x)`	floor of `x` (rounds down)	`floor(3.141593)` returns 3.0
`log(x)`	natural logarithm of `x` (base e)	`log(2)` returns 0.693147
`log10(x)`	common logarithm of `x` (base 10)	`log10(2)` returns 0.30103
`pow(x,p)`	`x` to the power `p`	`pow(2,3)` returns 8.0
`sin(x)`	sine of `x` (in radians)	`sin(2)` returns 0.909297
`sqrt(x)`	square root of `x`	`sqrt(2)` returns 1.41421
`tan(x)`	tangent of `x` (in radians)	`tan(2)` returns -2.18504

Notice that every mathematical function returns a `double` type. If an integer is passed to the function, it is promoted to a `double` before the function processes it.

The table below lists some of the more useful header files in the Standard C++ Library.

Some of the Header Files in the Standard C++ Library

Header File	Description
`<cassert>`	Defines the `assert()` function
`<ctype>`	Defines functions to test characters
`<cfloat>`	Defines constants relevant to floats
`<climits>`	Defines the integer limits on your local system
`<cmath>`	Defines mathematical functions
`<cstdio>`	Defines functions for standard input and output
`<cstdlib>`	Defines utility functions
`<cstring>`	Defines functions for processing strings
`<ctime>`	Defines time and date functions

These are derived from the Standard C Library. They are used the same way that Standard C++ header files such as `<iostream>` are used. For example, if you want to use the random number function `rand()` from the `<cstdlib>` header file, include the following preprocessor directive at the beginning of your main program file:

```
#include <cstdlib>
```

The Standard C Library is described in greater detail in Chapter 8 and in Appendix F.

5.3 USER-DEFINED FUNCTIONS

The great variety of functions provided by the Standard C++ Library is still not sufficient for most programming tasks. Programmers also need to be able to define their own functions.

EXAMPLE 5.3 A `cube()` Function

Here is a simple example of a user-defined function:

```
int cube(int x)
{ // returns cube of x:
  return x*x*x;
}
```

The function returns the cube of the integer passed to it. Thus the call `cube(2)` would return 8.

A user-defined function has two parts: its head and its body. The syntax for the *head* of a function is

return-type name`(`*parameter-list*`)`

This specifies for the compiler the function's *return type*, its *name*, and its *parameter list*. In Example 5.3, the function's return type is `int`, its name is `cube`, and its parameter list is `int x`. So its head is

```
int cube(int x)
```

The *body* of a function is the block of code that follows its head. It contains the code that performs the function's action, including the `return` statement that specifies the value that the function sends back to the place where it was called. The body of the cube function is

```
{ // returns cube of x:
  return x*x*x;
}
```

This is about as simple a body as a function could have. Usually the body is much larger. But the function's head typically fits on a single line.

Note that `main()` itself is a function. Its head is

```
int main()
```

and its body is the program itself. Its return type is `int`, its name is `main`, and its parameter list is empty.

A function's *return statement* serves two purposes: it terminates the execution of the function, and it returns a value to the calling program. Its syntax is

`return` *expression*`;`

where `expression` is any expression whose value could be assigned to a variable whose type is the same as the function's return type.

5.4 TEST DRIVERS

Whenever you create your own function, you should immediately test it with a simple program. Such a program is called a *test driver* for the function. Its only purpose is to test the function. It is a temporary, *ad hoc* program that should be "quick and dirty." That means that you need not include all the usual niceties such as user prompts, output labels, and documentation. Once you have used it to test your function thoroughly you can discard it.

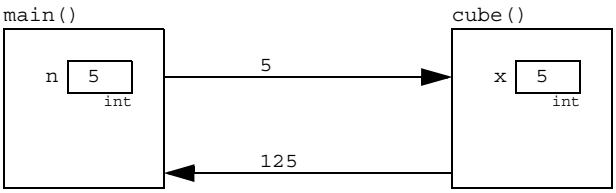
EXAMPLE 5.4 A Test Driver for the `cube()` Function

Here is a complete program that includes the definition of the `cube()` function from Example 5.4 together with a test driver for it:

```
int cube(int x)
{ // returns cube of x:
  return x*x*x;
}

int main()
{ // tests the cube() function:
  int n=1;
  while (n != 0)
  { cin >> n;
    cout << "\tcube(" << n << ") = " << cube(n) << endl;
  }
}
```

```
5
        cube(5) = 125
-6
        cube(-6) = -216
0
        cube(0) = 0
```

This reads integers and prints their cubes until the user inputs the sentinel value 0. Each integer read is passed to the `cube()` function by the call `cube(n)`. The value returned by the function replaces the expression `cube(n)` and then is passed to the output object `cout`.



We can visualize the relationship between the `main()` function and the `cube()` function like this:

The `main()` function passes the value 5 to the `cube()` function, and the `cube()` function returns the value 125 to the `main()` function. The argument `n` is passed by value to the formal parameter `x`. This simply means that `x` is assigned the value of `n` when the function is called.

Note that the `cube()` function is defined above the `main()` function in the example. This is because the C++ compiler must know about the `cube()` function before it is used in `main()`.

The next example shows a user-defined function named `max()` which returns the larger of the two `int`s passed to it. This function has two arguments.

EXAMPLE 5.5 A Test Driver for the `max()` Function

Here is a function with two parameters. It returns the larger of the two values passed to it.

```
int max(int x, int y)
{ // returns larger of the two given integers:
  if (x < y) return y;
  else return x;
}
```

```
int main()
{ // tests the max() function:
  int m, n;
  do
  { cin >> m >> n;
    cout << "\tmax(" << m << "," << n << ") = " << max(m,n) << endl;
  }
  while (m != 0);
}
```

```
5 8
        max(5,8) = 8
4 -3
        max(4,-3) = 4
0 0
        max(0,0) = 0
```

Notice that the function has more than one `return` statement. The first one that is reached terminates the function and returns the indicated value to the calling program.

A `return` statement is like a `break` statement. It is a jump statement that jumps out of the function that contains it. Although usually found at the end of the function, a `return` statement may be put anywhere that any other statement could appear within a function.

5.5 FUNCTION DECLARATIONS AND DEFINITIONS

The last two examples illustrate one method of defining a function in a program: the complete definition of the function is listed above the main program. This is the simplest arrangement and is good for test drivers.

Another, more common arrangement is to list only the function's header above the main program, and then list the function's complete definition (head and body) below the main program. This is illustrated in the next example.

In this arrangement, the function's declaration is separated from its definition. A function *declaration* is simply the function's head, followed by a semicolon. A function *definition* is the complete function: header and body. A function declaration is also called a function *prototype*.

A function declaration is like a variable declaration; its purpose is simply to provide the compiler with all the information it needs to compile the rest of the file. The compiler does not need to know how the function works (its body). It only needs to know the function's name, the number and types of its parameters, and its return type. This is precisely the information contained in the function's head.

Also like a variable declaration, a function declaration must appear above any use of the function's name. But the function definition, when listed separately from the declaration, may appear anywhere outside the `main()` function and is usually listed after it or in a separate file.

The variables that are listed in the function's parameter list are called *parameters*. They are local variables that exist only during the execution of the function. Their listing in the parameter list constitutes their declaration. In the example above, the parameters are `x` and `y`.

The variables that are listed in the function's calls are called the *arguments*. Like any other variable in the main program, they must be declared before they are used in the call. In the example above, the arguments are `m` and `n`.

In these examples, the arguments are *passed by value*. This means that their values are assigned to the function's corresponding parameters. So in the previous example, the value of `m` is assigned to `x` and the value of `n` is assigned to `y`. When passed by value, arguments may be constants or general expressions. For example, the `max()` function could be called by `max(44,5*m-n)`. This would assign 44 to `x` and the value of the expression `5*m-n` to `y`.

EXAMPLE 5.6 The `max()` Function with Declaration Separate from Definition

This program is the same test driver for the same `max()` function as in Example 5.6. But here the function's declaration appears above the main program and the function's definition follows it:

```
int max(int,int);
// returns larger of the two given integers:

int main()
{ // tests the max() function:
  int m, n;
  do
  { cin >> m >> n;
    cout << "\tmax(" << m << "," << n << ") = " << max(m,n) << endl;
  }
  while (m != 0);
}

int max(int x, int y)
{ if (x < y) return y;
  else return x;
}
```

Notice that the formal parameters `x` and `y` are listed in the header in the definition (as usual) but not in the declaration.

Function declarations are very similar to variable declarations, especially if the function has no parameters. For example, in a program that processes strings, you might need a variable named `length` to store the length of a string. But a reasonable alternative would be to have a function that computes the length of the string wherever it is needed, instead of storing and updating the value. The function would be declared as

```
int length();
```

whereas the variable would be declared as

```
int length;
```

The only difference is that the function declaration includes the parentheses `()`. In reality, the two alternatives are quite different, but syntactically they are nearly the same when they are used. In cases like this, one can regard a function as a kind of an "active variable;" *i.e.*, a variable that can do things.

EXAMPLE 5.7 SEPARATE COMPILATION

Function definitions are often compiled independently in separate files. For example, all the functions declared in the Standard C++ Library are compiled separately. One reason for separate compilation is "information hiding"—that is, information that is necessary for the complete

compilation of the program but not essential to the programmer's understanding of the program is hidden. Experience shows that information hiding facilitates the understanding and thus success of large software projects.

EXAMPLE 5.8 The `max()` Function Compiled Separately

This shows one way that the `max` function and its test driver could be compiled separately. The test driver is in a file named `test_max.cpp` and the function is in a separate file named `max.cpp`.

`test_max.cpp`

```
int max(int,int);
// returns larger of the two given integers:

int main()
{ // tests the max() function:
  int m, n;
  do
  { cin >> m >> n;
    cout << "\tmax(" << m << "," << n << ") = " << max(m,n) << endl;
  }
  while (m != 0);
}
```

`max.cpp`

```
int max(int x, int y)
{ if (x < y) return y;
  else return x;
}
```

The actual commands that you would use to compile these files together depend on the system you are using. In UNIX you could do it like this:

```
$ c++ -c max.c
$ c++ -c test_max.c
$ c++ -o test_max test_max.o max.o
$ test_max
```

(Here the dollar sign is the system prompt.) The first command compiles the `max` function, the second command compiles the test driver separately, the third command links them together to produce the executable module `test_max`, which is then run by the command on the fourth line.

One advantage of compiling functions separately is that they can be tested separately before the program(s) that call them are written. Once you know that the `max` function works properly, you can forget about how it works and save it as a "black box" ready to be used whenever it is needed. This is how the functions in the math library are used. It is the "off-the-shelf software" point of view.

Another advantage of separate compilation is the ease with which one module can be replaced by another equivalent module. For example, if you happen to discover a better way to compute the maximum of two integers, you can compile and test that function and then link that module with whatever programs were using the previous version of the `max()` function.

5.6 LOCAL VARIABLES AND FUNCTIONS

A *local variable* is simply a variable that is declared inside a block. It is accessible only from within that block. Since the body of a function itself is a block, variables declared within a function are local to that function; they exist only while the function is executing. A function's formal parameters (arguments) are also regarded as being local to the function.

The next two examples show functions with local variables.

EXAMPLE 5.9 The Factorial Function

The factorial numbers were introduced in Example 4.9 on page 65. The *factorial* of a positive integer n is the number $n!$ obtained by multiplying n by all the positive integers less than n:

$$n! = (n)(n-1)\cdots(3)(2)(1)$$

For example, $5! = (5)(4)(3)(2)(1) = 120$.

Here is an implementation of the factorial function:

```
long fact(int n)
{ // returns n! = n*(n-1)*(n-2)*...*(2)(1)
  if (n < 0) return 0;
  int f = 1;
  while (n > 1)
    f *= n--;
  return f;
}
```

This function has two *local variables*: `n` and `f`. The parameter `n` is local because it is declared in the function's parameter list. The variable `f` is local because it is declared within the body of the function.

Here is a test driver for the factorial function:

```
long fact(int);
// returns n! = n*(n-1)*(n-2)*...*(2)(1)

int main()
{ // tests the factorial() function:
  for (int i=-1; i < 6; i++)
    cout << " " << fact(i);
  cout << endl;
}
```

```
0 1 1 2 6 24 120
```

This program could be compiled separately, or it could be placed in the same file with the function and compiled together.

EXAMPLE 5.10 The Permutation Function

A *permutation* is an arrangement of elements taken from a finite set. The permutation function $P(n,k)$ gives the number of different permutations of any k items taken from a set of n items. One way to compute this function is by the formula

$$P(n, k) = \frac{n!}{(n-k)!}$$

For example,

$$P(5, 2) = \frac{5!}{(5-2)!} = \frac{5!}{3!} = \frac{120}{6} = 20$$

So there are 20 different permutations of 2 items taken from a set of 5. For example, here are the 20 different permutations of length 2 taken from the set {A, B, C, D, E}: AB, AC, AD, AE, BC, BD, BE, CD, CE, DE, BA, CA, DA, EA, CB, DB, EB, DC, EC, ED.

The code below implements this formula for the permutation function:

```
long perm(int n, int k)
{ // returns P(n,k), the number of permutations of k from n:
  if (n < 0 || k < 0 || k > n) return 0;
  return fact(n)/fact(n-k);
}
```

Notice that the condition `(n < 0 || k < 0 || k > n)` is used to handle the cases where either parameter is out of range. In these cases the function returns an "impossible" value, 0, to indicate that its input was erroneous. That value would then be recognized by the calling program as an "error flag."

Here is a test driver for the `perm()` function:

```
long perm(int,int);
// returns P(n,k), the number of permutations of k from n;

int main()
{ // tests the perm() function:
  for (int i = -1; i < 8; i++)
  { for (int j=-1; j <= i+1; j++)
      cout << " " << perm(i,j);
    cout << endl;
  }
}
```

```
0 0
0 1 0
0 1 1 0
0 1 2 2 0
0 1 3 6 6 0
0 1 4 12 24 24 0
0 1 5 20 60 120 120 0
0 1 6 30 120 360 720 720 0
0 1 7 42 210 840 2520 5040 5040 0
```

Note that the test driver checks the "exceptional cases where `i` < 0, `j` < 0, and `j` > `i`. Such values are called *boundary values* because they lie on the boundary of the output set (where `perm()` returns 0).

5.7 `void` FUNCTIONS

A function need not return a value. In other programming languages, such a function is called a *procedure* or a *subroutine*. In C++, such a function is identified simply by placing the keyword `void` where the function's return type would be.

A type specifies a set of values. For example, the type `short` specifies the set of integers from –32,768 to 32,767. The `void` type specifies the empty set. Consequently, no variable can be declared with `void` type. A `void` function is simply one that returns no value.

EXAMPLE 5.11 A Function that Prints Dates

```
void printDate(int,int,int);
// // prints the given date in literal form;
```

```
int main()
{ // tests the printDate() function:
  int month, day, year;
  do
  { cin >> month >> day >> year;
    printDate(month,day,year);
  }
  while (month > 0);
}

void printDate(int m, int d, int y)
{ // prints the given date in literal form:
  if (m < 1 || m > 12 || d < 1 || d > 31 || y < 0)
  { cerr << "Error: parameter out of range.\n";
    return;
  }
  switch (m)
  { case  1:  cout << "January ";   break;
    case  2:  cout << "February ";  break;
    case  3:  cout << "March ";     break;
    case  4:  cout << "April ";     break;
    case  5:  cout << "May ";       break;
    case  6:  cout << "June ";      break;
    case  7:  cout << "July ";      break;
    case  8:  cout << "August ";    break;
    case  9:  cout << "September "; break;
    case 10:  cout << "October ";   break;
    case 11:  cout << "November ";  break;
    case 12:  cout << "December ";  break;
  }
  cout << d << ", " << y << endl;
}
```

```
12 7 1941
December 7, 1941
5 16 1994
May 16, 1994
0 0 0
Error: parameter out of range.
```

The `printDate()` function returns no value. Its only purpose is to print the date. So its return type is `void`. The function uses a `switch` statement to print the month as a literal, and it prints the day and year as integers.

Note that the function returns without printing anything if the parameters are obviously out of range (*e.g.*, `m > 12` or `y < 0`). But impossible values such as `February 31, 1996` would be printed. Corrections for these anomalies are left as exercises.

Since a `void` function does not return a value, it need not include a `return` statement. If it does have a `return` statement, then it should appear simply as

```
return;
```

with no expression following the keyword `return`. In this case, the purpose of the `return` statement is simply to terminate the function.

A function with no return value is an action. Accordingly, it is usually best to use a verb phrase for its name. For example, the above function is named `printDate` instead of some noun phrase like `date`.

5.8 BOOLEAN FUNCTIONS

In some situations it is helpful to use a function to evaluate a condition, typically within an `if` statement or a `while` statement. Such functions are called *boolean functions* after the British logician George Boole (1815-1864) who developed boolean algebra.

EXAMPLE 5.12 Classifying Characters

The following program classifies the 128 *ASCII characters* (see Appendix A):

```
#include <cctype>  // defines the functions isdigit(), islower(), etc.
#include <iostream>  // defines the cout object
using namespace std;

void printCharCategory(char c);
// prints the category to which the given character belongs;

int main()
{ // tests the printCharCategory() function:
  for (int c=0; c < 128; c++)
    printCharCategory(c);
}

void printCharCategory(char c)
{ // prints the category to which the given character belongs:
  cout << "The character [" << c << "] is a ";
  if      (isdigit(c)) cout << "digit.\n";
  else if (islower(c)) cout << "lower-case letter.\n";
  else if (isupper(c)) cout << "capital letter.\n";
  else if (isspace(c)) cout << "white space character.\n";
  else if (iscntrl(c)) cout << "control character.\n";
  else if (ispunct(c)) cout << "punctuation mark.\n";
  else                 cout << "Error.\n";
}
```

The `void` function `printCharCategory()` calls the six boolean functions `isdigit()`, `islower()`, `isupper()`, `isspace()`, `iscntrl()`, and `ispunct()`. Each of these functions is predefined in the `<cctype>` header file. These functions are used to test objects' character type (*i.e.*, "c type").

Here is part of the output:

```
The character [ ] is a white space character.
The character [!] is a punctuation mark.
The character ["] is a punctuation mark.
The character [#] is a punctuation mark.
The character [$] is a punctuation mark.
```

The complete output contains 128 lines.

This example illustrates several new ideas. The main idea is the use of the boolean functions `isdigit()`, `islower()`, `isupper()`, `isspace()`, `iscntrl()`, and `ispunct()`. For example, the call `isspace(c)` tests the character `c` to determine whether it is a white space character. (There are six *white space characters*: the *horizontal tab character* `\t`, the *newline character* `\n`, the *vertical tab character* `\v`, the *form feed character* `\f`, the *carriage return character* `\r`, and the *space character*.) If `c` is any of these characters, then the function returns a nonzero integer for `true`; otherwise it returns 0 for `false`. Placing the call as the condition in the `if` statement causes the corresponding output statement to execute if and only if `c` is one of these characters.

Each character is tested within the `printCharCategory()` function. Although the program could have been written without this separate function, its use modularizes the program, making it more structured. We are conforming here to the general programming principle that recommends that every task be relegated to a separate function.

Functions such as `isdigit()` and `ispunct()` which are defined in the C header files (such as `<cctype>`) were originally defined for the C programming language. Since that language does not have a standard boolean type, those boolean functions return an integer instead of true or false. But since those C++ boolean values are stored as integers (see Section 2.2), the conversion from integer value to `bool` value is automatic.

EXAMPLE 5.13 A Function that Tests Primality

Here is a boolean function that determines whether an integer is a prime number:

```
bool isPrime(int n)
{ // returns true if n is prime, false otherwise:
  float sqrtn = sqrt(n);
  if (n < 2) return false;           // 0 and 1 are not primes
  if (n < 4) return true;            // 2 and 3 are the first primes
  if (n%2 == 0) return false;        // 2 is the only even prime
  for (int d=3; d <= sqrtn; d += 2)
    if (n%d == 0) return false;      // n has a nontrivial divisor
  return true;                       // n has no nontrivial divisors
}
```

This function works by looking for a divisor `d` of the given number `n`. It tests divisibility by the value of the condition `(n%d == 0)`. This will be true precisely when `d` is a divisor of `n`. In that case, `n` cannot be a prime number, so the function immediately returns `false`. If the `for` loop finishes without finding any divisors of `n`, then the function returns `true`.

We can stop searching for divisors once we get past the square root of `n` because if `n` is a product `d*a`, then one of these factors must be less than or equal to the square root of `n`. We define the `sqrtn` outside the loop so that it only has to be evaluated once.

It is also more efficient to check for even numbers `(n%2 == 2)` first. This way, once we get to the `for` loop, we need only check for odd divisors. This is done by incrementing the divider `d` by 2 on each iteration.

Here is a test driver and a test run for the `isPrime()` function:

```
#include <cmath>     // defines the sqrt() function
#include <iostream>  // defines the cout object
using namespace std;

bool isPrime(int);
// returns true if n is prime, false otherwise;
```

```
int main()
{ for (int n=0; n < 80; n++)
    if (isPrime(n)) cout << n << " ";
  cout << endl;
}
```

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79
```

Notice that, like the "c-type" functions in the previous example, a verb phrase is used for the name of this function. The name `isPrime` makes the function's use more readable for humans: the code

```
if (isPrime(n)) . . .
```

is almost the same as the ordinary English phrase "if *n* is prime..."

It should be noted that this function is not optimal. In searching for divisors, we need only check prime numbers, because every composite (non-prime) number is a unique product of primes. To modify this function so that it checks only prime divisors requires that the primes be stored as they are found. That requires using an array. (See Problem 6.22 on page 144.)

EXAMPLE 5.14 A Leap Year Function

A *leap year* is a year in which one extra day (February 29) is added to the regular calendar. Most of us know that the leap years are the years that are divisible by 4. For example, 1992 and 1996 are leap years. Most people, however, do not know that there is an exception to this rule: centennial years are not leap years. For example, 1800 and 1900 are not leap years. Furthermore, there is an exception to the exception: centennial years which are divisible by 400 are leap years. Thus, the year 2000 is a leap year.

Here is a boolean function that implements this definition:

```
bool isLeapYear(int y)
{ // returns true iff y is a leap year:
  return y % 4 == 0 && y % 100 != 0 || y % 400 == 0;
}
```

The compound condition `y % 4 == 0 && y % 100 != 0 || y % 400 == 0` will be true precisely when `y` is divisible by 4 but not by 100 unless it is also divisible by 400. In these cases the function returns `true`; in all other cases it returns `false`.

Here is a test driver and test run for the function:

```
bool isLeapYear(int);
// returns true iff y is a leap year;

int main()
{ // tests the isLeapYear() function:
  int n;
  do
  { cin >> n;
    if (isLeapYear(n)) cout << n << " is a leap year.\n";
    else cout << n << " is not a leap year.\n";
  }
  while (n > 1);
}
```

The output is

```
2000
2000 is a leap year.
2001
2001 is not a leap year.
0
0 is a leap year.
```

5.9 I/O FUNCTIONS

Functions are particularly useful for encapsulating tasks that require messy details that are not very relevant to the primary task of the program. For example, in processing personnel records, you might have a program that requires interactive input of a user's age. By relegating this task to a separate function, you can encapsulate the details needed to ensure correct data entry without distracting the main program.

We have already seen examples of output functions. The only purpose of the `printDate()` function in Example 5.11 on page 96 was to print the date represented by its input parameters. Instead of sending information back to the calling function, it sends its information to the standard output (*i.e.*, the computer screen). An input function like the one described above is analogous. Instead of receiving its information through its parameters, it reads it from standard input (*i.e.*, the keyboard).

The next example illustrates an input function. The `while (true)` control of the loop in this example makes it look like an infinite loop: the condition `(true)` is always true. But the loop is actually controlled by the `return` statement which not only terminates the loop but also terminates the function.

EXAMPLE 5.15 A Function for Reading the User's Age

Here is a simple function that prompts the user for his/her age and then returns it. It is "robust" in the sense that it rejects any unreasonable integer input. It repeatedly requests input until it receives an integer in the range 0 to 120:

```
int age()
{ // prompts the user to input his/her age, and returns that value:
  int n;
  while (true)
  { cout << "How old are you: ";
    cin >> n;
    if (n < 0) cout << "\a\tYour age could not be negative.";
    else if (n > 120) cout << "\a\tYou could not be over 120.";
    else return n;
    cout << "\n\tTry again.\n";
  }
}
```

As soon as the input received from `cin` is acceptable, the function terminates with a `return` statement, sending the input back to the calling function. If the input is not acceptable (either `n < 0` or `n > 120`), then the *system beep* is sounded by printing the character `'\a'` and a comment printed. Then the user is asked to "Try again."

Note that this is an example of a function whose `return` statement is not at the end of the function.

Here is a test driver and output from a sample run:

```
int age();
// prompts the user to input his/her age, and returns that value;

int main()
{ // tests the age() function:
  int a = age();
  cout << "\nYou are " << a << " years old.\n";
}
```

```
How old are you: 125
        You could not be over 120.
        Try again.
How old are you: -3
        Your age could not be negative.
        Try again.
How old are you: 99

You are 99 years old.
```

Notice that the function's parameter list is empty. But even though it has no input parameters, the parentheses `()` must be included both in the function's header and in every call to the function.

5.10 PASSING BY REFERENCE

Until now, all the parameters that we have seen in functions have been *passed by value.* That means that the expression used in the function call is evaluated first and then the resulting value is assigned to the corresponding parameter in the function's parameter list before the function begins executing. For example, in the call `cube(x)`, if `x` has the value 4, then the value 4 is passed to the local variable `n` before the function begins to execute its statements. Since the value 4 is used only locally inside the function, the variable `x` is unaffected by the function. Thus the variable `x` is a *read-only* parameter.

The pass-by-value mechanism allows for more general expressions to be used in place of an argument in the function call. For example the `cube()` function could also be called as `cube(3)`, or as `cube(2*x-3)`, or even as `cube(2*sqrt(x)-cube(3))`. In each case, the expression within the parentheses is evaluated to a single value and then that value is passed to the function.

The read-only, pass-by-value method of communication is usually what we usually want for functions. It makes the functions more self-contained, protecting them against accidental side effects. However, there are some situations where a function needs to change the value of the parameter passed to it. That can be done by passing it *by reference*.

To pass a parameter by reference instead of by value, simply append an ampersand, `&`, to the type specifier in the functions parameter list. This makes the local variable a reference to the argument passed to it. So the argument is *read-write* instead of read-only. Then any change to the local variable inside the function will cause the same change to the argument that was passed to it.

Note that parameters that are passed by value are called *value parameters*, and parameters that are passed by reference are called *reference parameters*.

EXAMPLE 5.16 The `swap()` Function

This little function is widely used in sorting data:

```
void swap(float& x, float& y)
{ // exchanges the values of x and y:
  float temp = x;
  x = y;
  y = temp;
}
```

Its sole purpose is to interchange the two objects that are passed to it. This is accomplished by declaring the formal parameters `x` and `y` as reference variables: `float& x, float& y`. The reference operator `&` makes `x` and `y` synonyms for the arguments passed to the function.

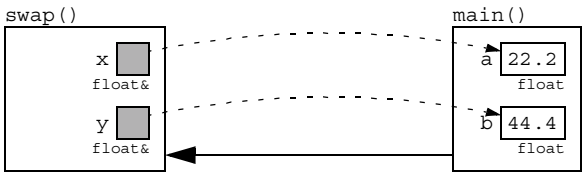
Here is a test driver and output from a sample run:

```
void swap(float&, float&);
// exchanges the values of x and y;

int main()
{ // tests the swap() function:
  float a = 22.2, b = 44.4;
  cout << "a = " << a << ", b = " << b << endl;
  swap(a,b);
  cout << "a = " << a << ", b = " << b << endl;
}
```

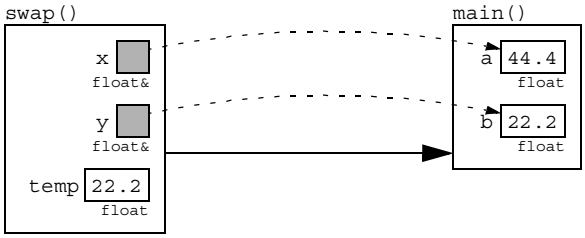
```
a = 22.2, b = 44.4
a = 44.4, b = 22.2
```

When the call `swap(a,b)` executes, the function creates its local references `x` and `y`, so that `x` is the function's local name for `a`, and `y` is the function's local name for `b`. Then the function's three statements execute: the local variable `temp` is declared and initialized with the value of `x` (which is `a`); then `x` (which is `a`) is assigned the value of `y` (which is `b`); then `y` (which is `b`) is assigned the value of `temp`. So `a` ends up with the value 44.4, and `b` ends up with the value 22.2:

Upon the call `swap(a,b)`:



Upon the return:



Note that the function declaration

```
void swap(float&, float&);
```

includes the reference operator `&` for each reference parameter, even though the parameters are omitted.

Some programmers write the reference operator `&` as a prefix to the parameter, like this:

```
void swap(float &x, float &y)
```

instead of as a suffix to its type as done here. That style is more common among C programmers. In C++, we think of `x` as the parameter and `float&` as its type. But the compiler will accept `float& x`, `float &x`, `float & x`, or even `float&x`. It's mostly a matter of taste.

EXAMPLE 5.17 Passing By Value and Passing By Reference

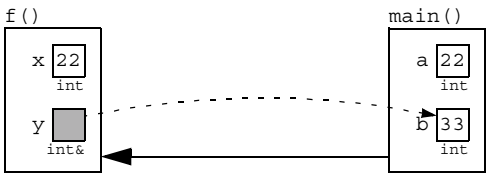
This example shows the difference between passing by value and passing by reference:

```
void f(int,int&);
// changes reference argument to 99:;

int main()
{ // tests the f() function:
  int a = 22, b = 44;
  cout << "a = " << a << ", b = " << b << endl;
  f(a,b);
  cout << "a = " << a << ", b = " << b << endl;
  f(2*a-3,b);
  cout << "a = " << a << ", b = " << b << endl;
}

void f(int x, int& y)
{ // changes reference argument to 99:
  x = 88;
  y = 99;
}
```

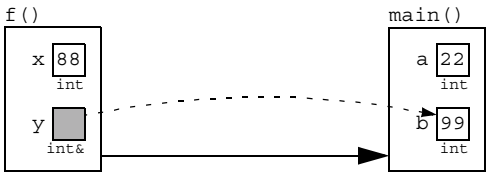
```
a = 22, b = 44
a = 22, b = 99
a = 22, b = 99
```

The call `f(a,b)` passes `a` by value to `x` and it passes `b` by reference to `y`. So `x` is a local variable that is assigned `a`'s value of 22, while `y` is an alias for the variable `b` whose value is 33. The function assigns 88 to `x`, but that has no effect on `a`. But when it assigns 99 to `y`, it is really assigning 99 to `b`, because `y` is an alias for `b`. So when the function terminates, `a` still has its original value 22, while `b` has the new value 99. The argument `a` is read-only, while the argument `b` is read-write.

Upon the call `f(a,b)`:



Upon the return:



The next table summarizes the differences between passing by value and passing by reference.

Passing By Value Versus Passing By Reference

Passing By Value	Passing By Reference
`int x;`	`int &x;`
The parameter x is a local variable.	The parameter `x` is a local reference.
It is a duplicate of the argument.	It is a synonym for the argument.
It cannot change the argument.	It can change the argument.
The argument passed by value may be a constant, a variable, or an expression.	The argument passed by reference must be a variable.
The argument is read-only.	The argument is read-write.

A common situation where reference parameters are needed is where the function has to return more than one value. It can only return one value directly with a `return` statement. So if more than one value must be returned, reference parameters can do the job.

EXAMPLE 5.18 Returning More than One Value

This function returns two values by using two reference parameters: the `area` and `circumference` of a circle whose radius has the given length `r`:

```
void computeCircle(double& area, double& circumference, double r)
{ // returns the area and circumference of a circle with radius r:
  const double PI = 3.141592653589793;
  area = PI*r*r;
  circumference = 2*PI*r;
}
```

Here is a test driver and output from a sample run:

```
void computeCircle(double&, double&, double);
// returns the area and circumference of a circle with radius r;

int main()
{ // tests the computeCircle() function:
  double r, a, c;
  cout << "Enter radius: ";
  cin >> r;
  computeCircle(a, c, r);
  cout << "area = " << a << ", circumference = " << c << endl;
}
```

```
Enter radius: 100
area = 31415.9, circumference = 628.319
```

Note that the output parameters `area` and `circumference` are listed first in the parameter list, to the left of the input parameter `r`. This standard C style is consistent with the format of assignment statements: `y = x`, where the information (the value) flows from the read-only variable `x` on the right to the read-write variable `y` on the left.

5.11 PASSING BY CONSTANT REFERENCE

There are two good reasons for passing a parameter by reference. If the function has to change the value of the argument, as the `swap()` function did, then it must be passed by reference. Also, if the argument that is passed to a function takes up a lot of storage space (for example, a one-megabyte graphics image), then it is more efficient to pass it by reference to prevent it from being duplicated. However, this also allows the function to change the value (*i.e.*, contents) of the argument. If you don't want the function to change its contents (for example, if the purpose of the function is to print the object), then passing by reference can be risky. Fortunately, C++ provides a third alternative: passing by *constant reference*. It works the same way as passing by reference, except that the function is prevented from changing the value of the parameter. The effect is that the function has access to the argument by means of its formal parameter alias, but the value of that formal parameter may not be changed during the execution of the function. A parameter that is passed by value is called "read-only" because it cannot write (*i.e.*, change) the contents of that parameter.

EXAMPLE 5.19 Passing By Constant Reference

This illustrates the three ways to pass a parameter to a function:

```
void f(int x, int& y, const int& z)
{ x += z;
  y += z;
  cout << "x = " << x << ", y = " << y << ", z = " << z << endl;
}
```

The first parameter `a` is passed by value, the second parameter `b` is passed by reference, and the third parameter `c` is passed by constant reference:

```
void f(int, int&, const int&);
int main()
{ // tests the f() function:
  int a = 22, b = 33, c = 44;
  cout << "a = " << a << ", b = " << b << ", c = " << c << endl;
  f(a,b,c);
  cout << "a = " << a << ", b = " << b << ", c = " << c << endl;
  f(2*a-3,b,c);
  cout << "a = " << a << ", b = " << b << ", c = " << c << endl;
}
```

```
a = 22, b = 33, c = 44
x = 66, y = 77, z = 44
a = 22, b = 77, c = 44
x = 85, y = 121, z = 44
a = 22, b = 121, c = 44
```

The function changes the formal parameters `x` and `y`, but it would not be able to change `z`. The function's change of `x` has no effect upon the argument `a` because it was passed by value. The function's change of `y` causes the same change on the argument `b` because it was passed by reference.

Passing parameters by constant reference is used mostly in functions that process large objects, such as arrays and class instances that are described in later chapters. Objects of fundamental types (integers, floats, *etc*.) are usually passed either by value (if you don't want the function to change them) or by reference (if you do want the function to change them).

5.12 INLINE FUNCTIONS

A function call involves substantial overhead. Extra time and space have to be used to invoke the function, pass parameters to it, allocate storage for its local variables, store the current variables and the location of execution in the main program, *etc.* In some cases, it is better to avoid all this by specifying the function to be `inline`. This tells the compiler to replace each call to the function with explicit code for the function. To the programmer, an inline function appears the same as an ordinary function, except for the use of the `inline` specifier.

EXAMPLE 5.20 Inlining the Cube Function

This is the same `cube()` function as in Example 5.3 on page 90:

```
inline int cube(int x)
{ // returns cube of x:
  return x*x*x;
}
```

The only difference is that the `inline` keyword has been added as a prefix to the function's head. This tells the compiler to replace the expression `cube(n)` in the main program with the actual code `(n)*(n)*(n)`. So this test program

```
int main()
{ // tests the cube() function:
  cout << cube(4) << endl;
  int x, y;
  cin >> x;
  y = cube(2*x-3);
}
```

will actually be compiled as though it were this program:

```
int main()
{ // tests the cube() function:
  cout << (4)*(4)*(4) << endl;
  int x, y;
  cin >> x;
  y = (2*X+3)*(2*X+3)*(2*X+3);
}
```

When the compiler replaces the `inline` function call with the function's actual code, we say that it *expands* the inline function.

The C++ Standard does not actually require the compiler to expand `inline` functions. It only "advises" the compiler to do so. A compiler that doesn't follow this "advice" could still be validated as a Standard C++ compiler. On the other hand, some Standard C++ compilers may expand some simple functions even if they are not declared to be `inline`.

Warning: use of inlined function can cause negative side effects. For example, inlining a 40-line function that is called in 26 different locations would add at least 1000 lines of unnoticed source code to your program. Inlined functions can also limit the portability of your code across platforms.

5.13 SCOPE

The scope of variable names was described in Section 3.5. The *scope* of a name consists of that part of the program where it can be used. It begins where the name is declared. If that declaration is inside a function (including the `main()` function), then the scope extends to the end of the innermost block that contains the declaration.

A program can have several objects with the same name if their scopes are nested or disjoint. This is illustrated by the next example, which is an elaboration of Example 3.7 on page 40.

EXAMPLE 5.21 Nested and Parallel Scopes

In this example, `f()` and `g()` are global functions, and the first `x` is a global variable. So their scope includes the entire file. This is called *file scope*. The second `x` is declared inside `main()` so it has *local scope*; *i.e.*, it is accessible only from within `main()`. The third `x` is declared inside an internal block, so its scope is restricted to that internal block. Each `x` scope overrides the scope of the previously declared `x`, so there is no ambiguity when the identifier x is referenced. The *scope resolution operator* `::` is used to access the last `x` whose scope was overridden; in this case, the global `x` whose value is 11:

```
void f();    // f() is global
void g();    // g() is global
int x = 11;  // this x is global

int main()
{ int x = 22;
  { int x = 33;
    cout << "In block inside main(): x = " << x << endl;
  }                                       // end scope of internal block
  cout << "In main(): x = " << x << endl;
  cout << "In main(): ::x = " << ::x << endl;    // accesses global x
  f();
  g();
}                                                 // end scope of main()

void f()
{ int x = 44;
  cout << "In f(): x = " << x << endl;
}                                                   // end scope of f()

void g()
{ cout << "In g(): x = " << x << endl;
}                                                   // end scope of g()
```

```
In block inside main(): x = 33
In main(): x = 22
In main(): ::x = 11
In f(): x = 44
In g(): x = 11
```

The `x` initialized with 44 has scope limited to the function `f()` which is parallel to `main()`; but its scope is also nested within the global scope of the first `x`, so its scope overrides that of both the first `x` within `f()`. In this example, the only place where the scope of the first `x` is not overridden is within the function `g()`.

5.14 OVERLOADING

C++ allows you to use the same name for different functions. As long as they have different parameter type lists, the compiler will regard them as different functions. To be distinguished, the parameter lists must either contain a different number of parameters, or there must be at least one position in their parameter lists where the types are different.

EXAMPLE 5.22 Overloading the `max()` Function

Example 5.6 on page 93 defined a `max()` function for two integers. Here we define two other `max()` functions in the same program:

```
int max(int, int);
int max(int, int, int);

int main()
{ cout << max(99,77) << " " << max(55,66,33);
}

int max(int x, int y)
{ // returns the maximum of the two given integers:
  return (x > y ? x : y);
}

int max(int x, int y, int z)
{ // returns the maximum of the three given integers:
  int m = (x > y ? x : y);  // m = max(x,y)
  return (z > m ? z : m);
}
```

```
99 66
```

Three different functions, all named `max`, are defined here. The compiler checks their parameter lists to determine which one to use on each call. For example, the first call passes two `int`s, so the version that has two `int`s in its parameter list is called. (If that version had been omitted, then the system would promote the two ints 99 and 77 to the doubles 99.0 and 77.0 and then pass them to the version that has two doubles in its parameter list.)

Overloaded functions are widely used in C++. Their value will become more apparent with the use of classes in Chapter 12.

5.15 THE `main()` FUNCTION

Every C++ program requires a function named `main()`. In fact, we can think of the complete program itself as being made up of the `main()` function together with all the other functions that are called either directly or indirectly from it. The program starts by calling `main()`.

Since `main()` is a function with return type `int`, it is normal to end its block with

```
return 0;
```

although most compilers do not require this. Some compilers allow it to be omitted but will issue a warning when it is. The value of the integer that is returned to the operating system should be the number of errors counted; the value 0 is the default.

The `return` statement in `main()` can be used to terminate the program abnormally, as the next example illustrates.

EXAMPLE 5.23 Using the `return` Statement to Terminate a Program

```
int main()
{ // prints the quotient of two input integers:
  int n, d;
  cout << "Enter two integers: ";
  cin >> n >> d;
  if (d == 0) return 0;
  cout << n << "/" << d << " = " << n/d << endl;
}
```

```
Enter two integers: 99 17
99/17 = 5
```

If the user inputs 0 for d, the program will terminate without output:

```
Enter two integers: 99 0
```

In any function, the `return` statement will terminate the current function and return control to the invoking function. That's why a `return` statement in `main()` terminates the program. There are actually four ways to terminate a program abnormally (*i.e.*, before execution reaches the end of the main block):

1. use a `return` statement in `main()`;
2. call the `exit()` function;
3. call the `abort()` function;
4. throw an uncaught exception.

The `exit()` and `abort()` functions are described in Appendix F.

The `exit()` function is defined in the `<cstdlib>` header. It is useful for terminating a program from within a function other than `main()`. This is illustrated by the next example.

EXAMPLE 5.24 Using the `exit()` Function to Terminate a Program

```
#include <cstdlib>   // defines the exit() function
#include <iostream>  // defines the cin and cout objects
using namespace std;
double reciprocal(double x);

int main()
{ double x;
  cin >> x;
  cout << reciprocal(x);
}

double reciprocal(double x)
{ // returns the reciprocal of x:
  if (x == 0) exit(1);  // terminate the program
  return 1.0/x;
}
```

If the user enters 0 for `x`, the program will terminate from within the `reciprocal()` function without attempting to divide by it.

5.16 DEFAULT ARGUMENTS

In C++ the number of arguments that a function has can vary during run-time. This is done by providing default values for the optional arguments.

EXAMPLE 5.25 Default Parameters

This function evaluates the third degree polynomial $a_0 + a_1x + a_2x^2 + a_3x^3$. The actual evaluation is done using Horner's Algorithm, grouping the calculations as $a_0 + (a_1 + (a_2 + a_3x)x)x$ for greater efficiency:

```
double p(double, double, double=0, double=0, double=0);

int main()
{ // tests the p() function:
  double x = 2.0003;
  cout << "p(x,7) = " << p(x,7) << endl;
  cout << "p(x,7,6) = " << p(x,7,6) << endl;
  cout << "p(x,7,6,5) = " << p(x,7,6,5) << endl;
  cout << "p(x,7,6,5,4) = " << p(x,7,6,5,4) << endl;
}
double p(double x, double a0, double a1, double a2, double a3)
{ // returns a0 + a1*x + a2*x^2 + a3*x^3:
  return a0 + (a1 + (a2 + a3*x)*x)*x;
}
```

```
p(x,7) = 7
p(x,7,6) = 19.0018
p(x,7,6,5) = 39.0078
p(x,7,6,5,4) = 71.0222
```

The call `p(x,a0,a1,a2,a3)` evaluates the third-degree polynomial $a_0 + a_1x + a_2x^2 + a_3x^3$. But since `a1`, `a2`, and `a3` all have the default value 0, the function can also be called by `p(x,a0)` to evaluate the constant polynomial a_0, or by `p(x,a0,a1)` to evaluate the first-degree polynomial $a_0 + a_1x$, or by `p(x,a0,a1,a2)` to evaluate the second-degree polynomial $a_0 + a_1x + a_2x^2$.

Note how the default values of 0 are given in the function prototype. For example, the call `p(x,7,6,5)`, which is equivalent to the call `p(x,7,6,5,0)`, evaluates the second degree polynomial $7 + 6x + 5x^2$.

In the example above, the function may be called with 2, 3, 4, or 5 arguments. So the effect of allowing default parameter values is really to allow a variable number of arguments passed to the function.

If a function has default parameter values, then the function's parameter list must show all the parameters that have default values to the right of those that don't, like this:

```
void f(int a, int b, int=4, int=7, int=3);  // OK
void g(int a, int=2, int=4, int, int=3);  // ERROR
```

In other words, all "optional" parameters must be listed last.

Review Questions

5.1 What are the advantages of using functions to modularize a program?
5.2 What is the difference between a function's declaration and its definition?
5.3 Where can the declaration of a function be placed?
5.4 When does a function need an `include` directive?
5.5 What is the advantage of putting a function's definition in a separate file?
5.6 What is the advantage of compiling a function separately?
5.7 What are the differences between passing a parameter by value and by reference?
5.8 What are the differences between passing a parameter by reference and by constant reference?
5.9 Why is a parameter that is passed by value referred to as "read-only"? Why is a parameter that is passed by reference referred to as "read-write"?
5.10 What is wrong with the following declaration:

```
int f(int a, int b=0, int c);
```

Problems

5.1 In Example 5.14, the following expression was used to test whether `y` is a leap year:

```
y % 4 == 0 && y % 100 != 0 || y % 400 == 0
```

This expression is not the most efficient form. If `y` is not divisible by 4, it will still test the condition `y % 400 == 0` which would have to be false. C++ implements "short circuiting," which means that subsequent parts of a compound condition are tested only when necessary. Find an equivalent compound condition that is more efficient due to short circuiting.

5.2 Describe how a `void` function with one reference parameter can be converted into an equivalent non-`void` function with one value parameter.

5.3 Write a simple program like the one in Example 5.2 on page 88 to check the trigonometry $\cos 2x = 2\cos^2 x - 1$.

5.4 Write a program like the one in Example 5.2 that checks the identity: $\cos^2 x + \sin^2 x = 1$.

5.5 Write a program like the one in Example 5.2 that checks the identity: $b^x = e^{(x \log b)}$.

5.6 Write and test the following `min` function that returns the smallest of four given integers:

```
int min(int,int,int,int);
```

5.7 Write and test the following `max()` function that uses the `max(int,int)` function from Example 5.5 on page 91 to find and return the largest of four given integers:

```
int max(int,int,int);
```

5.8 Write and test the following `min()` function that uses a `min(int,int)` function to find and return the smallest of four given integers:

```
int min(int,int,int,int);
```

5.9 Write and test the following `average()` function that returns the average of four numbers:

```
float average(float x1, float x2, float x3, float x4)
```

5.10 Write and test the following `average()` function that returns the average of up to four positive numbers:

```
float average(float x1, float x2=0, float x3=0, float x4=0)
```

5.11 Implement the factorial function `fact()` with a `for` loop. (See Example 4.9 on page 65.) Determine which values of `n` will cause `fact(n)` to overflow.

5.12 A more efficient way to compute the permutations function $P(n,k)$ is by the formula

$$P(n,k) = (n)(n-1)(n-2)\cdots(n-k+2)(n-k+1)$$

This means the product of the k integers from n down to $n - k + 1$. Use this formula to rewrite and test the `perm()` function from Example 5.10.

5.13 The *combination function* $C(n,k)$ gives the number of different (unordered) k-element subsets that can be found in a given set of n elements. The function can be computed from the formula

$$C(n, k) = \frac{n!}{k!(n-k)!}$$

Implement this formula.

5.14 The combinations function $C(n,k)$ can be computed from the formula

$$C(n, k) = \frac{P(n, k)}{k!}$$

Use this formula to rewrite and test the `comb()` function implemented in Problem 5.13.

5.15 A more efficient way to compute $C(n,k)$ is shown by the formula

$C(n,k) = (((((((n/1)(n–1))/2)(n–2))/3)\cdots(n–k+2))/(k–1))(n–k+1))/k$

This alternates divisions and multiplications, each time multiplying by the next decremented value from `n` and then dividing by the next incremented value from 1. Use this formula to rewrite and test the `comb()` function implemented in Problem 5.13. Hint: Use a `for` loop like the one in Problem 5.12.

5.16 *Pascal's Triangle* is a triangular array of numbers that begins like this:

```
                1
              1   1
            1   2   1
          1   3   3   1
        1   4   6   4   1
      1   5  10  10   5   1
    1   6  15  20  15   6   1
  1   7  21  35  35  21   7   1
1   8  28  56  70  56  28   8   1
```

Each number in Pascal's Triangle is one of the combinations $C(n,k)$. (See Problem 5.13.) If we count the rows and the diagonal columns starting with 0, then the number $C(n,k)$ is in row n and column k. For example, the number $C(6,2) = 15$ is in row number 6 and column number 2. Write a program that uses the `comb()` function to print Pascal's Triangle down to row number 12.

5.17 Write and test the `digit()` function:

```
int digit(int n, int k)
```

This function returns the `k`th digit of the positive integer `n`. For example, if `n` is the integer 29,415, then the call `digit(n, 0)` would return the digit 5, and the call `digit(n, 2)` would return the digit 4. Note that the digits are numbered from right to left beginning with the "zeroth digit."

5.18 Write and test a function that implements the *Euclidean Algorithm* to return the greatest common divisor of two given positive integers. See Problem 4.14 on page 67.

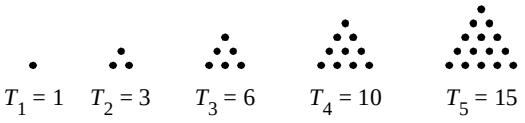
5.19 Write and test a function that uses the greatest common divisor function (Problem 5.18) to return the *least common multiple* of two given positive integers.

5.20 Write and test the following `power()` function that returns `x` raised to the power `n`, where `n` can be any integer:

```
double power(double x, int p);
```

Use the algorithm that would compute x^{20} by multiplying 1 by x 20 times.

5.21 The ancient Greeks classified numbers geometrically. For example, a number was called "triangular" if that number of pebbles could be arranged in a symmetric triangle. The first ten triangular numbers are 0, 1, 3, 6, 10, 15, 21, 28, 36, and 45. Write and test the boolean function:




```
int isTriangular(int n)
```

This function returns 1 if the given integer `n` is a triangular number, and 0 otherwise.

5.22 Write and test the following `isSquare()` function that determines whether the given integer is a square number:

```
int isSquare(int n)
```

The first ten square numbers are 0, 1, 4, 9, 16, 25, 36, 49, 64, and 81.

5.23 Write and test the following `isPentagonal()` function that determines whether the given integer is a pentagonal number:

```
int isPentagonal(int n)
```

The first ten pentagonal numbers are 0, 1, 5, 12, 22, 35, 51, 70, 92, and 117.

5.24 Write and test the following `computeCircle()` function that returns the area `a` and the circumference `c` of a circle with given radius `r`:

```
void computeCircle(float& a, float& c, float r).
```

5.25 Write and test the following `computeTriangle()` function that returns the area `a` and the perimeter `p` of a triangle with given side lengths `a`, `b`, and `c`:

```
void computeTriangle(float& a, float& p, float a, float b, float
c)
```

5.26 Write and test the following `computeSphere()` function that returns the volume `v` and the surface area `s` of a sphere with given radius `r`:

```
void computeSphere(float& v, float& s, float r).
```

Answers to Review Questions

5.1 A separately compiled function can be regarded as an independent "black box" which performs a specific task. Once the function has been thoroughly tested, the programmer need not be concerned about how it works. This frees the programmer to concentrate on the development of the main program. Moreover, if a better way of implementing the function is found later, it can replace the previous version without affecting the main program.

5.2 A function's declaration (also called its *prototype*) is essentially only the function's header. A function's definition is the complete function: header and body block. The declaration provides only the information needed to call the function: its name, its parameter types, and its return type; it is the *interface* between the function and its caller. The definition gives all the information about the function, including the details of how it works; it is the function's *implementation*.

5.3 A function may be declared anywhere as long as its declaration is above all references to the function. So the declaration must come before any calls to it, and if its definition is separate then it too must come after its declaration.

5.4 An `include` directive is used to include other files. Typically, function declarations and/or definitions are listed in a separate "header" file (with `.h` file extension). If only the declarations are in the header file, then the definitions would be compiled separately in other files.

5.5 The advantage of putting a function's definition in a separate header file is that it doesn't have to be brought into the editor when changes are made to the functions that call it.

5.6 The advantage of compiling a function separately is that it does not need to be recompiled when the functions that call it are recompiled.

5.7 A parameter passed by value is duplicated by its corresponding argument. A parameter passed by reference is simply renamed by its corresponding argument.

5.8 A parameter passed by constant reference cannot be changed by the function to which it is passed.

5.9 A parameter that is passed by value cannot be changed (rewritten).

5.10 The function has a default value for a parameter (`b`) that precedes a parameter (`c`) that has no default value. This violates the requirement that all default parameters be listed after all the other parameters in the function's parameter list.

Solutions to Problems

5.1 The compound condition

```
y%4 == 0 && (y % 100 != 0 || y % 400 == 0)
```

is equivalent and more efficient. The two can be seen to be equivalent by checking their values in the four possibilities, represented by the four `y` values 1995, 1996, 1900, and 2000. This condition is more efficient because if `y` is not divisible by 4 (the most likely case), then it will not test `y` further.

5.2 Convert the reference parameter into a return value. For example, the function

```
void f(int& n)
{ n *= 2;
}
```

is equivalent to the function

```
int g(int n)
{ return 2*n;
}
```

The two functions are invoked differently:

```
int x=22, y=44;
f(x);      // double the value of x
y = g(y);  // double the value of y
```

But in both cases, the effect is to double the value of the parameter.

5.3 This is similar to Example 5.2:

```
int main()
{ for (float x=0; x < 1; x += 0.1)
    cout << cos(2*x) << '\t' << 2*cos(x)*cos(x) - 1 << endl;
}
```

```
1       1
0.980067        0.980067
0.921061        0.921061
0.825336        0.825336
0.696707        0.696707
0.540302        0.540302
0.362358        0.362358
0.169967        0.169967
-0.0291997      -0.0291997
-0.227202       -0.227202
```

The equal values show that the identity is true for the 10 values of `x` tested.

5.4 This is similar to Example 5.2:

```
int main()
{ for (double x=0; x < 2; x += 0.2)
  { double s=sin(x);
    double c=cos(x);
    cout << s*s << "\t" << c*c << "\t" << s*s+c*c << endl;
  }
}
```

```
0       1       1
0.0394695       0.96053 1
0.151647        0.848353        1
0.318821        0.681179        1
0.5146  0.4854  1
0.708073        0.291927        1
0.868697        0.131303        1
0.971111        0.0288888       1
0.999147        0.000852612     1
0.948379        0.0516208       1
0.826822        0.173178        1
```

5.5 This is similar to Example 5.2:

```
int main()
{ double b=2;
  double lg2=log(2);
  for (double x=0; x < 2; x += 0.2)
    cout << pow(b,x) << "\t" << exp(x*lg2) << endl;
}
```

```
1       1
1.1487  1.1487
1.31951 1.31951
1.51572 1.51572
1.7411  1.7411
2       2
2.2974  2.2974
2.63902 2.63902
3.03143 3.03143
3.4822  3.4822
4       4
```

5.6 This tests a function that returns the minimum of four integers:

```
int min(int,int,int,int);
int main()
{ cout << "Enter four integers: ";
  int w, x, y, z;
  cin >> w >> x >> y >> z;
  cout << "Their minimum is " << min(w,x,y,z) << endl;
}
int min(int n1, int n2, int n3, int n4)
{ int min=n1;
  if (n2 < min) min = n2;
  if (n3 < min) min = n3;
  if (n4 < min) min = n4;
  return min;
}
```

```
Enter four integers: 44 88 22 66
Their minimum is 22
```

5.7 This tests a function that returns the maximum of three integers:

```
int max(int,int,int);
int main()
{ cout << "Enter three integers: ";
  int x, y, z;
  cin >> x >> y >> z;
  cout << "Their maximum is " << max(x,y,z) << endl;
}
```

```
int max(int, int);
int max(int x, int y, int z)
{ int max(int,int);
  return max(max(x,y),z);
}
int max(int x, int y)
{ // returns the maximum of the two given integers:
  if (x < y) return y;
  else return x;
}
```

```
Enter three integers: 44 88 66
Their maximum is 88
```

5.8 This tests a function that returns the minimum of four integers:

```
int min(int,int,int,int);
int main()
{ cout << "Enter four integers: ";
  int w, x, y, z;
  cin >> w >> x >> y >> z;
  cout << "Their minimum is " << min(w,x,y,z) << endl;
}
int min(int,int);
int min(int n1, int n2, int n3, int n4)
{ int m12=min(n1,n2);
  int m34=min(n3,n4);
  return (m12 < m34 ? m12 : m34);
}
int min(int m, int n)
{ return (m < n ? m : n);
}
```

```
Enter four integers: 44 88 22 66
Their minimum is 22
```

5.9 This tests a function that returns the average of four numbers:

```
double ave(double,double,double,double);
double main()
{ cout << "Enter four numbers: ";
  double w, x, y, z;
  cin >> w >> x >> y >> z;
  cout << "Their average is " << ave(w,x,y,z) << endl;
}
double ave(double x1, double x2, double x3, double x4)
{ return (x1 + x2 + x3 + x4)/4.0;
}
```

```
Enter four numbers: 44 88 22 66
Their average is 55
```

5.10 This tests a function that returns the average of four or fewer numbers:

```
double ave(double,double=0,double=0,double=0);
double main()
{ cout << "Enter four non-zero numbers: ";
  double w, x, y, z;
  cin >> w >> x >> y >> z;
  cout << "The average of the first one is " << ave(w) << endl;
```

```
  cout << "The average of the first two is " << ave(w,x) << endl;
  cout << "The average of the first three is " << ave(w,x,y)<<endl;
  cout << "The average of all four is " << ave(w,x,y,z) << endl;
}
double ave(double x1, double x2, double x3, double x4)
{ double sum = x1 + x2 + x3 + x4;
  if (x2 == 0) return sum;
  if (x3 == 0) return sum/2.0;
  if (x4 == 0) return sum/3.0;
  return sum/4.0;
}
```

```
Enter four non-zero numbers: 44 88 22 66
The average of the first one is 44
The average of the first two is 66
The average of the first three is 51.3333
The average of all four is 55
```

5.11 This tests the factorial function:

```
long fact(int n);
int main()
{ for (int i=-1; i<20; i++)
    cout << "fact(" << i << ") = " << fact(i) << endl;
}
long fact(int n)
{ if (n < 2) return 1;
  long f=1;
  for (int i=2; i <= n; i++)
    f *= i;
  return f;
}
```

```
fact(-1) = 1
fact(0) = 1
fact(1) = 1
fact(2) = 2
fact(3) = 6
fact(4) = 24
fact(5) = 120
fact(6) = 720
fact(7) = 5040
fact(8) = 40320
fact(9) = 362880
fact(10) = 3628800
fact(11) = 39916800
fact(12) = 479001600
fact(13) = 1932053504
fact(14) = 1278945280
fact(15) = 2004310016
fact(16) = 2004189184
fact(17) = -288522240
fact(18) = -898433024
fact(19) = 109641728
```

This overflows when `n` = 13 on machines that implement the `long` type with 32-bits.

5.12 This tests the permutation function:

```
long perm(int n, int k);
int main()
{ for (int i = -1; i < 6; i++)
  { for (int j = -1; j <= i+1; j++)
      cout << " " << perm(i,j);
    cout << endl;
  }
}
long perm(int n, int k)
{ if (n < 0 || k < 0 || k > n) return 0;
  int p = 1;
  for (int i = 1; i <= k; i++, n--)
    p *= n;
  return p;
}
```

```
0 0
0 1 0
0 1 1 0
0 1 2 2 0
0 1 3 6 6 0
0 1 4 12 24 24 0
0 1 5 20 60 120 120 0
```

5.13 This tests the combination function:

```
long comb(int n, int k);
int main()
{ for (int i = -1; i < 6; i++)
  { for (int j = -1; j <= i+1; j++)
      cout << " " << comb(i,j);
    cout << endl;
  }
}
long fact(int n);
long comb(int n, int k)
{ if (n < 0 || k < 0 || k > n) return 0;
  return fact(n)/(fact(k)*fact(n-k));
}
long fact(int n)
{ if (n < 2) return 1;
  long f=1;
  for (int i=2; i <= n; i++)
    f *= i;
  return f;
}
```

```
0 0
0 1 0
0 1 1 0
0 1 2 1 0
0 1 3 3 1 0
0 1 4 6 4 1 0
0 1 5 10 10 5 1 0
```

Note that the `fact()` function must be declared above the `comb()` function because it is used by `comb()`. But it does not need to be declared above `main()` because it is not used there.

5.14 This tests the combination function:

```
long comb(int n, int k);
int main()
{ for (int i = -1; i < 9; i++)
  { for (int j = -1; j <= i+1; j++)
      cout << " " << comb(i,j);
    cout << endl;
  }
}
long perm(int,int);
long fact(int);
long comb(int n, int k)
{ if (n < 0 || k < 0 || k > n) return 0;
  return perm(n,k)/fact(k);
}
long perm(int n, int k)
{ if (n < 0 || k < 0 || k > n) return 0;
  int p = 1;
  for (int i = 1; i <= k; i++, n--)
    p *= n;
  return p;
}
long fact(int n)
{ if (n < 2) return 1;
  long f=1;
  for (int i=2; i <= n; i++)
    f *= i;
  return f;
}
```

The output is the same as for Problem 5.13.

5.15 This tests the combination function:

```
long comb(int n, int k);
int main()
{ for (int i = -1; i < 9; i++)
  { for (int j = -1; j <= i+1; j++)
      cout << " " << comb(i,j);
    cout << endl;
  }
}
long comb(int n, int k)
{ if (n < 0 || k < 0 || k > n) return 0;
  long c = 1;
  for (int i=1; i<=k; i++, n--)
    c = c*n/i;
  return c;
}
```

The output is the same as for Problem 5.13.

5.16 This prints Pascal's Triangle:

```
long comb(int n, int k);
int main()
{ const m = 13;
  for (int i = 0; i < m; i++)
```

```
  { for (int j = 1; j < m-i; j++)
      cout << setw(2) << "";  // print whitespace
    for (int j = 0; j <= i; j++)
      cout << setw(4) << comb(i,j);
    cout << endl;
  }
}
long comb(int n, int k)
{ if (n < 0 || k < 0 || k > n) return 0;
  long c = 1;
  for (int i=1; i<=k; i++, n--)
    c = c*n/i;
  return c;
}
```

```
                          1
                        1   1
                      1   2   1
                    1   3   3   1
                  1   4   6   4   1
                1   5  10  10   5   1
              1   6  15  20  15   6   1
            1   7  21  35  35  21   7   1
          1   8  28  56  70  56  28   8   1
        1   9  36  84 126 126  84  36   9   1
      1  10  45 120 210 252 210 120  45  10   1
    1  11  55 165 330 462 462 330 165  55  11   1
  1  12  66 220 495 792 924 792 495 220  66  12   1
```

5.17 This tests a function that extracts a digit from an integer:

```
int digit(long,int);
int main()
{ int n, k;
  cout << "Integer: ";
  cin >> n;
  do
  { cout << "Digit: ";
    cin >> k;
    cout << "Digit number " << k << " of " << n
         << " is " << digit(n, k) << endl;
  } while (k > 0);
}
int digit(long n, int k)
{ for (int i = 0; i < k; i++)
    n /= 10;  // remove right-most digit
  return n % 10;
}
```

```
Integer: 876543210
Digit: 4
Digit number 4 of 876543210 is 4
Digit: 7
Digit number 7 of 876543210 is 7
Digit: 0
Digit number 0 of 876543210 is 0
```

5.18 This tests the greatest common divisor function:

```
long gcd(long,long);
int main()
{ int m, n;
  cout << "Enter two positive integers: ";
  cin >> m >> n;
  cout << "gcd(" << m << "," << n << ") = " << gcd(m,n) << endl;
}
long gcd(long m, long n)
{ // returns the greatest common divisor of m and n:
  if (m<n) swap(m,n);
  assert(n >= 0);
  while (n>0)
  { long r=m%n;
    m = n;
    n = r;
  }
  return m;
}
```

```
Enter two positive integers: 144 192
gcd(144,192) = 48
```

5.19 This tests the least common multiple function:

```
long lcm(long,long);
int main()
{ int m, n;
  cout << "Enter two positive integers: ";
  cin >> m >> n;
  cout << "lcm(" << m << "," << n << ") = " << lcm(m,n) << endl;
}
long gcd(long,long);
long lcm(long m,long n)
{ return m*n/gcd(m,n);
}
long gcd(long m,long n)
{ if (m < n) swap(m,n);
  while (n>0)
  { int r = m%n;
    m = n;
    n = r;
  }
  return m;
}
```

```
Enter two positive integers: 144 192
lcm(144,192) = 576
```

5.20 This tests the power function:

```
double pow(double,int);
int main()
{ cout << "Enter a positive float x and an integer n: ";
  double x;
  int n;
  cin >> x >> n;
```

```
  cout << "pow(" << x << "," << n << ") = " << pow(x,n) << endl;
}
double pow(double x, int n)
{ if (x == 0) return 0;
  if (n == 0) return 1;
  double y=1;
  for (int i=0; i < n; i++)
    y *= x;
  for (int i=0; i > n; i--)
    y /= x;
  return y;
}
```

```
Enter a positive float x and an integer n: 2.0 -3
pow(2,-3) = 0.125
```

5.21 This tests a boolean function that tests integers for triangularity:

```
int isTriangular(int);
int main()
{ const int MAX=12;
  for (int i=0; i<MAX; i++)
    if (isTriangular(i)) cout << i << " is triangular.\n";
    else cout << i << " is not triangular.\n";
}
int isTriangular(int n)
{ int x=0, y=0, dy=1;
  while (y < n)
    y += dy++;
  if (y == n) return true;
  else return false;
}
```

```
0 is triangular.
1 is triangular.
2 is not triangular.
3 is triangular.
4 is not triangular.
5 is not triangular.
6 is triangular.
7 is not triangular.
8 is not triangular.
9 is not triangular.
10 is triangular.
11 is not triangular.
```

5.22 This tests a boolean function that tests integers for squares:

```
int isSquare(int);
int main()
{ const int MAX=20;
  for (int i=0; i<MAX; i++)
    if (isSquare(i)) cout << i << " is square.\n";
    else cout << i << " is not square.\n";
}
int isSquare(int n)
{ int i=0;
  while (i*i<n)
```

```
    ++i;
  if (i*i == n) return true;
  else return false;
}
```

```
0 is square.
1 is square.
2 is not square.
3 is not square.
4 is square.
5 is not square.
6 is not square.
7 is not square.
8 is not square.
9 is square.
10 is not square.
11 is not square.
```

5.23 This tests a boolean function that tests integers for pentangularity:

```
int isPentagonal(int);
int main()
{ const int MAX=40;
  for (int i=0; i<MAX; i++)
    if (isPentagonal(i)) cout << i << " is pentagonal.\n";
    else cout << i << " is not pentagonal.\n";
}
int isPentagonal(int n)
{ int x=0, y=0, dy=1;
  while (y < n)
  { y += dy;
    dy += 3;
  }
  if (y == n) return true;
  else return false;
}
```

```
0 is pentagonal.
1 is pentagonal.
2 is not pentagonal.
3 is not pentagonal.
4 is not pentagonal.
5 is pentagonal.
6 is not pentagonal.
7 is not pentagonal.
8 is not pentagonal.
9 is not pentagonal.
10 is not pentagonal.
11 is not pentagonal.
12 is pentagonal.
13 is not pentagonal.
```

5.24 This tests a function that has reference parameters:

```
void computeCircle(double& area, double& circ, double r);
int main()
{ double a, c, r;
  cout << "Enter the radius: ";
  cin >> r;
```

```
  computeCircle(a,c,r);
  cout << "The area of a circle of radius " << r << " is " << a
       << "\nand its circumference is " << c << endl;
}
void computeCircle(double& area, double& circ, double r)
{ const double PI=3.141592653589793;
  area = PI*r*r;
  circ = 2*PI*r;
}
```

```
Enter the radius: 10
The area of a circle of radius 10 is 314.159
and its circumference is 62.8319
```

5.25 This tests a function that has reference parameters:

```
void computeTriangle(float& a, float& p, float x,float y,float z);
int main()
{ float a, p, x, y, z;
  cout << "Enter the sides: ";
  cin >> x >> y >> z;
  computeTriangle(a,p,x,y,z);
  cout << "The area of the triangle is " << a
       << "\nand its perimeter is " << p << endl;
}
void computeTriangle(float& a, float& p, float x, float y, float
z)
{ p = x + y + z;
  float s = p/2.0;  // the semiperimeter of the triangle
  a = sqrt(s*(s-x)*(s-y)*(s-z));  // Heron's formula
}
```

```
Enter the sides: 30 50 40
The area of the triangle is 600
and its perimeter is 120
```

5.26 This tests a function that has reference parameters:

```
void computeSphere(double& a, double& v, double r);
int main()
{ double a, v, r;
  cout << "Enter the radius: ";
  cin >> r;
  computeSphere(a,v,r);
  cout << "The area of a sphere of radius " << r << " is " << a
       << "\nand its volume is " << v << endl;
}
void computeSphere(double& a, double& v, double r)
{ const double PI=3.141592653589793;
  a = 4.0*PI*r*r;
  v = a*r/3.0;
}
```

```
Enter the radius: 10
The area of a sphere of radius 10 is 1256.64
and its volume is 4188.79
```

Chapter 6

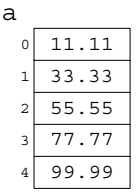
Arrays

6.1 INTRODUCTION

An *array* is a sequence of objects all of which have the same type. The objects are called the *elements* of the array and are numbered consecutively 0, 1, 2, 3, ... These numbers are called *index values* or *subscripts* of the array. The term "subscript" is used because as a mathematical sequence, an array would be written with subscripts: a_0, a_1, a_2, …. The subscripts locate the element's position within the array, thereby giving *direct access* into the array.

If the name of the array is `a`, then `a[0]` is the name of the element that is in position 0, `a[1]` is the name of the element that is in position 1, *etc*. In general, the *i*th element is in position *i*–1. So if the array has *n* elements, their names are `a[0]`, `a[1]`, `a[2]`, …, `a[n-1]`.

We usually visualize an array as a series of adjacent storage compartments that are numbered by their index values. For example, the diagram here shows an array named `a` with 5 elements: `a[0]` contains 11.11, `a[1]` contains 33.33, `a[2]` contains 55.55, `a[3]` contains 77.77, and `a[4]` contains 99.99. The diagram actually represents a region of the computer's memory because an array is always stored this way with its elements in a contiguous sequence.



The method of numbering the *i*th element with index *i*–1 is called *zero-based indexing*. It guarantees that the index of each array element is equal to the number of "steps" from the initial element `a[0]` to that element. For example, element `a[3]` is 3 steps from element `a[0]`.

Virtually all useful programs use arrays. If several objects of the same type are to be used in the same way, it is usually simpler to encapsulate them into an array.

6.2 PROCESSING ARRAYS

An array is a *composite object*: it is composed of several elements with independent values. In contrast, an ordinary variable of a primitive type is called a *scalar object*.

The first example shows that array elements can be assigned and accessed the same as ordinary scalar objects.

EXAMPLE 6.1 Using Direct Access on Arrays

```
int main()
{ double a[3];
  a[2] = 55.55;
  a[0] = 11.11;
  a[1] = 33.33;
  cout << "a[0] = " << a[0] << endl;
  cout << "a[1] = " << a[1] << endl;
  cout << "a[2] = " << a[2] << endl;
}
```

```
a[0] = 11.11
a[1] = 33.33
a[2] = 55.55
```

The first line declares `a` to be an array of 3 elements of type **`double`**. The next three lines assign values to those elements.

Arrays are usually processed with **`for`** loops.

EXAMPLE 6.2 Printing a Sequence in Order

This program reads five numbers and then prints them in reverse order:

```
int main()
{ const int SIZE=5;  // defines the size N for 5 elements
  double a[SIZE];    // declares the array's elements as type double
  cout << "Enter " << SIZE << " numbers:\t";
  for (int i=0; i<SIZE; i++)
    cin >> a[i];
  cout << "In reverse order: ";
  for (int i=SIZE-1; i>=0; i--)
    cout << "\t" << a[i];
}
```

```
Enter 5 numbers:        11.11   33.33   55.55   77.77   99.99
In reverse order:       99.99   77.77   55.55   33.33   11.11
```

The first line defines the symbolic constant `SIZE` to be 5 elements. The second line declares `a` to be an array of 5 elements of type **`double`**. Then the first **`for`** loop reads 5 values into the array, and the second **`for`** loop prints them in reverse order.

The syntax for an array declaration is

`type array-name``[`*`array-size`*`];`

where *`type`* is the array's element type and *`array-size`* is its number of elements. The declaration in Example 6.1

```
double a[SIZE];
```

declares `a` to be an array of 5 elements, each of type **`double`**. Standard C++ requires *`array-size`* to be a positive integer constant. So it must be either a symbolic constant as in Example 6.1, or an integer literal like this:

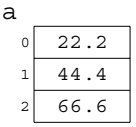
```
double a[5];
```

Generally, it is better to use a symbolic constant since the same size value is likely to be used in **`for`** loops that process the array.

6.3 INITIALIZING AN ARRAY

In C++, an array can be initialized with an optional *initializer list*, like this:

```
float a[] = {22.2, 44.4, 66.6 };
```



The values in the list are assigned to the elements of the array in the order that they are listed. The size of the array is set to be equal to the number of values in the initializer list. So this single line of code declares `a` to be an array of 3 floats and then initializes those for elements with the four values given in the list.

EXAMPLE 6.3 Initializing an Array

This program initializes the array `a` and then prints its values:

```
int main()
{ float a[] = { 22.2, 44.4, 66.6 };
  int size = sizeof(a)/sizeof(float);
  for (int i=0; i<size; i++)
    cout << "\ta[" << i << "] = " << a[i] << endl;
}
```

```
        a[0] = 22.2
        a[1] = 44.4
        a[2] = 66.6
```

The first line declares `a` to be the array of 3 elements described above. The second line uses the `sizeof()` function to compute the actual number of elements in the array. The value of `sizeof(float)` is 4 because on this machine a **`float`** value occupies 4 bytes in memory. The value of `sizeof(a)` is 12 because the complete array occupies 12 bytes in memory. Therefore, the value of `size` is computed to be 12/4 = 3.

An array can be "zeroed out" by declaring it with an initializer list together with an explicit size value, like this:

```
float a[7] = { 55.5, 66.6, 77.7 };
```

This array is declared to have 7 elements of type **`float`**; then its initializer list initializes the first 3 elements with the given values and the remaining 4 elements with the value 0.

a

0	55.5
1	66.6
2	77.7
3	0.0
4	0.0
6	0.0
7	0.0

EXAMPLE 6.4 Initializing an Array with Trailing Zeros

This program initializes the array `a` and then prints its values:

```
int main()
{ float a[7] = { 22.2, 44.4, 66.6 };
  int size = sizeof(a)/sizeof(float);
  for (int i=0; i<size; i++)
    cout << "\ta[" << i << "] = " << a[i] << endl;
}
```

```
        a[0] = 22.2
        a[1] = 44.4
        a[2] = 66.6
        a[3] = 0
        a[4] = 0
        a[5] = 0
        a[6] = 0
```

Note that the number of values in an array's initializer list cannot exceed its size:

```
float a[3] = { 22.2, 44.4, 66.6, 88.8 };  // ERROR: too many values!
```

An array can be initialized to be all zeros by using an empty initializer list. So, for example, the following three declarations are equivalent:

```
float a[ ] = { 0, 0, 0, 0, 0, 0, 0, 0, 0 };
float a[9] = { 0, 0 };
float a[9] = { 0, 0, 0, 0, 0, 0, 0, 0, 0 };
```

But note that this is not the same as using no initializer list. Just as with a variable of fundamental type, if an array is not initialized it will contain "garbage" values.

EXAMPLE 6.5 An Uninitialized Array

This program initializes the array `a` and then prints its values:

```
int main()
{ const int SIZE=4;  // defines the size N for 4 elements
  float a[SIZE];     // declares the array's elements as type float
  for (int i=0; i<SIZE; i++)
    cout << "\ta[" << i << "] = " << a[i] << endl;
}
```

```
        a[0] = 6.01838e-39
        a[1] = 9.36651e-39
        a[2] = 6.00363e-39
        a[3] = 0
```

Note that the values in the uninitialized array may or may not be zero; it depends upon how that part of memory was used prior to the execution of this program.

Note that an initialization is not the same as an assignment. Arrays can be initialized, but they cannot be assigned:

```
float a[7] = { 22.2, 44.4, 66.6 };
float b[7] = { 33.3, 55.5, 77.7 };
b = a;  // ERROR: arrays cannot be assigned!
```

Nor can an array be used to initialize another array:

```
float a[7] = { 22.2, 44.4, 66.6 };
float b[7] = a;  // ERROR: arrays cannot be used as initializers!
```

6.4 ARRAY INDEX OUT OF BOUNDS

In some programming languages, an index variable will not be allowed to go beyond the bounds set by the array's definition. For example, in Pascal, if an array `a` is defined to be indexed from 0 to 3, then the reference `a[6]` will crash the program. This is a security device that does not exist for arrays in C++ (or C). As the next example shows, the index variable may run far beyond its defined range without any error being detected by the computer.

EXAMPLE 6.6 Allowing an Array Index to Exceed its Bounds

This program has a run-time error: it accesses a part of memory that is not allocated:

```
int main()
{ const int SIZE=4;
  float a[SIZE] = { 33.3, 44.4, 55.5, 66.6 };
  for (int i=0; i<7; i++)  // ERROR: index is out of bounds!
    cout << "\ta[" << i << "] = " << a[i] << endl;
}
```

```
        a[0] = 33.3
        a[1] = 44.4
        a[2] = 55.5
        a[3] = 66.6
        a[4] = 5.60519e-45
        a[5] = 6.01888e-39
        a[6] = 6.01889e-39
```

The last three values printed are garbage values, left from the previous use of those bytes in memory.

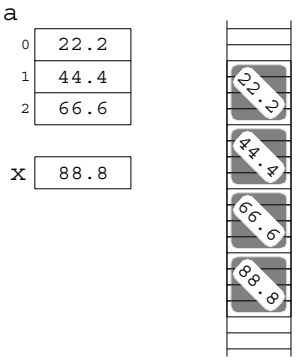
Allowing an array index to exceed its bounds can cause disastrous side effects, as the next example shows.

EXAMPLE 6.7 Causing Side Effects

This program inadvertently changes the value of a variable when it accesses a nonexistent element of an array:

```
int main()
{ const int SIZE=4;
  float a[] = { 22.2, 44.4, 66.6 };
  float x=11.1;
  cout << "x = " << x << endl;
  a[3] = 88.8;  // ERROR: index is out of bounds!
  cout << "x = " << x << endl;
}
```

```
x = 11.1
x = 88.8
```



The variable `x` is declared after the array `a`, so the system allocates a 4-byte block of memory to `x` that immediately follows the 12 bytes of memory that it allocates to the 3 elements of `a`. Consequently, the 16 contiguous bytes of memory that `a` and `x` occupy are configured as though `x` were `a[3]`. So when the program assigns 88.8 to `a[3]` (which does not exist), it actually changes the value of `x` to 88.8. This is depicted in the diagram on the right which represents 20 contiguous bytes of memory; the four bytes used to store 88.8 immediately follow the four bytes used to store 66.6.

This is the worst kind of run-time error. It changes the value of a variable which is completely independent and not even mentioned in the code where the change occurs. This kind of error is called a *side effect*. It can have disastrous consequences because it may not be detected.

It is the C++ programmer's responsibility to ensure that array index values are kept in range. As Example 6.7 shows, the penalty for shirking that responsibility can be severe if the resulting side effects are not detected.

The next example shows that a different kind of run-time error can occur if an array index is allowed to get too big.

EXAMPLE 6.8 Causing Unhandled Exceptions

This program crashes because the array index gets too big:

```
int main()
{ const int SIZE=4;
  float a[] = { 22.2, 44.4, 66.6 };
  float x=11.1;
  cout << "x = " << x << endl;
  a[3333] = 88.8;  // ERROR: index is out of bounds!
  cout << "x = " << x << endl;
}
```

When run on a Windows workstation, this program generates the alert panel shown here. This little window is reporting that the program attempted to access memory location 0040108e. That location is outside the segment of memory that was allocated to the process that is running the program. So the Windows operating system aborted the program.

The run-time error that occurred in Example 6.8 is called an *unhandled exception* because there is no code in the program to respond to the error. It is possible to include code in C++ programs so that the program won't crash. Such code is called an *exception handler.*

Unlike some other programming languages (*e.g.*, Pascal and Java), the Standard C++ compiler will not allow arrays to be assigned and it will not restrict array indexes from exceeding their bounds. It is the programmer's responsibility to prevent these compile-time and run-time errors. The reward for this extra responsibility is faster, more efficient code. If those benefits are not important to your application, then you should use Standard C++ `vector` objects instead of arrays. (See Chapter 10.)

6.5 PASSING AN ARRAY TO A FUNCTION

The code `float a[]` that declares an array `a` in the previous examples tells the compiler two things: the name of the array is `a`, and the array's elements have type **`float`**. The symbol `a` stores the array's memory address. So the code `float a[]` provides all the information that the compiler needs to declare the array. The size of the array (*i.e.*, the number of elements in the array) does not need to be conveyed to the compiler. C++ requires the same information to be passed to a function that uses an array as a parameter.

EXAMPLE 6.9 Passing an Array to a Function that Returns its Sum

```
int sum(int[],int);
int main()
{ int a[] = { 11, 33, 55, 77 };
  int size = sizeof(a)/sizeof(int);
  cout << "sum(a,size) = " << sum(a,size) << endl;
}
int sum(int a[], int n)
{ int sum=0;
  for (int i=0; i<n; i++)
    sum += a[i];
  return sum;
}
sum(a,size) = 176
```

The function's parameter list is **`(int a[], int n)`**. The function prototype, which is used to declare the function above `main()`, uses **`(int[],int)`**; this is the same as in the prototype except that the names of the parameters are omitted. (They can be included.) The function call, which occurs inside `main()`, uses **`sum(a,size)`**; this lists the names of the parameters without their types. Note that the actual name of the type for the object `a` is **`int[]`**.

When an array is passed to a function, as in the call `sum(a,size)` in Example 6.9, the value of array name `a` is actually the memory address of the first element (`a[0]`) in the array. The function uses that address value to access and possibly modify the contents of the array. So passing an array to a function is similar to passing a variable by reference: the function can change the values of the array's elements. This is illustrated in the next example.

EXAMPLE 6.10 Input and Output Functions for an Array

This program uses a `read()` function to input values into the array `a` interactively. Then it uses a `print()` function to print the array:

```
void read(int[],int&);
void print(int[],int);
int main()
{ const int MAXSIZE=100;
  int a[MAXSIZE]={0}, size;
  read(a,size);
  cout << "The array has " << size << " elements: ";
  print(a,size);
}
void read(int a[], int& n)
{ cout << "Enter integers.  Terminate with 0:\n";
  n = 0;
  do
  { cout << "a[" << n << "]: ";
    cin >> a[n];
  } while (a[n++] != 0 && n < MAXSIZE);
  --n;  // don't count the 0
}
void print(int a[], int n)
{ for (int i=0; i<n; i++)
    cout << a[i] << " ";
}
```

```
Enter integers.  Terminate with 0:
a[0]: 11
a[1]: 22
a[2]: 33
a[3]: 44
a[4]: 0
The array has 4 elements: 11 22 33 44
```

The `read()` function changes the values of the array `a` and the value of the size parameter `n`. Since `n` is a scalar variable, it must be passed by reference to allow the function to change its value. Since `a` is an array variable, it must be passed by value and the function is able to change the values its elements.

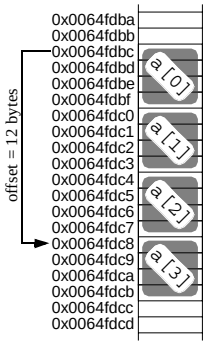
Note that the size of the array has to be passed explicitly to the function that processes the array. In C++ a function is unable to compute the size of an array passed to it.

Example 6.10 shows that a function can change the values of an array's elements even though the array variable is passed by value. That is possible because the value of the array variable itself is the memory address of the first element of the array. Passing the value of that address to the function gives the function all the information it needs to access and change that part of memory where the array is stored. This is accomplished by a direct calculation of the elements'

locations from the given memory address and the array index. For example, in the input statement

```
cin >> a[n];
```

in Example 6.10, when `n` = 3 the system computes that the memory address of `a[3]` is 3×4 = 12 bytes past the memory address of `a[0]`. That address is passed (by value) to the function in the variable `a`, so the explicit address of `a[3]` is obtained. For example, suppose that `a[0]` is stored in the four contiguous bytes starting at location 0x0064fdbc ("0x0064fdbc" is hexadecimal notation for 6,618,556). Then the calculated address for `a[3]` is 0x0064fdc8 (hexadecimal for 6,618,568 = 6,618,556 + 12). In this calculation, the number 12 is called the *offset* for the element `a[3]`. (See Problem 6.5 on page 143. Also see Appendix G for information on hexadecimal numerals.)



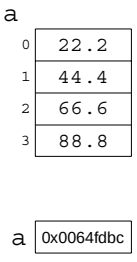
Note that an array name (whose value is a memory address) itself is a constant, so it cannot be changed anywhere. This simply means that the array cannot be moved to another place in memory.

EXAMPLE 6.11 Printing the Memory Location of an Array

This program prints the value of the address stored in an array name.

```
int main()
{ int a[] = { 22, 44, 66, 88 };
  cout << "a = " << a;  // the address of a[0]
}
```

```
a = 0x0064fdec
```



The array name `a` has two interpretations. It is used with an index to name each element of the array, so it identifies the complete composite object. But as a variable, its value is the memory address of the first byte of the first element `a[0]` in the array.

6.6 THE LINEAR SEARCH ALGORITHM

Computers are probably used more for the storage and retrieval of information than for any other purpose. Data is often stored in a sequential structure such as an array. The simplest way to find an object in an array is start at the beginning and inspect each element, one after the other, until the object is found. This method is called the *Linear Search algorithm*.

EXAMPLE 6.12 The Linear Search

This program tests a function that implements the Linear Search algorithm:

```
int index(int,int[],int);
int main()
{ int a[] = { 22, 44, 66, 88, 44, 66, 55 };
  cout << "index(44,a,7) = " << index(44,a,7) << endl;
  cout << "index(50,a,7) = " << index(50,a,7) << endl;
}
```

```
int index(int x, int a[], int n)
{ for (int i=0; i<n; i++)
    if (a[i] == x) return i;
  return n;  // x not found
}
index(44,a,7) = 1
index(50,a,7) = 7
```

6.7 THE BUBBLE SORT ALGORITHM

The Linear Search algorithm is not very efficient. It obviously would not be a good way to find a name in the telephone book. We can do this common task more efficiently because the names are sorted in alphabetical order. To use an efficient searching algorithm on a sequential data structure such as an array, we must first sort the structure to put its elements in order.

There are many algorithms for sorting an array. Although not as efficient as most others, the Bubble Sort is one of the simplest sorting algorithms. It proceeds through a sequence of iterations, each time moving the next largest item into its correct position. On each iteration, it compares each pair of consecutive elements, moving the larger element up.

EXAMPLE 6.13 The Bubble Sort

This program tests a function that implements the Bubble Sort algorithm.

```
void print(float[],int);
void sort(float[],int);
int main()
{ float a[] = {55.5, 22.5, 99.9, 66.6, 44.4, 88.8, 33.3, 77.7};
  print(a,8);
  sort(a,8);
  print(a,8);
}
void sort(float a[], int n)
{ // bubble sort:
  for (int i=1; i<n; i++)
    // bubble up max{a[0..n-i]}:
    for (int j=0; j<n-i; j++)
      if (a[j] > a[j+1]) swap(a[j],a[j+1]);
    // INVARIANT: a[n-1-i..n-1] is sorted
}
```

```
55.5, 22.5, 99.9, 66.6, 44.4, 88.8, 33.3, 77.7
22.5, 33.3, 44.4, 55.5, 66.6, 77.7, 88.8, 99.9
```

The `sort()` function uses two nested loops. The inside **`for`** loop compares pairs of adjacent elements and swaps them whenever they are out of order. This way, each element "bubbles up" past all the elements that are less than it.

6.8 THE BINARY SEARCH ALGORITHM

The binary search uses the "divide and conquer" strategy. It repeatedly divides the array into two pieces and then searches the piece that could contain the target value.

EXAMPLE 6.14 The Binary Search Algorithm

This program tests a function that implements the Binary Search algorithm. It uses the same test driver that was used in Example 6.12 on page 133 to test the Linear Search algorithm:

```
int index(int,int[],int);
int main()
{ int a[] = { 22, 33, 44, 55, 66, 77, 88 };
  cout << "index(44,a,7) = " << index(44,a,7) << endl;
  cout << "index(60,a,7) = " << index(60,a,7) << endl;
}
int index(int x, int a[], int n)
{ // PRECONDITION: a[0] <= a[1] <= ... <= a[n-1];
  // binary search:
  int lo=0, hi=n-1, i;
  while (lo <= hi)
  { i = (lo + hi)/2; // the average of lo and hi
    if (a[i] == x) return i;
    if (a[i] < x) lo = i+1;  // continue search in a[i+1..hi]
    else hi = i-1;           // continue search in a[lo..i-1]
  }
  return n;  // x was not found in a[0..n-1]
}
```

```
index(44,a,7) = 2
index(60,a,7) = 7
```

Note that the array is already sorted before the Binary Search is applied. That requirement is expressed in the PRECONDITION specified as a comment in the function's code.

On each iteration of the **while** loop, the middle element `a[i]` of the subarray `a[lo..hi]` (*i.e.*, all the elements from `a[lo]` to `a[hi]`) is examined. If it is not the target `x`, then the search continues either on the upper half `a[i+1..hi]` or on the lower half `a[lo..i-1]`. If `(a[i] < x)`, then `x` could not be in the lower half (since the array is sorted into increasing order), so the lower half can be ignored and the search continued on only the upper half. Similarly, if the condition `(a[i] < x)` is false, then the search is continued on only the lower half. So on each iteration of the loop, the scope of the search is reduced by about 50%. The loop stops either when `x` is found at `a[i]` and the function returns, or when `lo` > `hi`. In that latter case, the subarray `a[lo..hi]` is empty, meaning that `x` was not found, so the function returns `n`.

Here is a trace of the call `index(44,a,7)`. When the loop begins, `x` = 44, `n` = 7, `lo` = 0, and `hi` = 6; the middle element of the array `a[0..6]` is `a[3]` = 55 which is greater than `x`, so `hi` gets reset to `i-1` = 2. On the second iteration, `lo` = 0 and `hi` = 2; the middle element of the subarray `a[0..2]` is `a[1]` = 33 which is less than `x`, so `lo` gets reset to `i+1` = 2. On the third iteration, `lo` = 2 and `hi` = 2; the middle element of the subarray `a[2..2]` is `a[2]` = 44 which is equal to `x`, so the function returns 2, indicating that the target `x` is was found at `a[2]`.

lo	hi	i	a[i]	??	x
0	6	3	55	>	44
	2	1	33	<	44
2		2	44	==	44

Here is a trace of the call `index(60,a,7)`. When the loop begins, `x` = 44, `n` = 7, `lo` = 0, and `hi` = 6; the middle element of the array `a[0..6]` is `a[3]` = 55 which is less than `x`, so `lo` gets reset to `i+1` = 4. On the second iteration, `lo` = 4 and `hi` = 6; the middle element of the subarray `a[4..6]` is `a[5]` = 77 which is

lo	hi	i	a[i]	??	x
0	6	3	55	<	60
4		5	77	>	60
	4	4	66	>	60

greater than x, so hi gets reset to i-1 = 4. On the third iteration, lo = 4 and hi = 4; the middle element of the subarray a[4..4] is a[4] = 66 which is greater than x, so hi gets reset to i-1 = 3. That terminates the loop, so the function returns 7, indicating that the target x was not found.

The Binary Search algorithm is significantly different from the Linear Search algorithm. The most important distinction is that the Binary Search works only on sorted arrays. The benefit of that requirement is that the Binary Search is much faster than the Linear Search. For example, on an array of 100 elements, the Linear Search could take up to 100 iterations, but the Binary Search will not need more than 8 iterations, no matter what the target is. That is because the Binary Search runs in *logarithmic time*; *i.e.*, the number of iterations cannot exceed $\lg n + 1$, where n is the size of the array and $\lg n$ is the binary (base 2) logarithm of n. When $n = 100$, $\lg n + 1 = 7.64$. Note that in Example 6.14, $n = 7$ elements, so $\lg n + 1 = 3.81$; this means that no more than 3 iterations will ever be needed.

A third distinction between the two algorithms is that the Linear Search returns the smallest index i for which a[i] == x. But the Binary Search is not specific: if there are multiple copies of x, you cannot be sure which one is located by the returned index.

Since the Binary Search requires that the array be sorted, it is useful to have a separate function that tests that condition.

EXAMPLE 6.15 Determining whether an Array is Sorted

This program tests a boolean function that determines whether a given array is nondecreasing.

```
bool isNondecreasing(int a[], int n);
int main()
{ int a[] = { 22, 44, 66, 88, 44, 66, 55 };
  cout << "isNondecreasing(a,4) = " << isNondecreasing(a,4) << endl;
  cout << "isNondecreasing(a,7) = " << isNondecreasing(a,7) << endl;
}
bool isNondecreasing(int a[], int n)
{ // returns true iff a[0] <= a[1] <= ... <= a[n-1]:
  for (int i=1; i<n; i++)
    if (a[i]<a[i-1]) return false;
  return true;
}
```

```
isNondecreasing(a,4) = 1
isNondecreasing(a,7) = 0
```

If the function finds any adjacent pair (a[i-1],a[i]) of elements that decrease (*i.e.*, a[i]<a[i-1]), then it returns **false**. If that doesn't happen, then it returns **true**, meaning that the array is nondecreasing.

Note that the boolean values **true** and **false** are printed as the integers 1 and 0; that is how they are stored in memory.

If the precondition in Example 6.14 that the array be sorted is not true, the Binary search function search() will not work correctly. Such conditions can be checked automatically using the assert() function defined in the <cassert> header. This function takes a boolean argument. If the argument is **false**, the function terminates the program and reports the fact to the operating system. If the argument is **true**, the program continues unaffected.

EXAMPLE 6.16 Using the `assert()` Function to Enforce a Precondition

This program tests an improved version of the `search()` function from Example 6.14. This version uses the `isNondecreasing()` function from Example 6.15 to determine whether the array is sorted. It passes the resulting boolean return value to the `assert()` function so that the search will not be carried out if the array is not sorted:

```
#include <cassert>   // defines the assert() function
#include <iostream>  // defines the cout object
using namespace std;
int index(int x, int a[], int n);
int main()
{ int a[] = { 22, 33, 44, 55, 66, 77, 88, 60 };
  cout << "index(44,a,7) = " << index(44,a,7) << endl;
  cout << "index(44,a,7) = " << index(44,a,8) << endl;
  cout << "index(60,a,7) = " << index(60,a,8) << endl;
}
bool isNondecreasing(int a[], int n);
int index(int x, int a[], int n)
{ // PRECONDITION: a[0] <= a[1] <= ... <= a[n-1];
  // binary search:
  assert(isNondecreasing(a,n));
  int lo=0, hi=n-1, i;
  while (lo <= hi)
  { i = (lo + hi)/2;
    if (a[i] == x) return i;
    if (a[i] < x) lo = i+1;  // continue search in a[i+1..hi]
    else hi = i-1;           // continue search in a[lo..i-1]
  }
  return n;  // x was not found in a[0..n-1]
}
index(44,a,7) = 2
```

Here, the array `a[]` is not completely sorted. But its first 7 elements are in order. So on the first call `index(44,a,7)`, the `index()` function makes the call `isNondecreasing(a,7)` which returns the boolean value **`true`** to the `assert()` function, and the output is the same as in Example 6.14. But on the second call `index(44,a,8)`, the subsequent call `isNondecreasing(a,8)` returns the boolean value **`false`** to the `assert()` function which then aborts the program, causing Windows to display the alert panel shown here.

6.9 USING ARRAYS WITH ENUMERATION TYPES

Enumeration types were described in Chapter 2. They are naturally processed with arrays.

EXAMPLE 6.17 Enumerating the Days of the Week

This program defines an array `high[]` of seven **`float`**s, representing the high temperatures for the seven days of a week:

```
int main()
{ enum Day { SUN, MON, TUE, WED, THU, FRI, SAT };
  float high[SAT+1] = {88.3, 95.0, 91.2, 89.9, 91.4, 92.5, 86.7};
  for (int day = SUN; day <= SAT; day++)
    cout << "The high temperature for day " << day
         << " was " << high[day] << endl;
}
```

```
The high temperature for day 0 was 88.3
The high temperature for day 1 was 95.0
The high temperature for day 2 was 91.2
The high temperature for day 3 was 89.9
The high temperature for day 4 was 91.4
The high temperature for day 5 was 92.5
The high temperature for day 6 was 86.7
```

The array size is `SAT+1` because `SAT` has the integer value 6 and the array needs 7 elements.

The `int` variable `day`, declared as an index in the **`for`** loop, takes the values `SUN`, `MON`, `TUE`, `WED`, `THU`, `FRI`, or `SAT`. Remember that they are actually stored as the integers 0, 1, 2, 3, 4, 5, and 6.

Note that it is not possible to print the names of the symbolic constants.

The advantage of using enumeration constants this way is that they render your code "self-documenting." For example, in Example 6.17 the **`for`** loop control

```
for (int day = SUN; day <= SAT; day++)
```

speaks for itself.

6.10 TYPE DEFINITIONS

Enumeration types are one way for programmers to define their own types. For example,

```
enum Color { RED, ORANGE, YELLOW, GREEN, BLUE, VIOLET };
```

defines the type `Color` which can then be used to declare variables like this:

```
Color shirt = BLUE;
Color car[] = { GREEN, RED, BLUE, RED };
float wavelength[VIOLET+1] = {420, 480, 530, 570, 600, 620};
```

Here, `shirt` is a variable whose value can be any one of the 6 values of the type `Color` and is initialized to have the value `BLUE`, `car` is an array of 4 such `Color` type variables indexed from 0 to 3, and `wavelength` is an array of 6 `float` type variables indexed from `RED` to `VIOLET`.

C++ also provides a way to rename existing types. The keyword **`typedef`** declares a new name (*i.e.*, a synonym or alias) for a specified type. The syntax is

`typedef` *`type alias`*`;`

where *`type`* is the given type and *`alias`* is the new name. For example, if you are used to programming in Pascal, you might want to use these type aliases:

```
typedef long Integer;
typedef double Real;
```

Then you could declare variables like this:

```
Integer n = 22;
const Real PI = 3.141592653589793;
Integer frequency[64];
```

Note the syntax for the **`typedef`** of an array type:

`typedef` *`element-type alias`*`[];`

It shows that the number of elements in an array is not part of its type.

A **typedef** statement does not define a new type; it only provides a synonym for an existing type. For example, the `sum()` function defined in Example 6.9 on page 131 could be called by

```
cout << sum(frequency,4);
```

even though the `frequency[]` array is declared (above) to have elements of type `Integer`. There is no conflict in the parameter because `Integer` and **int** are merely different names for the same type.

The next example shows another use for `typedef`s.

EXAMPLE 6.18 The Bubble Sort Again

This is the same program as in Example 6.13 on page 134. The only change is the **typedef** for the type name `Sequence` which is then used in the parameter lists and the declaration of `a` in `main()`:

```
typedef float Sequence[];
void sort(Sequence,int);
void print(Sequence,int);
int main()
{ Sequence a = {55.5, 22.5, 99.9, 66.6, 44.4, 88.8, 33.3, 77.7};
  print(a,8);
  sort(a,8);
  print(a,8);
}
void sort(Sequence a, int n)
{ for (int i=n-1; i>0; i--)
    for (int j=0; j<i; j++)
      if (a[j] > a[j+1]) swap(a[j],a[j+1]);
}
```

Note the `typedef`:

```
typedef float Sequence[];
```

The brackets `[]` appear after the alias type name `Sequence`. This alias is then used without brackets to declare array variables and formal parameters.

6.11 MULTIDIMENSIONAL ARRAYS

The arrays we have used previously have all been *one-dimensional*. This means that they are *linear*; *i.e.*, sequential. But the element type of an array can be almost any type, including an array type. An array of arrays is called a *multidimensional array.* A one-dimensional array of one-dimensional arrays is called a two-dimensional array; a one-dimensional array of two-dimensional arrays is called a three-dimensional array; *etc.*

The simplest way to declare a multidimensional array is like this:

```
double a[32][10][4];
```

This is a three-dimensional array with dimensions 32, 10, and 4. The statement

```
a[25][8][3] = 99.99
```

would assign the value 99.99 to the element identified by the multi-index (25,8,3).

EXAMPLE 6.19 Reading and Printing a Two-Dimensional Array

This program shows how a two-dimensional array can be processed:

```
void read(int a[][5]);
void print(cont int a[][5]);
```

```
int main()
{ int a[3][5];
  read(a);
  print(a);
}
void read(int a[][5])
{ cout << "Enter 15 integers, 5 per row:\n";
  for (int i=0; i<3; i++)
  { cout << "Row " << i << ": ";
    for (int j=0; j<5; j++)
    cin >> a[i][j];
  }
}
void print(const int a[][5])
{ for (int i=0; i<3; i++)
  { for (int j=0; j<5; j++)
      cout << " " << a[i][j];
    cout << endl;
  }
}
```

```
Enter 15 integers, 5 per row:
Row 0: 44 77 33 11 44
Row 1: 60 50 30 90 70
Row 2: 85 25 45 45 55
 44 77 33 11 44
 60 50 30 90 70
 85 25 45 45 55
```

Notice that in the functions' parameter lists, the first dimension is left unspecified while the second dimension (5) is specified. This is because the two-dimensional array `a[][]` is stored as a one-dimensional array of three 5-element arrays. The compiler does not need to know how many of these 5-element arrays are to be stored, but it does need to know that they are 5-element arrays.

When a multi-dimensional array is passed to a function, the first dimension is not specified, while all the remaining dimensions are specified.

EXAMPLE 6.20 Processing a Two-Dimensional Array of Quiz Scores

```
const NUM_STUDENTS = 3;
const NUM_QUIZZES = 5;
typedef int Score[NUM_STUDENTS][NUM_QUIZZES];
void read(Score);
void printQuizAverages(Score);
void printClassAverages(Score);
int main()
{ Score score;
  cout << "Enter " << NUM_QUIZZES << " scores for each student:\n";
  read(score);
  cout << "The quiz averages are:\n";
  printQuizAverages(score);
  cout << "The class averages are:\n";
  printClassAverages(score);
}
```

```
void read(Score score)
{ for (int s=0; s<NUM_STUDENTS; s++)
  { cout << "Student " << s << ": ";
    for (int q=0; q<NUM_QUIZZES; q++)
      cin >> score[s][q];
  }
}
void printQuizAverages(Score score)
{ for (int s=0; s<NUM_STUDENTS; s++)
  { float sum = 0.0;
    for (int q=0; q<NUM_QUIZZES; q++)
      sum += score[s][q];
    cout << "\tStudent " << s << ": " << sum/NUM_QUIZZES << endl;
  }
}
void printClassAverages(Score score)
{ for (int q=0; q<NUM_QUIZZES; q++)
  { float sum = 0.0;
    for (int s=0; s<NUM_STUDENTS; s++)
      sum += score[s][q];
    cout << "\tQuiz " << q << ": " << sum/NUM_STUDENTS << endl;
  }
}
```

```
Enter 5 quiz scores for each student:
Student 0: 8 7 9 8 9
Student 1: 9 9 9 9 8
Student 2: 5 6 7 8 9
The quiz averages are:
        Student 0: 8.2
        Student 1: 8.8
        Student 2: 7
The class averages are:
        Quiz 0: 7.33333
        Quiz 1: 7.33333
        Quiz 2: 8.33333
        Quiz 3: 8.33333
        Quiz 4: 8.66667
```

This uses a **typedef** to define the alias `Score` for the two-dimensional array type. This makes the function headers more readable.

The `printQuizAverages()` function prints the average of each of the 3 rows of scores, while the `printClassAverages()` function prints the average of each of the 5 columns of scores.

EXAMPLE 6.21 Processing a Three-Dimensional Array

This program simply counts the number of zeros in a three-dimensional array:

```
int numZeros(int a[][4][3], int n1, int n2, int n3);
int main()
{ int a[2][4][3] = { { {5,0,2}, {0,0,9}, {4,1,0}, {7,7,7} },
                     { {3,0,0}, {8,5,0}, {0,0,0}, {2,0,9} }
                   };
  cout << "This array has " << numZeros(a,2,4,3) << " zeros:\n";
}
```

```
int numZeros(int a[][4][3], int n1, int n2, int n3)
{ int count = 0;
  for (int i = 0; i < n1; i++)
    for (int j = 0; j < n2; j++)
      for (int k = 0; k < n3; k++)
        if (a[i][j][k] == 0) ++count;
  return count;
}
This array has 11 zeros:
```

Notice how the array is initialized: it is a 2-element array of 4-element arrays of 3 elements each. That makes a total of 24 elements. It could have been initialized like this:

```
int a[2][4][3]={5,0,2,0,0,9,4,1,0,7,7,7,3,0,0,8,5,0,0,0,0,2,0,9};
```

or like this:

```
int a[2][4][3]={{5,0,2,0,0,9,4,1,0,7,7,7},{3,0,0,8,5,0,0,0,0,2,0,9}};
```

But these are more difficult to read and understand than the three-dimensional initializer list.

Also notice the three nested **for** loops. In general, processing a *d*-dimensional array is done with *d* **for** loops, one for each dimension.

Review Questions

6.1 How many different types can the elements of an array have?

6.2 What type and range must an array's subscript have?

6.3 What values will the elements of an array have when it is declared if it does not include an initializer?

6.4 What values will the elements of an array have when it is declared if it has an initializer with fewer values than the number of elements in the array?

6.5 What happens if an array's initializer has more values than the size of the array?

6.6 How does an **enum** statement differ from a **typedef** statement?

6.7 When a multi-dimensional array is passed to a function, why does C++ require all but the first dimension to be specified in the parameter list?

Solved Programming Problems

6.1 Modify the program in Example 6.1 on page 126 so that each input is prompted and each output is labeled, like this:

```
Enter 5 numbers
        a[0]: 11.11
        a[1]: 33.33
        a[2]: 55.55
        a[3]: 77.77
        a[4]: 99.99
In reverse order, they are:
        a[4] = 99.99
        a[3] = 77.77
        a[2] = 55.55
        a[1] = 33.33
        a[0] = 11.11
```

6.2 Modify the program in Example 6.1 on page 126 so that it fills the array in reverse and then prints them in the order that they are stored, like this:

```
Enter 5 numbers:
        a[4]: 55.55
        a[3]: 66.66
        a[2]: 77.77
        a[1]: 88.88
        a[0]: 99.99
In reverse order, they are:
        a[0] = 99.99
        a[1] = 88.88
        a[2] = 77.77
        a[3] = 66.66
        a[4] = 55.55
```

6.3 Modify the program in Example 6.9 on page 131 so that it tests the following function:

```
float ave(int[] a, int n);
// returns the average of the first n elements of a[]
```

6.4 Modify the program in Example 6.10 on page 132 so that it prints the array, its sum, and its average. (See Example 6.9 on page 131 and Problem 6.3.)

6.5 Modify the program in Example 6.11 on page 133 so that it prints the memory address and its contents for each element of an array. For an array named `a`, use the expressions `a`, `a+1`, `a+2`, *etc*. to obtain the addresses of `a[0]`, `a[1]`, `a[2]`, *etc*., and use the expressions `*a`, `*(a+1)`, `*(a+2)`, *etc*. to obtain the contents of those locations. Declare the array as

```
unsigned int a[];
```

so that the array element values will be printed as integers when inserted into the **`cout`** stream.

6.6 Modify the program in Example 6.12 on page 133 so that it returns the last location of the target instead of the first.

6.7 Modify the program in Example 6.15 on page 136 so that it returns true if and only if the array is nonincreasing.

6.8 Write and test the following function that returns the minimum value among the first `n` elements of the given array:

```
float min(float a[], int n);
```

6.9 Write and test the following function that returns the index of the first minimum value among the first `n` elements of the given array:

```
int minIndex(float a[], int n);
```

6.10 Write and test the following function that returns through its reference parameters both the maximum and the minimum values stored in an array:

```
void getExtremes(float& min, float& max, float a[], int n);
```

6.11 Write and test the following function that returns through its reference parameters both the largest and the second largest values (possibly equal) stored in an array:

```
void largest(float& max1, float& max2, float a[], int n);
```

6.12 Write and test the following function that removes an item from an array:

```
void remove(float a[], int& n, int i);
```

The function removes `a[i]` by shifting all the elements above that position are down and decrementing `n`.

6.13 Write and test the following function that attempts to remove an item from an array:

```
bool removeFirst(float a[], int& n, float x);
```

The function searches the first `n` elements of the array `a` for the item `x`. If `x` is found, its first occurrence is removed, all the elements above that position are shifted down, `n` is decre-

mented, and **true** is returned to indicate a successful removal. If x is not found, the array is left unchanged and **false** is returned. (See Problem 6.12.)

6.14 Write and test the following function that removes items from an array:

```
void removeAll(float a[], int& n, float x);
```

The function removes all occurrences of x among the first n elements of the array a and decreases the value of n by the number removed. (See Problem 6.13.)

6.15 Write and test the following function:

```
void rotate(int a[], int n, int k);
```

The function "rotates" the first n elements of the array a, k positions to the right (or –k positions to the left if k is negative). The last k elements are "wrapped" around to the beginning of the array. For example, the call `rotate(a,8,3)` would transform the array {22,33,44,55,66,77,88,99} into {77,88,99,22,33,44,55,66}. The call `rotate(a,8,-5)` would have the same effect.

6.16 Write and test the following function:

```
void append(int a[], int m, int b[], int n);
```

The function appends the first n elements of the array b onto the end of the first m elements of the array a. It assumes that a has room for at least m + n elements. For example, if a is {22,33,44,55,66,77,88,99} and b is {20,30,40,50,60,70,80,90} then the call `append(a,5,b,3)` would transform a into {22,33,44,55,66,20,30,40}. Note that b is left unchanged and only n elements of a are changed.

6.17 Write and test the function

```
void insert(float a[], int& n, float x)
```

This function inserts the item x into the sorted array a of n elements and increments n. The new item is inserted at the location that maintains the sorted order of the array. This requires shifting elements forward to make room for the new x. (Note that this requires the array to have at least n+1 elements allocated.)

6.18 Implement the *Insertion Sort* algorithm for sorting an array of n elements. In this algorithm, the main loop index i runs from 1 to *n*–1. On the ith iteration, the element a[i] is "inserted" into its correct position among the subarray a[0..i]. This is done by shifting one position up all the elements in the subarray that are greater than a[i]. Then a[i] is copied into the gap between the elements that are less than or equal to a[i] and those that are greater. (Hint: use the `insert()` algorithm from Problem 6.17.)

6.19 Implement the *Selection Sort* algorithm for sorting an array of n elements. This algorithm has *n*–1 iterations, each selecting the next largest element a[j] and swapping it with the element that is in the position where a[j] should be. So on the first iteration it selects the largest of all the elements and swaps it with a[n-1], and on the second iteration it selects the largest from the remaining unsorted elements a[0..n-2] and swaps it with a[n-2], *etc*. On its *i*th iteration it selects the largest from the remaining unsorted elements a[0..n-i] and swaps it with a[n-i]. (Hint: use the same loops as in Example 6.13 on page 134.)

6.20 Rewrite and test the Bubble Sort function presented in Example 6.13 on page 134, as an *indirect sort*. Instead of moving the actual elements of the array, sort an index array instead.

6.21 Write and test the function

```
int frequency(float a[], int n, int x);
```

This function counts the number of times the item x appears among the first n elements of the array a and returns that count as the frequency of x in a.

6.22 Implement the *Sieve of Eratosthenes* to find prime numbers. Define a boolean array named isPrime[SIZE], set its values isPrime[0] and isPrime[1] **false** (2 is the first

prime), and set all the other elements **true**. Then for each `i` from 4 to `SIZE-1`, set `isPrime[i]` **false** if `i` is divisible by 2 (*i.e.*, `i%2` = 0). Then for each `i` from 6 to `SIZE-1`, set `isPrime[i]` **false** if `i` is divisible by 3. Repeat this process for each possible divisor from 2 to `SIZE/2`. When finished, all the `i`s for which `isPrime[i]` is still **true** are the prime numbers. They are the numbers that have fallen through the sieve.

6.23 Write and test the following function:

```
void reverse(int a[], int n);
```

The function reverses the first `n` elements of the array. For example, the call `reverse(a,5)` would transform the array {22,33,44,55,66,77,88,99} into {66,55,44,33,22,77,88,99}.

6.24 Write and test the following function:

```
bool isSymmetric(int a[], int n);
```

The function returns true if and only if the array obtained by reversing the first `n` elements is the same as the original array. For example, if `a` is {22,33,44,55,44,33,22} then the call `isSymmetric(a,7)` would return **true**, but the call `isSymmetric(a,4)` would return **false**. Warning: The function should leave the array unchanged.

6.25 Write and test the following function:

```
void add(float a[], int n, float b[]);
```

The function adds the first `n` elements of `b` to the corresponding first `n` elements of `a`. For example, if `a` is {2.2,3.3,4.4,5.5,6.6,7.7,8.8,9.9} and `b` is {6.0,5.0,4.0,3.0,2.0,1.0}, then the call `add(a,5,b)` would transform `a` into {8.2,8.3,8.4,8.5,8.6,7.7,8.8,9.9}.

6.26 Write and test the following function:

```
void multiply(float a[], int n, float b[]);
```

The function multiplies the first `n` elements of `a` by the corresponding first `n` elements of `b`. For example, if `a` is the array {2.2,3.3,4.4,5.5,6.6,7.7,8.8,9.9} and `b` is the array {4.0,–3.0,2.0,–1.0,0.0,0.0}, then the call `multiply(a,5,b)` would transform `a` into the array {8.8,–9.9,8.8,–5.5,0.0,7.7,8.8,9.9}.

6.27 Write and test the following function:

```
float innerProduct(float a[], int n, float b[]);
```

The function returns the *inner product* (also called the "dot product" or "scalar product") of the first `n` elements of `a` with the first `n` elements of `b`. This is defined as the sum of the products of corresponding terms. For example, if `a` is the array {2.2,3.3,4.4,5.5,6.6,7.7,8.8,9.9} and `b` is the array {4.0,–3.0,2.0,–1.0,0.0,0.0}, then the call `innerProduct(a,5,b)` would return (2.2)(4.0) + (3.3)(–3.0) + (2.2)(4.0) + (5.5)(–1.0) + (6.6)(0.0) = 2.2.

6.28 Write and test the following function:

```
float outerProduct3(float p[][3], float a[], float b[]);
```

The function returns the *outer product* of the first 3 elements of `a` with the first 3 elements of `b`. For example, if `a` is the array {2.2,3.3,4.4} and `b` is the array {2.0,–1.0,0.0}, then the call `outerProduct(p,a,b)` would transform the two-dimensional array `p` into

$$\begin{matrix} 4.4 & -2.2 & 0.0 \\ 6.6 & -3.3 & 0.0 \\ 8.8 & -4.4 & 0.0 \end{matrix}$$

Its element `p[i][j]` is the product of `a[i]` with `b[j]`.

6.29 Write and test a function that implements the *Perfect Shuffle* of a one-dimensional array with an even number of elements. For example, it would replace the array `{11,22,33,44,55,66,77,88}` with the array `{11,55,22,66,33,77,44,88}`.

6.30 Write and test the function that "rotates" 90° clockwise a two-dimensional square array of `int`s. For example, it would transform the array

11 22 33
44 55 66
77 88 99

into the array

77 44 11
88 55 22
99 66 33

6.31 Write and run a program that reads an unspecified number of numbers and then prints them together with their deviations from their mean.

6.32 Write and test the following function:

```
double stdev(double x[], int n);
```

The function returns the *standard deviation* of a data set of n numbers $x_0, \ldots, x_{n-1}$ defined by the formula

$$s = \sqrt{\frac{\sum_{i=0}^{n-1} (x_i - \bar{x})^2}{n-1}}$$

where $\bar{x}$ is the mean of the data. This formula says: square each deviation `(x[i] - mean)`; sum those squares; divide that square root by `n-1`; take the square root of that sum.

6.33 Extend the program from Problem 6.31 so that it also computes and prints the Z-scores of the input data. The *Z-scores* of the n numbers $x_0, \ldots, x_{n-1}$ are defined by $z_i = (x_i - \bar{x})/s$. They normalize the given data so that they are centered about 0.0 and have standard deviation 1.0. Use the function defined in Problem 6.32.

6.34 In the imaginary "good old days" when a grade of "C" was considered "average," teachers of large classes would often "curve" their grades according to the following distribution:

A: $1.5 \le z$
B: $0.5 \le z < 1.5$
C: $-0.5 \le z < 0.5$
D: $-1.5 \le z < -0.5$
F: $z < -1.5$

If the grades were *normally distributed* (*i.e.*, their density curve is bell-shaped), then this algorithm would produce about 7% A's, 24% B's, 38% C's, 24% D's, and 7% F's. Here the z values are the Z scores described in Problem 6.33. Extend the program from Problem 6.33 so that it prints the "curved" grade for each of the test scores read.

6.35 Write and test a function that creates Pascal's Triangle in the square matrix that is passed to it. For example, if the two-dimensional array `a` and the integer 4 were passed to the function, then it would load the following into `a`:

1 0 0 0 0
1 1 0 0 0
1 2 1 0 0
1 3 3 1 0
1 4 6 4 1

6.36 In the theory of games and economic behavior, founded by John von Neumann, certain two-person games can be represented by a single two-dimensional array, called the *payoff matrix*. Players can obtain optimal strategies when the payoff matrix has a saddle point. A *saddle point* is an entry in the matrix that is both the minimax and the maximin. The *minimax*

of a matrix is the minimum of the column maxima, and the *maximin* is the maximum of the row minima. The optimal strategies are possible when these two values are equal. Write a program that prints the minimax and the maximin of a given matrix.

Answers to Review Questions

6.1 Only one: all of an array's elements must be the same type.

6.2 An array's subscript must be an integer type with range from 0 to *n*-1, where *n* is the array's size.

6.3 In the absence of an initializer, the elements of an array will have unpredictable initial values.

6.4 If the array's initializer has fewer values than the array size, then the specified values will be assigned to the lowest numbered elements and the remaining elements will automatically be initialized to zero.

6.5 It is an error to have more initial values than the size of the array.

6.6 An **`enum`** statement defines an enumeration type which is a new unsigned integer type. A **`typedef`** merely defines a synonym for an existing type.

6.7 When a multi-dimensional array is passed to a function, all dimensions except the first must be specified so that the compiler will be able to compute the location of each element of the array.

Solutions to Problems

6.1 Example 6.1 modified with input prompts and output labels:

```
int main()
{ const int SIZE=5;
  double a[SIZE];
  cout << "Enter " << SIZE << " numbers:\n";
  for (int i=0; i<SIZE; i++)
  { cout << "\ta[" << i << "]: ";
    cin >> a[i];
  }
  cout << "In reverse order, they are:\n";
  for (int i=SIZE-1; i>=0; i--)
    cout << "\ta[" << i << "] = " << a[i] << endl;
}
```

6.2 Example 6.1 modified so that inputs are stored in reverse:

```
int main()
{ const int SIZE=5;
  double a[SIZE];
  cout << "Enter " << SIZE << " numbers:\n";
  for (int i=SIZE-1; i>=0; i--)
  { cout << "\ta[" << i << "]: ";
    cin >> a[i];
  }
  cout << "In reverse order, they are:\n";
  for (int i=0; i<SIZE; i++)
    cout << "\ta[" << i << "] = " << a[i] << endl;
}
```

6.3 Example 6.9 modified so that it tests a function that returns the average of the elements of an array:

```
float ave(int[],int);
int main()
{ int a[] = { 11, 33, 55, 77 };
```

```
  int size = sizeof(a)/sizeof(int);
  cout << "ave(a,size) = " << ave(a,size) << endl;
}
float ave(int a[], int n)
{ float sum=0.0;
  for (int i=0; i<n; i++)
    sum += a[i];
  return sum/n;
}
```

6.4 Example 6.10 modified so that that it prints the array, its sum, and its average:

```
void read(int[],int&);
void print(int[],int);
int sum(int[],int);
float ave(int[],int);
int main()
{ const int MAXSIZE=100;
  int a[MAXSIZE]={0}, size;
  read(a,size);
  cout << "The array has " << size << " elements: ";
  print(a,size);
  cout << "\nIts sum is " << sum(a,size)
       << "\nand its average is " << ave(a,size) << endl;
}
```

The function definitions are the same as in Example 6.9, Example 6.10, and Problem 6.3.

6.5 Example 6.11 modified so that that it prints the memory locations and their contents for each element of an array:

```
int main()
{ unsigned int a[] = { 22, 44, 66, 88 };
  cout << "a   = " << a   << ", *a     = " << *a     << endl;
  cout << "a+1 = " << a+1 << ", *(a+1) = " << *(a+1) << endl;
  cout << "a+2 = " << a+2 << ", *(a+2) = " << *(a+2) << endl;
  cout << "a+3 = " << a+3 << ", *(a+3) = " << *(a+3) << endl;
}
```

```
a   = 0x0064fdbc, *a     = 22
a+1 = 0x0064fdc0, *(a+1) = 44
a+2 = 0x0064fdc4, *(a+2) = 66
a+3 = 0x0064fdc8, *(a+3) = 88
```

The `0x` that prefixes each memory location indicates that those are hexadecimal (base 16) values. (Most computers express memory addresses in hexadecimal notation.) Note that each address is 4 bytes past its predecessor; that shows that **`unsigned int`** objects occupy 4 bytes in memory.

6.6 Example 6.12 modified so that that it prints the memory locations and their contents for each element of an array:

```
int index(int,int[],int);
int main()
{ int a[] = { 22, 44, 66, 88, 44, 66, 55 };
  cout << "index(44,a,7) = " << index(44,a,7) << endl;
  cout << "index(50,a,7) = " << index(50,a,7) << endl;
}
int index(int x, int a[], int n)
{ for (int i=n-1; i>=0; i--)
```

```
    if (a[i] == x) return i;
  return n;
}
index(44,a,7) = 4
index(50,a,7) = 7
```

6.7 Example 6.15 modified so that that it determines whether the array is nonincreasing:

```
bool isNonincreasing(int a[], int n)
{ for (int i=1; i<n; i++)
    if (a[i]>a[i-1]) return false;
  return true;
}
```

6.8

```
float min(float a[], int n)
{ assert(n >= 0);
  float min=a[0];
  for (int i=1; i<n; i++)
    if (a[i] < min) min = a[i];
  return min;
}
```

6.9

```
int minIndex(float a[], int n)
{ assert(n >= 0);
  int j=0;
  for (int i=1; i<n; i++)
    if (a[i] < a[j]) j = i;
  return j;
}
```

6.10

```
void getExtremes(float& min, float& max, float a[], int n)
{ assert(n >= 0);
  min = max = a[0];
  for (int i=1; i<n; i++)
    if (a[i] < min) min = a[i];
    else if (a[i] > max) max = a[i];
}
```

6.11

```
void largest(float& max1, float& max2, float a[], int n)
{ assert(n >= 1);
  if (n == 1) return a[0];
  int i1=0, i2;
  for (int i=1; i<n; i++)
    if (a[i] > a[i1]) i1 = i;
  max1 = a[i1];
  i2 = ( i1 == 0 ? 1 : 0 );
  for (int i=i2+1; i<n; i++)
    if (i != i1 && a[i] > a[i2]) i2 = i;
  max2 = a[i2];
}
```

6.12

```
void remove(float a[], int& n, int i)
{ for (int j=i+1; j<n; j++)
    a[j-1] = a[j];
  --n;
}
```

6.13

```
bool removeFirst(float a[], int& n, float x)
{ for (int i=0; i<n; i++)
    if (a[i] == x)
```

```
      { for (int j=i+1; j<n; j++)
          a[j-1] = a[j];
        --n;
        return true;
      }
    return false;
}
```

6.14

```
void removeAll(float a[], int& n, float x)
{ for (int i=0; i<n; i++)
    if (a[i] == x)
    { for (int j=i+1; j<n; j++)
        a[j-1] = a[j];
      --n;
    }
}
```

6.15

```
void rotate(int a[], int n, int k)
{ const int MAXOFFSET=100;
  assert(k < MAXOFFSET);
  int temp[MAXOFFSET];
  if (k > 0)
  { for (int j=0; j<k; j++)      // copy k elements into temp[]
      temp[j] = a[n-k+j];
    for (int i=n-1; i>=k; i--)  // shift n-k elements
      a[i] = a[i-k];
    for (int i=0; i<k; i++)      // copy k elements back to a[]
      a[i] = temp[i];
  }
  if (k < 0)
  { for (int j=0; j<-k; j++)    // copy -k elements into temp[]
      temp[j] = a[j];
    for (int i=0; i<n+k; i++)   // shift n+k elements
      a[i] = a[i-k];
    for (int i=n+k; i<n; i++)   // copy -k elements back to a[]
      a[i] = temp[i-n-k];
  }
}
```

6.16

```
void append(int a[], int m, int b[], int n)
{ for (int j=0; j<n; j++)  // copy n elements into a[]
    a[m+j] = b[j];
}
```

6.17

```
void insert(float a[], int& n, float x)
{ int j=n;
  while (j>0 && a[j-1]>x)
    a[j--] = a[j-1];
  a[j] = x;
  ++n;
}
```

6.18

```
void sort(float a[], int n)
{ // insertion sort:
  for (int i=1; i<n; i++)
  { // insert a[i] among a[0..i-1]:
    float x=a[i];
```

```
    int j=i;
    while (j>0 && a[j-1]>x)
      a[j--] = a[j-1];
    a[j] = x;
    // INVARIANT: a[0..i] is sorted
  }
}
```

6.19

```
void sort(float a[], int n)
{ // selection sort:
  for (int i=1; i<n; i++)
  { // select a[k] = max{a[0],a[1],...,a[n-i]}:
    int k=0;
    for (int j=1; j<=n-i; j++)
      if (a[j]>a[k]) k = j;
    swap(a[k],a[n-i]);
    // INVARIANT: a[n-1-i..n-1] is sorted
  }
}
```

6.20

```
void sort(float a[], int indx[], int n)
{ // indirect bubble sort:
  for (int i=1; i<n; i++)
    // bubble up max{a[0],a[1],...,a[n-i]}:
    for (int j=0; j<n-i; j++)
      if (a[indx[j]] > a[indx[j+1]]) swap(indx[j],indx[j+1]);
    // INVARIANT: a[indx[n-1-i]] <= a[indx[n-i]] <= ..a[indx[n-1]]
}
```

6.21

```
int frequency(float[],int,int);
int main()
{ float a[] = {561, 508, 400, 301, 329, 599, 455, 400, 346, 346,
               329, 375, 561, 390, 399, 400, 401, 561, 405, 405,
               455, 508, 473, 329, 561, 505, 329, 455, 561, 599,
               561, 455, 346, 301, 455, 561, 399, 599, 508, 508};
  int n=40, x;
  cout << "Item: ";
  cin >> x;
  cout << x << " has frequency " << frequency(a,n,x) << endl;
}
int frequency(float a[], int n, int x)
{ int count = 0;
  for (int i=0; i<n; i++)
    if (a[i] == x) ++count;
  return count;
}
```

```
Item: 400
400 has frequency 3
```

6.22

```
#include <iomanip>   // defines the setw() function
#include <iomanip>   // defines the setw() function
#include <iostream>  // defines the cout object
using namespace std;
const int SIZE = 400;
void sieve(bool[],int);
void print(bool[],int);
```

```
int main()
{ // prints all the prime numbers less than SIZE:
  bool isPrime[SIZE] = {0};
  sieve(isPrime,SIZE);
  print(isPrime,SIZE);
}
void sieve(bool isPrime[], int n)
{ // sets isPrime[i] = false iff i is not prime:
  for (int i=2; i<n; i++)
    isPrime[i] = true;      // assume all i > 1 are prime
  for (int p=2; p<=n/2; p++)
    for (int m=2*p; m<n; m += p)
      isPrime[m] = false;   // no multiple of p is prime
}
void print(bool a[], int n)
{ // prints each i for which isPrime[i] is true:
  for (int i=1; i<n; i++)
    if (a[i]) cout << setw(3) << i;
    else cout << setw(3) << (i%20==0?'\n':' ');
}
```

```
    2  3     5     7          11    13          17    19
      23                29    31                37
 41    43          47                53                59
 61                67          71    73                79
       83                89                      97
101   103         107   109         113
                  127         131               137   139
                        149   151               157
      163         167               173               179
181                             191   193         197   199
                                211
      223         227   229         233               239
241                             251               257
      263               269   271               277
281   283                             293
                  307         311   313         317
                              331               337
                  347   349         353               359
                  367               373               379
      383               389                     397
```

6.23
```
void reverse(int a[], int n)
{ for (int i=0; i<n/2; i++)
    swap(a[i],a[n-1-i]);
}
```

6.24
```
bool isSymmetric(int a[], int n)
{ for (int i=0; i<n/2; i++)
    if (a[i] != a[n-1-i]) return false;
  return true;
}
```

6.25
```
void add(float a[], int n, float b[])
{ for (int i=0; i<n; i++)
    a[i] += b[i];
}
```

6.26
```
void multiply(float a[], int n, float b[])
{ for (int i=0; i<n; i++)
    a[i] *= b[i];
}
```

6.27
```
float innerProduct(float a[], int n, float b[])
{ float p=0;
  for (int i=0; i<n; i++)
    p += a[i]*b[i];
  return p;
}
```

6.28
```
void outerProduct3(float p[][3], float a[], float b[])
{ for (int i=0; i<3; i++)
    for (int j=0; j<3; j++)
      p[i][j] = a[i]*b[j];
}
```

6.29
```
void shuffle(int a[], int n
{ //  The Perfect Shuffle for an even number of elements:
  assert(n <= SIZE);
  int temp[SIZE];
  for (int i=0; i<n/2; i++)
  { temp[2*i] = a[i];
    temp[2*i+1] = a[n/2+i];
  }
  for (int i=0; i<n; i++)
    a[i] = temp[i];
}
```

6.30
```
const int SIZE=3;
typedef int Matrix[SIZE][SIZE];
void print(Matrix);
void rotate(Matrix);
int main()
{ // tests the rotate() function:
  Matrix m = { 11, 22, 33, 44, 55, 66, 77, 88, 99 };
  print(m);
  rotate(m);
  print(m);
}
void print(Matrix a)
{ for (int i=0; i<SIZE; i++)
  { for (int j=0; j<SIZE; j++)
      cout << a[i][j] << "\t";
    cout << endl;
  }
  cout << endl;
}
void rotate(Matrix m)
{ Matrix temp;
  for (int i=0; i<SIZE; i++)
    for (int j=0; j<SIZE; j++)
      temp[i][j] = m[SIZE-j-1][i];
  for (int i=0; i<SIZE; i++)
    for (int j=0; j<SIZE; j++)
```

```
      m[i][j] = temp[i][j];
}
```

6.31

```
const int SIZE = 100;
void read(double[],int&);
double mean(double[],int);
int main()
{ double x[SIZE];
  int n=0;
  read(x,n);
  double m = mean(x,n);
  cout << "mean = " << m << endl;
  for (int i = 0; i < n; i++)
    cout << "x[" << i << "] = " << x[i]
         << ", dev[i] = " << x[i] - m << endl;
}
void read(double x[], int& n)
{ cout << "Enter data.  Terminate with 0:\n";
  while (n<SIZE)
  { cout << "x[" << n << "]: ";
    cin >> x[n];
    if (x[n] == 0) break;
    else ++n;
  }
}
double mean(double x[], int n)
{ double sum=0;
  for (int i=0; i<n; i++)
    sum += x[i];
  return sum/n;
}
```

```
Enter data.  Terminate with 0:
x[0]: 1.23
x[1]: 7.65
x[2]: 0
mean = 4.44
x[0] = 1.23, dev[i] = -3.21
x[1] = 7.65, dev[i] = 3.21
```

6.32

```
double stdev(double a[], int n)
{ assert(n > 1);
  double sum=0;
  for (int i=0; i<n; i++)
    sum += a[i];
  double mean = sum/n;
  sum=0;
  double deviation;
  for (int i=0; i<n; i++)
  { deviation = a[i] - mean;
    sum += deviation*deviation;
  }
  return sqrt(sum/(n-1));
}
```

6.33
```
int main()
{ double x[] = { 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9 };
  int n=8;
  print(x,n);
  double m = mean(x,n);
  double s = stdev(x,n);
  cout << "mean = " << m << ", std dev = " << s << endl;
  for (int i=0; i<n; i++)
    cout << "x[" << i << "] = " << x[i]
         << ", z[" << i << "] = " << (x[i] - m)/s << endl;
}
```

6.34
```
int main()
{ double x[] = { 2.5, 4.5, 6.3, 6.7, 7.2, 7.5, 7.8, 9.9 };
  int n=8;
  print(x,n);
  double m = mean(x,n);
  double s = stdev(x,n);
  cout << "mean = " << m << ", std dev = " << s << endl;
  for (int i=0; i<n; i++)
  { double z = (x[i] - m)/s;
    cout << "x[" << i << "] = " << x[i]
         << ", z[" << i << "] = " << z;
    if (z >= 1.5) cout << " = A" << endl;
    else if (z >= 0.5) cout << " = B" << endl;
    else if (z >= -0.5) cout << " = C" << endl;
    else if (z >= -1.5) cout << " = D" << endl;
    else  cout << " = F" << endl;
  }
}
```

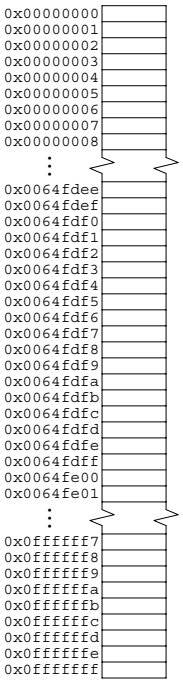
6.35
```
void build_pascal(int p[][SIZE], int n)
{ assert(n > 0 && n < SIZE);
  for (int i=0; i<SIZE; i++)
    for (int j=0; j<SIZE; j++)
      if (i>n || j>i) p[i][j] = 0;
      else if (j==0 || j==i) p[i][j] = 1;
      else p[i][j] = p[i-1][j-1] + p[i-1][j];
}
```

6.36
```
double max_of_col(Matrix m, int n, int j)
{ double max=m[0][j];
  for (int i=1; i<n; i++)
    if (m[i][j]>max) max = m[i][j];
  return max;
}
double minimax(Matrix m, int n)
{ assert(n>0 && n < SIZE);
  double minimax=max_of_col(m,n,0);
  for (int j=1; j<n; j++)
  { double mm = max_of_col(m,n,j);
    if (mm<minimax) minimax = mm;
  }
  return minimax;
}
```

Chapter 7

Pointers and References

7.1 THE REFERENCE OPERATOR

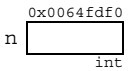


Computer memory can be imagined as a very large array of bytes. For example, a computer with 256 MB of RAM (256 megabytes of random-access memory) actually contains an array of 268,435,456 (2^{28}) bytes. As an array, these bytes are indexed from 0 to 268,435,455. The index of each byte is its memory *address*. So a 256 MB computer has memory addresses ranging from 0 to 268,435,455, which is `0x00000000` to `0x0fffffff` in hexadecimal (see Appendix G). The diagram at right represents that array of bytes, each with its hexadecimal address.

A variable declaration associates three fundamental attributes to the variable: its *name*, its *type*, and its memory *address*. For example, the declaration

```
int n;
```

associates the name `n`, the type `int`, and the address of some location in memory where the value of `n` is stored. Suppose that address is `0x0064fdf0`. Then we can visualize `n` like this:



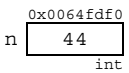
The variable itself is represented by the box. The variable's name `n` is on the left of the box, the variable's address `0x0064fdf0` is above the box, and the variable's type `int` is below the box.

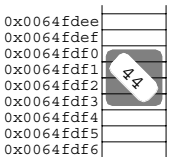
On most computers, variables of type `int` occupy 4 bytes in memory. So the variable `n` shown above would occupy the 4-byte block of memory represented by the shaded rectangle in the diagram at right, using bytes `0x0064fdf0`, `0x0064fdf1`, `0x0064fdf2`, and `0x0064fdf3`. Note that the address of the object is the address of the first byte in the block of memory where the object is stored.

If the variable is initialized, like this:

```
int n=44;
```

then the two representations look like this:





The variable's value 44 is stored in the four bytes allocated to it.

In C++, you can obtain the address of a variable by using the *reference operator* `&`, also called the *address operator*. The expression `&n` evaluates to the address of the variable `n`.

EXAMPLE 7.1 Printing Pointer Values

```
int main()
{ int n=44;
  cout << "n = " << n << endl;     // prints the value of n
  cout << "&n = " << &n << endl;  // prints the address of n
}
```

```
n = 44
&n = 0x0064fdf0
```

The output shows that the address of `n` is `0x0064fdf0`. You can tell that the output `0x0064fdf0` must be an address because it is given in hexadecimal form, identified by its `0x` prefix. The decimal form for this number is 6,618,608. (See Appendix G.)

Displaying the address of a variable this way is not very useful. The reference operator `&` has other more important uses. We saw one use in Chapter 5: designating reference parameters in a function declaration. That use is closely tied to another: declaring reference variables.

7.2 REFERENCES

A *reference* is an alias or synonym for another variable. It is declared by the syntax

type`&` *ref-name* `=` *var-name*`;`

where *type* is the variable's type, *ref-name* is the name of the reference, and *var-name* is the name of the variable. For example, in the declaration

```
int& rn=n;  // r is a synonym for n
```

`rn` is declared to be a reference to the variable `n`, which must already have been declared.

EXAMPLE 7.2 Using References

This declares `rn` as a reference to `n`:

```
int main()
{ int n=44;
  int& rn=n;  // r is a synonym for n
  cout << "n = " << n << ", rn = " << rn << endl;
  --n;
  cout << "n = " << n << ", rn = " << rn << endl;
  rn *= 2;
  cout << "n = " << n << ", rn = " << rn << endl;
}
```

```
n = 44, rn = 44
n = 43, rn = 43
n = 86, rn = 86
```

The two identifiers `n` and `rn` are different names for the same variable; they always have the same value. Decrementing `n` changes both `n` and `nr` to 32. Doubling `rn` increases both `n` and `rn` to 64.

Like constants, references must be initialized when they are declared. But unlike a constant, a reference must be initialized to a variable, not a literal:

```
int& rn=44;  // ERROR: 44 is not a variable!
```

(Some compilers may allow this, issuing a warning that a temporary variable had to be created to allocate memory to which the reference `rn` can refer.)

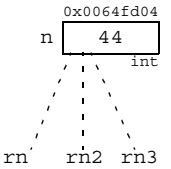
Although a reference must be initialized to a variable, references are not variables. A variable is an object; *i.e.*, a block of contiguous bytes in memory used to store accessible information. Different objects must occupy disjoint blocks of memory.

EXAMPLE 7.3 References Are Not Separate Variables

```
int main()
{ int n=44;
  int& rn=n;  // r is a synonym for n
  cout << "  &n = " << &n << ",  &rn = " << &rn << endl;
  int& rn2=n;   // r is another synonym for n
  int& rn3=rn;  // r is another synonym for n
  cout << "&rn2 = " << &rn2 << ", &rn3 = " << &rn3 << endl;
}
```

```
  &n = 0x0064fde4,  &rn = 0x0064fde4
&rn2 = 0x0064fde4, &rn3 = 0x0064fde4
```

The first line of output shows that `n` and `rn` have the same address: `0x0064fde4`. Thus they are merely different names for the same object. The second line of output shows that an object can have several references, and that a reference to a reference is the same as a reference to the object to which it refers. In this program, there is only one object: an **`int`** named `n` with address `0x0064fde4`. The names `rn`, `rn2`, and `rn3` are all references to that same object.



In C++, the reference operator **`&`** is used for two distinct purposes. When applied as a prefix to the name of an object, it forms an expression that evaluates to the address of the object. When applied as a suffix to a type `T`, it names the derived type "reference to `T`". For example, `int&` is the type "reference to **`int`**". So in Example 7.3, `n` is declared to have type **`int`** and `rn` is declared to have type reference to **`int`**.

C++ actually has five kinds of derived types. If `T` is a type, then **`const T`** is the derived type "constant `T`", **`T()`** is the derived type "function returning `T`", **`T[]`** is the derived type "array of `T`", **`T&`** is the derived type "reference to `T`", and **`T*`** is the derived type "pointer to `T`".

References are used mostly for reference parameters (See Section 5.10 on page 102.). We see now that they work the same way as reference variables: they are merely synonyms for other variables. Indeed, a reference parameter for a function is really just a reference variable whose scope is limited to the function.

7.3 POINTERS

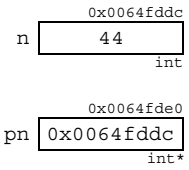
The reference operator **`&`** returns the memory address of the variable to which it is applied. We used this in Example 7.1 on page 157 to print the address. We can also store the address in another variable. The type of the variable that stores an address is called a *pointer*. Pointer variables have the derived type "pointer to `T`", where `T` is the type of the object to which the pointer points. As mentioned in Section 7.2, that derived type is denoted by **`T*`**. For example, the address of an **`int`** variable can be stored in a pointer variable of type **`int*`**.

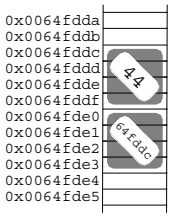
EXAMPLE 7.4 Using Pointer Variables

This program defines the **int** variable n and the **int*** variable pn:

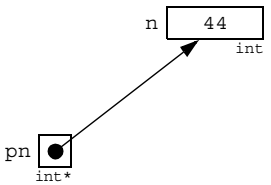
```
int main()
{ int n=44;
  cout << "n = " << n << ", &n = " << &n << endl;
  int* pn=&n;  // pn holds the address of n
  cout << "        pn = " << pn << endl;
  cout << "&pn = " << &pn << endl;
}
```

```
n = 44, &n = 0x0064fddc
        pn = 0x0064fddc
&pn = 0x0064fde0
```





The variable n is initialized to 44. Its address is 0x0064fddc. The variable pn is initialized to &n which is the address of n, so the value of pn is 0x0064fddc, as the second line of output shows. But pn is a separate object, as the third line of output shows: it has the distinct address 0x0064fde0.



The variable pn is called a "pointer" because its value "points" to the location of another value. The value of a pointer is an address. That address depends upon the state of the individual computer on which the program is running. In most cases, the actual value of that address (here, 0x0064fddc) is not relevant to the issues that concern the programmer. So diagrams like the one above are usually drawn more simply like this. This captures the essential features of n and pn: pn is a pointer to n, and n has the value 44. A pointer can be thought of as a "locator": it locates another object.

7.4 THE DEREFERENCE OPERATOR

If pn points to n, we can obtain the value of n directly from p; the expression *pn evaluates to the value of n. This evaluation is called "dereferencing the pointer" pn, and the symbol * is called the *dereference operator.*

EXAMPLE 7.5 Dereferencing a Pointer

This is the same program as in Example 7.4 with one more line of code:

```
int main()
{ int n=44;
  cout << "n = " << n << ", &n = " << &n << endl;
  int* pn=&n;  // pn holds the address of n
  cout << "        pn = " << pn << endl;
  cout << "&pn = " << &pn << endl;
  cout << "*pn = " << *pn << endl;
}
```

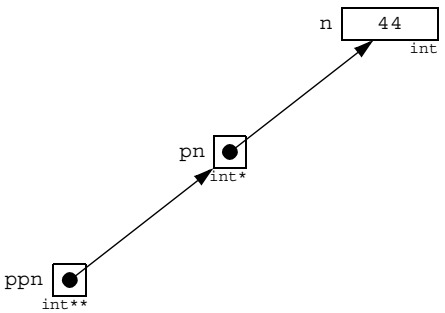
```
n = 44, &n = 0x0064fdcc
        pn = 0x0064fdcc
&pn = 0x0064fdd0
*pn = 44
```

This shows that `*pn` is an alias for `n`: they both have the value 44.

EXAMPLE 7.6 Pointers to Pointers

```
This continues to build upon the program from Example 7.4:
int main()
{ int n=44;
  cout << "    n = " << n << endl;
  cout << "   &n = " << &n << endl;
  int* pn=&n;  // pn holds the address of n
  cout << "   pn = " << pn << endl;
  cout << "  &pn = " << &pn << endl;
  cout << "  *pn = " << *pn << endl;
  int** ppn=&pn;  // ppn holds the address of pn
  cout << "  ppn = " << ppn << endl;
  cout << " &ppn = " << &ppn << endl;
  cout << " *ppn = " << *ppn << endl;
  cout << "**ppn = " << **ppn << endl;
}
```

```
    n = 44
   &n = 0x0064fd78
   pn = 0x0064fd78
  &pn = 0x0064fd7c
  *pn = 44
  ppn = 0x0064fd7c
 &ppn = 0x0064fd80
 *ppn = 0x0064fd78
**ppn = 44
```



The variable `ppn` points to `pn` which points to `n`. So `*ppn` is an alias for `pn`, just as `*pn` is an alias for `n`. Therefore `**ppn` is also an alias for `n`.

Note in Example 7.6 that each of the three variables `n`, `pn`, and `ppn`, has a different type: `int`, `int*`, and `int**`. In general, if `T1` and `T2` are different types, then any of their derived types will also be different. So although `pn` and `ppn` are both pointers, they are not the same type: `pn` has type pointer to `int`, while `ppn` has type pointer to **`int*`**.

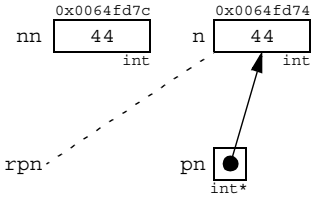
The reference operator **`&`** and the dereference operator **`*`** are inverses: `n == *p` whenever `p == &n`. This can also be expressed as `*&n == n` and `&*p == p`.

EXAMPLE 7.7 Referencing Is the Opposite of Dereferencing

```
This also builds upon the program from Example 7.4:
int main()
{ int n=44;
  cout << "    n = " << n << endl;
  cout << "   &n = " << &n << endl;
  int* pn=&n;  // pn holds the address of n
```

```
  cout << "   pn = " << pn << endl;
  cout << "  &pn = " << &pn << endl;
  cout << "  *pn = " << *pn << endl;
  int nn=*pn;  // nnn is a duplicate of n
  cout << "   nn = " << nn << endl;
  cout << "  &nn = " << &nn << endl;
  int& rpn=*pn;  // rpn is a reference for n
  cout << "  rpn = " << rpn << endl;
  cout << " &rpn = " << &rpn << endl;
}
```

```
    n = 44
   &n = 0x0064fd74
   pn = 0x0064fd74
  &pn = 0x0064fd78
  *pn = 44
   nn = 44
  &nn = 0x0064fd7c
  rpn = 44
 &rpn = 0x0064fd74
```



Here `p` points to the integer named `np` and `rnp` is a reference that is initialized to the value to which `pn` points. So `pn` references `n` and `rpn` dereferences `pn`. Therefore `rpn` is an alias for `n`; *i.e.*, they are different names for the same object. The output verifies this: `n` and `rpn` have the same address: `0x0064fd74`.

7.5 DERIVED TYPES

Like the reference operator **`&`**, the dereference operator **`*`** is used for two distinct purposes. When applied as a prefix to a pointer to an object, it forms an expression that evaluates to that object's value. When applied as a suffix to a type `T`, it names the derived type "pointer to `T`". For example, `int*` is the type "pointer to **`int`**".

As mentioned above, there are five kinds of *derived types* in C++. Here are some declarations of derived types:

```
const int C = 33;            // const int
int& rn = n;                 // reference to int
int* pn = &n;                // pointer to int
int a[] = { 33, 66 };        // array of int
int f() = { return 33; };    // function returning int
```

A derived type can derive from any other type. So many combinations are possible:

```
int* const Pn=44;            // constant pointer to an int
const int* pN=&N;            // pointer to a constant int
const int* const PN=&N;      // constant pointer to a constant int
float& ar[] = { x, y };      // array of 2 references to floats
float* ap[] = { &x, &y };    // array of 2 pointers to floats
long& r() { return n; }      // function returning reference to long
long* p() { return &n; }     // function returning pointer to long
long (*pf)() { return 44; }  // pointer to function returning long
```

Some derived types require the assistance of `typedef`s:

```
typedef char Word[255];      // type array of 255 chars
Word& pa=a;                  // reference to an array of 255 chars
Word* pa=&a;                 // pointer to an array of 255 chars
```

7.6 OBJECTS AND LVALUES

The Annotated C++ Reference Manual **[Ellis]** states: "An *object* is a region of storage. An *lvalue* is an expression referring to an object or function." Originally, the terms "lvalue" and "rvalue" referred to things that appeared on the left and right sides of assignments. But now "lvalue" is more general.

The simplest examples of lvalues are names of objects, *i.e.*, variables:

```
int n;
n = 44;  // n is an lvalue
```

The simplest examples of things that are not lvalues are literals:

```
44 = n;  // ERROR: 44 is not an lvalue
```

But symbolic constants are lvalues:

```
const int MAX = 65535;  // MAX is an lvalue
```

even though they cannot appear on the left side of an assignment:

```
MAX = 21024;  // ERROR: MAX is constant
```

Lvalues that can appear on the left side of an assignment are called *mutable lvalues*; those that cannot are called *immutable lvalues*. A variable is a mutable lvalue; a constant is an immutable lvalue. Other examples of mutable lvalues include subscripted variables and dereferenced pointers:

```
int a[8];
a[5] = 22;   // a[5] is a mutable lvalue
int* p = &n;
*p = 77;     // *p is a mutable lvalue
```

Other examples of immutable lvalues include arrays, functions, and references.

In general, an lvalue is anything whose address is accessible. Since an address is what a reference variable needs when it is declared, the C++ syntax requirement for such a declaration specifies an lvalue:

```
type& refname = lvalue;
```

For example, this is a legal declaration of a reference:

```
int& r = n;          //  OK: n is an lvalue
```

but these are illegal:

```
int& r = 44;         //  ERROR: 44 is not an lvalue
int& r = n++;        //  ERROR: n++ is not an lvalue
int& r = cube(n);    //  ERROR: cube(n) is not an lvalue
```

7.7 RETURNING A REFERENCE

A function's return type may be a reference provided that the value returned is an lvalue which is not local to the function. This restriction means that the returned value is actually a reference to an lvalue that exists after the function terminates. Consequently that returned lvalue may be used like any other lvalue; for example, on the left side of an assignment:

EXAMPLE 7.8 Returning a Reference

```
int& max(int& m, int& n)   // return type is reference to int
{ return (m > n ? m : n);  // m and n are non-local references
}
```

```
int main()
{ int m = 44, n = 22;
  cout << m << ", " << n << ", " << max(m,n) << endl;
  max(m,n) = 55;          // changes the value of m from 44 to 55
  cout << m << ", " << n << ", " << max(m,n) << endl;
}
```

```
44, 22, 44
55, 22, 55
```

The `max()` function returns a reference to the larger of the two variables passed to it. Since the return value is a reference, the expression `max(m,n)` acts like a reference to `m` (since `m` is larger than `n`). So assigning 55 to the expression `max(m,n)` is equivalent to assigning it to `m` itself.

EXAMPLE 7.9 Using a Function as an Array Subscript

```
float& component(float* v, int k)
{ return v[k-1];
}

int main()
{ float v[4];
  for (int k = 1; k <= 4; k++)
    component(v,k) = 1.0/k;
  for (int i = 0; i < 4; i++)
    cout << "v[" << i << "] = " << v[i] << endl;
}
```

```
v[0] = 1
v[1] = 0.5
v[2] = 0.333333
v[3] = 0.25
```

The `component()` function allows vectors to be accessed using the scientific "1-based indexing" instead of the default "0-based indexing." So the assignment `component(v,k) = 1.0/k` is really the assignment `v[k+1] = 1.0/k`. We'll see a better way to do this in Chapter 10.

7.8 ARRAYS AND POINTERS

Although pointer types are not integer types, some integer arithmetic operators can be applied to pointers. The affect of this arithmetic is to cause the pointer to point to another memory location. The actual change in address depends upon the size of the fundamental type to which the pointer points.

Pointers can be incremented and decremented like integers. However, the increase or decrease in the pointer's value is equal to the size of the object to which it points:

EXAMPLE 7.10 Traversing an Array with a Pointer

This example shows how a pointer can be used to traverse an array.

```
int main()
{ const int SIZE = 3;
  short a[SIZE] = {22, 33, 44};
```

```
  cout << "a = " << a << endl;
  cout << "sizeof(short) = " << sizeof(short) << endl;
  short* end = a + SIZE;       // converts SIZE to offset 6
  short sum = 0;
  for (short* p = a; p < end; p++)
  { sum += *p;
    cout << "\t p = " << p;
    cout << "\t *p = " << *p;
    cout << "\t sum = " << sum << endl;
  }
  cout << "end = " << end << endl;
}
```

```
  a = 0x3fffd1a
  sizeof(short) = 2
           p = 0x3fffd1a   *p = 22         sum = 22
           p = 0x3fffd1c   *p = 33         sum = 55
           p = 0x3fffd1e   *p = 44         sum = 99
  end = 0x3fffd20
```

The second line of output shows that on this machine `short` integers occupy 2 bytes. Since `p` is a pointer to `short`, each time it is incremented it advances 2 bytes to the next `short` integer in the array. That way, `sum += *p` accumulates their sum of the integers. If `p` were a pointer to `double` and `sizeof(double)` were 8 bytes, then each time `p` is incremented it would advance 8 bytes.

Example 7.10 shows that when a pointer is incremented, its value is increased by the number `SIZE` (in bytes) of the object to which it points. For example,

```
float a[8];
float* p = a;       //  p points to a[0]
++p;                // increases the value of p by sizeof(float)
```

If `float`s occupy 4 bytes, then `++p;` increases the value of `p` by 4, and `p += 5;` increases the value of `p` by 20. This is how an array can be traversed: by initializing a pointer to the first element of the array and then repeatedly incrementing the pointer. Each increment moves the pointer to the next element of the array.

We can also use a pointer for direct access into the array. For example, we can access `a[5]` by initializing the pointer to `a[0]` and then adding 5 to it:

```
float* p = a;       //  p points to a[0]
p += 5;             //  now p points to a[5]
```

So once the pointer is initialized to the starting address of the array, it works like an index.

Warning: In C++ it is possible to access and even modify unallocated memory locations. This is risky and should generally be avoided. For example,

```
float a[8];
float* p = a[7];    // p points to last element in the array
++p;                // now p points to memory past last element!
*p = 22.2;          // TROUBLE!
```

The next example shows an even tighter connection between arrays and pointers: the name of an array itself is a `const` pointer to the first element of the array. It also shows that pointers can be compared.

EXAMPLE 7.11 Examining the Addresses of Array Elements

```
int main()
{ short a[] = {22, 33, 44, 55, 66};
  cout << "a = " << a << ", *a = " << *a << endl;
  for (short* p = a; p < a + 5; p++)
    cout << "p = " << p << ", *p = " << *p << endl;
}
```

```
a = 0x3fffd08, *a = 22
p = 0x3fffd08, *p = 22
p = 0x3fffd0a, *p = 33
p = 0x3fffd0c, *p = 44
p = 0x3fffd0e, *p = 55
p = 0x3fffd10, *p = 66
```

Initially, `a` and `p` are the same: they are both pointers to `short` and they have the same value (`0x3fffd08`). Since `a` is a constant pointer, it cannot be incremented to traverse the array. Instead, we increment `p` and use the exit condition `p < a + 5` to terminate the loop. This computes `a + 5` to be the hexadecimal address `0x3fffd08 + 5*sizeof(short) = 0x3fffd08 + 5*2 = 0x3fffd08 + 0xa = 0x3fffd12`, so the loop continues as long as `p < 0x3fffd12`.

The array subscript operator `[]` is equivalent to the dereference operator `*`. They provide direct access into the array the same way:

```
a[0] == *a
a[1] == *(a + 1)
a[2] == *(a + 2), etc.
```

So the array a could be traversed like this:

```
for (int i = 0; i < 8; i++)
cout << *(a + i) << endl;
```

The next example illustrates how pointers can be combined with integers to move both forward and backward in memory.

EXAMPLE 7.12 Pattern Matching

In this example, the `loc` function searches through the first `n1` elements of array `a1` looking for the string of integers stored in the first `n2` elements of array `a2` inside it. If found, it returns a pointer to the location within `a1` where `a2` begins; otherwise it returns the `NULL` pointer.

```
short* loc(short* a1, short* a2, int n1, int n2)
{ short* end1 = a1 + n1;
  for (short* p1 = a1; p1 < end1; p1++)
    if (*p1 == *a2)
    { int j;
      for (j = 0; j < n2; j++)
        if (p1[j] != a2[j]) break;
      if (j == n2) return p1;
    }
  return 0;
}

int main()
{ short a1[9] = {11, 11, 11, 11, 11, 22, 33, 44, 55};
```

```
  short a2[5] = {11, 11, 11, 22, 33};
  cout << "Array a1 begins at location\t" << a1 << endl;
  cout << "Array a2 begins at location\t" << a2 << endl;
  short* p = loc(a1, a2, 9, 5);
  if (p)
  { cout << "Array a2 found at location\t" << p << endl;
    for (int i = 0; i < 5; i++)
      cout << "\t" << &p[i] << ": " << p[i]
           << "\t" << &a2[i] << ": " << a2[i] << endl;
  }
  else cout << "Not found.\n";
}
```

```
  Array a1 begins at location     0x3fffd12
  Array a2 begins at location     0x3fffd08
  Array a2 found at location      0x3fffd16
          0x3fffd16: 11   0x3fffd08: 11
          0x3fffd18: 11   0x3fffd0a: 11
          0x3fffd1a: 11   0x3fffd0c: 11
          0x3fffd1c: 22   0x3fffd0e: 22
          0x3fffd1e: 33   0x3fffd10: 33
```

The pattern matching algorithm uses two loops. The outer loop is controlled by the pointer `p1` which points to elements in array `a1` where the inner loop will begin checking for a match with array `a2`. The inner loop is controlled by the integer `j` which is used to compare corresponding elements of the two arrays. If a mismatch is found, the inner loop aborts and the outer loop continues by incrementing `p1` to look for a match starting with the next element of `a1`. If the inner loop is allowed to finish, then the condition `(j == n2)` will be true and the current location pointed to by `p1` is returned.

The test driver verifies that the match has indeed been found by checking the actual addresses.

EXAMPLE 7.13 THE `new` OPERATOR

When a pointer is declared like this:

```
float* p;  //  p is a pointer to a float
```

it only allocates memory for the pointer itself. The value of the pointer will be some memory address, but the memory at that address is not yet allocated. This means that storage could already be in use by some other variable. In this case, `p` is uninitialized: it is not pointing to any allocated memory. Any attempt to access the memory to which it points will be an error:

```
*p = 3.14159;   // ERROR: no storage has been allocated for *P
```

A good way to avoid this problem is to initialize pointers when they are declared:

```
float x = 3.14159;  // x contains the value 3.14159
float* p = &x;      // p contains the address of x
cout << *p;         // OK: *p has been allocated
```

In this case, accessing `*p` is no problem because the memory needed to store the float 3.14159 was automatically allocated when `x` was declared; `p` points to the same allocated memory.

Another way to avoid the problem of a dangling pointer is to allocate memory explicitly for the pointer itself. This is done with the `new` operator:

```
float* q;
q = new float;      // allocates storage for 1 float
*q = 3.14159;       // OK: *q has been allocated
```

The `new` operator returns the address of a block of *s* unallocated bytes in memory, where *s* is the size of a float. (Typically, `sizeof(float)` is 4 bytes.) Assigning that address to `q` guarantees that `*q` is not currently in use by any other variables.

The first two of these lines can be combined, thereby initializing `q` as it is declared:

```
float* q = new float;
```

Note that using the `new` operator to initialize `q` only initializes the pointer itself, not the memory to which it points. It is possible to do both in the same statement that declares the pointer:

```
float* q = new float(3.14159);
cout << *q;          // ok: both q and *q have been initialized
```

In the unlikely event that there is not enough free memory to allocate a block of the required size, the `new` operator will return `0` (the `NULL` pointer):

```
double* p = new double;
if (p == 0) abort();  // allocator failed: insufficient memory
else *p = 3.141592658979324;
```

This prudent code calls an `abort()` function to prevent dereferencing the `NULL` pointer.

Consider again the two alternatives to allocating memory:

```
float x = 3.14159;              // allocates named memory
float* p = new float(3.14159);  // allocates unnamed memory
```

In the first case, memory is allocated at compile time to the named variable `x`. In the second case, memory is allocated at run time to an unnamed object that is accessible through `*p`.

EXAMPLE 7.14 THE `delete` OPERATOR

The `delete` operator reverses the action of the `new` operator, returning allocated memory to the free store. It should only be applied to pointers that have been allocated explicitly by the `new` operator:

```
float* q = new float(3.14159);
delete q;             // deallocates q
*q = 2.71828;         // ERROR: q has been deallocated
```

Deallocating `q` returns the block of `sizeof(float)` bytes to the free store, making it available for allocation to other objects. Once `q` has been deallocated, it should not be used again until after it has been reallocated. A deallocated pointer, also called a *dangling pointer*, is like an uninitialized pointer: it doesn't point to anything.

A pointer to a constant cannot be deleted:

```
const int * p = new int;
delete p;           // ERROR: cannot delete pointer to const
```

This restriction is consistent with the general principle that constants cannot be changed.

Using the `delete` operator for fundamental types (`char`, `int`, `float`, `double`, *etc.*) is generally not recommended because little is gained at the risk of a potentially disastrous error:

```
float x = 3.14159;  // x contains the value 3.14159
float* p = &x;      // p contains the address of x
delete p;           // RISKY: p was not allocated by new
```

This would deallocate the variable `x`, a mistake that can be very difficult to debug.

7.9 DYNAMIC ARRAYS

An array name is really just a constant pointer that is allocated at compile time:

```
float a[20];        // a is a const pointer to a block of 20 floats
float* const p = new float[20];   // so is p
```

Here, both `a` and `p` are constant pointers to blocks of 20 floats. The declaration of `a` is called *static binding* because it is allocated at compile time; the symbol is bound to the allocated memory even if the array is never used while the program is running.

In contrast, we can use a non-constant pointer to postpone the allocation of memory until the program is running. This is generally called *run-time binding* or *dynamic binding*:

```
float* p = new float[20];
```

An array that is declared this way is called a *dynamic array.*

Compare the two ways of defining an array:

```
float a[20];               // static array
float* p = new float[20];  // dynamic array
```

The static array `a` is created at compile time; its memory remains allocated throughout the run of the program. The dynamic array `p` is created at run time; its memory allocated only when its declaration executes. Furthermore, the memory allocated to the array `p` is deallocated as soon as the `delete` operator is invoked on it:

```
delete [] p;               // deallocates the array p
```

Note that the subscript operator `[]` must be included this way, because `p` is an array.

EXAMPLE 7.15 Using Dynamic Arrays

The `get()` function here creates a dynamic array:

```
void get(double*& a, int& n)
{ cout << "Enter number of items: ";  cin >> n;
  a = new double[n];
  cout << "Enter " << n << " items, one per line:\n";
  for (int i = 0; i < n; i++)
  { cout << "\t" << i+1 << ": ";
    cin >> a[i];
  }
}
void print(double* a, int n)
{ for (int i = 0; i < n; i++)
  cout << a[i] << " ";
  cout << endl;
}
int main()
{ double* a;    // a is simply an unallocated pointer
  int n;
  get(a,n);     // now a is an array of n doubles
  print(a,n);
  delete [] a;  // now a is simply an unallocated pointer again
  get(a,n);     // now a is an array of n doubles
  print(a,n);
}
```

```
Enter number of items: 4
Enter 4 items, one per line:
            1: 44.4
            2: 77.7
            3: 22.2
            4: 88.8
44.4 77.7 22.2 88.8
Enter number of items: 2
Enter 2 items, one per line:
            1: 3.33
            2: 9.99
3.33 9.99
```

Inside the `get()` function, the `new` operator allocates storage for `n doubles` after the value of `n` is obtained interactively. So the array is created "on the fly" while the program is running.

Before `get()` is used to create another array for `a`, the current array has to be deallocated with the `delete` operator. Note that the subscript operator `[]` must be specified when deleting an array.

Note that the array parameter `a` is *a pointer that is passed by reference*:

```
void get(double*& a, int& n)
```

This is necessary because the `new` operator will change the value of `a` which is the address of the first element of the newly allocated array.

7.10 USING `const` WITH POINTERS

A pointer to a constant is different from a constant pointer. This distinction is illustrated in the following example.

EXAMPLE 7.16 Constant Pointers and Pointers to Constants

This fragment declares four variables: a pointer `p`, a constant pointer `cp`, a pointer `pc` to a constant, and a constant pointer `cpc` to a constant:

```
int n = 44;                  // an int
int* p = &n;                 // a pointer to an int
++(*p);                      // ok: increments int *p
++p;                         // ok: increments pointer p
int* const cp = &n;          // a const pointer to an int
++(*cp);                     // ok: increments int *cp
++cp;                        // illegal: pointer cp is const
const int k = 88;            // a const int
const int * pc = &k;         // a pointer to a const int
++(*pc);                     // illegal: int *pc is const
++pc;                        // ok: increments pointer pc
const int* const cpc = &k;   // a const pointer to a const int
++(*cpc);                    // illegal: int *cpc is const
++cpc;                       // illegal: pointer cpc is const
```

Note that the reference operator `*` may be used in a declaration with or without a space on either side. Thus, the following three declarations are equivalent:

```
int* p;    // indicates that p has type int* (pointer to int)
int * p;   // style sometimes used for clarity
int *p;    // old C style
```

7.11 ARRAYS OF POINTERS AND POINTERS TO ARRAYS

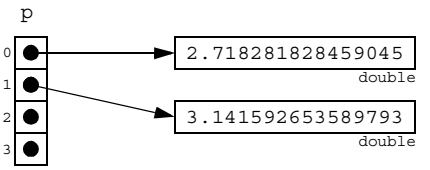
The elements of an array may be pointers. Here is an array of 4 pointers to type `double`:

```
double* p[4];
```

Its elements can allocated like any other pointer:

```
p[0] = new double(2.718281828459045);
p[1] = new double(3.141592653589793);
```

We can visualize this array like this.



The next example illustrates a useful application of pointer arrays. It shows how to sort a list indirectly by changing the pointers to the elements instead of moving the elements themselves. This is equivalent to the Indirect Bubble Sort shown in Problem 5.12.

EXAMPLE 7.17 Indirect Bubble Sort

```
void sort(float* p[], int n)
{ float* temp;
  for (int i = 1; i < n; i++)
    for (int j = 0; j < n-i; j++)
      if (*p[j] > *p[j+1])
      { temp = p[j];
        p[j] = p[j+1];
        p[j+1] = temp;
      }
}
```

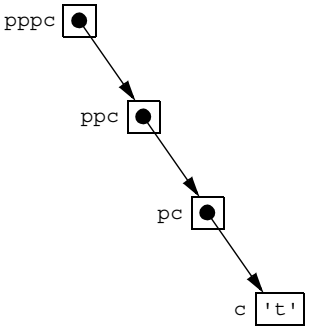
On each iteration of the inner loop, if the `floats` of adjacent pointers are out of order, then the pointers are swapped.

7.12 POINTERS TO POINTERS

A pointer may point to another pointer. For example,

```
char c = 't';
char* pc = &c;
char** ppc = &pc;
char*** pppc = &ppc;
***pppc = 'w';  // changes value of c to 'w'
```

We can visualize these variables like this:



The assignment `***pppc = 'w'` refers to the contents of the address `pc` that is pointed to by the address `ppc` that is pointed to by the address `pppc`.

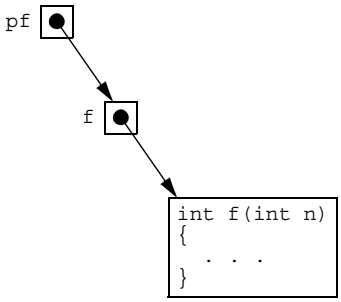
7.13 POINTERS TO FUNCTIONS

Like an array name, a function name is actually a constant pointer. We can think of its value as the address of the code that implements the function.

A pointer to a function is simply a pointer whose value is the address of the function name. Since that name is itself a pointer, a pointer to a function is just a pointer to a constant pointer. For example,

```
int f(int);      // declares function f
int (*pf)(int); // declares function pointer pf
pf = &f;         // assigns address of f to pf
```

We can visualize the function pointer like this:



The value of function pointers is that they allow us to define functions of functions. This is done by passing a function pointer as a parameter to another function.

EXAMPLE 7.18 The Sum of a Function

The `sum()` function has two parameters: the function pointer `pf` and the integer `n`:

```
int sum(int (*)(int), int);
int square(int);
int cube(int);

int main()
{ cout << sum(square,4) << endl;     // 1 + 4 + 9 + 16
  cout << sum(cube,4) << endl;       // 1 + 8 + 27 + 64
}
```

The call `sum(square,4)` computes and returns the sum `square(1) + square(2) + square(3) + square(4)`. Since `square(k)` computes and returns `k*k`, the `sum()` function returns `1 + 4 + 9 + 16 = 30`.

Here are the function definitions and the output:

```
int sum(int (*pf)(int k), int n)
{ // returns the sum f(0) + f(1) + f(2) + . . . + f(n-1):
  int s = 0;
  for (int i = 1; i <= n; i++)
    s += (*pf)(i);
  return s;
}

int square(int k)
{ return k*k;
}

int cube(int k)
{ return k*k*k;
}
```

```
30
100
```

The `sum()` function evaluates the function to which `pf` points, at each of the integers `1` through `n`, and returns the sum of these `n` values.

Note that the declaration of the function pointer parameter `pf` in the `sum()` function's parameter list requires the dummy variable `k`.

7.14 NUL, NULL, AND void

The constant 0 (zero) has type int. Nevertheless, this symbol can be assigned to all the fundamental types:

```
char c = 0;       // initializes c to the char '\0'
short d = 0;      // initializes d to the short int 0
int n = 0;        // initializes n to the int 0
unsigned u = 0;   // initializes u to the unsigned int 0
float x = 0;      // initializes x to the float 0.0
double z = 0;     // initializes z to the double 0.0
```

In each case, the object is initialized to the number 0. In the case of type char, the character c becomes the *null character*; denoted by '\0' or NUL, it is the character whose ASCII code is 0.

The values of pointers are memory addresses. These addresses must remain within that part of memory allocated to the executing process, with the exception of the address 0x0. This is called the NULL pointer. The same constant applies to pointers derived from any type:

```
char* pc = 0;       // initializes pc to NULL
short* pd = 0;      // initializes pd to NULL
int* pn = 0;        // initializes pn to NULL
unsigned* pu = 0;   // initializes pu to NULL
float* px = 0;      // initializes px to NULL
double* pz = 0;     // initializes pz to NULL
```

The NULL pointer cannot be dereferenced. This is a common but fatal error:

```
int* p = 0;
*p = 22;          // ERROR: cannot dereference the NULL pointer
```

A reasonable precaution is to test a pointer before attempting to dereference it:

```
if (p) *p = 22;  // ok
```

This tests the condition (p != NULL) because that condition is true precisely when p is nonzero.

The name void denotes a special fundamental type. Unlike all the other fundamental types, void can only be used in a derived type:

```
void x;   // ERROR: no object can have type void
void* p;  // OK
```

The most common use of the type void is to specify that a function does not return a value:

```
void swap(double&, double&);
```

Another, different use of void is to declare a pointer to an object of unknown type:

```
void* p = q;
```

This use is most common in low-level C programs designed to manipulate hardware resources.

Review Questions

7.1 How do you access the memory address of a variable?

7.2 How do you access the contents of the memory location whose address is stored in a pointer variable?

7.3 Explain the difference between the following two declarations:

```
int n1=n;
int& n2=n;
```

7.4 Explain the difference between the following two uses of the reference operator `&`:

```
int& r = n;
p = &n;
```

7.5 Explain the difference between the following two uses of the indirection operator `*`:

```
int* q = p;
n = *p;
```

7.6 True or false? Explain:

a. If `(x == y)` then `(&x == &y)`.

b. If `(x == y)` then `(*x == *y)`.

7.7 ***a.*** What is a "dangling pointer"?

b. What dire consequences could result from dereferencing a dangling pointer?

c. How can these dire consequences be avoided?

7.8 What is wrong with the following code:

```
int& r = 22;
```

7.9 What is wrong with the following code:

```
int* p = &44;
```

7.10 What is wrong with the following code:

```
char c = 'w';
char p = &c;
```

7.11 Why couldn't the variable `ppn` in Example 7.6 on page 160 be declared like this:

```
int** ppn=&&n;
```

7.12 What is the difference between "static binding" and "dynamic binding"?

7.13 What is wrong with the following code:

```
char c = 'w';
char* p = c;
```

7.14 What is wrong with the following code:

```
short a[32];
for (int i = 0; i < 32; i++)
*a++ = i*i;
```

7.15 Determine the value of each of the indicated variables after the following code executes. Assume that each integer occupies 4 bytes and that `m` is stored in memory starting at byte `0x3fffd00`.

```
int m = 44;
int* p = &m;
int& r = m;
int n = (*p)++;
int* q = p - 1;
r = *(--p) + 1;
++*q;
```

a. `m`

b. `n`

c. `&m`

d. `*p`

e. `r`

f. `*q`

7.16 Classify each of the following as a mutable lvalue, an immutable lvalue, or a non-lvalue:

a. `double x = 1.23;`

b. `4.56*x + 7.89`

c. `const double Y = 1.23;`

d. `double a[8] = {0.0};`
e. `a[5]`
f. `double f() { return 1.23; }`
g. `f(1.23)`
h. `double& r = x;`
i. `double* p = &x;`
j. `*p`
k. `const double* p = &x;`
l. `double* const p = &x;`

7.17 What is wrong with the following code:

```
float x = 3.14159;
float* p = &x;
short d = 44;
short* q = &d;
p = q;
```

7.18 What is wrong with the following code:

```
int* p = new int;
int* q = new int;
cout << "p = " << p << ", p + q = " << p + q << endl;
```

7.19 What is the only thing that you should ever do with the `NULL` pointer?

7.20 In the following declaration, explain what type `p` is, and describe how it might be used:

```
double**** p;
```

7.21 If `x` has the address `0x3fffd1c`, then what will values of `p` and `q` be for each of the following:

```
double x = 1.01;
double* p = &x;
double* q = p + 5;
```

7.22 If `p` and `q` are pointers to `int` and `n` is an `int`, which of the following are legal:

a. `p + q`
b. `p - q`
c. `p + n`
d. `p - n`
e. `n + p`
f. `n - q`

7.23 What does it mean to say that an array is really a constant pointer?

7.24 How is it possible that a function can access every element of an array when it is passed only the address of the first element?

7.25 Explain why the following three conditions are true for an array `a` and an int `i`:

```
a[i] == *(a + i);
*(a + i) == i[a];
a[i] == i[a];
```

7.26 Explain the difference between the following two declarations:

```
double * f();
double (* f)();
```

7.27 Write a declaration for each of the following:

a. an array of 8 `float`s;
b. an array of 8 pointers to `float`;
c. a pointer to an array of 8 `float`s;
d. a pointer to an array of 8 pointers to `float`;

e. a function that returns a `float`;
f. a function that returns a pointer to a `float`;
g. a pointer to a function that returns a `float`;
h. a pointer to a function that returns a pointer to a `float`;

Problems

7.1 Write a function that uses pointers to copy an array of `double`.

7.2 Write a function that uses pointers to search for the address of a given integer in a given array. If the given integer is found, the function returns its address; otherwise it returns `NULL`.

7.3 Write a function that is passed an array of `n` pointers to `float`s and returns a newly created array that contains those `n float` values.

7.4 Implement a function for integrating a function by means of Riemann sums. Use the formula

$$\int_a^b f(x)dx = \sum_{j=1}^{n} f(a+jh)h$$

7.5 Write a function that returns the *numerical derivative* of a given function *f* at a given point *x*, using a given tolerance *h*. Use the formula

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h}$$

7.6 Write a function that is passed an array of `n` pointers to `float`s and returns a pointer to the maximum of the `n float`s.

7.7 Write the following function that is passed an array of `n` pointers to `float`s and returns a newly created array that contains those `n float` values in reverse order.

```
float* mirror(float* p[], int n)
```

7.8 Write the following function that returns the number of bytes that `s` has to be incremented before it points to the null character `'\0'`:

```
unsigned len(const char* s)
```

7.9 Write the following function that copies the first *n* bytes beginning with `*s2` into the bytes beginning with `*s1`, where *n* is the number of bytes that `s2` has to be incremented before it points to the null character `'\0'`:

```
void cpy(char* s1, const char* s2)
```

7.10 Write the following function that copies the first *n* bytes beginning with `*s2` into the bytes beginning at the location of the first occurrence of the null character `'\0'` after `*s1`, where *n* is the number of bytes that `s2` has to be incremented before it points to the null character `'\0'`:

```
void cat(char* s1, const char* s2)
```

7.11 Write the following function that compares at most *n* bytes beginning with `s2` with the corresponding bytes beginning with `s1`, where *n* is the number of bytes that `s2` has to be incremented before it points to the null character `'\0'`. If all *n* bytes match, the function should return 0; otherwise, it should return either -1 or 1 according to whether the byte from `s1` is less than or greater than the byte from `s2` at the first mismatch:

```
int cmp(char* s1, char* s2)
```

7.12 Write the following function that searches the *n* bytes beginning with `s` for the character `c`, where *n* is the number of bytes that `s` has to be incremented before it points to the null character `'\0'`. If the character is found, a pointer to it is returned; otherwise return `NULL`:

```
char* chr(char* s, char c)
```

7.13 Write the following function that returns the sum of the `float`s pointed to by the first `n` pointers in the array `p`:

```
float sum(float* p[], int n)
```

7.14 Write the following function that changes the sign of each of the negative `float`s pointed to by the first `n` pointers in the array `p`:

```
void abs(float* p[], int n)
```

7.15 Write the following function that indirectly sorts the `float`s pointed to by the first `n` pointers in the array `p` by rearranging the pointers:

```
void sort(float* p[], int n)
```

7.16 Implement the *Indirect Selection Sort* using an array of pointers. (See Problem 6.19 on page 144 and Example 7.17 on page 170.)

7.17 Implement the *Indirect Insertion Sort*. (See Problem 6.18 on page 144 and Example 7.17 on page 170.)

7.18 Implement the *Indirect Perfect Shuffle*. (See Problem 6.29 on page 145.)

7.19 Rewrite the `sum()` function (Example 7.18 on page 171) so that it applies to functions with return type `double` instead of `int`. Then test it on the `sqrt()` function (defined in `<math.h>`) and the reciprocal function.

7.20 Apply the `riemann()` function (Problem 7.4 on page 173) to the following functions defined in `<math.h>`:

a. `sqrt()`, on the interval [1, 4];

b. `cos()`, on the interval [0, π/2];

c. `exp()`, on the interval [0, 1];

d. `log()`, on the interval [1, *e*].

7.21 Apply the `derivative()` function (Problem 7.5 on page 175) to the following functions defined in `<math.h>`:

a. `sqrt(), at the point x = 4;`

b. `cos(), at the point x = p/6;`

c. `exp(), at the point x = 0;`

d. `log(), at the point x =` 1.

7.22 Write the following function that returns the product of the `n` values $f(1)$, $f(2)$, ..., and $f(n)$. (See Example 7.18 on page 171.)

```
int product(int (*pf)(int k), int n)
```

7.23 Implement the *Bisection Method* for solving equations. Use the following function:

```
double root(double (*pf)(double x), double a, double b, int n)
```

Here, `pf` points to a function `f` that defines the equation $f(x) = 0$ that is to be solved, `a` and `b` bracket the unknown root x (*i.e.*, $a \le x \le b$), and `n` is the number of iterations to use. For example, if $f(x) = x^2 - 2$, then `root(f,1,2,100)` would return 1.414213562373095 (= $\sqrt{2}$), thereby solving the equation $x^2 = 2$. The Bisection Method works by repeatedly bisecting the interval and replacing it with the half that contains the root. It checks the sign of the product $f(a)\,f(b)$ to determine whether the root is in the interval $[a, b]$.

7.24 Implement the *Trapezoidal Rule* for integrating a function. Use the following function:

```
double trap(double (*pf)(double x), double a, double b, int n)
```

Here, `pf` points to the function `f` that is to be integrated, `a` and `b` bracket the interval $[a, b]$ over which f is to be integrated, and `n` is the number of subintervals to use. For example, the

call `trap(square,1,2,100)` would return 1.41421. The Trapezoidal Rule returns the sum of the areas of the *n* trapezoids that would approximate the area under the graph of *f*. For example, if $n = 5$, then it would return the following, where $h = (b–a)/5$, the width of each

$$\frac{h}{2}[f(a) + 2f(a + h) + 2f(a + 2h) + 2f(a + 3h) + 2f(a + 4h) + f(b)]$$

trapezoid.

Answers to Review Questions

7.1 Apply the address operator `&` to the variable `&x`.

7.2 Apply the dereference operator `*` to the variable `*p`.

7.3 The declaration `int n1=n;` defines `n1` to be a clone of `n`; it is a separate object that has the same value as `n`. The declaration `int& n2=n;` defines `n2` to be a synonym of `n`; it is the same object as `n`, with the same address.

7.4 The declaration `int& r = n;` declares `r` to be a reference (alias) for the `int` variable `n`. The assignment `p = &n;` assigns the address of `n` to the pointer `p`.

7.5 The declaration `int* q = p;` declares `q` to be a pointer (memory address) pointing to the same `int` to which `p` points. The assignment `n = *p;` assigns to `n` the `int` to which `p` points.

7.6 ***a.*** True: `&x == x` and `&y == y` because `&x` and `&y` are synonyms for `x` and `y`, respectively; so if `(x == y)` then they all have the same value.

b. False: different objects can have the same value, but different objects have different addresses.

7.7 ***a.*** A "dangling pointer" is a pointer that has not been initialized. It is dangerous because it could be pointing to unallocated memory, or inaccessible memory.

b. If a pointer pointing to unallocated memory is dereferenced, it could change the value of some unidentified variable. If a pointer pointing to inaccessible memory is dereferenced, the program will probably crash (*i.e.*, terminate abruptly).

c. Initialize pointers when they are declared.

7.8 You cannot have a reference to a constant; it's address is not accessible.

7.9 The reference operator `&` cannot be applied to a constant.

7.10 The variable `p` has type `char`, while the expression `&c` has type pointer to `char`. To initialize `p` to `&c`, `p` would have to be declared as type `char*`.

7.11 The declaration is invalid because the expression `&&n` is illegal. The reference operator & can be applied only to objects (variables and class instances). But `&n` is not an object, it is only a reference. References do not have addresses, so `&&n` does not exist.

7.12 Static binding is when memory is allocated at compile time, as with the array declaration:

```
double a[400];
```

Dynamic binding is when memory is allocated at run time, by means of the `new` operator:

```
double* p;
p = new double[400];
```

7.13 The variable `p` has type `char*`, while the expression `c` has type `char`. To initialize `p` to `c`, `p` would have the same type as c: either both `char` or both `char*`.

7.14 The only problem is that the array name a is a constant pointer, so it cannot be incremented. The following modified code would be okay:

```
short a[32];
short* p = a;
for (int i = 0; i < 32; i++)
*p++ = i*i;
```

7.15 ***a.*** `m` = 46

b. `n` = 44

c. `&m` = `0x3fffd00`

d. `*p` = 46
e. `r` = 46
f. `*q` = 46

7.16 ***a.*** mutable lvalue;
b. not an lvalue;
c. immutable lvalue;
d. immutable lvalue;
e. mutable lvalue;
f. immutable lvalue;
g. mutable lvalue if return type is a non-local reference; otherwise not an lvalue;
h. mutable lvalue;
i. mutable lvalue;
j. mutable lvalue, unless `p` points to a constant, in which case `*p` is an immutable lvalue;
k. mutable lvalue;
l. immutable lvalue;

7.17 The pointers `p` and `q` have different types: `p` is pointer to `float` while `q` is pointer to `short`. It is an error to assign the address in one pointer type to a different pointer type.

7.18 It is an error to add two pointers.

7.19 Test it to see if it is `NULL`. In particular, you should never try to dereference it.

7.20 `p` is a pointer to a pointer to a pointer to a pointer to a `double`. It could be used to represent a four-dimensional array.

7.21 The value of `p` is the same as the address of `x`: `0x3fffd1c`. The value of `q` depends upon `sizeof(double)`. If objects of type `double` occupy 8 bytes, then an offset of 8(5) = 40 is added to `p` to give `q` the hexadecimal value `0x3fffd44`.

7.22 The only expressions among these six that are illegal are `p + q` and `n - q`.

7.23 The name of an array is a variable that contains the address of the first element of the array. This address cannot be changed, so the array name is actually a constant pointer.

7.24 In the following code that adds all the elements of the array `a`, each increment of the pointer `p` locates the next element:

```
const SIZE = 3;
short a[SIZE] = {22, 33, 44};
short* end = a + SIZE;  // adds SIZE*sizeof(short) = 6 to a
for (short* p = a; p < end; p++)
sum += *p;
```

7.25 The value `a[i]` returned by the subscripting operator `[]` is the value stored at the address computed from the expression `a + i`. In that expression, `a` is a pointer to its base type `T` and `i` is an `int`, so the offset `i*sizeof(T)` is added to the address `a`. The same evaluation would be made from the expression `i + a` which is what would be used for `i[a]`.

7.26 The declaration `double * f();` declares `f` to be a function that returns a pointer to double. The declaration `double (* f)();` declares `*f` to be a pointer to a function that returns a double.

7.27 ***a.*** `float a[8];`
b. `float* a[8];`
c. `float (* a)[8];`
d. `float* (* a)[8];`
e. `float f();`
f. `float* f();`
g. `float (* f)();`
h. `float* (* f)();`

Solutions to Problems

7.1 The `copy()` function uses the `new` operator to allocate an array of `n doubles`. The pointer `p` contains the address of the first element of that new array, so it can be used for the name of the array, as in `p[i]`. Then after copying the elements of `a` into the new array, `p` is returned by the function

```
double* copy(double a[], int n)
{ double* p = new double[n];
  for (int i = 0; i < n; i++)
    p[i] = a[i];
  return p;
}

void print(double [], int);

int main()
{ double a[8] = {22.2, 33.3, 44.4, 55.5, 66.6, 77.7, 88.8, 99.9};
  print(a, 8);
  double* b = copy(a, 8);
  a[2] = a[4] = 11.1;
  print(a, 8);
  print(b, 8);
}
```

```
   22.2, 33.3, 44.4, 55.5, 66.6, 77.7, 88.8, 99.9
   22.2, 33.3, 11.1, 55.5, 11.1, 77.7, 88.8, 99.9
   22.2, 33.3, 44.4, 55.5, 66.6, 77.7, 88.8, 99.9
```

In this run we initialize `a` as an array of 8 `doubles`. We use a `print()` function to examine the contents of `a`. The `copy()` function is called and its return value is assigned to the pointer `b` which then serves as the name of the new array. Before printing `b`, we change the values of two of `a`'s elements in order to check that `b` is not the same array as `a`, as the last two `print()` calls confirm.

7.2 We use a `for` loop to traverse the array. If the `target` is found at `a[i]`, then its address `&a[i]` is returned. Otherwise, `NULL` is returned:

```
int* location(int a[], int n, int target)
{ for (int i = 0; i < n; i++)
    if (a[i] == target) return &a[i];
  return NULL;
}
```

The test driver calls the function and stores its return address in the pointer `p`. If that is nonzero (*i.e.*, not `NULL`), then it and the `int` to which it points are printed.

```
int main()
{ int a[8] = {22, 33, 44, 55, 66, 77, 88, 99}, * p, n;
  do
  { cin >> n;
    if (p = location(a, 8, n)) cout << p << ", " << *p << endl;
    else cout << n << " was not found.\n";
  } while (n > 0);
}
```

```
44
0x3fffcc4, 44
50
50 was not found.
99
0x3fffcd8, 99
90
90 was not found.
0
0 was not found.
```

7.3 We use a `for` loop to traverse the array until `p` points to the `target`:

```
float* duplicate(float* p[], int n)
{ float* const b = new float[n];
  for (int i = 0; i < n; i++)
    b[i] = *p[i];
  return b;
}

void print(float [], int);
void print(float* [], int);

int main()
{ float a[8] = {44.4, 77.7, 22.2, 88.8, 66.6, 33.3, 99.9, 55.5};
  print(a, 8);
  float* p[8];
  for (int i = 0; i < 8; i++)
    p[i] = &a[i];  // p[i] points to a[i]
  print(p, 8);
  float* const b = duplicate(p, 8);
  print(b, 8);
}
```

```
   44.4, 77.7, 22.2, 88.8, 66.6, 33.3, 99.9, 55.5
   44.4, 77.7, 22.2, 88.8, 66.6, 33.3, 99.9, 55.5
   44.4, 77.7, 22.2, 88.8, 66.6, 33.3, 99.9, 55.5
```

7.4 This function, named `riemann()`, is similar to the `sum()` function in Example 7.18. Its first argument is a pointer to a function that has one `double` argument and returns a `double`. In this test run, we pass it (a pointer to) the `cube()` function. The other three arguments are the boundaries `a` and `b` of the interval [*a*, *b*] over which the integration is being performed and the number `n` of subintervals to be used in the sum. The actual Riemann sum is the sum of the areas of the `n` rectangles based on these subintervals whose heights are given by the function being integrated:

```
double riemann(double (*)(double), double, double, int);
double cube(double);

int main()
{ cout << riemann(cube,0,2,10) << endl;
  cout << riemann(cube,0,2,100) << endl;
  cout << riemann(cube,0,2,1000) << endl;
  cout << riemann(cube,0,2,10000) << endl;
}

//  Returns [f(a)*h + f(a+h)*h + f(a+2h)*h + . . . + f(b-h)*h],
//  where h = (b-a)/n:
```

```
double riemann(double (*pf)(double t), double a, double b, int n)
{ double s = 0, h = (b-a)/n, x;
  int i;
  for (x = a, i = 0; i < n; x += h, i++)
    s += (*pf)(x);
  return s*h;
}

double cube(double t)
{ return t*t*t;
}
```

```
3.24
3.9204
3.992
3.9992
```

In this test run, we are integrating the function $y = x^3$ over the interval [0, 2]. By elementary calculus, the value of this integral is 4.0. The call `riemann(cube,0,2,10)` approximates this integral using 10 subintervals, obtaining 3.24. The call `riemann(cube,0,2,100)` approximates the integral using 100 subintervals, obtaining 3.9204. These sums get closer to their limit 4.0 as `n` increases. With 10,000 subintervals, the Riemann sum is 3.9992. Note that the only significant difference between this `riemann()` function and the `sum()` function in Example 7.18 is that the sum is multiplied by the subinterval width `h` before being returned.

7.5 This `derivative()` function is similar to the `sum()` function in Example 7.18, except that it implements the formula for the numerical derivative instead. It has three arguments: a pointer to the function *f*, the *x* value, and the tolerance *h*. In this test run, we pass it (pointers to) the `cube()` function and the `sqrt()` function.

```
#include <iostream>
#include <cmath>
using namespace std;
double derivative(double (*)(double), double, double);
double cube(double);

int main()
{ cout << derivative(cube, 1, 0.1) << endl;
  cout << derivative(cube, 1, 0.01) << endl;
  cout << derivative(cube, 1, 0.001) << endl;
  cout << derivative(sqrt, 1, 0.1) << endl;
  cout << derivative(sqrt, 1, 0.01) << endl;
  cout << derivative(sqrt, 1, 0.001) << endl;
}

//  Returns an approximation to the derivative f'(x):
double derivative(double (*pf)(double t), double x, double h)
{ return ((*pf)(x+h) - (*pf)(x-h))/(2*h);
}

double cube(double t)
{ return t*t*t;
}
```

```
    3.01
    3.0001
    3
    0.500628
    0.500006
    0.5
```

The derivative of the `cube()` function x^3 is $3x^2$, and its value at $x = 1$ is 3, so the numerical derivative should be close to 3.0 for small h. Similarly, the derivative of the `sqrt()` function $\sqrt{x}$ is $1/(2\sqrt{x})$, and its value at $x = 1$ is 1/2, so its numerical derivative should be close to 0.5 for small h.

7.6 The pointer `pmax` is used to locate the maximum `float`. It is initialized to have the same value as `p[0]` which points to the first `float`. Then inside the `for` loop, the `float` to which `p[i]` points is compared to the `float` to which `pmax` points, and `pmax` is updated to point to the larger `float` when it is detected. So when the loop terminates, `pmax` points to the largest `float`:

```
float* max(float* p[], int n)
{ float* pmax = p[0];
  for (int i = 1; i < n; i++)
  if (*p[i] > *pmax) pmax = p[i];
    return pmax;
}

void print(float [], int);
void print(float* [], int);

int main()
{ float a[8] = {44.4, 77.7, 22.2, 88.8, 66.6, 33.3, 99.9, 55.5};
  print(a, 8);
  float* p[8];
  for (int i = 0; i < 8; i++)
    p[i] = &a[i];  // p[i] points to a[i]
  print(p, 8);
  float* m = max(p, 8);
  cout << m << ", " << *m << endl;
}
```

```
    44.4, 77.7, 22.2, 88.8, 66.6, 33.3, 99.9, 55.5
    44.4, 77.7, 22.2, 88.8, 66.6, 33.3, 99.9, 55.5
    0x3fffcd4, 99.9
```

Here we have two (overloaded) `print()` functions: one to print the array of pointers, and one to print the `float`s to which they point. After initializing and printing the array `a`, we define the array `p` and initialize its elements to point to the elements of `a`. The call `print(p, 8)` verifies that `p` provides *indirect access* to `a`. Finally, the pointer `m` is declared and initialized with the address returned by the `max()` function. The last output verifies that `m` does indeed point to the largest `float` among those accessed by `p`.

Solutions to Problems 7.7-7.24 are available on-line at `projectEuclid.net`.

Chapter 8

C-Strings

8.1 INTRODUCTION

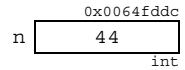
A *C-string* (also called a *character string*) is a sequence of contiguous characters in memory terminated by the NUL character `'\0'`. C-strings are accessed by variables of type `char*` (pointer to `char`). For example, if `s` has type `char*`, then

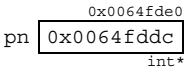
```
cout << s << endl;
```

will print all the characters stored in memory beginning at the address `s` and ending with the first occurrence of the NUL character.

The C header file `<cstring>` provides a wealth of special functions for manipulating C-strings. For example, the call `strlen(s)` will return the number of characters in the C-string `s`, not counting its terminating NUL character. These functions all declare their C-string parameters as pointers to `char`. So before we study these C-string operations, we need to review pointers. (See Section 7.3 on page 158.)

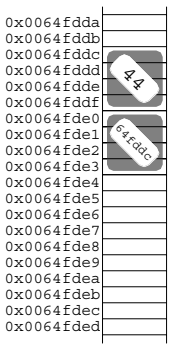
8.2 REVIEW OF POINTERS





A *pointer* is a memory address. For example, the following declarations define `n` to be an **`int`** with value 44 and `pn` to be a pointer containing the address of `n`:

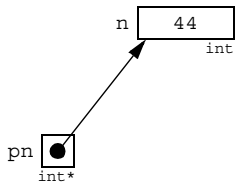
```
int n = 44;
int* pn = &n;
```



If we imagine memory to be a sequence of bytes with hexadecimal addresses, then we can picture `n` and `pn` as shown at right. This shows `n` stored at the address `64fddc` and `pn` stored at the address `64fde0`. The variable `n` contains value 44 and the variable `pn` contains the address value `64fddc`. The value of `pn` is the address of `n`. This relationship is usually represented by a simpler diagram like the one shown at right below. This shows two rectangles, one labeled `n` and one labeled `pn`. The rectangles represent storage locations in memory. The variable `pn` points to the variable `n`. We can access `n` through the pointer `pn` by means of the dereference operator **`*`**. For example, the statement

```
*pn = 77;
```

would change the value of `n` to 77.

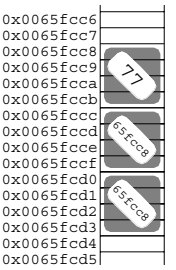


We can have more than one pointer pointing to the same object:

```
float* q = &x;
```

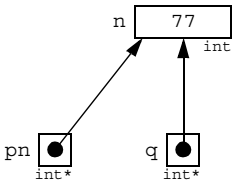
Now `*pn`, `*q`, and `x` are all names for the same object whose address is `64fddc` and whose current value is 77. This is shown in the diagram at right. Here, `q` is stored at the address `64fde4`. The value stored in `q` is the address `64fddc` of `n`.

The example below traces these definitions on a Windows workstation running Metrowerks CodeWarrior C++ on a Pentium III processor. As these diagrams indicate, memory is allocated in ascending order. The first object `n`, is stored at address `65fcc8`, occupying bytes `65fcc8–65fccb`. The second object, `pn`, is stored at address `65fccc`. The third object, `q`, is stored at address `65fcd0`.



EXAMPLE 8.1 Tracing Pointers

This program is similar to Example 7.5 on page 159:




```
int main()
{ int n=44;  // n holds the int 44
  cout << "int n=44;  // n holds the int 44:\n";
  cout << "\t\t    n = " << n << endl;
  cout << "\t\t   &n = " << &n << endl;
  int* pn=&n;  // pn holds the address of n
  cout << "int* pn=&n;  // pn holds the address of n:\n";
  cout << "\t\t    n = " << n << endl;
  cout << "\t\t   &n = " << &n << endl;
  cout << "\t\t   pn = " << pn << endl;
  cout << "\t\t  &pn = " << &pn << endl;
  cout << "\t\t  *pn = " << *pn << endl;
  *pn = 77;  // changes the value of n to 77
  cout << "*pn = 77;  // changes the value of n to 77:\n";
  cout << "\t\t    n = " << n << endl;
  cout << "\t\t   &n = " << &n << endl;
  cout << "\t\t   pn = " << pn << endl;
  cout << "\t\t  &pn = " << &pn << endl;
  cout << "\t\t  *pn = " << *pn << endl;
  int* q=&n;  // q also holds the address of n
  cout << "int* q=&n;  // q also holds the address of n:\n";
  cout << "\t\t    n = " << n << endl;
  cout << "\t\t   &n = " << &n << endl;
  cout << "\t\t   pn = " << pn << endl;
  cout << "\t\t  &pn = " << &pn << endl;
  cout << "\t\t  *pn = " << *pn << endl;
  cout << "\t\t    q = " << q << endl;
  cout << "\t\t   &q = " << &q << endl;
  cout << "\t\t   *q = " << *q << endl;
}
```

```
int* pn=&n;  // pn holds the address of n:
                    n = 44
                   &n = 0x0065fcc8
                   pn = 0x0065fcc8
                  &pn = 0x0065fccc
                  *pn = 44
*pn = 77;  // changes the value of n to 77:
                    n = 77
                   &n = 0x0065fcc8
                   pn = 0x0065fcc8
                  &pn = 0x0065fccc
                  *pn = 77
```

```
int* q=&n;  // q also holds the address of n:
                    n = 77
                   &n = 0x0065fcc8
                   pn = 0x0065fcc8
                  &pn = 0x0065fccc
                  *pn = 77
                    q = 0x0065fcc8
                   &q = 0x0065fcd0
                   *q = 77
```

If `p` is a pointer, then the statement `cout << *p` will always print the value of the object to which `p` points, and the statement `cout << p` will usually print the value of the address that is stored in `p`. The important exception to this second rule is when `p` is declared to have type **`char*`**.

8.3 C-STRINGS

In C++, a *C-string* is an array of characters with the following important features:

- An extra component is appended to the end of the array, and its value is set to the NUL character `'\0'`. This means that the total number of characters in the array is always 1 more than the string length.
- The C-string may be initialized with a string literal, like this:
  ```
  char str[] = "Bjarne";
  ```
  Note that this array has 7 elements: `'B'`, `'j'`, `'a'`, `'r'`, `'n'`, `'e'`, and `'\0'`.
- The entire C-string may be output as a single object, like this:
  ```
  cout << str;
  ```
  The system will copy characters from `str` to `cout` until the `NUL` character `'\0'` is encountered.
- The entire C-string may be input as a single object, like this:
  ```
  cin >> buffer;
  ```
  The system will copy characters from `cin` into `buffer` until a white space character is encountered. The user must ensure that buffer is defined to be a character string long enough to hold the input.
- The functions declared in the `<cstring>` header file may be used to manipulate C-strings. These include the string length function `strlen()`, the string copying functions `strcpy()` and `strncpy()`, the string concatenating functions `strcat()` and `strncat()`, the string comparing functions `strcmp()` and `strncmp()`, and the token extracting function `strtok()`. These functions are described in Section 8.8 on page 193.

EXAMPLE 8.2 C-Strings Are Terminated with the `NUL` Character

This little demo program shows that the `NUL` character `'\0'` is appended to the C-string:

```
int main()
{ char s[] = "ABCD";
  for (int i = 0; i < 5; i++)
    cout << "s[" << i << "] = '" << s[i] << "'\n";
}
```

```
s[0] = 'A'
s[1] = 'B'
s[2] = 'C'
s[3] = 'D'
s[4] = ''
```

When the `NUL` character is sent to `cout`, nothing is printed—not even a blank. This is seen by printing one apostrophe immediately before the character and another apostrophe immediately after the character.

8.4 STRING I/O

Input and output of C-strings are done in several ways in C++ programs. One way is to use the Standard C++ **`string`** class operators. Other methods are described here.

EXAMPLE 8.3 Ordinary Input and Output of C-Strings

This program reads words into a 79-character buffer:

```
int main()
{ char word[80];
  do
  { cin >> word;
    if (*word) cout << "\t\"" << word << "\"\n";
  } while (*word);
}
```

```
Today's date is March 12, 2000.
        "Today's"
        "date"
        "is"
        "March"
        "12,"
        "2000."
Tomorrow is Monday.
        "Tomorrow"
        "is"
        "Monday."
^Z
```

In this run, the `while` loop iterated 10 times: once for each word entered (including the Ctrl+Z that stopped the loop). Each word in the input stream `cin` is echoed to the output stream `cout`. Note that the output stream is not "flushed" until the input stream encounters the end of the line.

Each C-string is printed with a double quotation mark `"` on each side. This character must be designated by the character pair `\"` inside a C-string literal.

The expression `*word` controls the loop. It is the initial character in the C-string. It will be nonzero (*i.e.*, "true") as long as the C-string `word` contains a C-string of length greater than 0. The C-string of length 0, called the *empty C-string*, contains the `NUL` character `'\0'` in its first element. Entering Ctrl+Z+Enter+Entersends the end-of-file character in from `cin`. This loads the empty C-string into `word`, setting `*word` (which is the same as `word[0]`) to `'\0'` and stopping the loop. The last line of output shows only the Ctrl+Z echo, as `^Z`.

The Enter key may have to be pressed twice after Ctrl+Z is entered.

Note that punctuation marks (apostrophes, commas, periods, *etc.*) are included in the C-strings, but whitespace characters (blanks, tabs, newlines, *etc.*) are not.

The `do` loop in Example 8.3 could be replaced with:

```
cin >> word
while (*word)
{ cout << "\t\"" << word << "\"\n";
  cin >> word;
}
```

When Ctrl+Z is pressed, the call `cin >> word` assigns the empty C-string to `word`.

Example 8.3 and Example 8.1 illustrate an important distinction: the output operator `<<` behaves differently with pointers of type `char*` than with other pointer types. With a `char*` pointer, the operator outputs the entire character string to which the pointer points. But with any other pointer type, the operator will simply output the address of the pointer.

8.5 SOME `cin` MEMBER FUNCTIONS

The input stream object `cin` includes the input functions: `cin.getline()`, `cin.get()`, `cin.ignore()`, `cin.putback()`, and `cin.peek()`. Each of these function names includes the prefix "`cin.`" because they are "member functions" of the `cin` object.

The call `cin.getline(str,n)` reads up to `n` characters into `str` and ignores the rest.

EXAMPLE 8.4 The `cin.getline()` Function with Two Parameters

This program echoes the input, line by line:

```
int main()
{ char line[80];
  do
  { cin.getline(line, 80);
    if (*line) cout << "\t[" << line << "]\n";
  } while (*line);
}
```

Note that the condition `(*line)` will evaluate to "true" precisely when `line` contains a non-empty C-string, because only then will `line[0]` be different from the NUL character (ASCII value 0).

The call `cin.getline(str,n,ch)` reads all input up to the first occurrence of the delimiting character `ch` into `str`. If the specified character `ch` is the newline character `'\n'`, then this is equivalent to `cin.getline(str,n)`. This is illustrated in the next example where the delimiting character is the comma `','`.

EXAMPLE 8.5 The `cin.getline()` Function with Three Parameters

This program echoes the input, clause by clause:

```
int main()
{ char clause[80];
  do
  { cin.getline(clause, 80, ',');
    if (*clause) cout << "\t[" << clause << "]\n";
  } while (*clause);
}
```

```
Once upon a midnight dreary, while I pondered, weak and weary,
        [Once upon a midnight dreary]
        [ while I pondered]
        [ weak and weary]
Over a many quaint and curious volume of forgotten lore,
        [
Over a many quaint and curious volume of forgotten lore]
^Z
        [
]
```

Notice that the invisible endline character that follows "`weary,`" is stored as the first character of the next input line. Since the comma is being used as the delimiting character, the endline character is processed just like an ordinary character.

The `cin.get()` function is used for reading input character-by-character. The call `cin.get(ch)` copies the next character from the input stream `cin` into the variable `ch` and returns 1, unless the end of file is detected in which case it returns 0.

EXAMPLE 8.6 The `cin.get()` Function

This program counts the number of occurrences of the letter '`e`' in the input stream. The loop continues as long as the `cin.get(ch)` function is successful at reading characters into `ch`:

```
int main()
{ char ch;
  int count = 0;
  while (cin.get(ch))
    if (ch == 'e') ++count;
  cout << count << " e's were counted.\n";
}
```

```
Once upon a midnight dreary, while I pondered, weak and weary,
Over many a quaint and curious volume of forgotten lore,
^Z
11 e's were counted.
```

The opposite of `get` is `put`. The `cout.put()` function is used for writing to the output stream `cout` character-by-character. This is illustrated in the next example.

EXAMPLE 8.7 The `cout.put()` Function

This program echoes the input stream, capitalizing each word:

```
int main()
{ char ch, pre = '\0';
  while (cin.get(ch))
  { if (pre == ' ' || pre == '\n') cout.put(char(toupper(ch)));
    else cout.put(ch);
    pre = ch;
  }
}
```

```
Fourscore and seven years ago our fathers
Fourscore And Seven Years Ago Our Fathers
brought forth upon this continent a new nation,
Brought Forth Upon This Continent A New Nation,
^Z
```

The variable `pre` holds the previously read character. The idea is that if `pre` is a blank or the newline character, then the next character `ch` would be the first character of the next word. In that case, `ch` is replaced by its equivalent uppercase character `ch + 'A' - 'a'`.

The header file `<ctype.h>` declares the function `toupper(ch)` which returns the uppercase equivalent of `ch` if `ch` is a lowercase letter.

The `cin.putback()` function restores the last character read by a `cin.get()` back to the input stream `cin`. The `cin.ignore()` function reads past one or more characters in the input stream `cin` without processing them. Example 8.8 illustrates these functions.

The `cin.peek()` function can be used in place of the combination `cin.get()` and `cin.putback()` functions. The call

```
ch = cin.peek()
```

copies the next character of the input stream `cin` into the `char` variable `ch` without removing that character from the input stream. Example 8.9 shows how the `peek()` function can be used in place of the `get()` and `putback()` functions.

EXAMPLE 8.8 The `cin.putback()` and `cin.ignore()` Functions

This tests a function that extracts the integers from the input stream:

```
int nextInt();
int main()
{ int m = nextInt(), n = nextInt();
  cin.ignore(80,'\n');               //  ignore rest of input line
  cout << m << " + " << n << " = " << m+n << endl;
}
int nextInt()
{ char ch;
  int n;
  while (cin.get(ch))
    if (ch >= '0' && ch <= '9')  //  next character is a digit
    { cin.putback(ch);           //  put it back so it can be
      cin >> n;                  //  read as a complete int
      break;
    }
  return n;
}
```

```
What is 305 plus 9416?
305 + 9416 = 9721
```

The `nextInt()` function scans past the characters in `cin` until it encounters the first digit. In this run, that digit is 3. Since this digit will be part of the first integer 305, it is put back into `cin` so that the complete integer 305 can be read into `n` and returned.

EXAMPLE 8.9 The `cin.peek()` Function

This version of the `nextInt()` function is equivalent to the one in the previous example:

```
int nextInt()
{ char ch;
  int n;
  while (ch = cin.peek())
    if (ch >= '0' && ch <= '9')
    { cin >> n;
      break;
    }
    else cin.get(ch);
  return n;
}
```

The expression `ch = cin.peek()` copies the next character into `ch`, and returns 1 if successful. Then if `ch` is a digit, the complete integer is read into `n` and returned. Otherwise, the character is removed from `cin` and the loop continues. If the end-of-file is encountered, the expression `ch = cin.peek()` returns 0, stopping the loop.

8.6 STANDARD C CHARACTER FUNCTIONS

Example 8.7 on page 188 illustrates the `toupper()` function. This is one of a series of character manipulation function defined in the `<cctype>` header file. These are summarized in the following table.

`isalnum()`	`int isalnum(int c);` Returns nonzero if `c` is an alphabetic or numeric character; otherwise returns 0.
`isalpha()`	`int isalpha(int c);` Returns nonzero if `c` is an alphabetic character; otherwise returns 0.
`iscntrl()`	`int iscntrl(int c);` Returns nonzero if `c` is a control character; otherwise returns 0.
`isdigit()`	`int isdigit(int c);` Returns nonzero if `c` is a digit character; otherwise returns 0.
`isgraph()`	`int isgraph(int c);` Returns nonzero if `c` is any non-blank printing character; otherwise returns 0.
`islower()`	`int islower(int c);` Returns nonzero if `c` is a lowercase alphabetic character; otherwise returns 0.
`isprint()`	`int isprint(int c);` Returns nonzero if `c` is any printing character; otherwise returns 0.
`ispunct()`	`int ispunct(int c);` Returns nonzero if `c` is any printing character, except the alphabetic characters, the numeric characters, and the blank; otherwise returns 0.

`isspace()`	`int isspace(int c);` Returns nonzero if `c` is any white-space character, including the blank `' '`, the form feed `'\f'`, the newline `'\n'`, the carriage return `'\r'`, the horizontal tab `'\t'`, and the vertical tab `'\v'`; otherwise returns 0.
`isupper()`	`int isupper(int c);` Returns nonzero if `c` is an uppercase alphabetic character; otherwise returns 0.
`isxdigit()`	`int isxdigit(int c);` Returns nonzero if `c` is one of the 10 digit characters or one of the 12 hexadecimal digit letters: `'a'`, `'b'`, `'c'`, `'d'`, `'e'`, `'f'`, `'A'`, `'B'`, `'C'`, `'D'`, `'E'`, or `'F'`; otherwise returns 0.
`tolower()`	`int tolower(int c);` Returns the lowercase version of `c` if `c` is an uppercase alphabetic character; otherwise returns `c`.
`toupper()`	`int toupper(int c);` Returns the uppercase version of `c` if `c` is a lowercase alphabetic character; otherwise returns `c`.

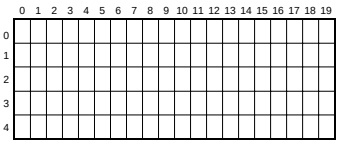
Note that these functions receive an `int` parameter `c` and they return an `int`. This works because `char` is an integer type. Normally, a `char` is passed to the function and the return value is assigned to a `char`, so we regard these as character-modifying functions.

8.7 ARRAYS OF STRINGS

Recall that a two-dimensional array is really a one-dimensional array whose components themselves are one-dimensional arrays. When those component arrays are C-strings, we have an array of C-strings.

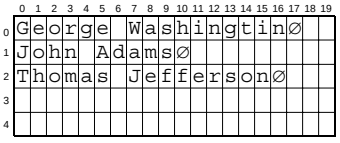
Example 8.10 declares the two-dimensional array `name` as

```
char name[5][20];
```



This declaration allocates 100 bytes, arranged like this:

Each of the 5 rows is a one-dimensional array of 20 characters and therefore can be regarded as a character string. These C-strings are accessed as `name[0]`, `name[1]`, `name[2]`, `name[3]`, `name[4]`. In the sample run shown in Example 8.10, the data would be stored like this:



Here, the symbol ∅ represents the `NUL` character `'\0'`.

EXAMPLE 8.10 An Array of Strings

This program reads in a sequence of C-strings, storing them in an array, and then prints them:

```
int main()
{ char name[5][20];
  int count=0;
  cout << "Enter at most 4 names with at most 19 characters:\n";
  while (cin.getline(name[count++], 20))
    ;
  --count;
```

```
  cout << "The names are:\n";
  for (int i=0; i<count; i++)
    cout << "\t" << i << ". [" << name[i] << "]" << endl;
}
Enter at most 8 names with at most 23 characters:
George Washington
John Adams
Thomas Jefferson
^Z
The names are:
        0. [George Washington]
        1. [John Adams]
        2. [Thomas Jefferson]
```

Note that all the activity in the `while` loop is done within its control condition:

```
cin.getline(name[count++],20)
```

This call to the `cin.getline()` function reads the next line into `name[count]` and then increments `count`. The function returns nonzero (*i.e.*, "true") if it was successful in reading a character string into `name[count]`. When the end-of-file is signalled (with **<Control-D>** or **<Control-Z>**), the `cin.getline()` function fails, so it returns 0 which stops the `while` loop. The body of this loop is empty, indicated by the line that contains nothing but a semicolon.

A more efficient way to store C-strings is to declare an array of pointers: `char* name[4];` Here, each of the 4 components has type `char*` which means that each `name[i]` is a C-string. This declaration does not initially allocate any storage for C-string data. Instead, we need to store all the data in a buffer C-string. Then we can set each `name[i]` equal to the address of the first character of the corresponding name in the buffer. This is done in Example 8.11. This method is more efficient because each component of `name[i]` uses only as many bytes as are needed to store the C-string (plus storage for one pointer). The trade-off is that the input routine needs a sentinel to signal when the input is finished.

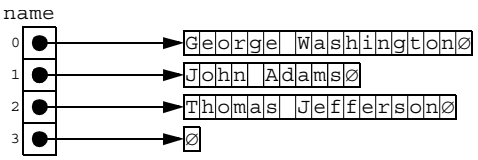
EXAMPLE 8.11 A String Array

This program illustrates the use of the `getline()` function with the sentinel character `'$'`. It is nearly equivalent to that in Example 8.10. It reads a sequence of names, one per line, terminated by the sentinel `'$'`. Then it prints the names which are stored in the array `name`:

```
int main()
{ char buffer[80];
  cin.getline(buffer,80,'$');
  char* name[4];
  name[0] = buffer;
  int count = 0;
  for (char* p=buffer; *p != '\0'; p++)
    if (*p == '\n')
    { *p = '\0';            //  end name[count]
      name[++count] = p+1;  //  begin next name
    }
  cout << "The names are:\n";
  for (int i=0; i<count; i++)
    cout << "\t" << i << ". [" << name[i] << "]" << endl;
}
```

The entire input is stored in `buffer` as the single C-string containing "`George Washington\nJohn Adams\nThomas Jefferson\n`". The `for` loop then scans through buffer using the pointer `p`. Each time `p` finds the `'\n'` character, it terminates the C-string in `name[count]` by appending the `NUL` character `'\0'` to it. Then it increments the counter `count` and stores the address `p+1` of the next character in `name[count]`.

The resulting array `name` looks like this: Note that the extra bytes that padded the ends of the names in Example 8.10 are not required here.



If the C-strings being stored are known at compile time, then the C-string array described above is quite a bit simpler to handle. Example 8.12 illustrates how to initialize a C-string array.

EXAMPLE 8.12 Initializing a String Array

This program is nearly equivalent to those in the previous two examples. It initializes the C-string array `name` and then prints its contents:

```
int main()
{ char* name[]
    = { "George Washington", "John Adams", "Thomas Jefferson" };
    cout << "The names are:\n";
    for (int i = 0; i < 3; i++)
        cout << "\t" << i << ". [" << name[i] << "]" << endl;
}
```

```
The names are:
        0. [George Washington]
        1. [John Adams]
        2. [Thomas Jefferson]
```

The storage of the data in the `name` array here is the same as in Example 8.11.

8.8 STANDARD C STRING FUNCTIONS

The C header file `<cstring>`, also called the *C-String Library*, includes a family of functions that are very useful for manipulating C-strings. Example 8.13 illustrates the simplest of these functions, the *C-string length function*, which returns the length of the C-string passed to it.

EXAMPLE 8.13 The `strlen()` Function

This program is a simple test driver for the `strlen()` function. The call `strlen(s)` simply returns the number of characters in `s` that precede the first occurrence of the `NUL` character `'\0'`

```
#include <cstring>
int main()
{ char s[] = "ABCDEFG";
  cout << "strlen(" << s << ") = " << strlen(s) << endl;
  cout << "strlen(\"\") = " << strlen("") << endl;
  char buffer[80];
  cout << "Enter string: ";  cin >> buffer;
  cout << "strlen(" << buffer << ") = " << strlen(buffer) << endl;
}
```

In some ways, C-strings behave like fundamental objects (*i.e.*, integers and reals). For example, they can be output to `cout` in the same way. But C-strings are structured objects, composed of smaller pieces (characters). So many of the operations that are provided for fundamental objects, such as the assignment operator (`=`), the comparison operators (`<`, `>`, `==`, `<=`, `>=`, and `!=`), and the arithmetic operators (`+`, *etc.*) are not available for C-strings. Some of the functions in the C String Library simulate these operations. In Chapter 12 we will learn how to write our own versions of these operations.

The next example illustrates three other C-string functions. These are used to locate characters and substrings within a given C-string.

EXAMPLE 8.14 The `strchr()`, `strrchr()`, and `strstr()` Functions

```
#include <cstring>
int main()
{ char s[] = "The Mississippi is a long river.";
  cout << "s = \"" << s << "\"\n";
  char* p = strchr(s, ' ');
  cout << "strchr(s, ' ') points to s[" << p - s << "].\n";
  p = strchr(s, 's');
  cout << "strchr(s, 's') points to s[" << p - s << "].\n";
  p = strrchr(s, 's');
  cout << "strrchr(s, 's') points to s[" << p - s << "].\n";
  p = strstr(s, "is");
  cout << "strstr(s, \"is\") points to s[" << p - s << "].\n";
  p = strstr(s, "isi");
  if (p == NULL) cout << "strstr(s, \"isi\") returns NULL\n";
}
```

```
s = "The Mississippi is a long river."
strchr(s, ' ') points to s[3].
strchr(s, 's') points to s[6].
strrchr(s, 's') points to s[17].
strstr(s, "is") points to s[5].
strstr(s, "isi") returns NULL
```

The call `strchr(s, ' ')` returns a pointer to the first occurrence of the blank character `' '` within the C-string `s`. The expression `p - s` computes the index (offset) 3 of this character within the C-string. (Remember that arrays used zero-based indexing, so the initial character `'T'` has index 0.) Similarly, the character `'s'` first appears at index 6 in `s`.

The call `strrchr(s, ' ')` returns a pointer to the last occurrence of the character `'s'` within the C-string `s`; this is `s[17]`.

The call `strstr(s, "is")` returns a pointer to the first occurrence of the substring `"is"` within the C-string `s`; this is at `s[5]`. The call `strstr(s, "isi")` returns the `NULL` pointer because `"isi"` does not occur anywhere within the C-string `s`.

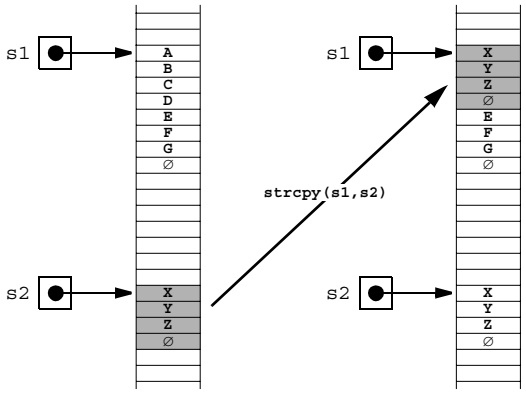
There are two functions that simulate the assignment operator for C-strings: `strcpy()` and `strncpy()`. The call `strcpy(s1,s2)` copies C-string `s2` into C-string `s1`. The call `strncpy(s1,s2,n)` copies the first `n` characters of C-string `s2` into C-string `s1`. Both functions return `s1`. These are illustrated in the next two examples.

EXAMPLE 8.15 The `strcpy()` Function

This program traces call `strcpy(s1,s2)`:

```
#include <cstring>
#include <iostream>
int main()
{ char s1[] = "ABCDEFG";
  char s2[] = "XYZ";
  cout << "Before strcpy(s1,s2):\n";
  cout << "\ts1 = [" << s1 << "], length = " << strlen(s1) << endl;
  cout << "\ts2 = [" << s2 << "], length = " << strlen(s2) << endl;
  strcpy(s1,s2);
  cout << "After strcpy(s1,s2):\n";
  cout << "\ts1 = [" << s1 << "], length = " << strlen(s1) << endl;
  cout << "\ts2 = [" << s2 << "], length = " << strlen(s2) << endl;
}
```

```
Before strcpy(s1,s2):
        s1 = [ABCDEFG], length = 7
        s2 = [XYZ], length = 3
After strcpy(s1,s2):
        s1 = [XYZ], length = 3
        s2 = [XYZ], length = 3
```



After `s2` is copied into `s1`, they are indistinguishable: both consist of the 3 characters `XYZ`. The effect of `strcpy(s1,s2)` can be visualized as shown at right. Since `s2` has length 3, `strcpy(s1,s2)` copies 4 bytes (including the `NUL` character, shown as ∅), overwriting the first 4 characters of `s1`. This changes the length of `s1` to 3.

Note that `strcpy(s1,s2)` creates a duplicate of C-string `s2`. The resulting two copies are distinct C-strings. Changing one of these C-strings later would have no effect upon the other C-string.

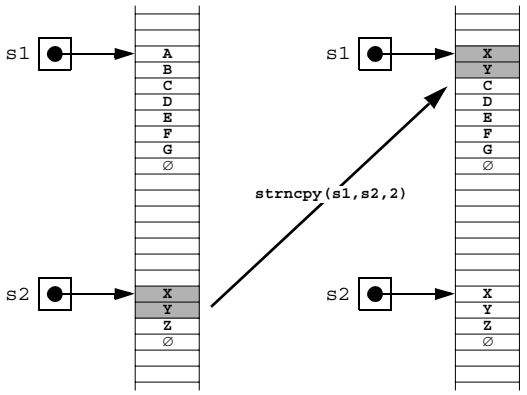
EXAMPLE 8.16 The Function `strncpy()`

This program traces calls `strncpy(s1,s2,n)`:

```
int main()
{ char s1[] = "ABCDEFG";
  char s2[] = "XYZ";
  cout << "Before strncpy(s1,s2,2):\n";
  cout << "\ts1 = [" << s1 << "], length = " << strlen(s1) << endl;
  cout << "\ts2 = [" << s2 << "], length = " << strlen(s2) << endl;
  strncpy(s1,s2,2);
  cout << "After strncpy(s1,s2,2):\n";
  cout << "\ts1 = [" << s1 << "], length = " << strlen(s1) << endl;
  cout << "\ts2 = [" << s2 << "], length = " << strlen(s2) << endl;
}
```

```
Before strncpy(s1,s2,2):
        s1 = [ABCDEFG], length = 7
        s2 = [XYZ], length = 3
After strncpy(s1,s2,2):
        s1 = [XYCDEFG], length = 7
        s2 = [XYZ], length = 3
```

The call `strncpy(s1,s2,2)` replaces the first 2 characters of `s1` with `XY`, leaving the rest of `s1` unchanged. The effect of `strncpy(s1,s2,2)` can be visualized as shown here. Since `s2` has length 3, `strncpy(s1,s2,2)` copies 2 bytes (excluding the `NUL` character ∅), overwriting the first 2 characters of `s1`. This has no effect upon the length of `s1` which is 7.



If `n` < `strlen(s2)`, as it is in the above example, then `strncpy(s1,s2,n)` simply copies the first `n` characters of `s2` into the beginning of `s1`. However, if `n` ≥ `strlen(s2)`, then `strncpy(s1,s2,n)` has the same effect as `strcpy(s1,s2)`: it makes `s1` a duplicate of `s2` with the same length.

The `strcat()` and `strncat()` functions work the same as the `strcpy()` and `strncpy()` functions except that the characters from the second C-string are copied onto the end of the first C-string. The term "`cat`" comes from the word "catenate" meaning "string together."

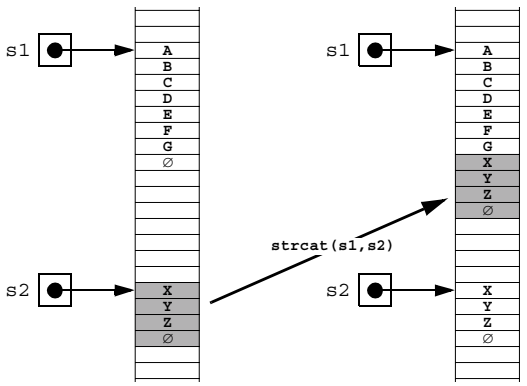
EXAMPLE 8.17 The String Concatenation Function `strcat()`

This program traces call `strcat(s1,s2)` which appends the C-string `s2` onto the end of `s1`:

```
int main()
{ char s1[] = "ABCDEFG";
  char s2[] = "XYZ";
  cout << "Before strcat(s1,s2):\n";
  cout << "\ts1 = [" << s1 << "], length = " << strlen(s1) << endl;
  cout << "\ts2 = [" << s2 << "], length = " << strlen(s2) << endl;
  strcat(s1,s2);
  cout << "After strcat(s1,s2):\n";
  cout << "\ts1 = [" << s1 << "], length = " << strlen(s1) << endl;
  cout << "\ts2 = [" << s2 << "], length = " << strlen(s2) << endl;
}
```

```
Before strcat(s1,s2):
        s1 = [ABCDEFG], length = 7
        s2 = [XYZ], length = 3
After strcat(s1,s2):
        s1 = [ABCDEFGXYZ], length = 10
        s2 = [XYZ], length = 3
```

The call `strcat(s1,s2)` appends `XYZ` onto the end of `s1`. It can be visualized as shown here. Since `s2` has length 3, `strcat(s1,s2)` copies 4 bytes (including the `NUL` character, shown as ∅), overwriting the `NUL` characters of `s1` and its following 3 bytes. The length of `s1` is increased to 10.



If any of the extra bytes following `s1` that are needed to copy `s2` are in use by any other object, then all of `s1` and its appended `s2` will be copied to some other free section of memory.

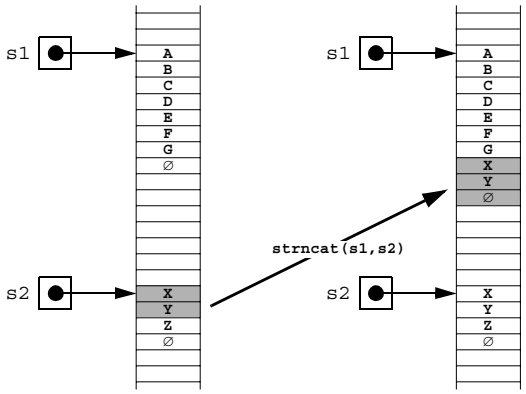
EXAMPLE 8.18 The Second String Concatenation Function `strncat()`

This program traces calls `strncat(s1,s2,n)`:

```
#include <cstring>
#include <iostream>
using namespace std;
int main()
{ // test-driver for the strncat() function:
  char s1[] = "ABCDEFG";
  char s2[] = "XYZ";
  cout << "Before strncat(s1,s2,2):\n";
  cout << "\ts1 = [" << s1 << "], length = " << strlen(s1) << endl;
  cout << "\ts2 = [" << s2 << "], length = " << strlen(s2) << endl;
  strncat(s1,s2,2);
  cout << "After strncat(s1,s2,2):\n";
  cout << "\ts1 = [" << s1 << "], length = " << strlen(s1) << endl;
  cout << "\ts2 = [" << s2 << "], length = " << strlen(s2) << endl;
}
```

```
Before strncat(s1,s2,2):
        s1 = [ABCDEFG], length = 7
        s2 = [XYZ], length = 3
After strncat(s1,s2,2):
        s1 = [ABCDEFGXY], length = 9
        s2 = [XYZ], length = 3
```

The call `strncat(s1,s2,2)` appends `XY` onto the end of `s1`. The effect can be visualized as shown here. Since `s2` has length 3, `strncat(s1,s2,2)` copies 2 bytes overwriting the `NUL` character of `s1` and the byte that follows it. Then it puts the `NUL` character in the next byte to complete the C-string `s1`. This increases its length to 9. (If either of the extra 2 bytes had been in use by some other object, then the entire 10 characters `ABCDEFGXYØ` would have been written in some other free part of memory.)



The next example illustrates the C-string *tokenize function*. Its purpose is to identify "tokens" within a given C-string: *e.g.*, words in a sentence.

EXAMPLE 8.19 The String Tokenize Function `strtok()`

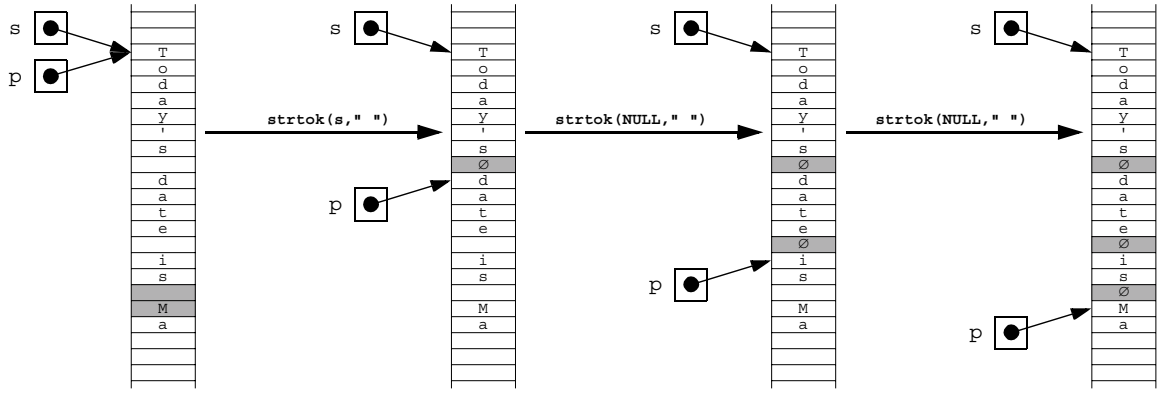
This program shows how `strtok()` is used to extract the individual words from a sentence.

```
#include <cstring>
#include <iostream>
using namespace std;
int main()
{ // test-driver for the strtok() function:
  char s[] = "Today's date is March 12, 2000.";
  char* p;
  cout << "The string is: [" << s << "]\nIts tokens are:\n";
  p = strtok(s, " ");
```

```
  while (p)
  { cout << "\t[" << p << "]\n";
    p = strtok(NULL, " ");
  }
  cout << "Now the string is: [" << s << "]\n";
}
```

```
The string is: [Today's date is March 12, 2000.]
Its tokens are:
        [Today's]
        [date]
        [is]
        [March]
        [12,]
        [2000.]
Now the string is: [Today's]
```

The call `p = strtok(s, " ")` sets the pointer `p` to point to the first token in the C-string `s` and changes the blank that follows `"Today's"` to the NUL character `'\0'` (denoted by Ø in the following diagram). This has the effect of making both `s` and `p` the C-string `"Today's"`. Then each successive call `p = strtok(NULL, " ")` advances the pointer `p` to the next non-blank character that follows the new NUL character, changing each blank that it passes into a NUL character, and changing the first blank that follows `*p` into a NUL character. This has the effect of making `p` the next substring that was delimited by blanks and is now delimited by NUL characters. This continues until p reaches the NUL character that terminated the original C-string `s`. That makes `p` NUL (*i.e.*, 0), which stops the `while` loop. The combined effect upon the original C-string `s` of all the calls to `strtok()` is to change every blank into a NUL. This "tokenizes" the C-string `s`, changing it into a sequence of distinct token strings, only the first of which is identified by `s`.



Note that the `strtok()` function changes the C-string that it tokenizes. Therefore, if you want to use the original C-string after you tokenize it, you should duplicate it with `strcpy()`.

Also note that the second parameter of the `strtok()` function is a C-string. This function uses all the characters in this C-string as delimiters in the first C-string. For example, to identify words in `s`, you might use `strtok(s, " ,:;.")`.

The `strpbrk()` function also uses a C-string of characters as a collection of characters. It generalizes the `strchr()` function, looking for the first occurrence in the first C-string of any of the characters in the second C-string.

EXAMPLE 8.20 The `strpbrk()` Function

```
#include <cstring>
#include <iostream>
using namespace std;
int main()
{ char s[] = "The Mississippi is a long river.";
  cout << "s = \"" << s << "\"\n";
  char* p = strpbrk(s, "nopqr");
  cout << "strpbrk(s, \"nopqr\") points to s[" << p - s << "].\n";
  p = strpbrk(s, "NOPQR");
  if (p == NULL) cout << "strpbrk(s, \"NOPQR\") returns NULL.\n";
}
```

```
s = "The Mississippi is a long river."
strpbrk(s, "nopqr") points to s[12].
strpbrk(s, "NOPQR") returns NULL.
```

The call `strpbrk(s, "nopqr")` returns the first occurrence in `s` of any of the five characters `'n'`, `'o'`, `'p'`, `'q'`, or `'r'`. The first of these found is the `'p'` at `s[12]`.

The call `strpbrk(s, "NOPQR")` returns the `NULL` pointer because none of these five characters occurs in `s`.

The following table summarizes some of the most useful functions declared in `<cstring>`. Note that `size_t` is a special integer type that is defined in the `<cstring>` file.

`memcpy()`	`void* memcpy(void* s1, const void* s2, size_t n);` Replaces the first `n` bytes of `*s1` with the first `n` bytes of `*s2`. Returns `s`.
`strcat()`	`char* strcat(char* s1, const char* s2);` Appends `s2` to `s1`. Returns `s1`.
`strchr()`	`char* strchr(const char* s, int c);` Returns a pointer to the first occurrence of `c` in `s`. Returns `NULL` if `c` is not in `s`.
`strcmp()`	`int strcmp(const char* s1, const char* s2);` Compares `s1` with substring `s2`. Returns a negative integer, zero, or a positive integer, according to whether `s1` is lexicographically less than, equal to, or greater than `s2`.
`strcpy()`	`char* strcpy(char* s1, const char* s2);` Replaces `s1` with `s2`. Returns `s1`.
`strcspn()`	`size_t strcspn(char* s1, const char* s2);` Returns the length of the longest substring of `s1` that begins with `s1[0]` and contains none of the characters found in `s2`.
`strlen()`	`size_t strlen(const char* s);` Returns the length of `s`, which is the number of characters beginning with `s[0]` that precede the first occurrence of the `NUL` character.
`strncat()`	`char* strncat(char* s1, const char* s2, size_t n);` Appends the first `n` characters of `s2` to `s1`. Returns `s1`. If `n` ≥ `strlen(s2)`, then `strncat(s1,s2,n)` has the same effect as `strcat(s1,s2)`.

`strncmp()`	`int strncmp(const char* s1, const char* s2, size_t n);` Compares the first `n` characters of `s1` with the first `n` characters of `s2`. Returns a negative integer, zero, or a positive integer, according to whether the first substring is lexicographically less than, equal to, or greater than the second. If `n` ≥ `strlen(s2)`, then `strncmp(s1,s2,n)` and `strcmp(s1,s2)` have the same effect.
`strncpy()`	`char* strncpy(char* s1, const char* s2, size_t n);` Replaces the first `n` characters of `s1` with the first `n` characters of `s2`. Returns `s1`. If `n` ≤ `strlen(s1)`, then the length of `s1` is not affected. If `n` ≥ `strlen(s2)`, then `strncpy(s1,s2,n)` and `strcpy(s1,s2)` have the same effect.
`strpbrk()`	`char* strpbrk(const char* s1, const char* s2);` Returns the address of the first occurrence in `s1` of any of the characters in `s2`. Returns `NULL` if none of the characters in `s2` appears in `s1`.
`strrchr()`	`char* strrchr(const char* s, int c);` Returns a pointer to the last occurrence of `c` in `s`. Returns `NULL` if `c` is not in `s`.
`strspn()`	`size_t strspn(char* s1, const char* s2);` Returns the length of the longest substring of `s1` that begins with `s1[0]` and contains only characters found in `s2`.
`strstr()`	`char* strstr(const char* s1, const char* s2);` Returns the address of the first occurrence of `s2` as a substring of `s1`. Returns `NULL` if `ch` is not in `s1`.
`strtok()`	`char* strtok(char* s1, const char* s2);` Tokenizes the C-string `s1` into tokens delimited by the characters found in C-string `s2`. After the initial call `strtok(s1, s2)`, each successive call `strtok(NULL, s2)` returns a pointer to next token found in `s1`. These calls change the C-string `s1`, replacing each delimiter with the `NUL` character `'\0'`.

Review Questions

8.1 Consider the following declarations for `s`:

```
char s[6];
char s[6] = {'H', 'e', 'l', 'l', 'o'};
char s[6] = "Hello";
char s[];
char s[] = new char[6];
char s[] = {'H', 'e', 'l', 'l', 'o'};
char s[] = "Hello";
char s[] = new("Hello");
char* s;
char* s = new char[6];
char* s = {'H', 'e', 'l', 'l', 'o'};
char* s = "Hello";
char* s = new("Hello");
```

a. Which of these is a valid declaration of a C++ character C-string?

b. Which of these is a valid declaration of a C++ character C-string of length 5, initialized to the C-string `"Hello"` and allocated at compile time?

c. Which of these is a valid declaration of a C++ character C-string of length 5, initialized to the C-string `"Hello"` and allocated at run time?

d. Which of these is a valid declaration of a C++ character C-string as a formal parameter for a function?

8.2 What is wrong with using the statement

```
cin >> s;
```

to read the input `"Hello, World!"` into a C-string `s`?

8.3 What does the following code print:

```
char s[] = "123 W. 42nd St., NY, NY 10020-1095";
int count = 0;
for (char* p = s; *p; p++)
if (isupper(*p)) ++count;
cout << count << endl;
```

8.4 What does the following code print:

```
char s[] = "123 W. 42nd St., NY, NY 10020-1095";
for (char* p = s; *p; p++)
if (isupper(*p)) *p = tolower(*p);
cout << s << endl;
```

8.5 What does the following code print:

```
char s[] = "123 W. 42nd St., NY, NY 10020-1095";
for (char* p = s; *p; p++)
if (isupper(*p)) (*p)++;
cout << s << endl;
```

8.6 What does the following code print:

```
char s[] = "123 W. 42nd St., NY, NY 10020-1095";
int count = 0;
for (char* p = s; *p; p++)
if (ispunct(*p)) ++count;
cout << count << endl;
```

8.7 What does the following code print:

```
char s[] = "123 W. 42nd St., NY, NY 10020-1095";
for (char* p = s; *p; p++)
if (ispunct(*p)) *(p-1) = tolower(*p);
cout << s << endl;
```

8.8 What is the difference between the following two statements, if `s1` and `s2` have type `char*`:

```
s1 = s2;
strcpy(s1,s2);
```

8.9 If `first` contains the C-string `"Rutherford"` and `last` contains the C-string `"Hayes"`, then what will be the effect of each of the following calls:

`a.int n = strlen(first);`

b. `char* s1 = strchr(first, 'r');`

c. `char* s1 = strrchr(first, 'r');`

d. `char* s1 = strpbrk(first, "rstuv");`

e. `strcpy(first, last);`

f. `strncpy(first, last, 3);`

g. `strcat(first, last);`

h. `strncat(first, last, 3);`

8.10 What do each of the following assign to `n`:

a.`int n = strspn("abecedarian","abcde");`

b. `int n = strspn("beefeater","abcdef");`

c. `int n = strspn("baccalaureate","abc");`

d. `int n = strcspn("baccalaureate","rstuv");`

8.11 What does the following code print:

```
char* s1 = "ABCDE";
char* s2 = "ABC";
if (strcmp(s1,s2) < 0) cout << s1 << " < " << s2 << endl;
else cout << s1 << " >= " << s2 << endl;
```

8.12 What does the following code print:

```
char* s1 = "ABCDE";
char* s2 = "ABCE";
if (strcmp(s1,s2) < 0) cout << s1 << " < " << s2 << endl;
else cout << s1 << " >= " << s2 << endl;
```

8.13 What does the following code print:

```
char* s1 = "ABCDE";
char* s2 = "";
if (strcmp(s1,s2) < 0) cout << s1 << " < " << s2 << endl;
else cout << s1 << " >= " << s2 << endl;
```

8.14 What does the following code print:

```
char* s1 = " ";
char* s2 = "";
if (strcmp(s1,s2) == 0) cout << s1 << " == " << s2 << endl;
else cout << s1 << " != " << s2 << endl;
```

Problems

8.1 Explain why the following alternative to Example 8.12 does not work:

```
int main()
{ char name[10][20], buffer[20];
  int count = 0;
  while (cin.getline(buffer,20))
    name[count++] = buffer;
  --count;
  cout << "The names are:\n";
  for (int i = 0; i < count; i++)
    cout << "\t" << i << ". [" << name[i] << "]" << endl;
}
```

8.2 Write the `strcpy()` function.

8.3 Write the `strncat()` function.

8.4 Write and test a function that returns the *plural* form of the singular English word that is passed to it.

8.5 Write a program that reads a sequence of names, one per line, and then sorts and prints them.

8.6 Write and test a function to reverse a C-string in place, without any duplication of characters.

8.7 Write and run the variation of the program in Example 8.3 that uses

```
while (cin >> word)
```

instead of

```
do..while (*word)
```

8.8 Write the `strchr()` function.

8.9 Write a function that returns the number of occurrences of a given character within a given C-string.

8.10 Write and test the `strrchr()` function.

8.11 Write and test the `strstr()` function.

8.12 Write and test the `strncpy()` function.

8.13 Write and test the `strcat()` function.

8.14 Write and test the `strcmp()` function.

8.15 Write and test the `strncmp()` function.

8.16 Write and test the `strspn()` function.

8.17 Write and test the `strcspn()` function.

8.18 Write and test the `strpbrk()` function.

8.19 Write a function that returns the number of words that contain a given character within a given C-string. (See Example 8.19.)

8.20 First, try to predict what the following program will do to the C-string `s`. (See Example 8.19 on page 197.) Then run the program to check your prediction.

```
int main()
{ char s[] = "###ABCD#EFG##HIJK#L#MN#####O#P#####";
  char* p;
  cout << "The string is: [" << s << "]\nIts tokens are:\n";
  p = strtok(s, "#");
  while (p)
  { cout << "\t[" << p << "]\n";
    p = strtok(NULL, "#");
  }
  cout << "Now the string is: [" << s << "]\n";
}
```

8.21 Write a program that reads one line of text and then prints it with all its letters capitalized.

8.22 Write a program that reads one line of text and then prints it with all its blanks removed.

8.23 Write a program that reads one line of text and then prints the number of words that were read.

8.24 Write a program that reads one line of text and then prints the same words in reverse order. For example, the input

```
today is Tuesday
```

would produce the output

```
Tuesday is today
```

Answers to Review Questions

8.1 Among the 13 declarations:

a. The following are valid declarations for a C++ character string:

```
char s[6];
char s[6] = {'H', 'e', 'l', 'l', 'o'};
char s[6] = "Hello";
char s[] = {'H', 'e', 'l', 'l', 'o'};
char s[] = "Hello";
char* s;
```

```
char* s = new char[6];
char* s = "Hello";
```

Warning: this last declaration only defines `s` to be a pointer to a string constant.

b. The following are valid declarations for a C++ character C-string of length 5, initialized to the C-string `"Hello"` and allocated at compile time:

```
char s[6] = {'H', 'e', 'l', 'l', 'o'};
char s[6] = "Hello";
char s[] = {'H', 'e', 'l', 'l', 'o'};
char s[] = "Hello";
char* s = "Hello";  // defines s as a pointer to a string constant
```

c. It is not possible to initialize a C-string like this at run time.

d. The following are valid declarations for a C++ character string as a formal parameter for a function:

```
char s[];
char* s;
```

8.2 This will read only as far as the first whitespace. For the given input, it would assign `"Hello,"` to `s`.

8.3 This counts the number of uppercase letters in the C-string `s`, so the output is `6`.

8.4 This changes all uppercase letters to lowercase in the C-string `s`:

```
123 w. 42nd st., ny, ny 10020-1095
```

Note that to change the case of a character `*p`, it must be assigned the return value of the function:

```
*p = tolower(*p);
```

8.5 This increments all uppercase letters, changing the `W` to an `X`, the `S` to a `T`, *etc.*:

```
123 X. 42nd Tt., OZ, OZ 10020-1095
```

8.6 This counts the number of punctuation characters in the C-string `s`, so the output is 5.

8.7 It changes each character that is followed by a punctuation character to that following character:

```
123 .. 42nd S.,, N,, NY 1002--1095
```

8.8 The assignment `s1 = s2` simply makes `s1` a synonym for `s2`; *i.e.*, they both point to the same character. The call `strcpy(s1,s2)` actually copies the characters of `s2` into the C-string `s1`, thereby duplicating the C-string.

8.9 ***a.*** This assigns the integer 10 to `n`.

b. This assigns the substring `"rford"` to `s1`.

c. This assigns the substring `"rd"` to `s1`.

d. This assigns the substring `"utherford"` to `s1`.

e. This copies `last` to `first`, so that `first` will also be the string `"Hayes"`.

f. This copies the substring `"Hay"` into the first part of `first`, making it `"Hayherford"`.

g. This appends `last` onto the end of `first`, making it `"RutherfordHayes"`.

h. This appends the substring `"Hay"` onto the end of `first`, making it `"RutherfordHay"`.

8.10 ***a.*** 7.

b. 6.

c. 5.

d. 7.

8.11 It prints: `ABCDE >= ABC`

8.12 It prints: `ABCDE < ABCE`

8.13 It prints: `ABCDE >=`

8.14 It prints: `!=`

Solutions to Problems

8.1 This does not work because the assignment

```
name[count] = buffer;
```

assigns the same pointer to each of the C-strings `name[0]`, `name[1]`, *etc*. Arrays cannot be assigned this way. To copy one array into another, use `strcpy()`, or `strncpy()`.

8.2 This copies the C-string `s2` into the C-string `s1`:

```
char* strcpy(char* s1, const char* s2)
{ char* p; for (p=s1; *s2; )
    *p++ = *s2++;
  *p = '\0';
  return s1;
}
```

The pointer `p` is initialized at the beginning of `s1`. On each iteration of the `for` loop, the character `*s2` is copied into the character `*p`, and then both `s2` and `p` are incremented. The loop continues until `*s2` is 0 (*i.e.*, the null character `'\0'`). Then the null character is appended to the C-string `s1` by assigning it to `*p`. (The pointer `p` was left pointing to the byte after the last byte copied when the loop terminated.) Note that this function does not allocate any new storage. So its first argument `s1` should already have been defined to be a character string with the same length as `s2`.

8.3 This function appends up to `n` characters from `s2` onto the end of `s1`. It is the same as the `strcat()` function except that its third argument `n` limits the number of characters copied:

```
char* strncat(char* s1, const char* s2, size_t n)
{ char* end; for (end=s1; *end; end++)  // find end of s1
    ;
  char* p; for (p=s2; *p && p-s2<n; )
    *end++ = *p++;
  *end = '\0';
  return s1;
}
```

The first **`for`** loop finds the end of C-string `s1`. That is where the characters from C-string `s2` are to be appended. The second **`for`** loop copies characters from `s2` to the locations that follow `s1`. Notice how the extra condition `p-s2<n` limits the number of characters copied to `n`: the expression `p-s2` equals the number of characters copied because it is the difference between `p` (which points to the next character to be copied) and `s2` (which points to the beginning of the C-string). Note that this function does not allocate any new storage. It requires that C-string `s1` have at least *k* more bytes allocated, where *k* is the smaller of `n` and the length of C-string `s2`.

8.4 This requires testing the last letter and the second from last letter of the word to be pluralized. We use pointers `p` and `q` to access these letters.

```
void pluralize(char* s)
{ int len = strlen(s);
  char* p = s + len - 1;    // last letter
  char* q = s + len - 2;    // last 2 letters
  if (*p == 'h' && (*q == 'c' || *q == 's')) strcat(p, "es");
  else if (*p == 's') strcat(p, "es");
  else if (*p == 'y')
    if (isvowel(*q)) strcat(p, "s");
    else strcpy(p, "ies");
  else if (*p == 'z')
    if (isvowel(*q)) strcat(p, "zes");
    else strcat(p, "es");
  else strcat(p, "s");
}
```

Two of the tests depend upon whether the second from last letter is a vowel, so we define a little boolean function `isvowel()` for testing that condition:

```
bool isvowel(char c)
{ return (c=='a' || c=='e' || c=='i' || c=='o' || c=='u');
}
```

The test driver repeatedly reads a word, prints it, pluralizes it, and prints it again. The loop terminates when the user enters a single blank for a word:

```
bool pluralize(char*);
int main()
{ char word[80];
  for (;;)
  { cin.getline(word, 80);
    if (*word == ' ') break;
    cout << "\tThe singular is [" << word << "].\n";
    pluralize(word);
    cout << "\t  The plural is [" << word << "].\n";
  }
}
```

```
        wish
            The singular is [wish].
              The plural is [wishes].
        hookah
            The singular is [hookah].
              The plural is [hookahs].
        bus
            The singular is [bus].
              The plural is [buses].
        toy
            The singular is [toy].
              The plural is [toys].
        navy
            The singular is [navy].
              The plural is [navies].
        quiz
            The singular is [quiz].
              The plural is [quizzes].
        quartz
            The singular is [quartz].
              The plural is [quartzes].
        computer
            The singular is [computer].
              The plural is [computers].
```

8.5 We assume that names have no more than 20 characters and that there will be no more than 25 names. We'll read all the input in at once and store it all in a single `buffer`. Since each name will be terminated with a `NUL` character, the `buffer` needs to be large enough to hold 25*(20 + 1) + 1 characters (25 21-character strings plus one last `NUL` character). The program is modularized into five function calls. The call `input(buffer)` reads everything into the `buffer`. The call `tokenize(name, numNames, buffer)` "tokenizes" the `buffer`, storing pointers to its names in the `name` array and returning the number of names in `numNames`. The call `print(name, numNames)` prints all the names that are stored in `buffer`. The call `sort(name, numNames)` does an *indirect sort* on the names stored in `buffer` by rearranging the pointers stored in the `name` array.

```
#include <cstring>
#include <iostream>
using namespace std;
const int NAME_LENGTH = 20;
const int MAX_NUM_NAMES = 25;
```

```
const int BUFFER_LENGTH = MAX_NUM_NAMES*(NAME_LENGTH + 1);
void input(char* buffer);
void tokenize(char** name, int& numNames, char* buffer);
void print(char** name, int numNames);
void sort(char** name, int numNames);
int main()
{ char* name[MAX_NUM_NAMES];
  char buffer[BUFFER_LENGTH+1];
  int numNames;
  input(buffer);
  tokenize(name, numNames, buffer);
  print(name, numNames);
  sort(name, numNames);
  print(name, numNames);
}
```

The entire input is done by the single call `cin.getline(buffer, BUFFER_LENGTH, '$')`. This reads characters until the "`$`" character is read, storing all the characters in `buffer`.

```
void input(char* buffer)
{ // reads up to 25 strings into buffer:
  cout << "Enter up to " << MAX_NUM_NAMES << " names, one per"
       << " line.  Terminate with \'$\'.\nNames are limited to "
       << NAME_LENGTH << " characters.\n";
  cin.getline(buffer, BUFFER_LENGTH, '$');
}
```

The `tokenize()` function uses the `strtok()` function to scan through the `buffer`, "tokenizing" each substring that ends with the newline character `'\n'` and storing its address in the `name` array. The `for` loop continues until `p` points to the sentinel `'$'`. Notice that the function's `name` parameter is declared as a `char**` because it is an array of pointers to `char`s. Also note that the counter `n` is declared as an `int&` (passed by reference) so that its new value is returned to `main()`.

```
void tokenize(char** name, int& n, char* buffer)
{ // copies address of each string in buffer into name array:
  char* p = strtok(buffer, "\n");        // p points to each token
  for (n = 0; p && *p != '$'; n++)
  { name[n] = p;
    p = strtok(NULL, "\n");
  }
}
```

The `print()` and `sort()` functions are similar to those seen before, except that both operate here indirectly. Both functions operate on the `name` array.

```
void print(char** name, int n)
{ // prints the n names stored in buffer:
  cout << "The names are:\n";
  for (int i = 0; i < n; i++)
    cout << "\t" << i+1 << ". " << name[i] << endl;
}
void sort(char** name, int n)
{ // sorts the n names stored in buffer:
  char* temp;
  for (int i = 1; i < n; i++)                          // Bubble Sort
    for (int j = 0; j < n-i; j++)
      if (strcmp(name[j], name[j+1]) > 0)
      { temp = name[j];
```

```
            name[j] = name[j+1];
            name[j+1] = temp;
          }
}
Enter up to 25 names, one per line.  Terminate with '$'.
Names are limited to 20 characters.
Washington, George
Adams, John
Jefferson, Thomas
Madison, James
Monroe, James
Adams, John Quincy
Jackson, Andrew
$The names are:
        1. Washington, George
        2. Adams, John
        3. Jefferson, Thomas
        4. Madison, James
        5. Monroe, James
        6. Adams, John Quincy
        7. Jackson, Andrew
The names are:
        1. Adams, John
        2. Adams, John Quincy
        3. Jackson, Andrew
        4. Jefferson, Thomas
        5. Madison, James
        6. Monroe, James
        7. Washington, George
```

On this sample run the user entered 7 names and then the sentinel "`$`". The names were then printed, sorted, and printed again.

8.6 The function first locates the end of the C-string. Then it swaps the first character with the last character, the second character with the second from last character, *etc.*:

```
void reverse(char* s)
{ char* end, temp;
  for (end = s; *end; end++)
    ;    // find end of s
  while (s < end - 1)
  { temp = *--end;
    *end = *s;
    *s++ = temp;
  }
}
```

The test driver uses the `getline()` function to read the C-string. Then it prints it, reverses it, and prints it again:

```
void reverse(char*);
int main()
{ char string[80];
  cin.getline(string, 80);
  cout << "The string is [" << string << "].\n";
  reverse(string);
  cout << "The string is [" << string << "].\n";
}
```

```
Today is Wednesday.
The string is [Today is Wednesday.].
The string is [.yadsendeW si yadoT].
```

8.7

```
int main()
{ char word[80];
   while (cin >> word)
     if (*word) cout << "\t\"" << word << "\"\n";
}
```

```
Today is Wednesday.
        "Today"
        "is"
        "Wednesday."
^Z
```

8.8

```
char* Strchr(const char* s, int c)
{ for (const char* p=s; p && *p; p++)
    if (*p==c) return (char*)p;
  return 0;
}
```

8.9

```
int numchr(const char* s, int c)
{ int n=0;
  for (const char* p=s; p && *p; p++)
    if (*p==c) ++n;
  return n;
}
```

8.10

```
char* Strrchr(const char* s, int c)
{ const char* pp=0;
  for (const char* p=s; p && *p; p++)
    if (*p==c) pp = p;
  return (char*)pp;
}
```

8.11

```
char* Strstr(const char* s1, const char* s2)
{ if (*s2==0) return (char*)s1;  // s2 is the empty string
  for ( ; *s1; s1++)
    if (*s1==*s2)
      for (const char* p1=s1, * p2=s2; *p1==*p2; p1++, p2++)
        if (*(p2+1)==0) return (char*)s1;
  return 0;
}
```

8.12

```
char* Strncpy(char* s1, const char* s2, size_t n)
{ char* p=s1;
  for ( ; n>0 && *s2; n--)
    *p++ = *s2++;
  for ( ; n>0; n--)
    *p++ = 0;
  return s1;
}
```

8.13
```
char* Strcat(char* s1, const char* s2)
{ char* p=s1;
  for ( ; *p; p++)
    ;
  for ( ; *s2; p++, s2++)
    *p = *s2;
  *p = 0;
  return s1;
}
```

8.14
```
int Strcmp(char* s1, const char* s2)
{ for ( ; *s1==*s2; s1++, s2++)
    if (*s1==0) return 0;
  return (int)(*s1-*s2);
}
```

8.15
```
int Strncmp(char* s1, const char* s2, size_t n)
{ for ( ; n>0; s1++, s2++, n--)
    if (*s1!=*s2) return (int)(*s1-*s2);
    else if (*s1==0) return 0;
  return 0;
}
```

8.16
```
size_t Strspn(const char* s1, const char* s2)
{ const char *p1, *p2;
  for ( p1 = s1 ; *p1; p1++)
    for ( p2 = s2 ;  ; p2++ )
      if ( *p2 == '\0' )  // end of s2 reached; no match found
         return ( p1 - s1 ) ;  // so *p1 is not in s2[]
      else if ( *p1 == *p2 ) // *p1 is not the one
         break ;   // aborts inner for loop
  return ( p1 - s1 ) ;  // returning length of s1
}
```

8.17
```
size_t Strcspn(const char* s1, const char* s2)
{ const char *p1, *p2;
  for ( p1 = s1 ; *p1; p1++)
    for ( p2 = s2 ; *p2 ; p2++ )
      if ( *p1 == *p2 )          // *p1 found in s2[]
         return ( p1 - s1 ) ;  // and p1-s1 is its index
  return ( p1 - s1 ) ;  // returning length of s1
}
```

8.18
```
char* Strpbrk(const char* s1, const char* s2)
{ const char *p1, *p2;
  for ( p1 = s1 ; *p1; p1++)
    for ( p2 = s2 ; *p2 ; p2++ )
      if ( *p1 == *p2 )          // *p1 found in s2[]
         return (char*) p1 ;       // so returns its address
  return NULL ;  // no character of s1 is in s2[]
}
```

8.19
```
int freqInWords(const char* sentence, char ch)
{ int count = 0 ;
  char* copy = new char[ strlen(sentence) ] ;
  copy = strcpy( copy, sentence ) ;
  if ( copy == NULL )  return 0 ;
  char *p = strtok(copy, "\t\n \v\f\r" ) ;
```

```
  while (p) {
    for ( int i = 0  ; p[i]  ; i++ )
      if (p[i] == ch)    // ch found in current word
      { count++ ;        // referenced by p
        break   ;        // finished with current word
      }   // end if (p[i] == ch)
    p = strtok(NULL, "\t\n \v\f\r"  ) ; // advance to next word
  }     // end while (p)
  return count ;  //
}
```

8.20

8.21

```
void capitalize(char* s)
{ if (s == NULL) return;
  for (char* p=s; *p; p++)
    if (*p>='a' && *p<='z')*p = (char)(*p - 'a' + 'A');
}
```

8.22

```
void removeBlanks( char* s)
{ if ( s == NULL ) return ;
  int j = 0 ;
  for ( int i = 0; s[i] ; i++ )
    if ( s[i] != ' ' ) s[j++] = s[i] ;
  s[j] = '\0' ;
}
```

8.23

```
int numWords( const char* s)
{ if ( s == NULL ) return 0 ;
  int wordCount = 0 ;
  char * Copy = new char[ strlen(s) ] ;
  Copy = strcpy( Copy, s ) ;
  char * p = strtok( Copy, "\n \v\t\f\r" ) ;
  while ( p )
  { char ch0 = p[0];  // check whether first char is letter
    if ( ((ch0 >= 'a') && (ch0 <= 'z') ) ||   // lowercase
         ((ch0 >= 'A') && (ch0 <= 'Z') ) )    // uppercase
          wordCount++ ;
    p = strtok( NULL, "\n \v\t\f\r" ) ;
  }
  return wordCount ;
}
```

8.24

```
char*  reverseWords(char* reverseS, const char* s)
{  if ( (reverseS == NULL) || (s == NULL) ) return NULL;
   char * Copy  = new char[ strlen(s) ] ;
   Copy = strcpy( Copy , s ) ;
   char * currentReverse = new char[ strlen(s) ] ;
   char * revPtr = reverseS ;
   *revPtr = '\0' ;  // reverse starts with no words
   char * pS;
   pS = strtok(Copy, " \t" ) ;  // words separated by space or tab
   while ( pS )
   {  // reverseS = currentWordInS + currentReverse
      currentReverse = strcpy( currentReverse, revPtr ) ;
      revPtr = addWords( revPtr, pS, currentReverse ) ;
     pS = strtok( NULL, " \t" ) ;  // advance pS to next word in s
```

```
   }    // end while (pS)
   return revPtr ;
}

char*  addWords( char* leftPLUSright, const char* left,
                 const char* right )
{    char * both = leftPLUSright ;
     const char * pLeft = left ;
     const char * pRight = right ;
     while ( *pLeft )
        *(both++) = *(pLeft++) ;
     if ( *left && *right )  // both words nonempty
        *(both++) = ' ' ;    // so put space between
     while ( *pRight )
        *(both++) = *(pRight++) ;
     *both = '\0' ;    // terminate new string with null character
     return leftPLUSright ;
}
```

Chapter 9

Standard C++ Strings

9.1 INTRODUCTION

The classic C-strings described in Chapter 8 are an important part of C++. They provide a very efficient means for fast data processing. But as with ordinary arrays, the efficiency of C-strings comes at a price: the risk of run-time errors, resulting primarily from their dependency upon the use of the `NUL` character as a string terminator.

Standard C++ strings provide a safe alternative to C-strings. By encapsulating the length of the string with the string itself, there is no direct reliance on string terminators.

9.2 FORMATTED INPUT

Recall the idea of a stream in C++ as a conduit through which data passes. Input passes through an `istream` object and output passes through an `ostream` object. The `istream` class defines the behavior of objects like `cin`. The most common behavior is the use of the *extraction operator* `>>` (also called the *input operator*). It has two operands: the `istream` object from which it is extracting characters, and the object to which it copies the corresponding value formed from those characters. This process of forming a typed value from raw input characters is called *formatting*.

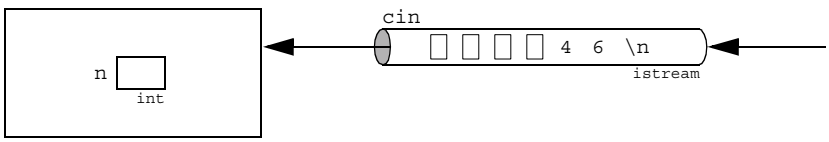
EXAMPLE 9.1 The Extraction Operator `>>` Performs Formatted Input

Suppose the code

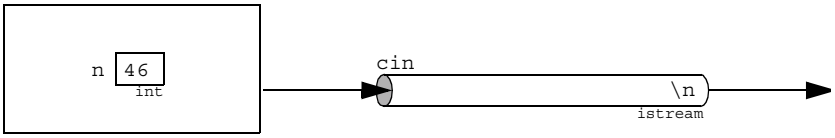
```
int n;
cin >> n;
```

executes on the input

```
46
```



This input actually contains the 7 characters: `' '`, `' '`, `' '`, `' '`, `'4'`, `'6'`, `'\n'` (four blanks followed by a 4, a 6, and the newline character). It could be viewed as coming through the input stream. The stream object `cin` scans characters one at a time. If the first character it sees is a whitespace character (a blank, a tab, a newline, *etc.*), it extracts it and ignores it. It continues to extract and ignore the characters in the stream until it encounters a non-whitespace character. In this example, that would be the `'4'`. Since the second operand of the expression `cin >> n` has type `int`, the `cin` object is looking for digits to form an integer. So after "eating" any preceding whitespace, it expects to find one of the 12 characters `'+'`, `'-'`, `'0'`, `'1'`, `'2'`, `'3'`, `'4'`, `'5'`, `'6'`, `'7'`, `'8'`, or `'9'`. If it encounters any of the other 244 characters, it will fail. In this case, it sees the `'4'`. So it extracts it and then continues, expecting more digits. As long as it encounters only digits, it continues to extract them. As soon as it sees a non-digit, it stops, leaving that non-digit in the stream. In this case, that means that `cin` will extract exactly 6 characters: the 4 blanks, the `'4'`, and the `'6'`. It discards the 4 blanks and then combines the `'4'` and the `'6'` to form the integer value 46. Then it copies that value into the object `n`.



After that extraction has finished, the newline character is still in the input stream. If the next input statement is another formatted input, then like all whitespace characters that newline character will be ignored.

The extraction operator `>>` *formats* the data that it receives through its input stream. This means that it extracts characters from the stream and uses them to form a value of the same type as its second operand. In the process it ignores all whitespace characters that precede the characters it uses. A direct consequence of this rule is that it is impossible to use the extraction operator to read whitespace characters. For that you must use an unformatted input function.

The operator expression

```
cin >> x
```

has a value that can be interpreted in a condition as boolean; *i.e.*, either `true` or `false` depending upon whether the input is successful. That allows such an expression to be used to control a loop.

EXAMPLE 9.2 Using the Extraction Operation to Control a Loop

```
int main()
{ int n;
  while (cin >> n)
    cout << "n = " << n << endl;
}
```

```
46
n = 46
22    44    66    88
n = 22
n = 44
n = 66
n = 88
33, 55, 77, 99
n = 33
```

The loop continues iterating as long as the integer data is separated by only whitespace. The first non-whitespace character, the comma `','` causes the input to fail, thereby stopping the loop.

9.3 UNFORMATTED INPUT

The `<iostream>` files define several functions inputting characters and C-strings that do not skip over whitespace. The most common are the `cin.get()` function for reading individual characters and the `cin.getline()` function for reading C-strings.

EXAMPLE 9.3 Inputting Characters with the `cin.get()` Function

```
while (cin.get(c))
{ if (c >= 'a' && c <= 'z') c += 'A' - 'a';  // capitalize c
  cout.put(c);
```

```
    if (c == '\n') break;
  }
```

This loop is controlled by the input expression `(cin.get(c))`. When the input stream object `cin` detects the end-of-file (signaled interactively by Ctrl+Z or Ctrl+D), the expression evaluates to `false` and stops the loop. This loop also terminates with a `break` statement after reading and processing the newline character `'\n'`. The `if` statement simply capitalizes all lowercase letters, and the `cout.put(c)` statement prints the character.

Here is a sample run:

```
Cogito, ergo sum!
COGITO, ERGO SUM!
```

EXAMPLE 9.4 Inputting C-Strings with the `cin.getline()` Function

This program shows how to read text data line-by-line into an array of C-strings:

```
const int LEN=32;         // maximum word length
const int SIZE=10;        // array size
typedef char Name[LEN];   // defines Name to be a C-string type
int main()
{ Name king[SIZE];        // defines king to be an array of 10 names
  int n=0;
  while(cin.getline(king[n++], LEN) && n<SIZE)
    ;
  --n;                    // now n == the number of names read
  for (int i=0; i<n; i++)
    cout << '\t' << i+1 << ". " << king[i] << endl;
}
```

The object `king` is an array of 10 objects of type `Name`. The `typedef` defines `Name` as a synonym for C-strings of 32 `char`s (31 non-null). The function call `cin.getline(king[n++], LEN)` reads characters from `cin` until either it has extracted `LEN`-1 characters or it encounters the newline character, whichever comes first. It copies these characters into the C-string `king[n]`. If it encounters the newline character, it extracts it and ignores it (*i.e.*, it does not copy it into the C-string). Then it increments `n`.

Note that the body of the `while` loop is empty. The loop stops when either `cin` detects the end-of-file or when `n == SIZE`. Since `n` starts at 0 and is incremented after the last name is read, its value is always 1 greater than the number of names read. So it gets decremented once at the end so that its value equals the number of names read. Then it is easy to print them or process them in other ways using a simple `for` loop.

`Kings.dat`

```
Kenneth II (971-995)
Constantine III (995-997)
Kenneth III (997-1005)
Malcolm II (1005-1034)
Duncan I (1034-1040)
Macbeth (1040-1057)
Lulach (1057-1058)
Malcolm III (1058-1093)
```

When input is read from this text file, the output is

```
        1. Kenneth II (971-995)
        2. Constantine III (995-997)
        3. Kenneth III (997-1005)
        4. Malcolm II (1005-1034)
        5. Duncan I (1034-1040)
        6. Macbeth (1040-1057)
        7. Lulach (1057-1058)
        8. Malcolm III (1058-1093)
```

9.4 THE STANDARD C++ `string` TYPE

Standard C++ defines its **`string`** type in the `<string>` header file. Objects of type `string` can be declared and initialized in several ways:

```
string s1;               // s1 contains 0 characters
string s2 = "New York";  // s2 contains 8 characters
string s3(60, '*');      // s3 contains 60 asterisks
string s4 = s3;          // s4 contains 60 asterisks
string s5(s2, 4, 2);     // s5 is the 2-character string "Yo"
```

If the `string` is not initialized, like `s1` here, then it represents the empty string containing 0 characters. A `string` can be initialized the same way a C-string is, like `s2` here. Or a `string` can be initialized to hold a given number of the same character, like `s3` here which holds 60 stars. Unlike a C-string, C++ `string` objects can be initialized with a copy of another existing `string` object, like `s4` here, or with a substring of an existing string, like `s5`. Note that the standard substring designator has three parts: the parent string (`s2`, here), the starting character (`s2[4]`, here), and the length of the substring (`2`, here).

Formatted input works the same way for C++ `string`s as it does for C-strings: preceding whitespace is skipped, and input is halted at the end of the first whitespace-terminated word. C++ `string`s have a `getline()` function that works almost the same way as the `cin.getline()` function for C-strings:

```
string s = "ABCDEFG";
getline(cin, s);         // reads the entire line of characters into s
```

They also use the subscript operator the same way that C-strings do:

```
char c = s[2];               // assigns 'C' to c
s[4] = '*';                  // changes s to "ABCD*FG"
```

Note that the array index always counts how many characters precede the indexed character. C++ `string`s can be converted to C-strings like this:

```
const char* cs = s.c_str();  // converts s into the C-string cs
```

The `c_str()` function has return type `const char*`.

The C++ `string` class also defines a `length()` function that can be used like this to determine how many characters are stored in a `string`:

```
cout << s.length() << endl;  // prints 7 for the string s == "ABCD*FG"
```

C++ `string`s can be compared using the relational operators like fundamentals types:

```
if (s2 < s5) cout << "s2 lexicographically precedes s5\n";
while (s4 == s3) //...
```

You can also concatenate and append `string`s using the `+` and `+=` operators:

```
string s6 = s + "HIJK";      // changes s6 to "ABCD*FGHIJK"
s2 += s5;                    // changes s2 to "New YorkYo"
```

The `substring()` function is used like this:

```
s4 = s6.substr(5,3);         // changes s4 to "FGH";
```

The `erase()` and `replace()` function work like this:

```
s6.erase(4, 2);              // changes s6 to "ABCDGHIJK"
s6.replace(5, 2, "xyz");     // changes s6 to "ABCDGxyzJK"
```

The `find()` function returns the index of the first occurrence of a given substring:

```
string s7 = "Mississippi River basin";
cout << s7.find("si") << endl;  // prints 3
cout << s7.find("so") << endl;  // prints 23, the length of the string
```

If the `find()` function fails, it returns the length of the string it was searching.

EXAMPLE 9.5 Using the Standard C++ `string` Type

This code adds a nonsense syllable after each "`t`" that precedes a vowel. For example, the sentence

```
The first step is to study the status of the C++ Standard.
```

is replaced by the sentence:

```
The first stegep is tego stegudy the stegatus of the C++ Stegandard.
```

It uses an auxiliary boolean function named `is_vowel()`:

```
string word;
int k;
while (cin >> word)
{ k = word.find("t") + 1;
  if (k < word.length() && is_vowel(word[k]))
    word.replace(k, 0, "eg");
  cout << word << ' ';
}
```

The `while` loop is controlled by the input, terminating when the end-of-file is detected. It reads one word at a time. If the letter `t` is found and if it is followed by a vowel, then `e.g.` is inserted between that `t` and the vowel.

9.5 FILES

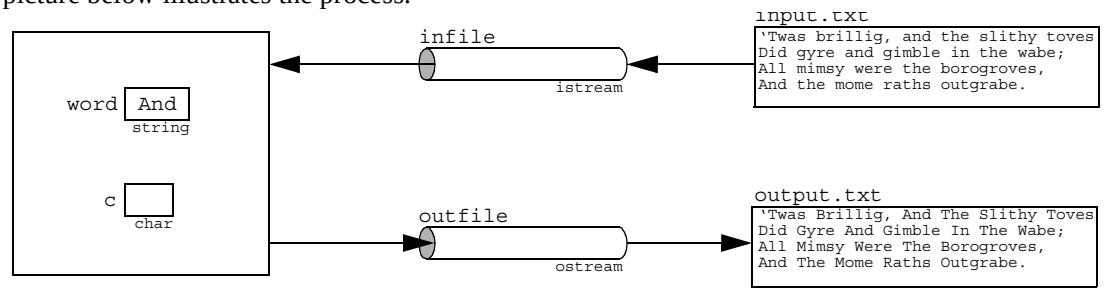
File processing in C++ is very similar to ordinary interactive input and output because the same kind of stream objects are used. Input from a file is managed by an `ifstream` object the same way that input from the keyboard is managed by the `istream` object `cin`. Similarly, output to a file is managed by an `ofstream` object the same way that output to the monitor or printer is managed by the `ostream` object `cout`. The only difference is that `ifstream` and `ofstream` objects have to be declared explicitly and initialized with the external name of the file which they manage. You also have to `#include` the `<fstream>` header file (or `<fstream.h>` in pre-Standard C++) that defines these classes.

EXAMPLE 9.6 Capitalizing All the Words in a Text File

Here is a complete program that reads words from the external file named `input.txt`, capitalizes them, and then writes them to the external file named `output.txt`:

```
#include <fstream>
#include <iostream>
using namespace std;
int main()
{ ifstream infile("input.txt");
  ofstream outfile("output.txt");
  string word;
  char c;
  while (infile >> word)
  { if (word[0] >= 'a' && word[0] <= 'z') word[0] += 'A' - 'a';
    outfile << word;
    infile.get(c);
    outfile.put(c);
  }
}
```

The picture below illustrates the process.

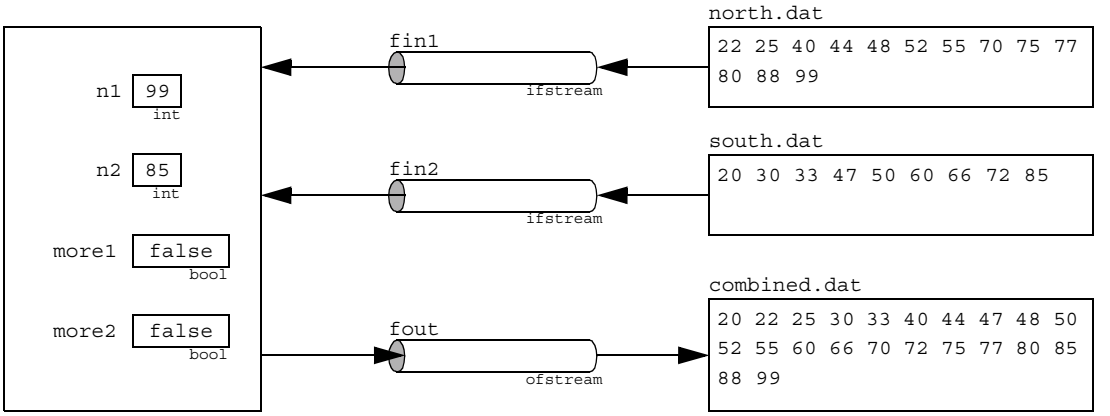


Notice that the program has four objects: an `ifstream` object named `infile`, an `ofstream` object named `outfile`, a `string` object named `word`, and a `char` object named `c`.

The advantage of using external files instead of command line redirection is that there is no limit to the number of different files that you can use in the same program.

EXAMPLE 9.7 Merging Two Sorted Data Files

This program merges two files into a third file. The numbers stored in the files `north.dat` and `south.dat` are sorted in increasing order. The program reads these two input files simultaneously and copies all their data to the file `combined.dat` so that they are all together in increasing order:




```
bool more(ifstream& fin, int& n)
{ if (fin >> n) return true;
  else return false;
}
bool copy(ofstream& fout, ifstream& fin, int& n)
{ fout << " " << n;
  return more(fin, n);
}
int main()
{ ifstream fin1("north.dat");
  ifstream fin2("south.dat");
  ofstream fout("combined.dat");
  int n1, n2;
  bool more1 = more(fin1, n1);
  bool more2 = more(fin2, n2);
```

```
  while (more1 && more2)
    if (n1 < n2) more1 = copy(fout, fin1, n1);
    else more2 = copy(fout, fin2, n2);
  while (more1)
    more1 = copy(fout, fin1, n1);
  while (more2)
    more2 = copy(fout, fin2, n2);
  fout << endl;
}
```

The `more()` function is used to read the data from the input files. Each call attempts to read one integer from the `fin` file to the reference parameter `n`. It returns `true` if it is successful, otherwise `false`. The `copy()` function writes the value of `n` to the `fout` file and then calls the `more()` function to read the next integer from the `fin` file into `n`. It also returns `true` if and only if it is successful.

The first two calls to the `more()` function read 22 and 20 into `n1` and `n2`, respectively. Both calls return `true` which allows the main `while` loop to begin. On that first iteration, the condition `(n1 < n2)` is false, so the `copy()` function copies 20 from `n2` into the `combined.dat` file and then calls the `more()` function again which reads 30 into `n2`. On the second iteration, the condition `(n1 < n2)` is true (because 22 < 30), so the `copy()` function copies 22 from `n1` into the `combined.dat` file and then calls the `more()` function again which reads 25 into `n1`. The next iteration writes 25 to the output file and then reads 40 into `n1`. The next iteration writes 30 to the output file and then reads 33 into `n2`. This process continues until 85 is written to the output file from `n2` and the next call to `more()` fails, assigning `false` to `more2`. That stops the main `while` loop. Then the second `while` loop iterates three times, copying the last three integers from `north.dat` to `combined.dat` before it sets `more1` to `false`. The last loop does not iterate at all.

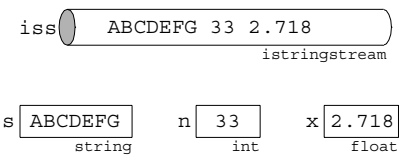
Note that file objects (`fin1`, `fin2`, `fout`) are passed to function the same way any other objects are passed. However, they must always be passed by reference.

9.6 STRING STREAMS

A *string stream* is a stream object that allows a `string` to be used as an internal text file. This is also called *in-memory I/O*. String streams are quite useful for buffering input and output. Their types `istringstream` and `ostringstream` are defined in the `<sstream>` header file.

EXAMPLE 9.8 Using an Output String Stream

This program creates four objects: a character string `s`, an integer `n`, a floating-point number `x`, and an output string stream `oss`:




```
#include <iostream>
#include <sstream>
#include <string>
using namespace std;
void print(ostringstream&);
int main()
{ string s="ABCDEFG";
  int n=33;
  float x=2.718;
  ostringstream oss;
```

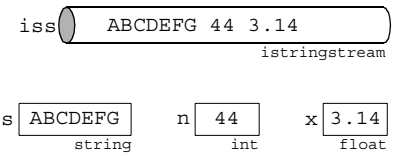
```
  print(oss);
  oss << s;
  print(oss);
  oss << " " << n;
  print(oss);
  oss << " " << x;
  print(oss);
}
void print(ostringstream& oss)
{ cout << "oss.str() = \"" << oss.str() << "\"" << endl;
}
```

```
oss.str() = ""
oss.str() = "ABCDEFG"
oss.str() = "ABCDEFG 33"
oss.str() = "ABCDEFG 33 2.718"
```

The output string stream object `oss` acts like the output stream object `cout`: the values of the string `s`, the integer `n`, and the number `x` are written to it by means of the insertion operator `<<`.

While the internal object `oss` is like an external text file, its contents can be accessed as a `string` object by the call `iss.str()`.

EXAMPLE 9.9 Using an Input String Stream



This program is similar to the one in Example 9.8 except that it reads from an input string stream `iss` instead of writing to an output string stream.:

```
void print(string&,int,float,istringstream&);
int main()
{ string s;
  int n=0;
  float x=0.0;
  istringstream iss("ABCDEFG 44 3.14");
  print(s,n,x,iss);
  iss >> s;
  print(s,n,x,iss);
  iss >> n;
  print(s,n,x,iss);
  iss >> x;
  print(s,n,x,iss);
}

void print(string& s, int n, float x, istringstream& iss)
{ cout << "s = \"" << s << "\", n = " << n << ", x = " << x
       << ", iss.str() = \"" << iss.str() << "\"" << endl;
}
```

```
s = "", n = 0, x = 0, iss.str() = "ABCDEFG 44 3.14"
s = "ABCDEFG", n = 0, x = 0, iss.str() = "ABCDEFG 44 3.14"
s = "ABCDEFG", n = 44, x = 0, iss.str() = "ABCDEFG 44 3.14"
s = "ABCDEFG", n = 44, x = 3.14, iss.str() = "ABCDEFG 44 3.14"
```

The input string stream object `iss` acts like the input stream object `cin`: values for the string `s`, the integer `n`, and the number `x` are read from it by means of the extraction operator `>>`. But the `iss` object also acts like an external file: reading from it does not change its contents.

Review Questions

9.1 What is the difference between a C-string and a C++ `string`?
9.2 What is the difference between formatted input and unformatted input?
9.3 Why can't whitespace be read with the extraction operator?
9.4 What is a stream?
9.5 How does C++ simplify the processing of strings, external files, and internal files?
9.6 What is the difference between sequential access and direct access?
9.7 What do the `seekg()` and `seekp()` functions do?
9.8 What do the `read()` and `write()` functions do?

Problems

9.1 Describe what the following code does:

```
char cs1[] = "ABCDEFGHIJ";
char cs2[] = "ABCDEFGH";
cout << cs2 << endl;
cout << strlen(cs2) << endl;
cs2[4] = 'X';
if (strcmp(cs1, cs2) < 0) cout << cs1 << " < " << cs2 << endl;
else cout << cs1 << " >= " << cs2 << endl;
char buffer[80];
strcpy(buffer, cs1);
strcat(buffer, cs2);
char* cs3 = strchr(buffer, 'G');
cout << cs3 << endl;
```

9.2 Describe what the following code does:

```
string s = "ABCDEFGHIJKLMNOP";
cout << s << endl;
cout << s.length() << endl;
s[8] = '!';
s.replace(8, 5, "xyz");
s.erase(6, 4);
cout << s.find("!");
cout << s.find("?");
cout << s.substr(6, 3);
s += "abcde";
string part(s, 4, 8);
string stars(8, '*');
```

9.3 Describe what happens when the code

```
string s;
int n;
float x;
cin >> s >> n >> x >> s;
```

executes on each of the following inputs:

a. `ABC 456 7.89 XYZ`
b. `ABC 4567 .89 XYZ`
c. `ABC 456 7.8 9XYZ`
d. `ABC456 7.8 9 XYZ`
e. `ABC456 7 .89 XYZ`
f. `ABC4 56 7.89XY Z`
g. `AB C456 7.89 XYZ`
h. `AB C 456 7.89XYZ`

9.4 Trace the execution of the merge program in Example 9.7 on page 218 on the following two data files:

north.dat

```
27 35 38 52 55 61 81 87
```

south.dat

```
31 34 41 45 49 56 63 74 92 95
```

Show each value of the variables `n1`, `n2`, `more1`, and `more2`, as they change.

9.5 Write a program that reads full names, one per line, and then prints them in the standard telephone directory format. For example, the input

```
Johann Sebastian Bach
George Frederic Handel
Carl Phillipp Emanuel Bach
Joseph Haydn
Johann Christian Bach
Wolfgang Amadeus Mozart
```

would be printed as:

```
Bach, Johann S.
Handel, George F.
Bach, Carl P. E.
Haydn, Joseph
Bach, Johann C.
Mozart, Wolfgang A.
```

9.6 Write a program that counts and prints the number of lines, words, and letter frequencies in its input. For example, the input:

```
Two roads diverged in a yellow wood,
And sorry I could not travel both
And be one traveler, long I stood
And looked down one as far as I could
To where it bent in the undergrowth;
```

would produce the output:

```
The input had 5 lines, 37 words,
and the following letter frequencies:
    A: 10   B: 3    C: 2    D: 13   E: 15   F: 1    G: 3    H: 4
    I: 7    J: 0    K: 1    L: 8    M: 0    N: 12   O: 20   P: 0
    Q: 0    R: 11   S: 5    T: 11   U: 3    V: 3    W: 6    X: 0
    Y: 2    Z: 0
```

9.7 Implement and test the following function:

```
void reduce(string& s);
// Changes all capital letters in s to lowercase
// and removes all non-letters from the beginning and end.
// EXAMPLE: if s == "'Tis,", then reduce(s) makes it "tis"
```

Hint: First write and test the following three boolean functions:

```
bool is_uppercase(char c);
bool is_lowercase(char c);
bool is_letter(char c);
```

9.8 Modify your program from Problem 9.6 so that it counts the frequencies of words instead of letters. For example, the input

```
[I] then went to Wm. and Mary college, to wit in the spring of
1760, where I continued 2 years.  It was my great good fortune,
and what probably fixed the destinies of my life that Dr. Wm.
Small of Scotland was then professor of Mathematics, a man
profound in most of the useful branches of science, with a happy
talent of communication, correct and gentlemanly manners, & an
enlarged & liberal mind.  He, most happily for me, became soon
```

```
attached to me & made me his daily companion when not engaged in
the school; and from his conversation I got my first views of the
expansion of science & of the system of things in which we are
placed.
```

would produce the output

```
The input had 11 lines and 120 words,
with the following frequencies:
                i:  3            then:  2            went:  1
               to:  3              wm:  2             and:  4
             mary:  1         college:  1             wit:  1
               in:  4             the:  6          spring:  1
               of: 11                :  6           where:  1
        continued:  1           years:  1              it:  1
              was:  2              my:  3           great:  1
             good:  1         fortune:  1            what:  1
         probably:  1           fixed:  1       destinies:  1
             life:  1            that:  1              dr:  1
            small:  1        scotland:  1       professor:  1
      mathematics:  1               a:  2             man:  1
         profound:  1            most:  2          useful:  1
         branches:  1         science:  2            with:  1
            happy:  1          talent:  1   communication:  1
          correct:  1     gentlemanly:  1         manners:  1
               an:  1        enlarged:  1         liberal:  1
             mind:  1              he:  1         happily:  1
              for:  1              me:  3          became:  1
             soon:  1        attached:  1            made:  1
              his:  2           daily:  1       companion:  1
             when:  1             not:  1         engaged:  1
           school:  1            from:  1    conversation:  1
              got:  1           first:  1           views:  1
        expansion:  1          system:  1          things:  1
            which:  1              we:  1             are:  1
           placed:  1
```

9.9 Write a program that right-justifies text. It should read and echo a sequence of left-justified lines and then print them in right-justified format. For example, the input

```
Listen, my children, and you shall hear
Of the midnight ride of Paul Revere,
On the eighteenth of April, in Seventy-five;
Hardly a man is now alive
Who remembers that famous day and year.
```

would be printed as

```
      Listen, my children, and you shall hear
         Of the midnight ride of Paul Revere,
On the eighteenth of April, in Seventy-five;
                    Hardly a man is now alive
      Who remembers that famous day and year.
```

9.10 Implement and test the following function:

```
string Roman(int n);
//  Returns the Roman numeral equivalent to the Hindu-Arabic
//  numeral n.
//  PRECONDITIONS: n > 0, n < 3888
//  EXAMPLES: Roman(1776) returns "MDCCLXXVI",
//    Roman(1812) returns "MDCCCXII", Roman(1945) returns "MCMXLV"
```

9.11 Implement and test the following function:

```
int HinduArabic(string s);
//  Returns the Hindu-Arabic numeral equivalent to the Roman
//  numeral given in the string s.
//  PRECONDITIONS: s contains a valid Roman numeral
//  EXAMPLES: HindArabic("MDCCLXXVI") returns 1776,
//    HindArabic("MDCCCXII") returns 1812
```

Note that this is the inverse of the `Roman()` function in Problem 9.10. [Hint: Write an auxiliary function `int v(string s, int i)` that returns the digit for the Roman numeral character `s[i]`; *e.g.*, `v("MDCCCXII", 1)` returns `500`.]

9.12 Implement Algorithm G.1 on page 403 to convert decimal numerals to hexadecimal:

```
string hexadecimal(int n);
//  Returns the hexadecimal numeral that represents n.
//  PRECONDITION: n >= 0
//  POSTCONDITION: each character in the returned string is a
//    hexadecimal digit and that string is the dexadecimal
//    equivalent of n
//  EXAMPLE: hexadecimal(11643) returns "2d7b"
```

[Hint: Write an auxiliary function `char c(int k)` that returns the hexadecimal character for the hexadecimal digit `k`; *e.g.*, `c(14)` returns `'e'`.]

9.13 Implement Algorithm G.2 on page 403 to convert hexadecimal numerals to decimal:

```
int decimal(string s);
//  Returns the decimal numeral that represents the hexadecimal
//  numeral stored in the string s.
//  PRECONDITION: s.length() > 0 and each s[i] is a hexadecimal
//    digit
//  POSTCONDITION: the returns value is the decimal equivalent
//  EXAMPLE: decimal("2d7b") returns 11643
```

Note that this is the inverse of the `hexadecimal()` function in Problem 9.12. [Hint: Write an auxiliary function `int v(string s, int i)` that returns the decimal digit for the hexadecimal character `s[i]`; *e.g.*, `v("2d7b", 3)` returns `12`.]

9.14 Implement and test the following function:

```
void reverse(string& s);
//  Reverses the string s.
//  POSTCONDITION: s[i] <--> s[len-i-1]
//  EXAMPLE: reverse(s) changes s = "ABCDEFG" into "GFEDCBA"
```

[Hint: Use a temporary string.]

9.15 Implement and test the following function:

```
bool is_palindrome(string s);
//  Returns true iff s is a palindrome
//  EXAMPLES: is_palindrome("RADAR") returns true,
//    is_palindrome("ABCD") returns false
```

9.16 Modify the program in Example 9.7 on page 218 so that it merges the two sorted files of names shown at the top of the next page, writing the resulting sorted lines both to a file named `Presidents.dat` and to `cout`:

[Hint: Use `getline(fin, s)`.]

Republicans

```
Bush, George Herbert Walker
Coolidge, Calvin
Eisenhower, Dwight David
Ford, Gerald Rudolph
Harding, Warren Gamaliel
Hoover, Herbert Clark
McKinley, William
Nixon, Richard Milhous
Reagan, Ronald Wilson
Roosevelt, Theodore
Taft, William Howard
```

Democrats

```
Carter, James Earl
Clinton, William Jefferson
Johnson, Lyndon Baines
Kennedy, John Fitzgerald
Roosevelt, Franklin
Truman, Harry S
Wilson, Woodrow
```

Answers to Review Questions

9.1 A C-string is an array of chars that uses the null character `'\0'` to mark the end of the string. A C++ string is an object whose `string` type is defined in the `<string>` file and which has a large repertoire of function, such as `length()` and `replace()`:

```
char cs[8] = "ABCDEFG";  // cs is a C-string
string s = "ABCDEFG";    // s is a C++ string
cout << s << " has " << s.length() << " characters.\n";
s.replace(4, 2, "yz");   // changes s to "ABCDyzG"
```

9.2 Formatted input uses the extraction operator `>>` which ignores whitespace. Unformatted input uses the `get()` and `getline()` functions. The `get()` function reads the next character in the input stream without ignoring whitespace. The `getline()` function reads all the rest of the characters in the input stream until it reaches the newline character `'\n'`, which it extracts and ignores.

9.3 Whitespace (blanks, tabs, newlines, *etc*.) cannot be read with the extraction operator because it ignores all whitespace.

9.4 A *stream* is an object that manages input and output between a program and a data source. C++ allows `<iostream>` objects for interactive I/O (*viz.*, `cin` and `cout`), `<fstream>` objects for external files, and `<sstream>` objects for internal files (string streams).

9.5 C++ simplifies the processing of strings, external files, and internal files, by defining the same family of functions and operations for all three. For example, the extraction operator >> works the same way for inputting a `double` from the keyboard, from an external file, or from a string stream.

9.6 Sequential access must begin at the beginning and access each element in order, one after the other. Direct access allows the access of any element directly by locating it by its index number or address. Arrays allow direct access. Magnetic tape has only sequential access, but CDs had direct access. If you are on a railroad train, to go from one car to another you must use sequential access. But when you board the train initially you have direct access. Direct access is faster than sequential access, but it requires some external mechanism (array index, file byte number, railroad platform).

9.7 The `seekg()` and `seekp()` functions position the get pointer and the put pointer, respectively, in an external file to allow direct access. For example, the call `input.seekg(24)` positions the get pointer at byte number 24 in the file bound to the file stream named `input`.

9.8 The `read()` and `write()` functions are used for direct access input and output, respectively, of external files. For example, the call `input.read(s.c_str(), n)` would copy `n` bytes to the string `s` directly from the file bound to the file stream named `input`.

Solutions to Problems

9.1

```
char cs1[] = "ABCDEFGHIJ";      // defines cs1 to be that C-string
char cs2[] = "ABCDEFGH";        // defines cs1 to be that C-string
cout << cs2 << endl;                              // prints: ABCDEFGH
cout << strlen(cs2) << endl;                              // prints: 8
cs2[4] = 'X';                           // changes cs2 to "ABCDXFGH"
if (strcmp(cs1, cs2) < 0) cout << cs1 << " < " << cs2 << endl;
else cout << cs1 << " >= " << cs2 << endl;
                                      // prints: ABCDEFGHIJ < ABCDXFGH
char buffer[80];  // defines buffer to be a C-string of < 80 chars
strcpy(buffer, cs1);              // changes buffer to "ABCDEFGHIJ"
strcat(buffer, cs2);     // changes buffer to "ABCDEFGHIJABCDXFGH"
char* cs3 = strchr(buffer, 'G');    // make cs3 point to buffer[6]
cout << cs3 << endl;                          // prints: GHIJABCDXFGH
```

9.2

```
string s = "ABCDEFGHIJKLMNOP";      // defines s to be that string
cout << s << endl;                     // prints: ABCDEFGHIJKLMNOP
cout << s.length() << endl;                              // prints: 16
s[8] = '!';                         // changes s to "ABCDEFGH!JKLMNOP"
s.replace(10, 5, "xyz");          // changes s to "ABCDEFGH!JxyzP"
s.erase(2, 4);                          // changes s to "ABGH!JxyzP"
cout << s.find("!") << endl;                              // prints: 4
cout << s.find("?") << endl;                             // prints: 10
cout << s.substr(3, 6) << endl;                   // prints: H!Jxyz
s += "abcde";                      // changes s to "ABGH!JxyzPabcde"
string part(s, 1, 10);           // defines part to be "BGH!JxyzPa"
string stars(8, '*');             // defines stars to be "********"
```

9.3

a. `ABC 456 7.89 XYZ`

Assigns `"ABC"` to `s`, 456 to `n`, 7.89 to `x`, and then `"XYZ"` to `s`.

b. `ABC 4567 .89 XYZ`

Assigns `"ABC"` to `s`, 4567 to `n`, 0.89 to `x`, and then `"XYZ"` to `s`.

c. `ABC 456 7.8 9XYZ`

Assigns `"ABC"` to `s`, 456 to `n`, 7.8 to `x`, and then `"9XYZ"` to `s`.

d. `ABC456 7.8 9 XYZ`

Assigns `"ABC456"` to `s`, and then crashes because 7.8 is not a valid integer literal.

e. `ABC456 7 .89 XYZ`

Assigns `"ABC456"` to `s`, 7 to `n`, 0.89 to `x`, and then `"XYZ"` to `s`.

f. `ABC4 5 67.89XY Z`

Assigns `"ABC4"` to `s`, 56 to `n`, and then crashes because `7.89XY` is not a valid `float` literal.

g. `AB C456 7.89 XYZ`

Assigns `"AB"` to `s` and then crashes because `C456` is not a valid integer literal. (Note that the hexidecimal numeral `c456`, which can also be written `C456`, would qualify as a valid integer literal. But on input, hexadecimal numerals must be prefixed with "`0x`", as in `0xc456`.)

h. `AB C 456 7.89XYZ`

Assigns `"ABC"` to `s` and then crashes because `C` is not a valid integer literal.

9.4 Tracing the merge program:

n1	n2	more1	more2
27	31	true	true
35			
	34		
	41		
38			
52			
	45		
	49		
	56		
55			
61			
	63		
81			
	74		
	92		
87		false	
	95		false

9.5

```
int main()
{ string word, first, last;
  char c;
  bool is_first, is_last = true;
  string name[32];
  int n=0;
  while (cin >> word)
  { cin.get(c);          // should be either a blank or a newline
    is_first = is_last;          //  current word is a first name
    is_last = bool(c == '\n');    //  current word is a last name
    if (is_first) first = word;
    else if (is_last) name[n++] = word + ", " + first;
    else first += " " + word.substr(0,1) + ".";    // add initial
  }
  --n;
  for (int i=0; i<n; i++)
    cout << '\t' << i+1 << ". " << name[i] << endl;
}
```

9.6

```
int main()
{ string word;
  const int SIZE=91;      // for frequency array (int('Z') == 90)
  int lines=0, words=0, freq[SIZE] = {0}, len;
  char c;
  while (cin >> word)
  { ++words;
    cin.get(c);
    if (c == '\n') ++lines;
    len = word.length();
    for (int i=0; i<len; i++)
    { c = word[i];
```

```
        if (c >= 'a' && c <= 'z') c += 'A' - 'a';    // capitalize c
        if (c >= 'A' && c <= 'Z') ++freq[c];              // count c
      }
    }
    cout << "The input had " << lines << " lines, " << words
         << " words,\nand the following letter frequencies:\n";
    for (int i=65; i<SIZE; i++)
    { cout << '\t' << char(i) << ": " << freq[i];
      if (i > 0 && i%8 == 0) cout << endl;      // print 8 to a line
    }
    cout << endl;
}
```

9.7

```
bool is_upper(char c)
{ return bool(c >= 'A' && c <= 'Z');
}
bool is_lower(char c)
{ return bool(c >= 'a' && c <= 'z');
}
bool is_letter(char c)
{ return bool(is_upper(c) || is_lower(c));
}
void reduce(string& s)
{ while (s.length() > 0 && !is_letter(s[0]))
    s.erase(0, 1);
  int k = s.length() - 1;
  while (k > 0 && !is_letter(s[k--]))
    s.erase(k+1, 1);
  int len = s.length();
  if (len == 0) return;
  for (int i=0; i<len; i++)
    if (is_upper(s[i])) s[i] += 'a' - 'A';
}
```

9.8

```
int main()
{ ifstream in("Pr0907.in");
  string s;
  const int SIZE=1000;     // assume at most 1000 different words
  string word[SIZE];                          // holds words read
  int lines=0, words=0, n=0, freq[SIZE]={0}, i;
  char c;
  while (in >> s)
  { reduce(s);
    if (s.length() == 0) continue;
    ++words;
    in.get(c);
    if (c == '\n') ++lines;                          // count line
    for (i=0; i<n; i++)
      if (word[i] == s) break;
    if (i == n) word[n++] = s;                // add word to list
    ++freq[i];                                       // count word
  }
  cout << "The input had " << lines << " lines and " << words
       << " words,\nwith the following frequencies:\n";
```

```
  for (int i=0; i<n; i++)
  { s = word[i];
    if (i > 0 && i%3 == 0) cout << endl;      // print 3 to a line
    cout << setw(16) << setiosflags(ios::right)
         << s.c_str() << ": " << setw(2) << freq[i];
  }
  cout << endl;
}
```

9.9

```
int main()
{ const int SIZE=100;  // maximum number of lines stored
  string line[SIZE], s;
  int n=0, len, maxlen=0;
  while (!cin.eof())
  { getline(cin, s);
    len = s.length();
    if (len > 0) cout << s << endl;
    if (len > maxlen) maxlen = len;
    line[n++] = s;
  }
  --n;                        // n == number of lines read
  for (int i=0; i<n; i++)
  { s = line[i];
    len = s.length();
    cout << string(maxlen-len, ' ') << s << endl;
  }
}
```

9.10

```
string Roman(int n)
{ int d3 = n/1000;  // the thousands digit
  string s(d3, 'M');
  n %= 1000;
  int d2 = n/100;  // the hundreds digit
  if (d2 == 9) s += "CM";
  else if (d2 >= 5)
  { s += "D";
    s += string(d2-5, 'C');
  }
  else if (d2 == 4) s += "CD";
  else s += string(d2, 'C');
  n %= 100;
  int d1 = n/10;  // the tens digit
  if (d1 == 9) s += "XC";
  else if (d1 >= 5)
  { s += "L";
    s += string(d1-5, 'X');
  }
  else if (d1 == 4) s += "XL";
  else s += string(d1, 'X');
  n %= 10;
  int d0 = n/1;  // the ones digit
  if (d0 == 9) s += "IX";
  else if (d0 >= 5)
  { s += "V";
```

```
    s += string(d0-5, 'I');
  }
  else if (d0 == 4) s += "IV";
  else s += string(d0, 'I');
  return s;
}
```

9.11

```
int v(string s, int i)
{ char c = s[i];
  if (c == 'M') return 1000;
  if (c == 'D') return 500;
  if (c == 'C') return 100;
  if (c == 'L') return 50;
  if (c == 'X') return 10;
  if (c == 'V') return 5;
  if (c == 'I') return 1;
  return 0;
}
int HindArabic(string s)
{ int n0=0, n1=0, n=0;
  for (int i=0; i<s.length(); i++)
  { n0 = n1;
    n += n1 = v(s,i);
    if (n1>n0) n -= 2*n0;
  }
  return n;
}
```

9.12

```
char c(int k)
{ assert(k >= 0 && k <= 15);
  if (k < 10) return char(k + '0');
  return char(k - 10 + 'a');
}
string hexadecimal(int n)
{ if (n == 0) return string(1, '0');
  string s;
  while (n > 0)
  { s = string(1, c(n%16)) + s;
    n /= 16;
  }
  return s;
}
```

9.13

```
int v(string s, int i)
{ char c = s[i];
  assert(c >= '0' && c <= '9' || c >= 'a' && c <= 'f');
  if (c >= '0' && c <= '9') return int(c - '0');
  else return int(c - 'a' + 10);
}
int decimal(string s)
{ int len = s.length();
  assert(len > 0);
  int n=0;
  for (int i=0; i<len; i++)
    n = 16*n + v(s,i);
```

```
  return n;
}
```

9.14

```
void reverse(string& s)
{ string temp = s;
  int len = s.length();
  for (int i=0; i<len; i++)
    s[i] = temp[len-i-1];
}
```

9.15

```
bool is_palindrome(string s)
{ int len = s.length();
  for (int i=0; i<len/2; i++)
    if (s[i] != s[len-i-1]) return false;
  return true;
}
```

9.16

```
bool more(ifstream& fin, string& s)
{ if (getline(fin, s)) return true;
  else return false;
}

bool copy(ofstream& fout, ifstream& fin, string& s)
{ fout << s << endl;
  cout << s << endl;
  return more(fin,s);
}
int main()
{ ifstream fin1("Democrats.dat");
  ifstream fin2("Republicans.dat");
  ofstream fout("Presidents.dat");
  string s1, s2;
  bool more1 = more(fin1, s1);
  bool more2 = more(fin2, s2);
  while (more1 && more2)
    if (s1 < s2) more1 = copy(fout, fin1, s1);
    else more2 = copy(fout, fin2, s2);
  while (more1)
    more1 = copy(fout, fin1, s1);
  while (more2)
    more2 = copy(fout, fin2, s2);
  fout << endl;
}
```

Chapter 10

Classes

10.1 INTRODUCTION

A *class* is like an array: it is a derived type whose elements have other types. But unlike an array, the elements of a class may have different types. Furthermore, some elements of a class may be functions, including operators.

Although any region of storage may generally be regarded as an "object", the word is usually used to describe variables whose type is a class. Thus "object-oriented programming" involves programs that use classes. We think of an object as a self-contained entity that stores its own data and owns its own functions. The functionality of an object gives it life in the sense that it "knows" how to do things on its own.

There is much more to object-oriented programming than simply including classes in your programs. However, that is the first step. An adequate treatment of the discipline lies far beyond an introductory outline such as this.

10.2 CLASS DECLARATIONS

Here is a declaration for a class whose objects represent rational numbers (*i.e.*, fractions):

```
class Ratio
{ public:
    void assign(int, int);
    double convert();
    void invert();
    void print();
  private:
    int num, den;
};
```

The declaration begins with the keyword `class` followed by the name of the class and ends with the required semicolon. The name of this class is `Ratio`.

The functions `assign()`, `convert()`, `invert()`, and `print()` are called *member functions* because they are members of the class. Similarly, the variables `num` and `den` are called *member data*. Member functions are also called *methods* and *services*.

In this class, all the member functions are designated as `public`, and all the member data are designated as `private`. The difference is that `public` members are accessible from outside the class, while `private` members are accessible only from within the class. Preventing access from outside the class is called "information hiding." It allows the programmer to compartmentalize the software which makes it easier to understand, to debug, and to maintain.

The following example shows how this class could be implemented and used.

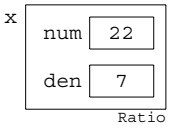
EXAMPLE 10.1 Implementing the `Ratio` Class

```
class Ratio
{ public:
    void assign(int, int);
    double convert();
    void invert();
    void print();
  private:
    int num, den;
};

int main()
{ Ratio x;
  x.assign(22,7);
  cout << "x = ";
  x.print();
  cout << " = " << x.convert() << endl;
  x.invert();
  cout << "1/x = ";  x.print();
  cout << endl;
}

void Ratio::assign(int numerator, int denominator)
{ num = numerator;
  den = denominator;
}

double Ratio::convert()
{ return double(num)/den;
}

void Ratio::invert()
{ int temp = num;
  num = den;
  den = temp;
}

void Ratio::print()
{ cout << num << '/' << den;
}
x = 22/7 = 3.14286
1/x = 7/22
```

Here `x` is declared to be an object of the `Ratio` class. Consequently, it has its own internal data members `num` and `den`, and it has the ability to call the four class member functions `assign()`, `convert()`, `invert()`, and `print()`. Note that a member function like `invert()` is called by prefixing its name with the name of its owner: `x.invert()`. Indeed, a member function can only be called this way. We say that the object `x` "owns" the call.

An object like `x` is declared just like an ordinary variable. Its type is `Ratio`. We can think of this type as a "user-defined type." C++ allows us to extend the definition of the programming language by

adding the new `Ratio` type to the collection of predefined numeric types `int`, `float`, *etc*. We can envision the object `x` like this:



Notice the use of the specifier `Ratio::` as a prefix to each function name. This is necessary for each member function definition that is given outside of its class definition. The *scope resolution operator* `::` is used to tie the function definition to the `Ratio` class. Without this specifier, the compiler would not know that the function being defined is a member function of the `Ratio` class. This can be avoided by including the function definitions within declaration, as shown below in Example 10.2.

When an object like the `Ratio` object `x` in Example 10.1 is declared, we say that the class has been *instantiated*, and we call the object an *instance* of the class. And just as we may have many variables of the same type, we may also have may instances of the same class:

```
Ratio x, y, z;
```

EXAMPLE 10.2 A Self-Contained Implementation of the `Ratio` Class

Here's the same `Ratio` class with the definitions of its member functions included within the class declaration:

```
class Ratio
{ public:
    void assign(int n, int d) { num = n; den = d; }
    double convert() { return double(num)/den; }
    void invert() { int temp = num; num = den; den = temp; }
    void print() { cout << num << '/' << den; }
  private:
    int num, den;
};
```

In most cases, the preferred style is to define the member functions outside of the class declaration, using the scope resolution operator as shown in Example 10.1. That format physically separates the function declarations from their definitions, consistent with the general principle of information hiding. In fact, the definitions are often put in a separate file and compiled separately. The point is that application programs that use the class need only know `what` the objects can do; they do not need to know how the objects do it. The function declarations tell what they do; the function definitions tell how they do it. This, of course, is how the predefined types (`int`, `double`, *etc*.) work: we know what the result should be when we divide one float by another, but we don't really know how the division is done (*i.e.*, what algorithm is implemented). More importantly, we don't want to know. Having to think about those details would distract us from the task at hand. This point of view is often called *information hiding* and is an important principle in object-oriented programming.

When the member function definitions are separated from the declarations, as in Example 10.1, the declaration section is called the *class interface*, and the section containing the member function definitions is called the *implementation*. The interface is the part of the class that the programmer needs to see in order to use the class. The implementation would normally be

concealed in a separate file, thereby "hiding" that information that the user (*i.e.*, the programmer) does not need to know about. These class implementations are typically done by implementors who work independently of the programmers who will use the classes that they have implemented.

10.3 CONSTRUCTORS

The `Ratio` class defined in Example 10.1 uses the `assign()` function to initialize its objects. It would be more natural to have this initialization occur when the objects are declared. That's how ordinary (predefined) types work:

```
int n = 22;
char* s = "Hello";
```

C++ allows this simpler style of initialization to be done for class objects using constructor functions.

A *constructor* is a member function that is invoked automatically when an object is declared. A constructor function must have the same name as the class itself, and it is declared without return type. The following example illustrates how we can replace the `assign()` function with a constructor.

EXAMPLE 10.3 A Constructor Function for the `Ratio` Class

```
class Ratio
{ public:
    Ratio(int n, int d) { num = n; den = d; }
    void print() { cout << num << '/' << den; }
  private:
    int num, den;
};

int main()
{ Ratio x(-1,3), y(22,7);
  cout << "x = ";
  x.print();
  cout << " and y = ";
  y.print();
}
```

```
    x = -1/3 and y = 22/7
```

The constructor function has the same effect as the `assign()` function had in Example 10.1: it initializes the object by assigning the specified values to its member data. When the declaration of `x` executes, the constructor is called automatically and the integers -1 and 3 are passed to its parameters `n` and `d`. The function then assigns these values to `x`'s `num` and `den` data members. So the declarations

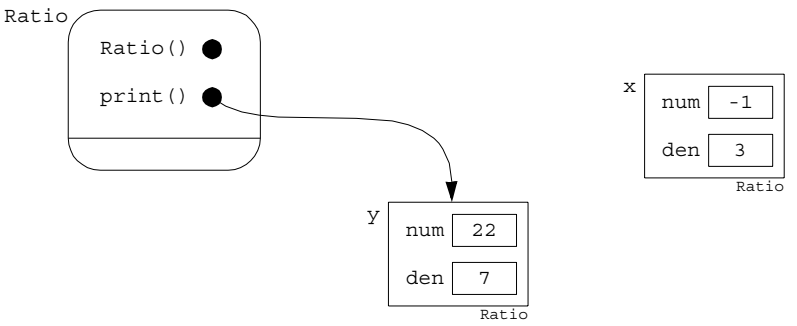
```
Ratio x(-1,3), y(22,7);
```

are equivalent to the three lines

```
Ratio x, y;
x.assign(-1,3);
y.assign(22,7);
```

A class's constructor "constructs" the class objects by allocating and initializing storage for the objects and by performing any other tasks that are programmed into the function. It literally creates a live object from a pile of unused bits.

We can visualize the relationships between the `Ratio` class itself and its instantiated objects like this:



The class itself is represented by a rounded box containing its member functions. Each function maintains a pointer, named "`this`", which points to the object that is calling it. The snapshot here represents the status during the execution of the last line of the program, when the object `y` is calling the `print()` function: `y.print()`. At that moment, the "`this`" pointer for the constructor is **`NULL`** because it is not being called.

A class may have several constructors. Like any other overloaded function, these are distinguished by their distinct parameter lists.

EXAMPLE 10.4 Adding More Constructors to the `Ratio` Class

```
class Ratio
{ public:
    Ratio() { num = 0; den = 1; }
    Ratio(int n) { num = n; den = 1; }
    Ratio(int n, int d) { num = n; den = d; }
    void print() { cout << num << '/' << den; }
  private:
    int num, den;
};

int main()
{ Ratio x, y(4), z(22,7);
  cout << "x = ";
  x.print();
  cout << "\ny = ";
  y.print();
  cout << "\nz = ";
  z.print();
}
```

```
x = 0/1
y = 4/1
z = 22/7
```

This version of the `Ratio` class has three constructors. The first has no parameters and initializes the declared object with the default values 0 and 1. The second constructor has one integer parameter and

initializes the object to be the fractional equivalent to that integer. The third constructor is the same as in Example 10.2.

Among the various constructors that a class may have, the simplest is the one with no parameters. It is called the *default constructor.* If this constructor is not explicitly declared in the class definition, then the system will automatically create it for the class. That is what happens in Example 10.1.

10.4 CONSTRUCTOR INITIALIZATION LISTS

Most constructors do nothing more than initialize the object's member data. Consequently, C++ provides a special syntactical device for constructors that simplifies this code. The device is an *initialization list.*

Here is the third constructor in Example 10.2, rewritten using an initialization list:

```
Ratio(int n, int d) : num(n), den(d) { }
```

The assignment statements in the function's body that assigned `n` to `num` and `d` to `den` are removed. Their action is handled by the initialization list shown in boldface. Note that the list begins with a colon and precedes the function body which is now empty.

Here is the `Ratio` class with its three constructors rewritten using initializer lists.

EXAMPLE 10.5 Using Initializer Lists in the `Ratio` Class

```
class Ratio
{ public:
    Ratio() : num(0), den(1) { }
    Ratio(int n) : num(n), den(1) { }
    Ratio(int n, int d) : num(n), den(d) { }
  private:
    int num, den;
};
```

Of course, these three separate constructors are not necessary. They can be combined into a single constructor, using default parameter values, as illustrated by the next example.

EXAMPLE 10.6 Using Default Parameter Values in the `Ratio` Class Constructor

```
class Ratio
{ public:
    Ratio(int n=0, int d=1) : num(n), den(d) { }
  private:
    int num, den;
};

int main()
{ Ratio x, y(4), z(22,7);
}
```

Here, `x` will represent 0/1, `y` will represent 4/1, and `z` will represent 22/7.

Recall that the default values are used when actual parameters are not passed. So in the declaration of the `Ratio` object `x` where no values are passed, the formal parameter `n` is given the default value 0

which is then assigned to `x.num`, and the formal parameter `d` is given the default value 1 which is then assigned to `x.den`. In the declaration of the object `y` where only the value 4 is passed, the formal parameter `n` is given that value 4 which is then assigned to `y.num`, and the formal parameter `d` is given the default value 1 which is then assigned to `y.den`. No default values are used in the declaration of `z`.

10.5 ACCESS FUNCTIONS

Although a class's member data are usually declared to be **`private`** to limit access to them, it is also common to include **`public`** member functions that provide read-only access to the data. Such functions are called *access functions*. (In Java, they are also called *getty methods*, because they usually use the word "get" in their names. This is in contrast to *setty methods* which are used to change the values of data members and use the word "set" in their name. Getty methods are read-only; setty methods are read-write.)

EXAMPLE 10.7 Access Functions in the `Ratio` Class

```
class Ratio
{ public:
    Ratio(int n=0, int d=1) : num(n), den(d) { }
    int numerator() const { return num; }
    int denominator() const { return den; }
  private:
    int num, den;
};

int main()
{ Ratio x(22,7);
  cout << x.numerator() << '/' << x.denominator() << endl;
}
```

The functions `numerator()` and `denominator()` return the values of the **`private`** member data.

Note the use of the `const` keyword in the declarations of the two access functions. This allows the functions to be applied to constant objects. (See Section 10.9.)

10.6 PRIVATE MEMBER FUNCTIONS

Class member data are usually declared to be **`private`** and member functions are usually declared to be **`public`**. But this dichotomy is not required. In some cases, it is useful to declare one or more member functions to be **`private`**. As such, these functions can only be used within the class itself; *i.e.*, they are local *utility functions*.

EXAMPLE 10.8 Using `private` Member Functions

```
class Ratio
{ public:
    Ratio(int n=0, int d=1) : num(n), den(d) { reduce(); }
    void print() const { cout << num << '/' << den << endl; }
  private:
```

```
    int num, den;
    void reduce();
};

int gcd(int,int);
void Ratio::reduce()
{ // enforce invariant(den > 0):
  if (num == 0 || den == 0)
  { num = 0;
    den = 1;
    return;
  }
  if (den < 0)
  { den *= -1;
    num *= -1;
  }
  // enforce invariant(gcd(num,den) == 1):
  if (den == 1) return;    // it's already reduced
  int sgn = (num<0?-1:1);  // no negatives to gcd()
  int g = gcd(sgn*num,den);
  num /= g;
  den /= g;
}
int gcd(int m, int n)
{ // returns the greatest common divisor of m and n:
  if (m<n) swap(m,n);
  while (n>0)
  { int r=m%n;
    m = n;
    n = r;
  }
  return m;
}

int main()
{ Ratio x(100,-360);
  x.print();
}
-5/18
```

This version includes the **`private`** function `reduce()` that uses the `gcd()` function (see Problem 5.18 on page 113) to reduce the fraction `num/den` to lowest terms. Thus the fraction 100/–360 is stored as –5/18.

Instead of having a separate `reduce()` function, we could have done the actual reduction within the constructor. But there are two good reasons for doing it this way. Combining the construction with the reduction would violate the software principle that separate tasks should be handled by separate functions. Moreover, the `reduce()` function will be needed later to reduce the results of arithmetic operations performed on `Ratio` objects.

Note that the keywords **public** and **private** are called access specifiers; they specify whether the members are accessible outside the class definition. The keyword **protected** is the third access specifier. It is described in Chapter 13.

10.7 THE COPY CONSTRUCTOR

Every class has at least two constructors. These are identified by their unique declarations:

```
X();              // default constructor
X(const X&);      // copy constructor
```

where X is the class identifier. For example, these two special constructors for a Widget class would be declared:

```
Widget();                // default constructor
Widget(const Widget&);   // copy constructor
```

The first of these two special constructors is called the *default constructor*; it is called automatically whenever an object is declared in the simplest form, like this:

```
Widget x;
```

The second of these two special constructors is called the *copy constructor*; it is called automatically whenever an object is copied (*i.e.*, duplicated), like this:

```
Widget y(x);
```

If either of these two constructors is not defined explicitly, then it is automatically defined implicitly by the system.

Note that the copy constructor takes one parameter: the object that it is going to copy. That object is passed by constant reference because it should not be changed.

When the copy constructor is called, it copies the complete state of an existing object into a new object of the same class. If the class definition does not explicitly include a copy constructor (as all the previous examples have not), then the system automatically creates one by default. The ability to write your own copy constructor gives you more control over your software.

EXAMPLE 10.9 Adding a Copy Constructor to the Ratio Class

```
class Ratio
{ public:
    Ratio(int n=0, int d=1) : num(n), den(d) { reduce(); }
    Ratio(const Ratio& r) : num(r.num), den(r.den) { }
    void print() { cout << num << '/' << den; }
  private:
    int num, den;
    void reduce();
};
int main()
{ Ratio x(100,360);
  Ratio y(x);
  cout << "x = ";
  x.print();
  cout << ", y = ";
  y.print();
}
```

```
x = 5/18, y = 5/18
```

The copy constructor copies the `num` and `den` fields of the parameter `r` into the object being constructed. When `y` is declared, it calls the copy constructor which copies `x` into `y`.

Note the required syntax for the copy constructor: it must have one parameter, which has the same class as that being declared, and it must be passed by constant reference: `const X&`.

The copy constructor is called automatically whenever

- an object is copied by means of a declaration initialization;
- an object is passed by value to a function;
- an object is returned by value from a function.

EXAMPLE 10.10 Tracing Calls to the Copy Constructor

```
class Ratio
{ public:
    Ratio(int n=0, int d=1) : num(n), den(d) { reduce(); }
    Ratio(const Ratio& r) : num(r.num), den(r.den)
    { cout << "COPY CONSTRUCTOR CALLED\n"; }
  private:
    int num, den;
    void reduce();
};

Ratio f(Ratio r)  // calls the copy constructor, copying ? to r
{ Ratio s = r;    // calls the copy constructor, copying r to s
  return s;       // calls the copy constructor, copying s to ?
}

int main()
{ Ratio x(22,7);
  Ratio y(x);  // calls the copy constructor, copying x to y
  f(y);
}
COPY CONSTRUCTOR CALLED
COPY CONSTRUCTOR CALLED
COPY CONSTRUCTOR CALLED
COPY CONSTRUCTOR CALLED
```

In this example, the copy constructor is called four times. It is called when `y` is declared, copying `x` to `y`; it is called when `y` is passed by value to the function `f`, copying `y` to `r`; it is called when `s` is declared, copying `r` to `s`; and it is called when the function `f` returns by value, even though nothing is copied there. Note that the initialization of `s` looks like an assignment. But as part of a declaration it calls the copy constructor just as the declaration of `y` does.

If you do not include a copy constructor in your class definition, then the compiler generates one automatically. This “default” copy constructor will simply copy objects bit-by-bit. In many cases, this is exactly what you would want. So in these cases, there is no need for an explicitly defined copy constructor.

However, in some important cases, a bit-by-bit copy will not be adequate. The **`string`** class, described in Chapter 9, is a prime example. In objects of that class, the relevant data member

holds only a pointer to the actual string, so a bit-by-bit copy would only duplicate the pointer, not the string itself. In cases like this, it is essential that you define your own copy constructor.

10.8 THE CLASS DESTRUCTOR

When an object is created, a constructor is called automatically to manage its birth. Similarly, when an object comes to the end of its life, another special member function is called automatically to manage its death. This function is called a *destructor*.

Each class has exactly one destructor. If it is not defined explicitly in the class definition, then like the default constructor, the copy constructor, and the assignment operator, the destructor is created automatically.

EXAMPLE 10.11 Including a Destructor in the `Ratio` Class

```
class Ratio
{ public:
    Ratio() { cout << "OBJECT IS BORN.\n"; }
    ~Ratio()  { cout << "OBJECT DIES.\n"; }
  private:
    int num, den;
};

int main()
{ { Ratio x;                        // beginning of scope for x
    cout << "Now x is alive.\n";
  }                                      // end of scope for x
  cout << "Now between blocks.\n";
  { Ratio y;
    cout << "Now y is alive.\n";
  }
}
```

```
OBJECT IS BORN.
Now x is alive.
OBJECT DIES.
Now between blocks.
OBJECT IS BORN.
Now y is alive.
OBJECT DIES.
```

The output here shows when the constructor and the destructor are called.

The class destructor is called for an object when it reaches the end of its scope. For a local object, this will be at the end of the block within which it is declared. For a `static` object, it will be at the end of the `main()` function.

Although the system will provide them automatically, it is considered good programming practice always to define the copy constructor, the assignment operator, and the destructor within each class definition.

10.9 CONSTANT OBJECTS

It is good programming practice to make an object constant if it should not be changed. This is done with the `const` keyword:

```
const char BLANK = ' ';
const int MAX_INT = 2147483647;
const double PI = 3.141592653589793;
void init(float a[], const int SIZE);
```

Like variables and function parameters, objects may also be declared to be constant:

```
const Ratio PI(22,7);
```

However, when this is done, the C++ compiler restricts access to the object's member functions. For example, with the `Ratio` class defined previously, the `print()` function could not be called for this object:

```
PI.print();    //  error: call not allowed
```

In fact, unless we modify our class definition, the only member functions that could be called for `const` objects would be the constructors and the destructor. To overcome this restriction, we must declare as constant those member functions that we want to be able to use with `const` objects.

A function is declared constant by inserting the `const` keyword between its parameter list and its body:

```
void print() const { cout << num << '/' << den << endl; }
```

This modification of the function definition will allow it to be called for constant objects:

```
const Ratio PI(22,7);
PI.print();    //  o.k. now
```

10.10 STRUCTURES

The C++ **`class`** is a generalization of the C **`struct`** (for "structure") which is a class with only **`public`** members and no functions. One normally thinks of a class as a structure that is given life by means of its member functions and which enjoys information hiding by means of **`private`** data members.

To remain compatible with the older C language, C++ retains the **`struct`** keyword which allows **`struct`**s to be defined. However, a C++ struct is essentially the same as a C++ **`class`**. The only significant difference between a C++ **`struct`** and a C++ **`class`** is with the default access specifier assigned to members. Although not recommended, C++ **`classes`** can be defined without explicitly specifying its member access specifier. For example,

```
class Ratio
{ int num, den;
};
```

is a valid definition of a `Ratio` class. Since the access specifier for its data members `num` and `den` is not specified, it is set by default to be **`private`**. If we make it a **`struct`** instead of a **`class`**, like this:

```
struct Ratio
{ int num, den;
};
```

then the data members are set by default to be **`public`**. But this could be corrected simply by specifying the access specifier explicitly:

```
struct Ratio
{ private:
    int num, den;
};
```

So the difference between a **`class`** and a C++ **`struct`** is really just cosmetic.

10.11 POINTERS TO OBJECTS

In many applications, it is advantageous to use pointers to objects (and `structs`). Here is a simple example:

EXAMPLE 10.12 Using Pointers to Objects

```
class X
{ public:
    int data;
};
int main()
{ X* p = new X;
  (*p).data = 22;          // equivalent to: p->data = 22;
  cout << "(*p).data = " << (*p).data << " = " << p->data << endl;
  p->data = 44;
  cout << "  p->data = " << (*p).data << " = " << p->data << endl;
}
(*p).data = 22 = 22
  p->data = 44 = 44
```

Since `p` is a pointer to an `X` object, `*p` is an `X` object, and `(*p).data` accesses its **`public`** member `data`. Note that parentheses are required in the expression `(*p).data` because the direct member selection operator "**`.`**" has higher precedence than the dereferencing operator "**`*`**". (See Appendix C.)

The two notations

```
(*p).data
p->data
```

have the same meaning. When working with pointers, the "arrow" symbol "`->`" is preferred because it is simpler and it suggests "the thing to which `p` points."

Here is a more important example:

EXAMPLE 10.13 A `Node` Class for Linked Lists

```
class Node
{ public:
    Node(int d, Node* q=0) : data(d), next(q) { }
    int data;
    Node* next;
};
```

This defines a `Node` class each of whose objects contain an `int` data member and a `next` pointer.

```
int main()
{ int n;
  Node* p;
```

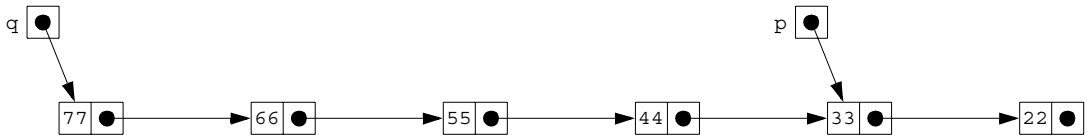
```
  Node* q=0;
  while (cin >> n)
  { p = new Node(n, q);
    q = p;
  }
  for ( ; p; p = p->next )
    cout << p->data << " -> ";
  cout << "*\n";
}
22 33 44 55 66 77 ^D
77 -> 66 -> 55 -> 44 -> 33 -> 22 -> *
```

First note that the definition of the `Node` class includes two references to the class itself. This is allowed because each reference is actually a pointer to the class. Also note that the constructor initializes both data members.

The program allows the user to create a linked list in reverse. Then it traverses the list, printing each `data` value.

The `while` loop continues reads `int`s into `n` until the user enters the end-of-file character (Ctrl+D). Within the loop, it gets a new node, inserts the `int` into its data member, and connects the new node to the previous node (pointed to by `q`). Finally, the `for` loop traverses the list, beginning with the node pointed to by `p` (which is the last node constructed) and continuing until `p` is `NUL`.

The list constructed in this example can be visualized like this:



10.12 STATIC DATA MEMBERS

Sometimes a single value for a data member applies to all members of the class. In this case, it would be inefficient to store the same value in every object of the class. That can be avoided by declaring the data member to be `static`. This is done by including the `static` keyword at the beginning of the variable's declaration. It also requires that the variable be defined globally. So the syntax looks like this:

```
class X
{ public:
  static int n;  // declaration of n as a static data member
};
int X::n = 0;    // definition of n
```

Static variables are automatically initialized to 0, so the explicit initialization in the definition is unnecessary unless you want it to have a non-zero initial value.

EXAMPLE 10.14 A `static` Data Member

The `Widget` class maintains a **`static`** data member `count` which keeps track of the number of `Widget` objects in existence globally. Each time a widget is created (by the constructor) the counter is incremented, and each time a widget is destroyed (by the destructor) the counter is decremented.

```
class Widget
{ public:
    Widget() { ++count; }
    ~Widget() { --count; }
    static int count;
};
int Widget::count = 0;

int main()
{ Widget w, x;
  cout << "Now there are " << w.count << " widgets.\n";
  { Widget w, x, y, z;
    cout << "Now there are " << w.count << " widgets.\n";
  }
  cout << "Now there are " << w.count << " widgets.\n";
  Widget y;
  cout << "Now there are " << w.count << " widgets.\n";
}
```

```
Now there are 2 widgets.
Now there are 6 widgets.
Now there are 2 widgets.
Now there are 3 widgets.
```

Notice how four widgets are created inside the inner block, and then they are destroyed when program control leaves that block, reducing the global number of widgets from 6 to 2.

A static data member is like an ordinary global variable: only one copy of the variable exists no matter how many instances of the class exist. The main difference is that it is a data member of the class, and so may be `private`.

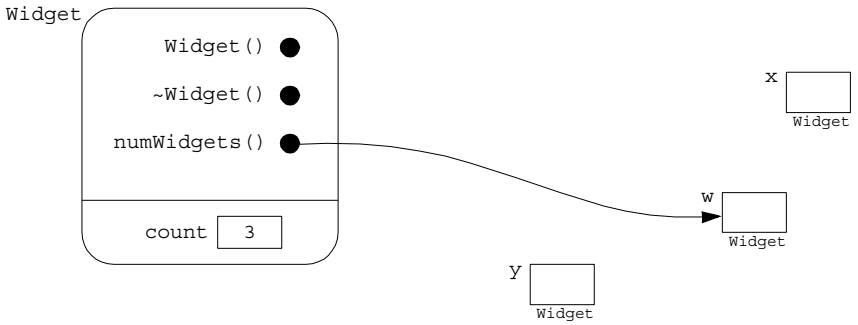
EXAMPLE 10.15 A `static` Data Member that is `private`

```
class Widget
{ public:
    Widget() { ++count; }
    ~Widget() { --count; }
    int numWidgets() { return count; }
  private:
    static int count;
};
int Widget::count = 0;

int main()
{ Widget w, x;
  cout << "Now there are " << w.numWidgets() << " widgets.\n";
  { Widget w, x, y, z;
```

```
    cout << "Now there are " << w.numWidgets() << " widgets.\n";
  }
  cout << "Now there are " << w.numWidgets() << " widgets.\n";
  Widget y;
  cout << "Now there are " << w.numWidgets() << " widgets.\n";
}
```

This works the same way as Example 10.2. But now that the `static` variable `count` is `private`, we need the access function `numWidgets()` to read `count` in `main()`.

The relationships among the class, its members, and its objects can be visualized like this:



The rounded box represents the class itself which contains the three member functions and the data member `count`. The `public` members are above the line and the `private` member(s) are below it. Each member function maintains a pointer (named "`this`") which points to the object that owns the current function call. This snapshot shows the status during the execution of the last line in the program: three widgets (`w`, `x`, and `y`) exist, and `w` is calling the `numWidgets()` function which returns the value of the private data member `count`. Note that this data member resides within the class itself; the class objects have no data.

10.13 `static` FUNCTION MEMBERS

Like any ordinary member function, the `numWidgets()` function in Example 10.2 requires that it be owned by some instance of the class. But since it returns the value of the `static` data member `count` which is independent of the individual objects themselves, it doesn't matter which object calls it. We had `w` call it each time, but we could just as well have had `x` or `y` or `z` call it when they exist. Moreover, we couldn't call it at all until after some object had been created. This is rather arbitrary. Since the action of the function is independent of the actual function objects, it would be better to make the calls independent of them too. This can be done simply by declaring the function to be `static`.

EXAMPLE 10.16 A `static` Function Member

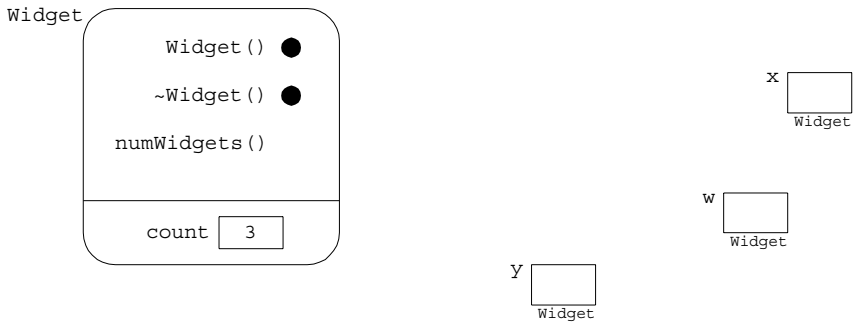
The `Widget` class maintains a `static` data member `count` which keeps track of the number of `Widget` objects in existence globally. Each time a widget is created (by the constructor) the counter is incremented, and each time a widget is destroyed (by the destructor) the counter is decremented.

```
class Widget
{ public:
    Widget() { ++count; }
```

```
    ~Widget() { --count; }
    static int num() { return count; }
  private:
    static int count;
};

int Widget::count = 0;

int main()
{ cout << "Now there are " << Widget::num() << " widgets.\n";
  Widget w, x;
  cout << "Now there are " << Widget::num() << " widgets.\n";
  { Widget w, x, y, z;
    cout << "Now there are " << Widget::num() << " widgets.\n";
  }
  cout << "Now there are " << Widget::num() << " widgets.\n";
  Widget y;
  cout << "Now there are " << Widget::num() << " widgets.\n";
}
```

Declaring the `numWidgets()` function to be `static` renders it independent of the class instances. So now it is invoked simply as a member of the `Widget` class using the scope resolution operator "`::`". This allows the function to be called before any objects have been instantiated.

The previous figure showing relationships among the class and its instances should now looks like this:



The difference is that now the member function `num()` has no "`this`" pointer. As a `static` member function, it is associated with the class itself, not with its instances.

Static member functions can access only `static` data from their own class.

Review Questions

10.1 Explain the difference between a `public` member and a `private` member of a class.
10.2 Explain the difference between the interface and the implementation of a class.
10.3 Explain the difference between a class member function and an application function.
10.4 Explain the difference between a constructor and a destructor.
10.5 Explain the difference between the default constructor and other constructors.
10.6 Explain the difference between the copy constructor and the assignment operator.
10.7 Explain the difference between an access function and a utility function.
10.8 Explain the difference between a `class` and a `struct` in C++.

10.9 What name must a constructor have?

10.10 What name must a destructor have?

10.11 How many constructors can a class have?

10.12 How many destructors can a class have?

10.13 How and why is the scope resolution operator `::` used in class definitions?

10.14 Which member functions are created automatically by the compiler if they are not included (by the programmer) in the class definition?

10.15 How many times is the copy constructor called in the following code:

```
Widget f(Widget u)
{ Widget v(u);
  Widget w = v;
  return w;
}

main()
{ Widget x;
  Widget y = f(f(x));
}
```

10.16 Why are the parentheses needed in the expression `(*p).data`?

Problems

10.1 Implement a `Point` class for three-dimensional points (*x*,*y*,*z*). Include a default constructor, a copy constructor, a `negate()` function to transform the point into its negative, a `norm()` function to return the point's distance from the origin (0,0,0), and a `print()` function.

10.2 Implement a `Stack` class for stacks of `int`s. Include a default constructor, a destructor, and the usual stack operations: `push()`, `pop()`, `isEmpty()`, and `isFull()`. Use an array implementation.

10.3 Implement a `Time` class. Each object of this class will represent a specific time of day, storing the hours, minutes, and seconds as integers. Include a constructor, access functions, a function `advance(int h, int m, int s)` to advance the current time of an existing object, a function `reset(int h, int m, int s)` to reset the current time of an existing object, and a `print()` function.

10.4 Implement a `Random` class for generating pseudo-random numbers.

10.5 Implement a `Person` class. Each object of this class will represent a human being. Data members should include the person's name, year of birth, and year of death. Include a default constructor, a destructor, access functions, and a print function.

10.6 Implement a `String` class. Each object of this class will represent a character string. Data members are the length of the string and the actual character string. In addition to constructors, destructor, access functions, and a print function, include a "subscript" function.

10.7 Implement a `Matrix` class for 2-by-2 matrices:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

Include a default constructor, a copy constructor, an `inverse()` function that returns the inverse of the matrix, a `det()` function that returns the determinant of the matrix, a Boolean function `isSingular()` that returns 1 or 0 according to whether the determinant is zero, and a `print()` function.

10.8 Implement a `Point` class for two-dimensional points (*x*, *y*). Include a default constructor, a copy constructor, a `negate()` function to transform the point into its negative, a `norm()` function to return the point's distance from the origin (0,0), and a `print()` function.

10.9 Implement a `Circle` class. Each object of this class will represent a circle, storing its radius and the *x* and *y* coordinates of its center as `float`s. Include a default constructor, access functions, an `area()` function, and a `circumference()` function.

Answers to Review Questions

10.1 A `public` member is accessible from outside the class; a `private` member is not.

10.2 The class interface consists of the member data and the member function prototypes (*i.e.* just the function declarations). The class implementation contains the definitions of the member functions.

10.3 A class member function is part of the class, so it has access to the class's `private` parts. An application function is declared outside the class, and so it does not have access to the class's `private` parts.

10.4 A constructor is a class member function that executes automatically whenever an object of that class is instantiated (*i.e.*, constructed). A destructor is a class member function that executes automatically whenever the scope of that object terminates (*i.e.*, is destructed).

10.5 The default constructor is the unique constructor that has no parameters (or the one whose parameters all have default values).

10.6 A class's copy constructor executes whenever an object of that class is copied by any mechanism except direct assignment. This includes initialization, passing a parameter by value, and returning by value.

10.7 An access function is a `public` class member function that returns the value of one of the class's data members. A utility function is a `private` class member function that is used only within the class to perform "technical" tasks.

10.8 A `class` and a `struct` in C++ are essentially the same. The only significant difference is that the default access level for a class is `private`, while that for a struct is `public`.

10.9 Every class constructor must have the same name as the class itself.

10.10 Every class destructor must have the same name as the class itself, prefixed with a tilde (~).

10.11 There is no limit to the number of constructors that a class may have. But since multiple constructors are function overloads, they all must be distinguishable by their parameter lists.

10.12 A class can have only one destructor.

10.13 The scope resolution operator `::` is used in general "to resolve external references." It is used in a class definition whenever the definition of a member function is given outside the scope of the class definition.

10.14 There are four class member functions that are created automatically by the compiler if they are not included (by the programmer) in the class definition: the default constructor, the copy constructor, the destructor, and the overloaded assignment operator.

10.15 The copy constructor is called 7 times in this code. Each call to the function `f` requires 3 calls to the copy constructor: when the parameter is passed by value to `u`, when `v` is initialized, and when `w` is returned by value. The seventh call is for the initialization `y`.

10.16 The parentheses are needed in the expression `(*p).data` because the direct member selection operator "`.`" has higher precedence than the dereferencing operator "`*`". (See Appendix C.)

Solutions to Problems

10.1 This implementation of a `Point` class uses the common device of ending the name of each data member with an underscore (_). This has the advantage of making it easy to match up the names of constructor parameters (`x`, `y`, and `z`) with their corresponding data members (`x_`, `y_`, and `z_`) without conflict.

```
#include <cmath>
#include <iostream>
using namespace std;
class Point
{ public:
    Point(float x=0, float y=0, float z=0): x_(x), y_(y), z_(z) {}
    Point(const Point& p) : x_(p.x_), y_(p.y_), z_(p.z_)  { }
    void negate() { x_ *= -1; y_ *= -1; z_ *= -1; }
    double norm() { return sqrt(x_*x_ + y_*y_ + z_*z_); }
    void print()
      { cout << '(' << x_ << "," << y_ << "," << z_ << ")"; }
  private:
    float x_, y_, z_;
};
```

10.2 In this implementation of a `Stack` class, `top` is always the index of the top element on the stack. The data member `size` is the size of the array that holds the stack items. So the stack is full when it contains that number of items. The constructor sets `size` to 10 as the default.

```
class Stack
{ public:
    Stack(int s=10) : size(s), top(-1) { a = new int[size]; }
    ~Stack() { delete [] a; }
    void push(const int& item) { a[++top] = item; }
    int pop() { return a[top--]; }
    bool isEmpty() const { return top == -1; }
    bool isFull() const { return top == (size-1); }
  private:
    int size;  // size of array
    int top;   // top of stack
    int* a;    // array to hold stack items
};
```

10.3

```
class Time
{ public:
    Time(int h=0, int m=0, int s=0)
      : hr(h), min(m), sec(s) { normalize(); }
    int hours() { return hr; }
    int minutes() { return min; }
    int seconds() { return sec; }
    void advance(int =0, int =0, int =1);
    void reset(int =0, int =0, int =0);
    void print() { cout << hr << ":" << min << ":" << sec; }
  private:
    int hr, min, sec;
    void normalize();
};
void Time::normalize()
```

```
{ min += sec/60;
  hr += min/60;
  hr = hr % 24;
  min = min % 60;
  sec = sec % 60;
}
void Time::advance(int h, int m, int s)
{ hr += h;
  min += m;
  sec += s;
  normalize();
}
void Time::reset(int h, int m, int s)
{ hr = h;
  min = m;
  sec = s;
  normalize();
}
```

10.4 This implementation of a Random class uses a utility function `normalize()`, which normalizes the `Time` object so that its three data members are in the correct range: $0 \leq$ `sec` < 60, $0 \leq$ `min` < 60, and $0 \leq$ `hr` < 24. It also uses the utility function `randomize()`, which implements the *Linear Congruential Algorithm* introduced by D. H. Lehmer in 1949. The utility function `_next()` updates the `_seed` by calling the `_randomize()` function a random number of times.

```
#include <climits>   // defines INT_MAX and ULONG_MAX constant
#include <ctime>     // defines time() function
#include <iomanip>   // defines the setw() function
#include <iostream>  // defines the cout object
using namespace std;
class Random
{ public:
    Random(long seed=0) { _seed = ( seed?seed:time(NULL) ); }
    void seed(long seed=0) { _seed = ( seed?seed:time(NULL) ); }
    int integer() { return _next(); }
    int integer(int min, int max)
      { return min +_next()%(max-min+1);}
    double real()
      { return double(_next())/double(INT_MAX); }
  private:
    unsigned long _seed;
    void _randomize()
      { _seed = (314159265*_seed + 13579)%ULONG_MAX;}
    int _next()
    { int iterations = _seed % 3;
      for (int i=0; i <= iterations; i++) _randomize();
      return int(_seed/2);
    }
};
int main()
{ Random random;
  for (int i = 1; i <= 10; i++)
    cout << setw(16) << setiosflags(ios::right)
         << random.integer()
         << setw(6) << random.integer(1,6)
```

```
          << setw(12) << setiosflags(ios::fixed | ios::left)
          << random.real() << endl;
}
```

The test driver makes 10 calls to each of the three random number functions, generating 10 pseudo-random integers in the range 0 to 2,147,483,647, 10 pseudo-random integers in the range 1 to 6, and 10 pseudo-random real numbers in the range 0.0 to 1.0.

10.5

```
class Person
{ public:
    Person(const char* =0, int =0, int =0);
    ~Person() { delete [] name_; }
    char* name() { return name_; }
    int born() { return yob_; }
    int died() { return yod_; }
    void print();
  private:
    int len_;
    char* name_;
    int yob_, yod_;
};
Person::Person(const char* name, int yob, int yod)
  : len_(strlen(name)),
    name_(new char[len_+1]),
    yob_(yob),
    yod_(yod)
{ memcpy(name_, name, len_+1);
}
void Person::print()
{ cout << "\tName: " << name_ << endl;
  if (yob_) cout << "\tBorn: " << yob_ << endl;
  if (yod_) cout << "\tDied: " << yod_ << endl;
}
```

To keep the object self-contained, `name_` is stored as a separate string. To facilitate this separate storage, we save its length in the data member `len_` and use the `memcpy()` function (defined in `string.h`) to copy the string `name` into the string `name_`. Then the destructor uses the delete operator to de-allocate this storage.

10.6 This implementation of a String class includes three constructors: the default constructor with optional parameter `size`, a constructor that allows an object to be initialized with an ordinary C string, and the copy constructor. The second access function is named `convert()` because it actually converts from type `String` to `char*` type. The "subscript" function is named `character()` because it returns one character in the string—the one indexed by the parameter `i`.

```
class String
{ public:
    String(short =0);                    // default constructor
    String(const char*);                 // constructor
    String(const String&);               // copy constructor
    ~String() { delete [] data; }        // destructor
    int length() const { return len; }   // access function
    char* convert() { return data; }     // access function
    char character(short i) { char c = data[i]; return c; }
    void print() { cout << data; }
  private:
    short len;     // number of (non-null) characters in string
    char* data;    // the string
```

```
};
String::String(short size) : len(size)
{ data = new char[len+1];
  for (int i=0; i < len; i++) data[i] = ' ';
  data[len] = '\0';
}
String::String(const char* str) : len(strlen(str))
{ data = new char[len+1];
  memcpy(data, str, len+1);
}
String::String(const String& str) : len(str.len)
{ data = new char[len+1];
  memcpy(data, str.data, len+1);
}
```

10.7

```
class Matrix
{ public:
    Matrix(double a=0, double b=0, double c=0, double d=0 )
       : a_(a), b_(b), c_(c), d_(d) { }
    Matrix(const Matrix& m)
       : a_(m.a_), b_(m.b_), c_(m.c_), d_(m.d_)  { }
    double det() { return a_*d_ - b_*c_; }
    int isSingular() { return det() == 0; }
    Matrix inverse();
    void print();
  private:
    double a_, b_, c_, d_;
};
Matrix Matrix::inverse()
{ double k = 1/det();
  Matrix temp(k*d_,-k*b_,-k*c_,k*a_);
  return temp;
}
void Matrix::print()
{ cout << a_ << "  " << b_ << '\n' << c_ << "  " << d_ << "\n";
}
```

10.8

```
class Point
{ public:
    Point( ) : _x(0.0) , _y(0.0) {}
    Point( double x, double y ): _x( x ) , _y( y ) {}
    Point( const Point & P ) {_x = P._x ;  _y = P._y ; }
    double norm() const { return sqrt( _x*_x + _y*_y ); }
    void  print() const
      { cout << "( " << _x << " , " << _y << " )" ; }
    void negate() { _x = -1.0 * _x ;  _y = -1.0 * _y ; }
  private:
    double _x ;
    double _y ;
};
```

10.9

```
class Circle
{ public:
    Circle( ) : _x(0.0) , _y(0.0), _radius(1.0) {}
    Circle( float x, float y, float radius )
```

```
      : _x( x ) , _y( y ),  _radius( radius) {}
    Circle( const Circle & C )
      { _x = C._x ;  _y = C._y ; _radius = C._radius; }
    float diameter() const { return 2.0 * _radius ; }
    float area() const
      { return 3.141592654 * _radius * _radius ; }
    float circumference() const
      { return 3.141592654 * diameter() ; }
    void  print() const
     { cout << "Center is at ( " << _x << " , " << _y
            << " ) and " << "Radius = "  << _radius  ; }
  private:
    float _x ;
    float _y ;
    float _radius ;
};
```

Chapter 11

Overloading Operators

11.1 INTRODUCTION

C++ includes a rich store of 45 operators. They are summarized in Appendix C. These operators are defined automatically for the fundamental types (`int`, `float`, *etc.*). When you define a class, you are actually creating a new type. Most of the C++ operators can be overloaded to apply to your new class type. This chapter describes how to do that.

11.2 OVERLOADING THE ASSIGNMENT OPERATOR

Of all the operators, the assignment operator `=` is probably used the most. Its purpose is to copy one object to another. Like the default constructor, the copy constructor, and the destructor, the assignment operator is created automatically for every class that is defined. But also like those other three member functions, it can be defined explicitly in the class definition.

EXAMPLE 11.1 Adding an Assignment Operator to the `Ratio` Class

Here is a class interface for the `Ratio` class, showing the default constructor, the copy constructor, and the assignment operator:

```
class Ratio
{ public:
    Ratio(int =0, int =1);          // default constructor
    Ratio(const Ratio&);            // copy constructor
    void operator=(const Ratio&);   // assignment operator
    // other declarations go here
  private:
    int num, den;
};
```

Note the required syntax for the assignment operator. The name of this member function is `operator=`. Its argument list is the same as that of the copy constructor: it contains a single argument of the same class, passed by constant reference.

Here is the implementation of the overloaded assignment operator:

```
void Ratio::operator=(const Ratio& r)
{ num = r.num;
  den = r.den;
}
```

It simply copies the member data from the object `r` to the object that owns the call.

11.3 THE `this` POINTER

C++ allows assignments to be chained together, like this:

```
x = y = z = 3.14;
```

This is executed first by assigning 3.14 to `z`, then to `y`, and finally to `x`. But, as Example 11.1 shows, the assignment operator is really a function named `operator=`. In this chain, the function is called three times. On its first call, it assigns 3.14 to `z`, so the input to the function is 3.14. On its second call, it assigns 3.14 to `y`, so its input again must be 3.14. So that value should be the output (*i.e.*, return value) of the first call. Similarly, the output of the second call should again be 3.14 to serve as the input to the third call. The three calls to this function are nested, like this:

```
f(x, f(y, f(z, 3.14)))
```

The point is that the assignment operator is a function that should return the value it assigns. Therefore, instead of the return type `void`, the assignment operator should return a reference to the same type as the object being assigned:

```
Ratio& operator=(const Ratio& r)
```

This allows assignments to be chained together.

EXAMPLE 11.2 The Preferred Prototype for an Overloaded Assignment Operator

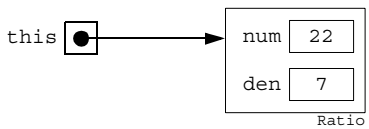
```
class Ratio
{ public:
    Ratio(int =0, int =1);           // default constructor
    Ratio(const Ratio&);             // copy constructor
    Ratio& operator=(const Ratio&);  // assignment operator
    // other declarations go here
  private:
    int num, den;
    // other declarations go here
};
```

The preferred syntax for the prototype of an overloaded assignment operator in a class `T` is

```
T& operator=(const T&);
```

The return type is a reference to an object of the same class `T`. But then this means that the function should return the object that is being assigned in order for the assignment chain to work. So when the assignment operator is being overloaded as a class member function, it should return the object that owns the call. Since there is no other name available for this owner object, C++ defines a special pointer, named **`this`**, which points to the owner object.

We can envision the `this` pointer like this:



Now we can give the correct implementation of the overloaded assignment operator:

EXAMPLE 11.3 Implementation of the Assignment Operator for the `Ratio` Class

```
Ratio& Ratio::operator=(const Ratio& r)
{ num = r.num;
  den = r.den;
  return *this;
}
```

Now assignments for the `Ratio` class can be chained together:

```
Ratio x, y, z(22,7);
x = y = z;
```

The correct implementation for an overloaded assignment operator in a class `T` is

```
T& T::operator=(const T& t)
{ // assign each member datum of t to the
  // corresponding member datum of the owner
  return *this;
}
```

Finally, note that an assignment is different from an initialization, even though they both use the equals sign:

```
Ratio x(22,7);  // this is an initialization
Ratio y(x);     // this is an initialization
Ratio z = x;    // this is an initialization
Ratio w;
w = x;             // this is an assignment
```

An initialization calls the copy constructor. An assignment calls the assignment operator.

11.4 OVERLOADING ARITHMETIC OPERATORS

All programming languages provide the standard arithmetic operators `+`, `-`, `*`, and `/` for numeric types. So it is only natural to define these for user-defined numeric types like the `Ratio` class. In older programming languages like C and Pascal, this is done by defining functions like this:

```
Ratio product(Ratio x, Ratio y)
{ Ratio z(x.num*y.num, x.den*y.den);
  return z;
}
```

This works. But the function has to be called in the conventional way:

```
z = product(x,y);
```

C++ allows such functions to be defined using the standard arithmetic operator symbols, so that they can be called more naturally:

```
z = x*y;
```

Like most operators in C++, the multiplication operator has a function name that uses the reserved word `operator`: its name is "`operator*`". Using this in place of "`product`" in the code above, we would expect the overloaded function to look something like this:

```
Ratio operator*(Ratio x, Ratio y)
{ Ratio z(x.num*y.num, x.den*y.den);
  return z;
}
```

But this is not a member function. If it were, we would have to set it up as in with only one argument. The `operator*` function requires two arguments.

Since the overloaded arithmetic operators cannot be member functions, they cannot access the `private` member data `num` and `den`. Fortunately, C++ allows an exception to this rule so that we can complete our definitions of the overloaded arithmetic functions. The solution is to declare the function as a `friend` of the `Ratio` class.

A `friend` function is a nonmember function that is given access to all members of the class within which it is declared. So it has all the privileges of a member function without actually being a member of the class. This attribute is used mostly with overloaded operators.

EXAMPLE 11.4 Declaring the Multiplication Operator as a `friend` Function

Here is the `Ratio` class declaration with the overloaded multiplication operator declared as a `friend` function:

```
class Ratio
{   friend Ratio operator*(const Ratio&, const Ratio&);
  public:
    Ratio(int =0, int =1);
    Ratio(const Ratio&);
    Ratio& operator=(const Ratio&);
    // other declarations go here
  private:
    int num, den;
    // other declarations go here
};
```

Note that the function prototype is inserted inside the class declaration, above the `public` section. Also note that the two arguments to the function are both passed by constant reference.

Now we can implement this nonmember just as we had expected:

```
Ratio operator*(const Ratio& x, const Ratio& y)
{ Ratio z(x.num * y.num, x.den * y.den);
  return z;
}
```

Note that the keyword `friend` is not used in the function implementation. Also note that the scope resolution prefix `Ratio::` is not used because this is not a member function.

Here is a little program that uses our improved `Ratio` class:

EXAMPLE 11.5 The `Ratio` Class with Assignment and Multiplication Operators

```
#include "Ratio.h"
int main()
{ Ratio x(22,7), y(-3,8), z;
  z = x;                          // assignment operator is called
  z.print();  cout << endl;
  x = y*z;                    // multiplication operator is called
  x.print();  cout << endl;
}
```

```
22/7
-33/28
```

Note that the `reduce()` function was called from within the constructor to reduce -66/56 to -33/58. (See Example 10.8 on page 238.)

11.5 OVERLOADING THE ARITHMETIC ASSIGNMENT OPERATORS

C++ allows you to combine arithmetic operations with the assignment operator; for example, using `x *= y` in place of `x = x * y`. These combination operators can all be overloaded for use in your own classes.

EXAMPLE 11.6 The `Ratio` Class with an Overloaded `*=` Operator

```
class Ratio
{ public:
    Ratio(int =0, int =1);
    Ratio& operator=(const Ratio&);
    Ratio& operator*=(const Ratio&);
    // other declarations go here
  private:
    int num, den;
    // other declarations go here
};
Ratio& Ratio::operator*=(const Ratio& r)
{ num = num*r.num;
  den = den*r.den;
  return *this;
}
```

The operator `operator*=` has the same syntax and nearly the same implementation as the basic assignment operator `operator=`. By returning `*this`, the operator can be chained, like this:

```
x *= y *= z;
```

It is also important to ensure that overloaded operators perform consistently with each other. For example, the following two lines should have the same effect, even though they call different operators:

```
x = x*y;
x *= y
```

11.6 OVERLOADING THE RELATIONAL OPERATORS

The six relational operators `<`, `>`, `<=`, `>=`, `==`, and `!=` can be overloaded the same way that the arithmetic operators are overloaded: as `friend` functions.

EXAMPLE 11.7 Overloading the Equality Operator `==` in the `Ratio` Class

Like other `friend` functions, the equality operator is declared above the `public` section of the class:

```
class Ratio
{   friend bool operator==(const Ratio&, const Ratio&);
    friend Ratio operator*(const Ratio&, const Ratio&);
    // other declarations go here
  public:
    Ratio(int =0, int =1);
    Ratio(const Ratio&);
    Ratio& operator=(const Ratio&);
    // other declarations go here
```

```
  private:
    int num, den;
    // other declarations go here
};
bool operator==(const Ratio& x, const Ratio& y)
{ return (x.num * y.den == y.num * x.den);
}
```

The test for equality of two fractions *a*/*b* and *c*/*d* is equivalent to the test *a***d* == *b***c*. So we end up using the equality operator for `int`s to define the equality operator for `Ratio`s.

Note that the relational operators return an `int` type, representing either "true" (1) or "false" (0).

11.7 OVERLOADING THE STREAM OPERATORS

C++ allows you to overload the stream insertion operator `>>` for customizing input and the stream deletion operator `<<` for customizing output. Like the arithmetic and relational operators, these should also be declared as `friend` functions.

For a class `T` with data member `d`, the syntax for the output operator is

```
friend ostream& operator<<(ostream& ostr, const T& t)
{ return ostr << t.d; }
```

Here, `ostream` is a standard class defined (indirectly) in the `iostream.h` header file. Note that all the parameters and the return value are passed by reference.

This function can then be called using the same syntax that we used for fundamental types:

```
cout << "x = " << x << ", y = " << y << endl;
```

Here is an example of how custom output can be written:

EXAMPLE 11.8 Overloading the Output Operator `<<` for the `Ratio` Class

```
class Ratio
{   friend ostream& operator<<(ostream&, const Ratio&);
  public:
    Ratio(int n=0, int d=1) : num(n), den(d) { }
    // other declarations go here
  private:
    int num, den;
    // other declarations go here
};
int main()
{ Ratio x(22,7), y(-3,8);
  cout << "x = " << x << ", y = " << y << endl;
}
ostream& operator<<(ostream& ostr, const Ratio& r)
{ return ostr << r.num << '/' << r.den;
}
```

```
x = 22/7, y = -3/8
```

When the second line of `main()` executes, the expression `cout << "x = "` executes first. This calls the standard output operator `<<`, passing the standard output stream `cout` and the string `"x = "` to it. As usual, this inserts the string into the output stream and then returns a reference to `cout`. This return value is then passed with the object `x` to the overloaded `<<` operator. This call to `operator<<` executes with `cout` in place of `ostr` and with `x` in place of `r`. The result is the execution of the line

```
return ostr << r.num << '/' << r.den;
```

which inserts `22/7` into the output stream and returns a reference to `cout`. Then another call to the standard output operator `<<` and another call to the overloaded operator are made, with the output (a reference to `cout`) of each call cascading into the next call as input. Finally, the last call to the standard output operator `<<` is made, passing `cout` and `endl`. This flushes the stream, causing the complete line

```
x = 22/7, y = -3/8
```

to be printed.

The syntax for overloading the input operator for a class `T` with data member `d` is

```
friend istream& operator>>(istream& istr, T& t)
{ return istr >> t.d; }
```

Here, `istream` is another standard class defined (indirectly) in the `iostream.h` header file.

Here is an example of how custom input can be written:

EXAMPLE 11.9 Overloading the Input Operator `>>` in the `Ratio` Class

```
class Ratio
{   friend istream& operator>>(istream&, Ratio&);
    friend ostream& operator<<(ostream&, const Ratio&);
  public:
    Ratio(int n=0, int d=1) : num(n), den(d) { }
    // other declarations go here
  private:
    int num, den;
    int gcd(int, int);
    void reduce();
};
int main()
{ Ratio x, y;
  cin >> x >> y;
  cout << "x = " << x << ", y = " << y << endl;
}
istream& operator>>(istream& istr, Ratio& r)
{ cout << "\t  Numerator: ";  istr >> r.num;
  cout << "\tDenominator: ";  istr >> r.den;
  r.reduce();
  return istr;
}
```

```
              Numerator: -10
            Denominator: -24
              Numerator: 36
            Denominator: -20
    x = 5/12, y = -9/5
```

This version of the input operator includes user prompts to facilitate input. It also includes a call to the utility function `reduce()`. Note that, as a `friend`, the operator can access this private function.

11.8 CONVERSION OPERATORS

In our original implementation of the `Ratio` class (Example 10.1 on page 233) we defined the member function `convert()` to convert from type `Ratio` to type `double`:

```
double convert() { return double(num)/den; }
```

This requires the member function to be called as

```
x.convert();
```

In keeping with our goal to make objects of the `Ratio` class behave like objects of fundamental types (*i.e.*, like ordinary variables), we would like to have a conversion function that could be called with a syntax that conforms to ordinary type conversions:

```
n = int(t);
y = double(x);
```

This can be done with a conversion operator.

Our `Ratio` class already has the facility to convert an object from `int` to `Ratio`:

```
Ratio x(22);
```

This is handled by the default constructor, which assigns 22 to `x.num` and 1 to `x.den`. This constructor also handles direct type conversions from type `int` to type `Ratio`:

```
x = Ratio(22);
```

Constructors of a given class are used to convert from another type to that class type.

To convert from the given class type to another type requires a different kind of member function. It is called a *conversion operator*, and it has a different syntax. If `type` is the type to which the object is to be converted, then the conversion operator is declared as

```
operator type();
```

For example, a member function of the `Ratio` class that returns an equivalent `float` would be declared as

```
operator float();
```

Or, if we want it to convert to type `double`, then we would declare it as

```
operator double();
```

And, if we want it to be usable for constant `Ratio`s (like `pi`), then we would declare it as

```
operator double() const;
```

Recall that, in our original implementation of the `Ratio` class (Example 10.1 on page 233) we defined the member function `convert()` for this purpose.

EXAMPLE 11.10 Adding a Conversion Operator to the `Ratio` Class

```
class Ratio
{ friend istream& operator>>(istream&, Ratio&);
  friend ostream& operator<<(ostream&, const Ratio&);
  public:
    Ratio(int n=0, int d=1) : num(n), den(d) { }
    operator double() const;
  private:
    int num, den;
};

int main()
{ Ratio x(-5,8);
  cout << "x = " << x << ", double(x) = " << double(x) << endl;
```

```
  const Ratio P(22,7);
  const double PI = double(P);
  cout << "P = " << P << ", PI = " << PI << endl;
}

Ratio::operator double() const
{ return double(num)/den;
}
```

```
x = -5/8, double(x) = -0.625
P = 22/7, PI = 3.14286
```

First we use the conversion operator `double()` to convert the `Ratio` object `x` into the `double` -0.625. Then we use it again to convert the constant `Ratio` object `p` into the constant `double pi`.

11.9 OVERLOADING THE INCREMENT AND DECREMENT OPERATORS

The increment operator `++` and the decrement operator `--` each have two forms: prefix and postfix. Each of these four forms can be overloaded. We'll examine the overloading of the increment operator here. Overloading the decrement operator works the same way.

When applied to integer types, the pre-increment operator simply adds 1 to the value of the object being incremented. This is a unary operator: its single argument is the object being incremented. The syntax for overloading it for a class named `T` is simply

```
T operator++();
```

So for our `Ratio` class, it is declared as

```
Ratio operator++();
```

EXAMPLE 11.11 Adding a Pre-Increment Operator to the `Ratio` Class

This example adds an overloaded pre-increment operator `++` to our `Ratio` class. Although we can make this function do whatever we want, it should be consistent with the action that the standard pre-increment operator performs on integer types. That adds 1 to the current value of the object before that value is used in the expression. This is equivalent to adding its denominator to its numerator:

$$\frac{22}{7} + 1 = \frac{22 + 7}{7} = \frac{29}{7}$$

So, we simply add `den` to `num` and then return `*this`, which is the object itself:

```
class Ratio
{   friend ostream& operator<<(ostream&, const Ratio&);
  public:
    Ratio(int n=0, int d=1) : num(n), den(d) { }
    Ratio operator++();
    // other declarations go here
  private:
    int num, den;
    // other declarations go here
};
int main()
{ Ratio x(22,7), y = ++x;
  cout << "y = " << y << ", x = " << x << endl;
```

```
}
Ratio Ratio::operator++()
{ num += den;
  return *this;
}
```

```
    y = 29/7, x = 29/7
```

Postfix operators have the same function name as the prefix operators. For example, both the pre-increment operator and the post-increment operator are named `operator++`. To distinguish them, C++ specifies that the prefix operator has one argument and the postfix operator has two arguments. (When used, they both appear to have one argument.) So the correct syntax for the prototype for an overloaded post-increment operator is

```
T operator++(int);
```

The required argument must have type `int`. This appears a bit strange because no integer is passed to the function when it is invoked. The integer argument is thus a *dummy argument*, required only so that the postfix operator can be distinguished from the corresponding prefix operator.

EXAMPLE 11.12 Adding a Post-Increment Operator to the `Ratio` Class

To be consistent with the ordinary post-increment operator for integer types, this overloaded version should not change the value of `x` until after it has been assigned to `y`. To do that, we need a temporary object to hold the contents of the object that owns the call. This is done by assigning `*this` to temp. Then this object can be returned after adding `den` to `num`.

```
class Ratio
{ friend ostream& operator<<(ostream&, const Ratio&);
  public:
    Ratio(int n=0, int d=1) : num(n), den(d) { }
    Ratio operator++();          // pre-increment
    Ratio operator++(int);       // post-increment
 private:
int num, den;
};

int main()
{ Ratio x(22,7), y = x++;
  cout << "y = " << y << ", x = " << x << endl;
}

Ratio Ratio::operator++(int)
{ Ratio temp = *this;
  num += den;
  return temp;
}
```

```
y = 22/7, x = 29/7
```

Note that the dummy argument in the `operator++` function is an unnamed `int`. It need not be named because it is not used. But it must be declared to distinguish the post-increment from the pre-increment operator.

11.10 OVERLOADING THE SUBSCRIPT OPERATOR

Recall that, if `a` is an array, then the expression `a[i]` really means nothing more than `*(a+i)`. This is because `a` is actually the address of the initial element in the array, so `a+i` is the address of the *i*th element, since the number of bytes added to `a` is `i` times the size of each array element.

The symbol `[]` denotes the *subscript operator.* Its name derives from the original use of arrays, where `a[i]` represented the mathematical symbol a_i. When used as `a[i]`, it has two operands: `a` and `i`. The expression `a[i]` is equivalent to `operator[](a, i)`. And as an operator, `[]` can be overloaded.

EXAMPLE 11.13 Adding a Subscript Operator to the `Ratio` Class

```
class Ratio
{   friend ostream& operator<<(ostream&, const Ratio&);
  public:
    Ratio(int n=0, int d=1) : num(n), den(d) { }
    int& operator[](int);
    // other declarations go here
  private:
    int num, den;
    // other declarations go here
};

int main()
{ Ratio x(22,7);
  cout << "x = " << x << endl;
  cout << "x[1] = " << x[1] << ", x[2] = " << x[2] << endl;
}

ostream& operator<<(ostream& ostr, const Ratio& r)
{ return ostr << r.num << "/" << r.den;
}

int& Ratio::operator[](int i)
{ if (i == 1) return num;
  else return den;
}
```

```
    x = 22/7
    x[1] = 22, x[2] = 7
```

The expression `x[1]` calls the subscript operator, passing 1 to i, which returns `x.num`. Similarly, `x[2]` returns `x.den`. If `i` has any value other than 1 or 2, then an error message is sent to `cerr`, the standard error stream, and then the `exit()` function is called.

This example is artificial. There is no advantage to accessing the fields of the `Ratio` object `x` with `x[1]` and `x[2]` instead of `x.num` and `x.den`. However, there are many important classes where the subscript is very useful. (See Problem 11.2.)

Note that the subscript operator is an access function, since it provides `public` access to `private` member data.

Review Questions

11.1 How is the `operator` keyword used?

11.2 What does `*this` always refer to?

11.3 Why can't the `this` pointer be used in nonmember functions?

11.4 Why should the overloaded assignment operator return `*this`?

11.5 What is the difference between the effects of the following two declarations:

```
Ratio y(x);
Ratio y = x;
```

11.6 What is the difference between the effects of the following two lines:

```
Ratio y = x;
Ratio y;  y = x;
```

11.7 Why can't `**` be overloaded as an exponentiation operator?

11.8 Why should the stream operators `<<` and `>>` be overloaded as `friend` functions?

11.9 Why should the arithmetic operators `+`, `-`, `*`, and `/` be overloaded as `friend` functions?

11.10 How is the overloaded pre-increment operator defintion distinguished from that of the overloaded post-increment operator?

11.11 Why is the `int` argument in the implementation of the post-increment operator left unnamed?

11.12 What mechanism allows the overloaded subscript operator `[]` to be used on the left side of an assignment statement, like this: `v[2] = 22`?

Problems

11.1 Implement the binary subtraction operator, the unary negation operator, and the less-than operator `<` for the `Ratio` class (see Example 11.4 on page 259).

11.2 Implement a `Vector` class, with a default constructor, a copy constructor, a destructor, and overloaded assignment operator, subscript operator, equality operator, stream insertion operator, and stream extraction operator.

11.3 Implement the addition and division operators for the `Ratio` class (see Example 11.5 on page 259).

11.4 Rewrite the overloaded input operator for the `Ratio` class (Example 11.9 on page 262) so that, instead of prompting for the numerator and denominator, it reads a fraction type as "`22/7`".

11.5 Implement an overloaded assignment operator `=` for the `Point` class (see Problem 10.1 on page 249).

11.6 Implement overloaded stream insertion operator `<<` for the `Point` class (see Problem 10.1 on page 249).

11.7 Implement overloaded comparison operators `==` and `!=` for the `Point` class (see Problem 10.1 on page 249).

11.8 Implement overloaded addition operator `+` and subtraction operator `-` for the `Point` class (see Problem 10.1 on page 249).

11.9 Implement an overloaded multiplication operator `*` to return the dot product of two `Point` objects (see Problem 10.1 on page 249).

Answers to Review Questions

11.1 The `operator` keyword is used to form the name of a function that overloads an operator. For example, the name of the function that overloads the assignment operator `=` is "`operator=`".

11.2 The expression `*this` always refers to the object that owns the call of the member function in which the expression appears. Therefore, it can only be used within member functions.

11.3 The keyword `this` is a pointer to the object that owns the call of the member function in which the expression appears.

11.4 The overloaded assignment operator should return `*this` so that the operator can be used in a cascade of calls, like this: `w = x = y = z;`

11.5 There is no difference. Both declarations use the copy constructor to create the object `y` as a duplicate of the object `x`.

11.6 The declaration `Ratio y = x;` calls the copy constructor. The code `Ratio y; y = x;` calls the default constructor and then the assignment operator.

11.7 The symbol `**` cannot be overloaded as an operator because it is not a C++ operator.

11.8 The stream operators `<<` and `>>` should be overloaded as `friend` functions because their left operands should be stream objects. If an overloaded operator is a member function, then its left operand is `*this`, which is an object of the class to which the function is a member.

11.9 The arithmetic operators `+`, `-`, `*`, and `/` should be overloaded as `friend` functions so that their left operands can be declared as `const`. This allows, for example, the use of an expression like `22 + x`. If an overloaded operator is a member function, then its left operand is `*this`, which is not `const`.

11.10 The overloaded pre-increment operator has no arguments. The overloaded post-increment operator has one (dummy) argument, of type `int`.

11.11 The `int` argument in the implementation of the post-increment operator is left unnamed because it is not used. It is a dummy argument.

11.12 By returning a reference, the overloaded subscript operator `[]` can be used on the left side of an assignment statement, like this: `v[2] = 22`. This is because, as a reference, `v[2]` is an lvalue.

Solutions to Problems

11.1 All three of these operators are implemented as `friend` functions to give them access to the `num` and `den` data members of their owner objects:

```
class Ratio
{   friend Ratio operator-(const Ratio&, const Ratio&);
    friend Ratio operator-(const Ratio&);
    friend bool operator<(const Ratio&, const Ratio&);
  public:
    Ratio(int =0, int =1);
    Ratio(const Ratio&);
    Ratio& operator=(const Ratio&);
    // other declarations go here
  private:
    int num, den;
    int gcd(int, int)
    int reduce();
};
```

The binary subtraction operator simply constructs and returns a `Ratio` object `z` that represents the difference `x - y`:

```
Ratio operator-(const Ratio& x, const Ratio& y)
{ Ratio z(x.num*y.den - y.num*x.den, x.den*y.den);
```

```
  z.reduce();
  return z;
}
```

Algebraically, the subtraction *a*/*b* - *c*/*d* is performed using the common denominator *bd*:

$$\frac{a}{b} - \frac{c}{d} = \frac{ad - bc}{bd}$$

So the numerator of `x - y` should be `x.num*y.den - y.num*x.den` and the denominator should be `x.den*y.den`. The function constructs the `Ratio` object `z` with that numerator and denominator. This algebraic formula can produce a fraction that is not in reduced form, even if x and y are. For example, 1/2 – 1/6 = (1·6 – 2·1)/(2·6) = 4/12. So we call the `reduce()` utility function before returning the resulting object `z`.

The unary negation operator overloads the symbol "`-`". It is distinguished from the binary subtraction operator by its parameter list; it has only one parameter:

```
Ratio Ratio::operator-(const Ratio& x)
{ Ratio y(-x.num, x.den);
  return y;
}
```

To negate a fraction *a*/*b* we simply negate its numerator: (-*a*)/*b*. So the newly constructed `Ratio` object `y` has the same denominator as `x` but its numerator is `-x.num`. The less-than operator is easier to do if we first modify our default constructor to ensure that every object's `den` value is positive. Then we can use the standard equivalence for the less-than operator:

$$\frac{a}{b} < \frac{c}{d} \Leftrightarrow ad < bc$$

```
bool operator<(const Ratio& x, const Ratio& y)
{ return (x.num*y.den < y.num*x.den);
}

Ratio::Ratio(int n, int d) : num(n), den(d)
{ if (d == 0) n = 0;
  else if (d < 0) { n *= -1; d *= -1; }
  reduce();
}
```

The modification ensuring that `den > 0` could instead be done in the `reduce()` function, since that utility should be called by every member function that allows `den` to be changed. However, none of our other member functions allows the sign of `den` to change, so by requiring it to be positive when the object is constructed we don't need to check the condition again.

11.2 Here is the class declaration:

```
class Vector
{   friend bool operator==(const Vector&, const Vector&);
    friend ostream& operator<<(ostream&, const Vector&);
    friend istream& operator>>(istream&, Vector&);
  public:
    Vector(int =1, double =0.0);            // default constructor
    Vector(const Vector&);                  // copy constructor
    ~Vector();                              // destructor
    const Vector& operator=(const Vector&); // assignment operator
    double& operator[](int) const;          // subscript operator
  private:
    int size;
    double* data;
};
```

Here is the implementation of the overloaded equality operator:

```
bool operator==(const Vector& v, const Vector& w)
{ if (v.size != w.size) return 0;
  for (int i = 0; i < v.size; i++)
  if (v.data[i] != w.data[i]) return 0;
  return 1;
}
```

It is a nonmember function which returns 1 or 0 according to whether the two vectors `v` and `w` are equal. If their sizes are not equal, then it returns 0 immediately. Otherwise it checks the corresponding elements of the two vectors, one at a time. If there is any mismatch, then again it returns 0 immediately. Only if the entire loop finishes without finding any mismatches can we conclude that the two vectors are equal and return 1.

Here is the implementation of the overloaded stream insertion operator:

```
ostream& operator<<(ostream& ostr, const Vector& v)
{ ostr << '(';
  for (int i = 0; i < v.size-1; i++) {
  ostr << v[i] << ", ";
  if ((i+1)%8 == 0) cout << "\n ";
  }
  return ostr << v[i] << ")\n";
}
```

This prints the vector like this: `(1.11111, 2.22222, 3.33333, 4.44444, 5.55556).` The conditional inside the loop allows the output to "wrap" around several lines neatly if the vector has more than 8 elements.

The output is sent to `ostr` which is just a local name for the output stream that is passed to the function. That would be `cout` if the function is called like this: `cout << v;`.

In the last line of the function, the expression `ostr << v[i] << ")\n"` makes two calls to the (standard) stream extraction operator. Those two calls return `ostr` as the value of this expression, and so that object `ostr` is then returned by this function.

Here is the overloaded stream extraction operator:

```
istream& operator>>(istream& istr, Vector& v)
{ for (int i = 0; i < v.size; i++)
  { cout << i << ": ";
    istr >> v[i];
  }
  return istr;
}
```

This implementation prompts the user for each element of the vector `v`. It could also be implemented without user prompts, simply reading the elements one at a time. Notice that the elements are read from the input stream `istr`, which is the first parameter passed in to the function. When the function is called like this: `cin >> v;` the standard input stream `cin` will be passed to the parameter `istr`, so the vector elements are actually read from `cin`. The argument `istr` is simply a local name for the actual input stream which probably will be `cin`. Notice that this argument is also returned, allowing a cascade of calls like this: `cin >> u >> v >> w;`.

Here is the implementation of the default constructor:

```
Vector::Vector(int sz, double t) : size(sz)
{ data = new double[size];
  for (int i = 0; i < size; i++)
    data[i] = t;
}
```

The declaration `Vector u;` would construct the vector `u` having 1 element with the value 0.0; the declaration `Vector v(4);` would construct the vector `v` with 4 elements all with the value 0.0; and the declaration `Vector w(8, 3.14159);` would construct the vector `w` with 8 elements all with the value 3.14159.

This constructor uses the initialization list `size(sz)` to assign the argument `sz` to the data member `size`. Then it uses the `new` operator to allocate that number of elements to the array `data`. Finally, it initializes each element with the value `t`.

The copy constructor is almost the same as the default constructor:

```
Vector::Vector(const Vector& v) : size(v.size)
{ data = new double[v.size];
  for (int i = 0; i < size; i++)
    data[i] = v.data[i];
}
```

It uses the data members of the vector argument `v` to initialize the object being constructed. So it assigns `v.size` to the new object's `size` member, and it assigns `v.data[i]` to the elements of the new object's `data` member.

The destructor simply restores the storage allocated to the `data` array and then sets `data` to `NULL` and `size` to 0:

```
Vector::~Vector()
{ delete [] data;
  data = NULL;
  size = 0;
}
```

The overloaded assignment operator creates a new object that duplicates the vector `v`:

```
const Vector& Vector::operator=(const Vector& v)
{ if (&v != this)
  { delete [] data;
    size = v.size;
    data = new double[v.size];
    for (int i = 0; i < size; i++)
      data[i] = v.data[i];
  }
  return *this;
}
```

The condition `(&v != this)` determines whether the object that owns the call is different from the vector `v`. If the address of `v` is the same as `this` (which is the address of the current object), then they are the same object and nothing needs to be done. This check is a safety precaution to guard against the possibility that an object might, directly or indirectly, be assigned to itself, like this: `w = v = w;`.

Before creating a new object, the function restores the allocated data array. Then it copies the vector `v` the same way that the copy constructor did.

The overloaded subscript operator simply returns the `i`th component of the object's `data` array:

```
double& Vector::operator[](int i) const
{ return data[i];
}
```

11.3

```
Ratio operator+(const Ratio& r1, const Ratio& r2)
{ Ratio r(r1.num*r2.den+r2.num*r1.den,r1.den*r2.den);
  r.reduce();
  return r;
}
Ratio operator/(const Ratio& r1, const Ratio& r2)
{ Ratio r(r1.num*r2.den,r1.den*r2.num);
```

```
  r.reduce();
  return r;
}
```

11.4

```
ostream& operator<<(ostream& ostr, const Ratio& r)
{ return ostr << r.num << "/" << r.den;
}
```

11.5

```
Point& Point::operator=(const Point& point)
{ _x = point._x;
  _y = point._y;
  _z = point._z;
  return *this;
}
```

11.6

```
ostream& operator<<(ostream& ostr, const Point& point)
{ return ostr << "(" << _x << "," << _y << "," << _z << ")";
}
```

11.7

```
bool Point::operator==(const Point& point) const
{ return _x == point._x && _y == point._y && _z == point._z;
}
bool Point::operator!=(const Point& point) const
{ return _x != point._x || _y != point._y || _z != point._z;
}
```

11.8

```
Point operator+(const Point& p1, const Point& p2)
{ return Point(p1._x+p2._x,p1._y+p2._y,p1._z+p2._z);
}
Point operator-(const Point& p1, const Point& p2)
{ return Point(p1._x-p2._x,p1._y-p2._y,p1._z-p2._z);
}
```

11.9

```
Point operator*(const double coef, const Point& point)
{ return Point(coef*point._x,coef*point._y,coef*point._z);
}
```

Chapter 12

Composition and Inheritance

12.1 INTRODUCTION

We often need to use existing classes to define new classes. The two ways to do this are called *composition* and the *inheritance*. This chapter describes both methods and shows how to decide when to use them.

12.2 COMPOSITION

Composition (also called *containment* or *aggregation*) of classes refers to the use of one or more classes within the definition of another class. When a data member of the new class is an object of another class, we say that the new class is a *composite* of the other objects.

EXAMPLE 12.1 A `Person` Class

Here is a simple definition for a class to represent people.

```
class Person
{ public:
    Person(char* n="", char* nat="U.S.A.", int s=1)
      : name(n), nationality(nat), sex(s) { }
    void printName() { cout << name; }
    void printNationality() { cout << nationality; }
  private:
    string name, nationality;
    int sex;
};

int main()
{ Person creator("Bjarne Stroustrup", "Denmark");
  cout << "The creator of C++ was ";
  creator.printName();
  cout << ", who was born in ";
  creator.printNationality();
  cout << ".\n";
}
```

```
The creator of C++ was Bjarne Stroustrup, who was born in Denmark.
```

This example illustrates the *composition* of the `string` class within the `Person` class. The next example defines another class that we can compose with the `Person` class to improve it:

EXAMPLE 12.2 A `Date` Class

```
class Date
{   friend istream& operator>>(istream&, Date&);
    friend ostream& operator<<(ostream&, const Date&);
  public:
    Date(int m=0, int d=0, int y=0) : month(m), day(d), year(y) { }
    void setDate(int m, int d, int y) { month = m; day = d; year = y; }
  private:
    int month, day, year;
};

istream& operator>>(istream& in, Date& x)
{ in >> x.month >> x.day >> x.year;
  return in;
}

ostream& operator<<(ostream& out, const Date& x)
{ static char* monthName[13] = {"", "January","February",
    "March", "April", "May", "June", "July", "August",
    "September", "October", "November", "December"};
  out << monthName[x.month] << ' ' << x.day << ", " << x.year;
  return out;
}

int main()
{ Date peace(11,11,1918);
  cout << "World War I ended on " << peace << ".\n";
  peace.setDate(8,14,1945);
  cout << "World War II ended on " << peace << ".\n";
  cout << "Enter month, day, and year: ";
  Date date;
  cin >> date;
  cout << "The date is " << date << ".\n";
}
```

```
World War I ended on November 11, 1918.
World War II ended on August 14, 1945.
Enter month, day, and year: 7 4 1776
The date is July 4, 1776.
```

The test driver tests the default constructor, the `setDate()` function, the overloaded insertion operator `<<`, and the overloaded extraction operator `>>`.

Now we can use the `Date` class inside the `Person` class to store a person's date of birth and date of death:

EXAMPLE 12.3 Composing the `Date` Class with the `Person` Class

```
#include "Date.h"

class Person
{ public:
```

```
    Person(char* n="", int s=0, char* nat="U.S.A.")
     : name(n), sex(s), nationality(nat) { }
    void setDOB(int m, int d, int y) { dob.setDate(m, d, y); }
    void setDOD(int m, int d, int y) { dod.setDate(m, d, y); }
    void printName() { cout << name; }
    void printNationality() { cout << nationality; }
    void printDOB() { cout << dob; }
    void printDOD() { cout << dod; }
  private:
    string name, nationality;
    Date dob, dod;                  // date of birth, date of death
    int sex;                        // 0 = female, 1 = male
};

int main()
{ Person author("Thomas Jefferson", 1);
  author.setDOB(4,13,1743);
  author.setDOD(7,4,1826);
  cout << "The author of the Declaration of Independence was ";
  author.printName();
  cout << ".\nHe was born on ";
  author.printDOB();
  cout << " and died on ";
  author.printDOD();
  cout << ".\n";
}
```

```
The author of the Declaration of Independence was Thomas Jefferson.
He was born on April 13, 1743 and died on July 4, 1826.
```

Notice again that we have used a member function of one class to define member functions of the composed class: the `setDate()` function is used to define the `setDOB()` and `setDOD()` functions.

Composition is often referred to as a "has-a" relationship because the objects of the composite class "have" objects of the composed class as members. Each object of the `Person` class "has a" `name` and a `nationality` which are `string` objects. Composition is one way of reusing existing software to create new software.

12.3 INHERITANCE

Another way to reuse existing software to create new software is by means of inheritance (also called *specialization* or *derivation*). This is often referred to as an "is-a" relationship because every object of the class being defined "is" also an object of the inherited class.

The common syntax for deriving a class `Y` from a class `X` is

```
class Y : public X {
// ...
};
```

Here `X` is called the *base class* (or *superclass*) and `Y` is called the *derived class* (or *subclass*). The keyword `public` after the colon specifies *public inheritance*, which means that `public` members of the base class become `public` members of the derived class.

EXAMPLE 12.4 Deriving a `Student` Class from the `Person` Class

Students are people. So it is natural to use the `Person` class to derive a `Student` class:

```
#include "Person.h"
class Student : public Person
{ public:
    Student(char* n, int s=0, char* i="")
     : Person(n, s), id(i), credits(0) { }
    void setDOM(int m, int d, int y) { dom.setDate(m, d, y); }
    void printDOM() { cout << dom; }
  private:
    string id;                    // student identification number
    Date dom;                     // date of matriculation
    int credits;                  // course credits
    float gpa;                    // grade-point average
};
```

The `Student` class inherits all the `public` functionality of the `Person` class, including the `Person()` constructor which it uses in its constructor to initialize `name` in the `Person` class. Note that this is a `private` member of the `Person` class, so it could not be accessed directly.

Here is a test driver for the `Student` class:

```
#include "Student.h"
int main()
{ Student x("Ann Jones", 0, "219360061");
  x.setDOB(5, 13, 1977);
  x.setDOM(8, 29, 1995);
  x.printName();
  cout << "\n\t        Born: ";  x.printDOB();
  cout << "\n\tMatriculated: ";  x.printDOM();  cout << endl;
}
```

```
Ann Jones
                Born: May 13, 1977
        Matriculated: August 29, 1995
```

12.4 `protected` CLASS MEMBERS

The `Student` class in Section 12.3 has a significant problem: it cannot directly access the `private` data members of its `Person` superclass: `name`, `nationality`, `DOB`, `DOD`, and `sex`. The lack of access on the first four of these is not serious because these can be written and read through the `Person` class's constructor and public access functions. However, there is no way to write or read a `Student`'s `sex`. One way to overcome this problem would be to make `sex` a data member of the `Student` class. But that is unnatural: `sex` is an attribute that all `Person` objects have, not just `Student`s. A better solution is to change the `private` access specifier to `protected` in the `Person` class. That allows access to these data members from derived classes.

EXAMPLE 12.5 The `Person` Class with `protected` Data Members

These are the same class definitions that were given in the two previous examples except that the `private` access specifier has been changed to `protected`, and we have added the access function `printSex()` to the `Student` class:

```
#include "Date.h"

class Person
{ public:
    Person(char* n="", int s=0, char* nat="U.S.A.")
     : name(n), sex(s), nationality(nat) { }
    void setDOB(int m, int d, int y) { dob.setDate(m, d, y); }
    void setDOD(int m, int d, int y) { dod.setDate(m, d, y); }
    void printName() { cout << name; }
    void printNationality() { cout << nationality; }
    void printDOB() { cout << dob; }
    void printDOD() { cout << dod; }
  protected:
    string name, nationality;
    Date dob, dod;                  // date of birth, date of death
    int sex;                        // 0 = female, 1 = male
};

class Student : public Person
{ public:
    Student(char* n, int s=0, char* i="")
     : Person(n, s), id(i), credits(0) { }
    void setDOM(int m, int d, int y) { dom.setDate(m, d, y); }
    void printDOM() { cout << dom; }
    void printSex() { cout << (sex ? "male" : "female"); }
  protected:
    string id;                      // student identification number
    Date dom;                       // date of matriculation
    int credits;                    // course credits
    float gpa;                      // grade-point average
};
```

Now all five data members defined in the `Person` class are accessible from its `Student` subclass, as seen by the following test driver:

```
int main()
{ Student x("Ann Jones", 0, "219360061");
  x.setDOB(5, 13, 1977);
  x.setDOM(8, 29, 1995);
  x.setDOD(7,4,1826);
  x.printName();
  cout << "\n\t        Born: ";  x.printDOB();
  cout << "\n\t         Sex: ";  x.printSex();
  cout << "\n\tMatriculated: ";  x.printDOM();
  cout << endl;
}
```

```
Ann Jones
                Born: May 13, 1977
                 Sex: female
        Matriculated: August 29, 1995
```

The `protected` access category is a balance between `private` and `public` categories: `private` members are accessible only from within the class itself and its `friend` classes;

`protected` members are accessible from within the class itself, its `friend` classes, its derived classes, and their `friend` classes; `public` members are accessible from anywhere within the file. In general, `protected` is used instead of `private` whenever it is anticipated that a subclass might be defined for the class.

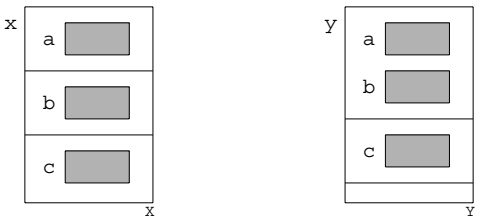
A subclass inherits all the `public` and `protected` members of its base class. This means that, from the point of view of the subclass, the `public` and `protected` members of its base class appear as though they actually were declared in the subclass. For example, suppose that class `X` and subclass `Y` are defined as

```
class X
{ public:
    int a;
  protected:
    int b;
  private:
    int c;
};

class Y : public X
{ public:
    int d;
};
```

and `x` and `y` are declared by

```
X x;
Y y;
```

Then we can visualize objects `x` and `y` as shown below.



The `public` member `a` of class `X` is inherited as a `public` member of `y`, and the `protected` member `b` of class `X` is inherited as a `protected` member of `y`. But the `private` member `c` of class `X` is not inherited by `y`. (The horizontal lines in each object indicate the separate `public`, `protected`, and `private` regions of the object.)

12.5 OVERRIDING AND DOMINATING INHERITED MEMBERS

If `Y` is a subclass of `X`, then `Y` objects inherit all the `public` and `protected` member data and member functions of `X`. For example, the `name` data and `printName()` function in the `Person` class are also members of the `Student` class.

In some cases, you might want to define a local version of an inherited member. For example, if `a` is a data member of `X` and if `Y` is a subclass of `X`, then you could also define a separate data member named `a` for `Y`. In this case, we say that the `a` defined in `Y` *dominates* the `a` defined in `X`. Then a reference `y.a` for an object `y` of class `Y` will access the `a` defined in `Y` instead of the `a` defined in `X`. To access the `a` defined in `X`, one would use `y.X::a`.

The same rule applies to member functions. If a function named `f()` is defined in `X` and another function named `f()` with the same signature is defined in `Y`, then `y.f()` invokes the latter function, and `y.X::f()` invokes the former. In this case, the local function `y.f()` *overrides* the `f()` function defined in `X` unless it is invoked as `y.X::f()`.

These distinctions are illustrated in the following example.

EXAMPLE 12.6 Dominating a Data Member and Overriding a Member Function

Here are two classes, `X` and `Y`, with `Y` inheriting from `X`.

```
class X
{ public:
    void f() { cout << "X::f() executing\n"; }
    int a;
};

class Y : public X
{ public:
    void f() { cout << "Y::f() executing\n"; }  // overrides X::f()
    int a;                                      // dominates X::a
};
```

But the members of `Y` have the same signatures as those in `X`. So `Y`'s member function `f()` overrides the `f()` defined in `X`, and `Y`'s data member `a` dominates the `a` defined in `X`.

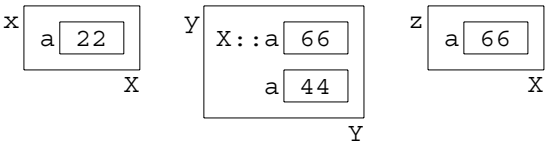
Here is a test driver for the two classes:

```
int main()
{ X x;
  x.a = 22;
  x.f();
  cout << "x.a = " << x.a << endl;
  Y y;
  y.a = 44;                  // assigns 44 to the a defined in Y
  y.X::a = 66;               // assigns 66 to the a defined in X
  y.f();                     // invokes the f() defined in Y
  y.X::f();                  // invokes the f() defined in X
  cout << "y.a = " << y.a << endl;
  cout << "y.X::a = " << y.X::a << endl;
  X z = y;
  cout << "z.a = " << z.a << endl;
}
```

```
X::f() executing
x.a = 22
Y::f() executing
X::f() executing
y.a = 44
y.X::a = 66
z.a = 66
```

Here, `y` has access to two different data members named `a` and two different functions `f()`. The defaults are the ones defined in the derived class `Y`. The scope resolution operator `::` is used in the form `X::` to override the defaults to access the corresponding members defined in the parent class `X`. When the `X` object `z` is initialized with `y`, its `X` members are used: `z.a` is assigned the value `y.X::a`.

This diagram illustrates the three objects `x`, `y`, and `z`:



Example 12.6 and most of the remaining examples in this chapter are designed to illustrate the intricacies of inheritance. They are not intended to exemplify common programming practice. Instead, they focus on specific aspects of C++ which can then be applied to more general and practical situations. In particular, the method of dominating data members as illustrated in Example 12.6 is rather unusual. Although it is not uncommon to override function members, dominating data members of the same type is rare. More common would be the reuse of the same data name with a different type, like this:

```
class Y : public X
{ public:
    double a;       // the data member a in class X had type int
}
```

In an inheritance hierarchy, default constructors and destructors behave differently from other member functions. As the following example illustrates, each constructor invokes its parent constructor before executing itself, and each destructor invokes its parent destructor after executing itself:

EXAMPLE 12.7 Parent Constructors and Destructors

```
class X
{ public:
    X() { cout << "X::X() constructor executing\n"; }
    ~X() { cout << "X::X() destructor executing\n"; }
};

class Y : public X
{ public:
    Y() { cout << "Y::Y() constructor executing\n"; }
    ~Y() { cout << "Y::Y() destructor executing\n"; }
};
class Z : public Y
{ public:
    Z(int n) { cout << "Z::Z(int) constructor executing\n"; }
    ~Z() { cout << "Z::Z() destructor executing\n"; }
};

int main()
{ Z z(44);
}
```

When `z` is declared, the `Z::Z(int)` constructor is called. Before executing, it calls the `Y::Y()` constructor which immediately calls the `X::X()` constructor. After the `X::X()` constructor has finished executing, control returns to the `Y::Y()` constructor which finishes executing. Then finally the `Z::Z()` constructor finishes executing. The effect is that all the parent default constructors execute in top-down order.

The same thing happens with the destructors, except that each destructor executes its own code before calling its parent destructor. So all the parent destructors execute in bottom-up order.

Here is a more realistic example:

EXAMPLE 12.8 Parent Constructors and Destructors

Here is a demo program that uses a base class `Person` and a derived class `Student`:

```
class Person
{ public:
    Person(const char* s)
      { name = new char[strlen(s)+1]; strcpy(name, s); }
    ~Person() { delete [] name; }
  protected:
    char* name;
};

class Student : public Person
{ public:
    Student(const char* s, const char* m) : Person(s)
      { major = new char[strlen(m)+1]; strcpy(major, m); }
    ~Student() { delete [] major; }
  private:
    char* major;
};

int main()
{ Person x("Bob");
  { Student y("Sarah", "Biology");
  }
}
```

When `x` is instantiated, it calls the `Person` constructor which allocates 4 bytes of memory to store the string "`Bob`". Then `y` instantiates, first calling the `Person` constructor which allocates 6 bytes to store the string "`Sarah`" and then allocating 8 more bytes of memory to store the string "`Biology`". The scope of `y` terminates before that of `x` because it is declared within an internal block. At that moment, `y`'s destructor deallocates the 8 bytes used for "`Biology`" and then calls the `Person` destructor which deallocates the 6 bytes used for "`Sarah`". Finally, the `Person` destructor is called to destroy `x`, deallocating the 4 bytes used for "`Bob`".

12.6 `private` ACCESS VERSUS `protected` ACCESS

The difference between `private` and `protected` class members is that subclasses can access `protected` members of a parent class but not `private` members. Since `protected` is more flexible, when would you want to make members `private`? The answer lies at the heart of the principle of information hiding: restrict access now to facilitate changes later. If you think you may want to modify the implementation of a data member in the future, then declaring it private will obviate the need to make any corollary changes in subclasses. Subclasses are independent of `private` data members.

EXAMPLE 12.9 The `person` Class with `protected` and `private` Data Members

Suppose that we need to know whether people (*i.e.*, `Person` objects) are high school graduates. We could just add a `protected` data member like `sex` that stores either 0 or 1. But we might decide later to replace it with data member(s) that contain more detailed information about the person's education. So, for now, we set up a `private` data member `hs` to prevent derived classes from accessing it directly:

```
class Person
{ public:
    Person(char* n="", int s=0, char* nat="U.S.A.")
     : name(n), sex(s), nationality(nat) { }
    // ...
  protected:
    string name, nationality;
    Date dob, dod;               // date of birth, date of death
    int sex;                     // 0 = female, 1 = male
    void setHSgraduate(int g) { hs = g; }
    int isHSgraduate() { return hs; }
  private:
    int hs;                      // = 1 if high school graduate
};
```

We include `protected` access functions to allow subclasses to access the information. If we do later replace the `hs` data member with something else, we need only modify the implementations of these two access functions without affecting any subclasses.

12.7 `virtual` FUNCTIONS AND POLYMORPHISM

One of the most powerful features of C++ is that it allows objects of different types to respond differently to the same function call. This is called *polymorphism* and it is achieved by means of `virtual` functions. Polymorphism is rendered possible by the fact that a pointer to a base class instance may also point to any subclass instance:

```
class X
{ // ...
}

class Y : public X                          // Y is a subclass if X
{ // ...
}

int main()
{ X* p;               // p is a pointer to objects of base class X
  Y y;
  p = &y;             // p can also point to objects of subclass Y
}
```

So if `p` has type `X*` ("pointer to type `X`"), then `p` can also point to any object whose type is a subclass of `X`. However, even when `p` is pointing to an instance of a subclass `Y`, its type is still `X*`. So an expression like `p->f()` would invoke the function `f()` defined in the base class.

Recall that `p->f()` is an alternate notation for `(*p).f()`. This invokes the member function `f()` of the object to which `p` points. But what if `p` is actually pointing to an object `y` of a subclass of the class to which `p` points, and what if that subclass `Y` has its own overriding version of `f()`? Which

`f()` gets executed: `X::f()` or `Y::f()`? The answer is that `p->f()` will execute `X::f()` because `p` had type `X*`. The fact that `p` happens to be pointing at that moment to an instance of subclass `Y` is irrelevant; it's the statically defined type `X*` of `p` that normally determines its behavior.

EXAMPLE 12.10 Using `virtual` Functions

This demo program declares `p` to be a pointer to objects of the base class `X`. First it assigns `p` to point to an instance `x` of class `X`. Then it assigns `p` to point to an instance `y` of the derived class `Y`.

```
class X
{ public:
    void f() { cout << "X::f() executing\n"; }
};

class Y : public X
{ public:
    void f() { cout << "Y::f() executing\n"; }
};

int main()
{ X x;
  Y y;
  X* p = &x;
  p->f();         // invokes X::f() because p has type X*
  p = &y;
  p->f();         // invokes X::f() because p has type X*
}
```

```
X::f() executing
X::f() executing
```

Two function calls `p->f()` are made. Both calls invoke the same version of `f()` that is defined in the base class `X` because `p` is declared to be a pointer to `X` objects. Having `p` point to `y` has no effect on the second call `p->f()`.

Transform `X::f()` into a *virtual function* by adding the keyword "`virtual`" to its declaration:

```
class X
{ public:
    virtual void f() { cout << "X::f() executing\n"; }
};
```

With the rest of the code left unchanged, the output now becomes

```
X::f() executing
Y::f() executing
```

Now the second call `p->f()` invokes `Y::f()` instead of `X::f()`.

This example illustrates *polymorphism*: the same call `p->f()` invokes different functions. The function is selected according to which class of object `p` points to. This is called *dynamic binding* because the association (*i.e.*, binding) of the call to the actual code to be executed is deferred until run time. The rule that the pointer's statically defined type determines which member function gets invoked is overruled by declaring the member function `virtual`.

Here is a more realistic example:

EXAMPLE 12.11 Polymorphism through `virtual` Functions

Here is a `Person` class with a `Student` subclass and a `Professor` subclass:

```
class Person
{ public:
    Person(char* s) { name = new char[strlen(s)+1]; strcpy(name, s); }
    void print() { cout << "My name is " << name << ".\n"; }
  protected:
    char* name;
};

class Student : public Person
{ public:
    Student(char* s, float g) : Person(s), gpa(g) { }
    void print()
    { cout << "My name is " << name << " and my G.P.A. is "
           << gpa << ".\n"; }
  private:
    float gpa;
};

class Professor : public Person
{ public:
    Professor(char* s, int n) : Person(s), publs(n) { }
    void print()
    { cout << "My name is " << name
           << " and I have " << publs << " publications.\n"; }
  private:
    int publs;
};

int main()
{ Person* p;
  Person x("Bob");
  p = &x;
  p->print();
  Student y("Tom", 3.47);
  p = &y;
  p->print();
  Professor z("Ann", 7);
  p = &z;
  p->print();
}
```

```
My name is Bob.
My name is Tom.
My name is Ann.
```

The `print()` function defined in the base class is not `virtual`. So the call `p->print()` always invokes that same base class function `Person::print()` because `p` has type `Person*`. The pointer `p` is *statically bound* to that base class function at compile time.

Now change the base class function `Person::print()` into a `virtual` function, and run the same program:

```
class Person
{ public:
    Person(char* s) { name = new char[strlen(s+1)]; strcpy(name, s); }
    virtual void print() { cout << "My name is " << name << ".\n"; }
  protected:
    char* name;
};
```

```
My name is Bob.
My name is Tom and my G.P.A. is 3.47
My name is Ann and I have 7 publications.
```

Now the pointer `p` is *dynamically bound* to the `print()` function of whatever object it points to. So the first call `p->print()` invokes the base class function `Person::print()`, the second call invokes the derived class function `Student::print()`, and the third call invokes the derived class function `Professor::print()`. We say that the call `p->print()` is *polymorphic* because its meaning changes according to circumstance.

In general, a member function should be declared as virtual whenever it is anticipated that at least some of its subclasses will define their own local version of the function.

12.8 VIRTUAL DESTRUCTORS

Virtual functions are overridden by functions that have the same signature and are defined in subclasses. Since the names of constructors and destructors involve the names of their different classes, it would seem that constructors and destructors could not be declared virtual. That is indeed true for constructors. However, an exception is made for destructors.

Every class has a unique destructor, either defined explicitly within the class definition or implicitly by the compiler. An explicit destructor may be defined to be virtual. The following example illustrates the value in defining a virtual destructor:

EXAMPLE 12.12 Memory Leaks

This program is similar to Example 12.6:

```
class X
{ public:
    X() { p = new int[2];  cout << "X().  "; }
    ~X() { delete [] p;  cout << "~X().\n"; }
  private:
    int* p;
};

class Y : public X
{ public:
    Y() { q = new int[1023];  cout << "Y(): Y::q = " << q << ".  "; }
    ~Y() { delete [] q;  cout << "~Y().  "; }
  private:
    int* q;
};
```

```
int main()
{ for (int i = 0; i < 8; i++)
  { X* r = new Y;
    delete r;
  }
}
```

```
X().  Y(): Y::q = 0x5821c.  ~X().
X().  Y(): Y::q = 0x5921c.  ~X().
X().  Y(): Y::q = 0x5a21c.  ~X().
X().  Y(): Y::q = 0x5b21c.  ~X().
X().  Y(): Y::q = 0x5c21c.  ~X().
X().  Y(): Y::q = 0x5d21c.  ~X().
X().  Y(): Y::q = 0x5e21c.  ~X().
X().  Y(): Y::q = 0x5f21c.  ~X().
```

Each iteration of the `for` loop creates a new dynamic object. As in Example 12.6, the constructors are invoked in top-down sequence: first `X()` and then `Y()`, allocating 4100 bytes of storage (using 4 bytes for each `int`). But since `r` is declared to be a pointer to `X` objects, only the `X` destructor is invoked, deallocating only 8 bytes. So on each iteration, 4092 bytes are lost! This loss is indicated by the actual values of the pointer `Y::q`.

To plug this leak, change the destructor `~X()` into a `virtual` function:

```
class X
{ public:
    X() { p = new int[2];  cout << "X().  "; }
    virtual ~X() { delete [] p;  cout << "~X().\n"; }
  private:
    int* p;
};
```

```
X().  Y(): Y::q = 0x5a220.  ~Y().  ~X().
X().  Y(): Y::q = 0x5a220.  ~Y().  ~X().
X().  Y(): Y::q = 0x5a220.  ~Y().  ~X().
X().  Y(): Y::q = 0x5a220.  ~Y().  ~X().
X().  Y(): Y::q = 0x5a220.  ~Y().  ~X().
X().  Y(): Y::q = 0x5a220.  ~Y().  ~X().
X().  Y(): Y::q = 0x5a220.  ~Y().  ~X().
X().  Y(): Y::q = 0x5a220.  ~Y().  ~X().
```

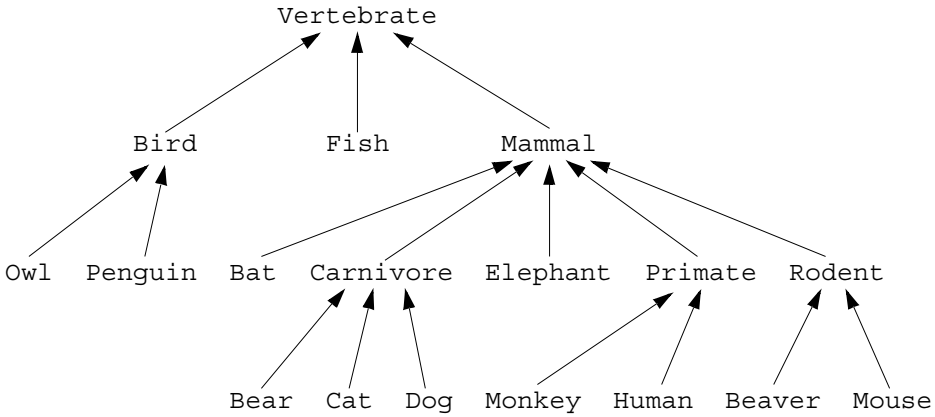
With the base class destructor declared `virtual`, each iteration of the `for` loop calls both destructors, thereby restoring all memory that was allocated by the `new` operator. This allows the same memory to be reused for the pointer `r`.

This example illustrates what is known as a *memory leak*. In a large-scale software system, this could lead to a catastrophe. Moreover, it is a bug that is not easily located. The moral is: declare the base class destructor `virtual` whenever your class hierarchy uses dynamic binding.

As noted earlier, these examples are contrived to illustrate specific features of C++ and are not meant to exemplify typical programming practice.

12.9 ABSTRACT BASE CLASSES

A well-designed object-oriented program will include a hierarchy of classes whose interrelationships can be described by a tree diagram like the one below. The classes at the leaves of this



tree (*e.g.*, `Owl`, `Fish`, `Dog`) would include specific functions that implement the behavior of their respective classes (*e.g.*, `Fish.swim()`, `Owl.fly()`, `Dog.dig()`). However, some of these functions may be common to all the subclasses of a class (*e.g.*, `Vertebrate.eat()`, `Mammal.suckle()`, `Primate.peel()`). Such functions are likely to be declared `virtual` in these base classes, and then overridden in their subclasses for specific implementations.

If a `virtual` function is certain to be overridden in all of its subclasses, then there is no need to implement it at all in its base class. This is done by making the `virtual` function "pure." A *pure virtual member function* is a virtual function that has no implementation in its class. The syntax for specifying a pure virtual member function is to insert the initializer "`=0;`" in place of the functions body, like this:

```
virtual int f() =0;
```

For example, in the `Vertebrate` class above, we might decide that the `eat()` function would be overridden in every one of its subclasses, and thus declare it as a pure virtual member function within its `Vertebrate` base class:

```
class Vertebrate
{ public:
    virtual void eat() =0;    // pure virtual function
};
class Fish : public Vertebrate
{ public:
    void eat(); // implemented specifically for Fish class elsewhere
};
```

The individual classes in a class hierarchy are designated as either "abstract" or "concrete" according to whether they have any pure virtual member functions. An *abstract base class* is a class that has one or more pure virtual member functions. A *concrete derived class* is a class that does not have any pure virtual member functions. In the example above, the `Vertebrate` class is an abstract base class, and the `Fish` class is a concrete derived class. Abstract base classes cannot be instantiated.

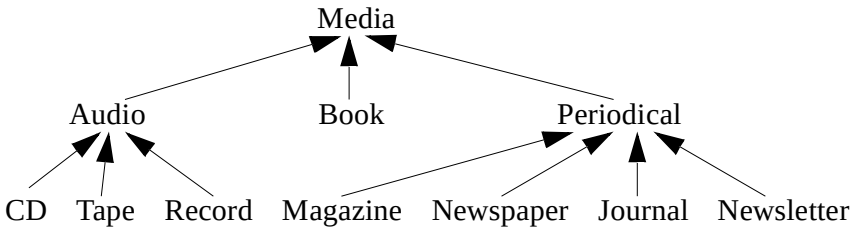
The existence of a pure virtual member function in a class requires that every one of its concrete derived subclasses implement the function. In the example above, if the methods `Vertebrate.eat()`, `Mammal.suckle()`, and `Primate.peel()` were the only pure virtual functions, then the abstract base classes ("ABCs") would be `Vertebrate`, `Mammal`, and `Primate`, and the other 15 classes would be concrete derived classes ("CDCs"). Each of these 15

CDCs would have its own implementation of the `eat()` function, the 11 CDCs of the `Mammal` class would have their own implementation of the `suckle()` function, and the 2 CDCs of the `Primate` class would have their own implementation of the `peel()` function.

An ABC is typically defined during the first stages of the process of developing a class hierarchy. It lays out the framework from which the details are derived in the ABC's subclasses. Its pure virtual functions prescribe a certain uniformity within the hierarchy.

EXAMPLE 12.13 A Hierarchy of `Media` Classes

Here is a hierarchy of classes to represent various media objects:



The primary ABC is the `Media` class:

```
class Media
{ public:
    virtual void print() =0;
    virtual char* id() =0;
  protected:
    string title;
};
```

It has two pure virtual functions and one data member.

Here is the concrete `Book` subclass:

```
class Book : Media
{ public:
    Book(string a="", string t="", string p="", string i="")
     : author(a), publisher(p), isbn(i) { title = t; }
    void print() { cout << title << " by " << author << endl; }
    char* id() { return isbn; }
  private:
    string author, publisher, isbn;
};
```

It implements the two virtual functions using its own member data.

Here is the concrete `CD` subclass:

```
class CD : Media
{ public:
    CD(string t="", string c="", string m="", string n="")
     : composer(c), make(m), number(n) { title = t; }
    void print() { cout << title << ", " << composer << endl; }
    char* id() { return make + " " + number; }
  private:
    string composer, make, number;
};
```

The `CD` class will also be a CDC of the `Audio` class, which will be another ABC. So when the `Audio` class is defined, its pure virtual functions will also have to be implemented in this `CD` class.

Here is the concrete `Magazine` subclass:

```
class Magazine : Media
{ public:
    Magazine(string t="", string i="", int v=0, int n=0)
     : issn(i), volume(v), number(n) { title = t; }
    void print()
    { cout << title << " Magazine, Vol. "
           << volume << ", No." << number << endl;
    }
    char* id() { return issn; };
  private:
    string issn, publisher;
    int volume, number;
};
```

The `Magazine` class will also be a CDC of the `Periodical` class, which will be another ABC. So when the `Periodical` class is defined, its pure virtual functions will also have to be implemented in this `Magazine` class.

Here is a test driver for the four classes defined above:

```
int main()
{ Book book("Bjarne Stroustrup", "The C++ Programming Language",
    "Addison-Wesley", "0-201-53992-6");
  Magazine magazine("TIME", "0040-781X", 145, 23);
  CD cd("BACH CANTATAS", "Johann Sebastian Bach",
    "ARCHIV", "D120541");
  book.print();
  cout << "\tid: " << book.id() << endl;
  magazine.print();
  cout << "\tid: " << magazine.id() << endl;
  cd.print();
  cout << "\tid: " << cd.id() << endl;
}
```

```
The C++ Programming Language by Bjarne Stroustrup
        id: 0-201-53992-6
TIME Magazine, Vol. 145, No.23
        id: 0040-781X
BACH CANTATAS, Johann Sebastian Bach
        id: ARCHIV D120541
```

Note that all the calls to the `print()` and `id()` functions are independent of their class implementations. So the implementations of these functions could be changed without making any changes to the program. For example, we could change the `Book::print()` function to

```
void print()
{ cout << title << " by " << author
       << ".\nPublished by " << publisher << ".\n";
}
```

and obtain the output

```
The C++ Programming Language by Bjarne Stroustrup.
Published by Addison-Wesley.
```

without any changes to the program.

12.10 OBJECT-ORIENTED PROGRAMMING

Object-oriented programming refers to the use of derived classes and virtual functions. A thorough treatment of object-oriented programming is beyond the scope of this book. See the books **[Bergin]**, **[Perry]**, and **[Wang]** listed in Appendix H for a more thorough treatment.

Suppose that you have three televisions, each equipped with its own video cassette recorder. Like most VCRs, yours are loaded with features and have confusing user manuals. Your three VCRs are all different, requiring different and complex operations to use them. Then one day you see on the shelf of your local electronics store a simple remote controller that can operate all kinds of VCRs. For example, it has a single "RECORD" button that causes whatever VCR it is pointed at to record the current TV program on the current tape. This marvelous device represents the essence of object-oriented programming ("OOP"): conceptual simplification of diverse *implementations* by means of a single *interface*. In this example, the interface is the remote controller, and the implementations are the (hidden) operations within the controller and the individual VCRs that carry out the requested functions ("RECORD", "STOP", "PLAY", *etc*.). The interface is the abstract base class below:

```
class VCR
{ public:
    virtual void on() =0;
    virtual void off() =0;
    virtual void record() =0;
    virtual void stop() =0;
    virtual void play() =0;
};
```

and the implementations are the concrete derived classes below:

```
class Panasonic : public VCR {
public:
void on();
void off();
void record();
void stop();
void play();
};

class Sony : public VCR {
public:
void on();
void off();
void record();
void stop();
void play();
};

class Mitsubishi : public VCR {
public:
void on();
void off();
void record();
void stop();
```

```
void play();
};
```

One important advantage of object-oriented systems is *extensibility*. This refers to the ease with which the system can be extended. In the example above, the VCR controller would be called "extensible" if it automatically works the same way on new VCRs that we might add in the future. The controller should not have to be modified when we extend our collection of VCRs, adding a Toshiba or replacing the Sony with an RCA.

In the object-oriented programming, we imagine two distinct points of view of the system: the view of the consumer (*i.e.*, the client or user) that shows what is to be done, and the view of the manufacturer (*i.e.*, the server or implementor) that shows how it is to be done. The consumer sees only the abstract base class, while the manufacturer sees the concrete derived classes. The customer's actions are generally called *operations*, as opposed to the manufacturer's implementations of these actions which are called generally *methods*. In C++, the actions are the pure virtual functions, and the methods are their implementations in the concrete derived classes. In this context, the abstract base class (the user's view) is called the system *interface*, and the concrete derived classes (the implementor's view) are called the system *implementation*:

The Two Views in an Object-Oriented Program

The System Interface	The System Implementation
(user's view)	(implementor's view)
shows what is done	shows how it is done
abstract base class	concrete derived classes
operations	methods
pure virtual functions	functions

This dichotomy is most effective when we use pointers to objects, as in Example 12.13. Then we can exploit dynamic binding to make the system interface even more independent from the system implementation. Extensibility is facilitated by the fact that only the newly added methods need to be compiled.

Review Questions

12.1 What is the difference between composition and inheritance?
12.2 What is the difference between `protected` and `private` members?
12.3 How do the default constructors and destructors behave in an inheritance hierarchy?
12.4 What is a `virtual` member function?
12.5 What is a pure `virtual` member function?
12.6 What is a memory leak?
12.7 How can virtual destructors plug a memory leak?
12.8 What is an abstract base class?
12.9 What is a concrete derived class?
12.10 What is the difference between static binding and dynamic binding?
12.11 What is polymorphism?
12.12 How does polymorphism promote extensibility?

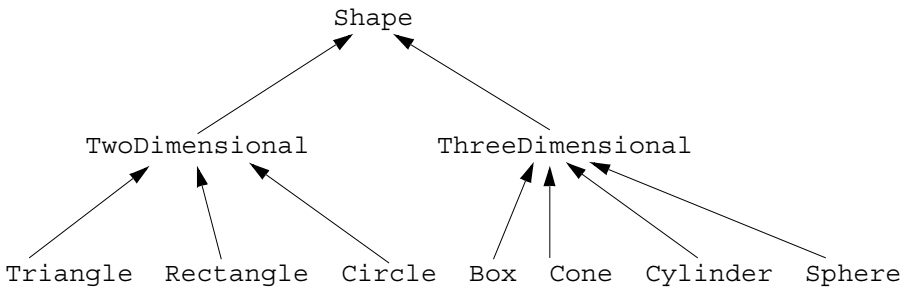
12.13 What is wrong with the following definitions:

```
class X
{ protected:
    int a;
};

class Y : public X
{ public:
    void set(X x, int c) { x.a = c; }
};
```

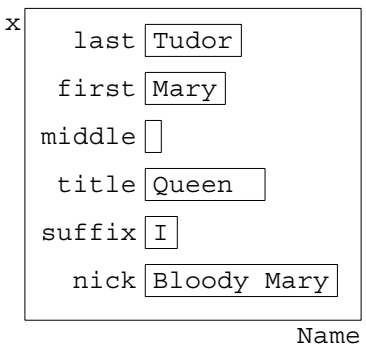
Problems

12.1 Implement a `Card` class, a composite `Hand` class, and a composite `Deck` class for playing poker.

12.2 Implement the following class hierarchy:



12.3 Define and test a `Name` class whose objects looks like the diagram at the top of the next page. Then modify the `Person` class so that `name` has type `Name` instead of type `string`.



Answers to Review Questions

12.1 Composition of classes refers to using one class to declare members of another class. Inheritance refers to deriving a subclass from a base class.

12.2 A `private` member is inaccessible from anywhere outside its class definition. A `protected` member is inaccessible from anywhere outside its class definition, with the exception that it is accessible from the definitions of derived classes.

12.3 In an inheritance hierarchy, each default constructor invokes its parent's default constructor before it executes itself, and each destructor invokes its parent's destructor after it executes itself. The effect is that all the parent default constructors execute in top-down order, and all the parent destructors execute in bottom-up order.

12.4 A `virtual` member function is a member function that can be overridden in a subclass.

12.5 A pure `virtual` function is a `virtual` member function that cannot be called directly; only its overridden functions in derived classes can be called. A pure `virtual` function is identified by the initializer `=0` at the end of its declaration.

12.6 A *memory leak* is the loss of access to memory in a program due to the wrong destructor being invoked. See Example 12.12 on page 285.

12.7 By declaring a base class destructor `virtual`, memory leaks as in Example 12.12 on page 285 can be prevented because after it is invoked its indicated subclass destructor(s) will also be invoked.

12.8 An abstract base class is a base class which includes at least one pure `virtual` function. Abstract base classes cannot be instantiated.

12.9 A concrete derived class is a subclass of an abstract base class that can be instantiated; *i.e.*, one which contains no pure `virtual` functions.

12.10 Static binding refers to the linking of a member function call to the function itself during compile time, in contrast to dynamic binding which postpones that linking until run time. Dynamic is possible in C++ by using virtual functions and by passing pointers to objects.

12.11 *Polymorphism* refers to the run-time binding that occurs when pointers to objects are used in classes that have `virtual` functions. The expressions `p->f()` will invoke the functions `f()` that is defined in the object to which `p` points. However, that object could belong to any one of a series of subclasses, and the selection of subclass could be made at run time. If the base-class function is `virtual`, then the selection (the "binding") of which `f()` to invoke is made at run time. So the expression `p->f()` can take "many forms."

12.12 Polymorphism promotes extensibility by allowing new subclasses and methods to be added to a class hierarchy without having to modify application programs that already use the hierarchy's interface.

12.13 The `protected` data member `a` can be accessed from the derived `Y` only if it is the member of the current object (*i.e.* only if it is `this->a`). `Y` cannot access `x.a` for any other object `x`.

Solutions to Problems

12.1 First we implement a `Card` class:

```
enum Rank {TWO=2, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN
           JACK, QUEEN, KING, ACE};
enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES };

class Card
{   friend class Hand;
    friend class Deck;
    friend ostream& operator<<(ostream&, const Card&);
  public:
    char rank() { return rank_; }
    char suit() { return suit_; }
  private:
    Card() { };
    Card(Rank rank, Suit suit) : rank_(rank), suit_(suit) { };
```

```
    Card(const Card& c) : rank_(c.rank_), suit_(c.suit_) { };
    ~Card() { };
    Rank rank_;
    Suit suit_;
};
```

This class uses enumeration types for a card's 13 possible ranks and 4 possible suits. Anticipating the implementation of `Hand` and `Deck` classes, we declare them here to be `friend` classes to the `Card` class. This will allow them to access the `private` members of the `Card` class. Notice that all three constructors and the destructor are declared to be `private`. This will prevent any cards to be created or destroyed except by the `Card`'s two `friend` classes.

Here is the implementation of the overloaded insertion operator `<<` for cards:

```
ostream& operator<<(ostream& ostr, const Card& card)
{ switch (card.rank_)
  { case TWO   : ostr << "two of ";    break;
    case THREE : ostr << "three of ";  break;
    case FOUR  : ostr << "four of ";   break;
    case FIVE  : ostr << "five of ";   break;
    case SIX   : ostr << "six of ";    break;
    case SEVEN : ostr << "seven of ";  break;
    case EIGHT : ostr << "eight of ";  break;
    case NINE  : ostr << "nine of ";   break;
    case TEN   : ostr << "ten of ";    break;
    case JACK  : ostr << "jack of ";   break;
    case QUEEN : ostr << "queen of ";  break;
    case KING  : ostr << "king of ";   break;
    case ACE   : ostr << "ace of ";    break;
  }
  switch (card.suit_)
  { case CLUBS    : ostr << "clubs";     break;
    case DIAMONDS : ostr << "diamonds";  break;
    case HEARTS   : ostr << "hearts";    break;
    case SPADES   : ostr << "spades";    break;
  }
  return ostr;
}
```

Here is the implementation of the `Hand` class:

```
#include "Card.h"
class Hand
{   friend class Deck;
  public:
    Hand(unsigned n=5) : size(n) { cards = new Card[n]; }
    ~Hand() { delete [] cards; }
    void display();
    int isPair();
    int isTwoPair();
    int isThreeOfKind();
    int isStraight();
    int isFlush();
    int isFullHouse();
    int isFourOfKind();
    int isStraightFlush();
```

```
  private:
    unsigned size;
    Card* cards;
    void sort();
};
```

It uses an array to store the cards in the hand. The `sort()` function is a private utility that is called by the `Deck` class after dealing the hand. It can be implemented by any simple sort algorithm such as the Bubble Sort. The `display()` function is also straightforward, using the insertion operator `<<` that is overloaded in the `Card` class.

The eight boolean functions that identify special poker hands are not so straightforward. Here is the implementation of the `isThreeOfKind()` function:

```
int Hand::isThreeOfKind()
{ if (cards[0].rank_ == cards[1].rank_
      && cards[1].rank_ == cards[2].rank_
      && cards[2].rank_ != cards[3].rank_
      && cards[3].rank_ != cards[4].rank_) return 1;
  if (cards[0].rank_ != cards[1].rank_
      && cards[1].rank_ == cards[2].rank_
      && cards[2].rank_ == cards[3].rank_
      && cards[3].rank_ != cards[4].rank_) return 1;
  if (cards[0].rank_ != cards[1].rank_
      && cards[1].rank_ != cards[2].rank_
      && cards[2].rank_ == cards[3].rank_
      && cards[3].rank_ == cards[4].rank_) return 1;
  return 0;
}
```

Since the hand is sorted by `rank_`, the only way there could be three cards of the same rank with the other two cards of different rank would be one of the three forms: *xxxyz*, *xyyyz*, or *xyzzz*. If any of these three forms is identified, then the function returns 1. If not it returns 0.

The `isPair()` function, the `isTwoPair()` function, the `isFullHouse()` function, and the `isFourOfKind()` function are similar to the `isThreeOfKind()` function.

The `isStraight()` function, the `isFlush()` function, and the `isStraightFlush()` function are also tricky. Here is the `isFlush()` function:

```
int Hand::isFlush()
{ for (int i = 1; i < size; i++)
    if (cards[i].suit_ != cards[0].suit_) return 0;
  return 1;
}
```

This compares the `suit_` of each of the second through fifth cards (`card[1]` through `card[4]`). If any of these are not the same, then we know immediately that the hand is not a flush and can return 0. If the loop terminates naturally, then all four pairs match and 1 is returned.

Here is the `Deck` class:

```
#include "Random.h"
#include "Hand.h"
class Deck
{ public:
    Deck();
    void shuffle();
    void deal(Hand&, unsigned =5);
  private:
    unsigned top;
    Card cards[52];
```

```
    Random random;
};
```

It uses the `Random` class in its `shuffle()` function. Note that the `random` object is declared as a private member since it is used only by another member function:

```
void Deck::deal(Hand& hand, unsigned size)
{ for (int i = 0; i < size; i++)
    hand.cards[i] = cards[top++];
  hand.sort();
}
```

The `top` member always locates the top of the deck; *i.e.*, the next card to be dealt. So the `deal()` function copies the top five cards off the deck into the `hand`'s `cards` array. Then it sorts the hand.

The `Deck`'s constructor initializes all 52 cards in the deck, in the order `two of clubs`, `three of clubs`, `four of clubs`, ..., `ace of spades`:

```
Deck::Deck()
{ for (int i = 0; i < 52; i++)
  { cards[i].rank_ = Rank(i%13);
    cards[i].suit_ = Suit(i%4);
  }
top = 0;
}
```

So if hands are dealt without shuffling first, the first hand would be the straight flush of two through six of clubs.

Finally, here is the `shuffle()` function:

```
void Deck::shuffle()
{ for (int i = 0; i < 52; i++)      // do 52 random swaps
  { int j = random.integer(0, 51);
    Card c = cards[i];
    cards[i] = cards[j];
    cards[j] = c;
  }
  top = 0;
}
```

It swaps the cards in each of the 52 elements with the card in a randomly selected element of the deck's `cards` array.

12.2 Here are the abstract base classes:

```
const double PI=3.14159265358979;
class Shape
{ public:
    virtual void print() = 0;
    virtual float area() = 0;
};
class TwoDimensional : public Shape
{ public:
    virtual float perimeter() = 0;
};
class ThreeDimensional : public Shape
{ public:
    virtual float volume() = 0;
};
```

Note that the `print()` function and the `area()` function prototypes are the same for all classes in this hierarchy, so their interfaces (pure `virtual` functions) are placed in the `Shape` base class. But only two-dimensional shapes have perimeters, and only three-dimensional shapes have volumes, so their interfaces are placed in the appropriate second-level ABCs.

Here are two of the seven concrete derived classes:

```
class Circle : public TwoDimensional
{ public:
    Circle(float r) : radius(r) { }
    void print() { cout << "Shape is a circle.\n"; }
    float perimeter() { return 2*PI*radius; }
    float area() { return PI*radius*radius; }
  private:
    float radius;
};
class Cone : public ThreeDimensional
{ public:
    Cone(float r, float h) : radius(r), height(h) { }
    void print();
    float area();
    float volume() { return PI*radius*radius*height/3; }
  private:
    float radius, height;
};
void Cone::print()
{ cout << "Cone: radius = " << radius << ", height = "
     << height << endl;
}
float Cone::area()
{ float s = sqrt(radius*radius + height*height);
  return  PI*radius*(radius + s);
}
```

The other five concrete derived classes are similar.

12.3 Here is the interface for the `Name` class:

```
class Name
{   friend ostream& operator<<(ostream&, const Name&);
    friend istream& operator>>(istream&, Name&);
  public:
    Name(char*, char*, char*, char*, char*, char*);
    string last() { return last_; }
    string first() { return first_; }
    string middle() { return middle_; }
    string title() { return title_; }
    string suffix() { return suffix_; }
    string nick() { return nick_; }
    void last(string s) { last_ = s; }
    void first(string s) { first_ = s; }
    void middle(string s) { middle_ = s; }
    void title(string s) { title_ = s; }
    void suffix(string s) { suffix_ = s; }
    void nick(string s) { nick_ = s; }
    void dump();
  private:
    string last_, first_, middle_, title_, suffix_, nick_;
};
```

Here is an implementation for the `Name` class:

```
Name::Name(char* last="", char* first="", char* middle="",
           char* title="", char* suffix="", char* nick="")
  : last_(last), first_(first), middle_(middle), title_(title),
         suffix_(suffix), nick_(nick) { }
void Name::dump()
{ cout << "\t   Last Name: " << last_   << endl;
  cout << "\t  First Name: " << first_  << endl;
  cout << "\tMiddle Names: " << middle_ << endl;
  cout << "\t       Title: " << title_  << endl;
  cout << "\t      Suffix: " << suffix_ << endl;
  cout << "\t    Nickname: " << nick_   << endl;
}
ostream& operator<<(ostream& out, const Name& x)
{ if (x.title_ != "") out << x.title_ << " ";
  out << x.first_ << " ";
  if (x.middle_ != "") out << x.middle_ << " ";
  out << x.last_;
  if (x.suffix_ != "") out << " " << x.suffix_;
  if (x.nick_ != "") out << ", \"" << x.nick_ << "\"";
  return out;
}
istream& operator>>(istream& in, Name& x)
{ char buffer[80];
  in.getline(buffer, 80, '|');
  x.last_ = buffer;
  in.getline(buffer, 80, '|');
  x.first_ = buffer;
  in.getline(buffer, 80, '|');
  x.middle_ = buffer;
  in.getline(buffer, 80, '|');
  x.title_ = buffer;
  in.getline(buffer, 80, '|');
  x.suffix_ = buffer;
  in.getline(buffer, 80);
  x.nick_ = buffer;
  return in;
}
```

Finally, here is the modified `Person` class:

```
#include "Date.h"
#include "Name.h"
class Person
{ public:
    Person(char* n="", int s=0, char* nat="U.S.A.")
     : name(n), sex(s), nationality(nat) { }
    void setDOB(int m, int d, int y) { dob.setDate(m, d, y); }
    void setDOD(int m, int d, int y) { dod.setDate(m, d, y); }
    void printName() { cout << name; }
    void printNationality() { cout << nationality; }
    void printDOB() { cout << dob; }
    void printDOD() { cout << dod; }
```

```
  protected:
    Name name;
    Date dob, dod;                  // date of birth, date of death
    int sex;                        // 0 = female, 1 = male
    string nationality;
};
```

Here is a test driver for the `Name` class, with test run:

```
#include <iostream.h>
#include "Name.h"

int main()
{ Name x("Bach", "Johann", "Sebastian");
  cout << x << endl;
  x.dump();
  x.last("Clinton");
  x.first("William");
  x.middle("Jefferson");
  x.title("President");
  x.nick("Bill");
  cout << x << endl;
  x.dump();
  cin >> x;
  cout << x << endl;
  cout << "x.last   = [" << x.last()   << "]\n";
  cout << "x.first  = [" << x.first()  << "]\n";
  cout << "x.middle = [" << x.middle() << "]\n";
  cout << "x.title  = [" << x.title()  << "]\n";
  cout << "x.suffix = [" << x.suffix() << "]\n";
  cout << "x.nick   = [" << x.nick()   << "]\n";
}
```

```
Johann Sebastian Bach
           Last Name: Bach
          First Name: Johann
       Middle Names: Sebastian
              Title:
             Suffix:
           Nickname:
President William Jefferson Clinton, "Bill"
           Last Name: Clinton
          First Name: William
       Middle Names: Jefferson
              Title: President
             Suffix:
           Nickname: Bill
Tudor|Mary||Queen|I|Bloody Mary
Queen Mary Tudor I, "Bloody Mary"
x.last   = [Tudor]
x.first  = [Mary]
x.middle = []
x.title  = [Queen]
x.suffix = [I]
x.nick   = [Bloody Mary]
```

Chapter 13

Templates and Iterators

13.1 INTRODUCTION

A *template* is an abstract recipe for producing concrete code. Templates can be used to produce functions and classes. The compiler uses the template to generate the code for various functions or classes, the way you would use a cookie cutter to generate cookies from various types of dough. The actual functions or classes generated by the template are called *instances* of that template.

The same template can be used to generate many different instances. This is done by means of *template parameters* which work much the same way for templates as ordinary parameters work for ordinary functions. But whereas ordinary parameters are place holders for objects, template parameters are place holders for types and classes.

The facility that C++ provides for instantiating templates is one of its major features and one that distinguishes it from most other programming languages. As a mechanism for automatic code generation, it allows for substantial improvements in programming efficiency.

13.2 FUNCTION TEMPLATES

In many sorting algorithms, we need to interchange a pair of elements. This simple task is often done by a separate function. For example, the following function swaps integers:

```
void swap(int& m, int& n)
{ int temp = m;
  m = n;
  n = temp;
}
```

If, however, we were sorting `string` objects, then we would need a different function:

```
void swap(string& s1, string& s2)
{ string temp = s1;
  s1 = s2;
  s2 = temp;
}
```

These two functions do the same thing. Their only difference is the type of objects they swap. We can avoid this redundancy by replacing both functions with a *function template*:

EXAMPLE 13.1 The `swap` Function Template

```
template <class T>
void swap(T& x, T& y)
{ T temp = x;
  x = y;
  y = temp;
}
```

The symbol `T` is called a *type parameter*. It is simply a place holder that is replaced by an actual type or class when the function is invoked.

A function template is declared the same way as an ordinary function, except that it is preceded by the specification

```
template <class T>
```

and the type parameter `T` may be used in place of ordinary types within the function definition. The use of the word `class` here means "any type." More generally, a template may have several type parameters, specified like this:

```
template <class T, class U, class V>
```

Function templates are called the same way ordinary functions are called:

```
int m = 22, n = 66;
swap(m, n);
string s1 = "John Adams", s2 = "James Madison";
swap(s1, s2);
Rational x(22/7), y(-3);
swap(x, y);
```

For each call, the compiler generates the complete function, replacing the type parameter with the type or class to which the arguments belong. So the call `swap(m,n)` generates the integer `swap` function shown above, and the call `swap(s1, s2)` generates the `swap` function for `string` objects.

Function templates are a direct generalization of function overloading. We could have written several overloaded versions of the `swap` function, one for each type that we thought we might need. The single `swap` function template serves the same purpose. But it is an improvement in two ways. It only has to be written once to cover all the different types that might be used with it. And we don't have to decide in advance which types we will use with it; any type or class can be substituted for the type parameter `T`. Function templates share source code among structurally similar families of functions.

EXAMPLE 13.2 The Bubble Sort Template

This is the Bubble Sort (Example 6.13 on page 134) and a print function for vectors of any base type.

```
template<class T>
void sort(T* v, int n)
{ for (int i = 1; i < n; i++)
    for (int j = 0; j < n-i; j++)
      if (v[j] > v[j+1]) swap(v[j], v[j+1]);
}

template<class T>
void print(T* v, int n)
{ for (int i = 0; i < n; i++)
  cout << " " << v[i];
  cout << endl;
}

int main()
{ short a[9] = {55, 33, 88, 11, 44, 99, 77, 22, 66};
```

```
    print(a,9);
    sort(a,9);
    print(a,9);
    string s[7] = {"Tom", "Hal", "Dan", "Bob", "Sue", "Ann", "Gus"};
    print(s,7);
    sort(s,7);
    print(s,7);
}
```

Here, both `sort()` and `print()` are function templates. The type parameter `T` is replaced by the type `short` in the first calls and by the class `string` in the second calls.

A function template works like an outline. The compiler uses the template to generate each version of the function that is needed. In the previous example, the compiler produces two versions of the `sort()` function and two versions of the `print()` function, one each for the type `short` and one each for the class `string`. The individual versions are called *instances* of the function template, and the process of producing them is called *instantiating* the template. A function that is an instance of a template is also called a *template function*. Using templates is a form of automatic code generation; it allows the programmer to defer more of the work to the compiler.

13.3 CLASS TEMPLATES

A *class template* works the same way as a function template except that it generates classes instead of functions. The general syntax is

```
template<class T,...> class X { ... };
```

As with function templates, a class template may have several template parameters. Moreover, some of them can be ordinary non-type parameters:

```
template<class T, int n, class U> class X { ... };
```

Of course, since templates are instantiated at compile time, values passed to non-type parameters must be constants:

```
template<class T, int n>
class X {};

int main()
{ X<float, 22> x1;    // OK
  const int n = 44;
  X<char, n> x2;      // OK
  int m = 66;
  X<short, m> x3;     // ERROR: m must be constant
}
```

Class templates are sometimes called *parameterized types*.

The member functions of a class template are themselves function templates with the same template header as their class. For example, the function `f()` declared in the class template

```
template<class T>
class X
{ T square(T t) { return t*t; }
};
```

is handled the same way that the following template function would be handled:

```
template<class T>
T square(T t) { return t*t; }
```

It is instantiated by the compiler, replacing the template parameter `T` with the type passed to it. Thus, the declaration

```
X<short> x;
```

generates the class and object

```
class X_short
{ short square(short t) { return t*t; }
};
X_short x;
```

except that your compiler may use some name other than `X_short` for the class.

EXAMPLE 13.3 A `Stack` Class Template

A *stack* is a simple data structure that simulates an ordinary stack of objects of the same type (*e.g.*, a stack of dishes) with the restrictions that an object can be inserted into the stack only at the top and an object can be removed from the stack only at the top. In other words, a stack is a linear data structure with access at only one end. A stack class abstracts this notion by hiding the implementation of the data structure, allowing access only by means of public functions that simulate the limited operations described above.

Here is a class template for generating `Stack` classes:

```
template<class T>
class Stack
{ public:
    Stack(int s = 100) : size(s), top(-1) { data = new T[size]; }
    ~Stack() { delete [] data; }
    void push(const T& x) { data[++top] = x; }
    T pop() { return data[top--]; }
    int isEmpty() const { return top == -1; }
    int isFull() const { return top == size - 1; }
  private:
    int size;
    int top;
    T* data;
};
```

This definition uses an array `data` to implement a stack. The constructor initializes the `size` of the array, allocates that many elements of type `T` to the array, and initializes its `top` pointer to –1. The value of `top` is always one less than the number of elements on the stack, and except when the stack is empty, `top` is the index in the array of the top element on the stack. The `push()` function inserts an object onto the stack, and the `pop()` function removes an object from the stack. A stack `isEmpty()` when its `top` has the value `-1`, and it `isFull()` when its `top` pointer has the value `size - 1`.

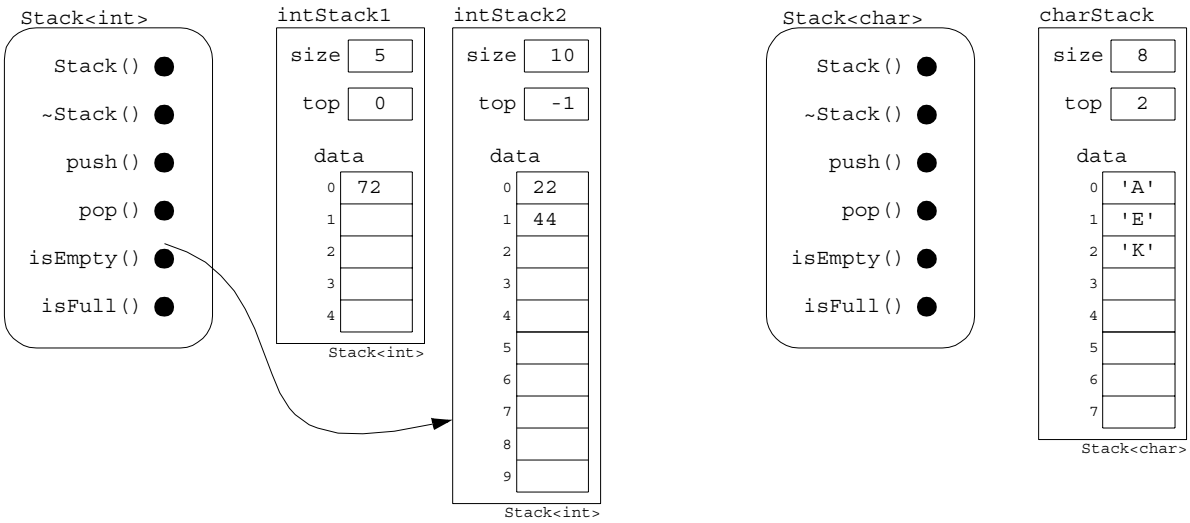
Here is a program to test the `Stack` template:

```
int main()
{ Stack<int> intStack1(5);
  Stack<int> intStack2(10);
  Stack<char> charStack(8);
  intStack1.push(77);
  charStack.push('A');
  intStack2.push(22);
  charStack.push('E');
```

```
    charStack.push('K');
    intStack2.push(44);
    cout << intStack2.pop() << endl;
    cout << intStack2.pop() << endl;
    if (intStack2.isEmpty()) cout << "intStack2 is empty.\n";
}
44
22
intStack2 is empty.
```

The template has one parameter `T` which will be used to specify the type of the objects stored on the stack. The first line declares `intStack1` to be a stack that can hold up to 5 `int`s. Similarly, `intStack2` is a stack that can hold up to 10 `int`s, and `charStack` is a stack that can hold up to 8 `char`s.

After pushing and popping objects on and off the stacks, the last line calls the `isEmpty()` function for `intStack2`. At that instant, the two `Stack` classes and three `Stack` objects look like this:



The call `intStack2.isEmpty()` returns 1 (*i.e.*, "true") because `intStack2.top` has the value –1 at that moment.

Note that there are two instances of the `Stack` class template: `Stack<int>` and `Stack<char>`. These are distinct classes, each generated by the compiler. Each class has its own six distinct member functions. For example, the two functions `Stack<int>::pop()` and `Stack<char>::pop()` are different: one returns an `int` and the other returns a `char`.

13.4 CONTAINER CLASSES

A *container* is simply an object that contains other objects. Ordinary arrays and stacks are containers. A *container class* is a class whose instances are containers. The `Stack<int>` and `Stack<char>` classes in Example 13.3 are container classes. Class templates are natural mechanisms for generating container classes because the contained objects' type can be specified using a template parameter.

A container is called *homogeneous* if all of its objects have the same type; otherwise it is called a *heterogeneous container*. Stacks, arrays, *etc.*, are homogeneous containers.

A *vector* is an indexed sequence of objects of the same type. The word is borrowed from mathematics where it originally referred to a three-dimensional point $\mathbf{x} = (x_1, x_2, x_3)$. Of course, that is just an array of 3 real numbers. The subscripts on the components are the same as the index values on the array, except that in C++ those values must begin with 0. Since subscripts cannot be written in source code, we use the bracket notation `[ ]` instead. So `x[0]` represents x_1, `x[1]` represents x_2, and `x[2]` represents x_3.

EXAMPLE 13.4 A `Vector` Class Template

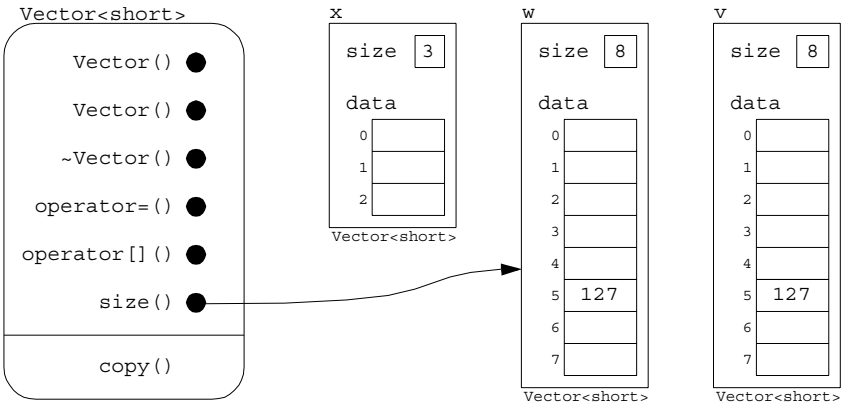
```
template<class T>
class Vector
{ public:
    Vector(unsigned n=8) : size(n), data(new T[size]) { }
    Vector(const Vector<T>& v) : size(v.size), data(new T[size])
    { copy(v); }
    ~Vector() { delete [] data; }
    Vector<T>& operator=(const Vector<T>&);
    T& operator[](unsigned i) const { return data[i]; }
    unsigned size() { return size; }
  protected:
    T* data;
    unsigned size;
    void copy(const Vector<T>&);
};

template<class T>
Vector<T>& Vector<T>::operator=(const Vector<T>& v)
{ size = v.size;
  data = new T[size];
  copy(v);
  return *this;
}

template<class T>
void Vector<T>::copy(const Vector<T>& v)
{ unsigned min_size = (size < v.size ? size : v.size);
  for (int i = 0; i < min_size; i++)
    data[i] = v.data[i];
}
```

Note that each implementation of a member function must be preceded by the same template designator that precedes the class declaration: `template<class T>`.

This template would allow the following code:

```
Vector<short> v;
v[5] = 127;
Vector<short> w = v, x(3);
cout << w.size();
```

Here `v` and `w` are both `Vector` objects with 8 elements of type `short`, and `x` is a `Vector` object with 3 elements of type `short`. The class and its three objects can be visualized from the diagram shown at the top of the next page. It shows the situation at the moment when the member function `w.size()` is executing. The class `Vector<short>` has been instantiated from the template, and three objects `v`,



w, and x have been instantiated from the class. Note that the copy() function is a protected utility function, so it cannot be invoked by any of the class instances.

Note that the expression v[5] is used on the left side of an assignment, even though this expression is a function call. This is possible because the subscript operator returns a reference to a Vector<T>, making it an *lvalue*.

Class templates are also called *parametrized types* because they act like types to which parameters can be passed. For example, the object b above has type Vector<double>, so the element type double acts like a parameter to the template Vector<T>.

13.5 SUBCLASS TEMPLATES

Inheritance works with class templates the same way as with ordinary class inheritance. To illustrate this technique, we will define a subclass template of the Vector class template defined in Example 13.4.

EXAMPLE 13.5 A Subclass Template for Vectors

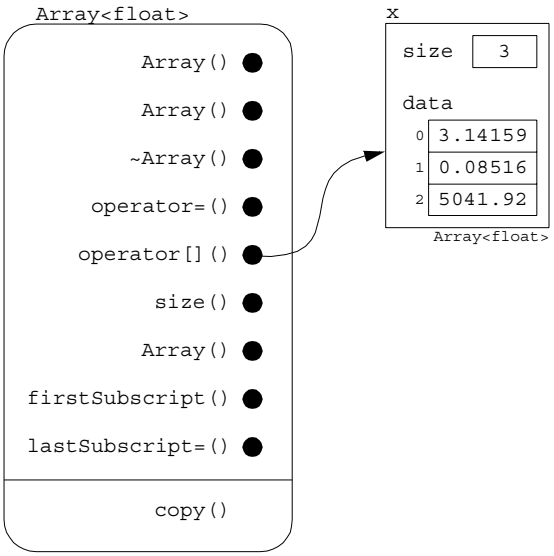
One problem with the Vector class as implemented by the template in Example 13.4 is that it requires *zero-based indexing*; *i.e.*, all subscripts must begin with 0. This is a requirement of the C++ language itself. Some other programming languages allow array indexes to begin with 1 or any other integer. We can add this useful feature to our Vector class template by declaring a subclass template:

```
template <class T>
class Array : public Vector<T> {
public:
    Array(int i, int j) : i0(i), Vector<T>(j-i+1) { }
    Array(const Array<T>& v) : i0(v.i0), Vector<T>(v) { }
    T& operator[](int i) const { return Vector<T>::operator[](i-i0); }
    int firstSubscript() const { return i0; }
    int lastSubscript() const { return i0+size-1; }
protected:
    int i0;
};
```

This Array class template inherits all the functionality of the Vector class template and also allows subscripts to begin with any integer. The first member function listed is a new constructor that allows the user to designate the first and last values of the subscript when the object is declared. The

second function is the copy constructor for this subclass, and the third function is the overloaded subscript operator. The last two functions simply return the first and last values of the subscript range.

Note how the two `Array` constructors invoke the corresponding `Vector` constructors, and how the `Array` subscript operator invokes the `Vector` subscript operator.



Here is a test driver and a sample run:

```
#include <iostream.h>
#include "Array.h"

int main()
{ Array<float> x(1,3);
  x[1] = 3.14159;
  x[2] = 0.08516;
  x[3] = 5041.92;
  cout << "x.size() = " << x.size() << endl;
  cout << "x.firstSubscript() = " << x.firstSubscript() << endl;
  cout << "x.lastSubscript() = " << x.lastSubscript() << endl;
  for (int i = 1; i <= 3; i++)
    cout << "x[" << i << "] = " << x[i] << endl;
}
```

```
x.size() = 3
x.firstSubscript() = 1
x.lastSubscript() = 3
x[1] = 3.14159
x[2] = 0.08516
x[3] = 5041.92
```

13.6 PASSING TEMPLATE CLASSES TO TEMPLATE PARAMETERS

We have already seen examples of passing a class to a template parameter:

```
Stack<Rational> s;    // a stack of Rational objects
Vector<string> a;     // a vector of string objects
```

Since template classes work like ordinary classes, we can also pass them to template parameters:

```
Stack<Vector<int>> s;      // a stack of Vector objects
Array<Stack<Rational>> a;  // an array of Stack objects
```

The next example shows how this "template nesting" can facilitate software reuse.

EXAMPLE 13.6 A `Matrix` Class Template

A *matrix* is essentially a two-dimensional vector. For example, a "2-by-3 matrix" is a table with 2 rows and 3 columns:

$$\begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix}$$

We can think of this as a 2-element vector, each of whose elements is a 3-element vector:

$$\begin{bmatrix} \begin{bmatrix} a & b & c \end{bmatrix} & \begin{bmatrix} d & e & f \end{bmatrix} \end{bmatrix}$$

The advantage of this point of view is that it allows us to reuse our `Vector` class template to define a new `Matrix` class template.

To facilitate the dynamic allocation of memory, we define a matrix as a vector of pointers to vectors:

```
Vector<Vector<T>*>
```

We are passing a class template pointer to the template parameter indicated by the outside angle brackets. This really means that when the `Matrix` class template is instantiated, the instances of the resulting class will contain vectors of pointers to vectors.

```
template<class T>
class Matrix
{ public:
    Matrix(unsigned r=1, unsigned c=1) : row(r)
      { for (int i=0; i<r; i++) row[i] = new Vector<T>(c); }
    ~Matrix() { for (int i=0; i<row.size(); i++) delete row[i]; }
    Vector<T>& operator[](unsigned i) const { return *row[i]; }
    unsigned rows() { return row.size(); }
    unsigned columns() { return row[0]->size();
  protected:
    Vector<Vector<T>*> row;
};
```

Here the only data member is `row`, a vector of pointers to vectors. As a vector, `row` can be used with the subscript operator: `row[i]` which returns a pointer to the vector that represents the `i`th row of the matrix.

The default constructor assigns to each `row[i]` a `new` vector containing `c` elements of type T. The destructor has to `delete` each of these vectors separately. The `rows()` and `columns()` functions return the number of rows and columns in the matrix. The number of rows is the value that the member function `size()` returns for the `Vector<Vector<T>*>` object `row`. The number of columns is the value that the member function `size()` returns for the `Vector<T>` object `*row[0]`, which can be referenced either by `(*row[0]).size()` or by `row[0]->size()`.

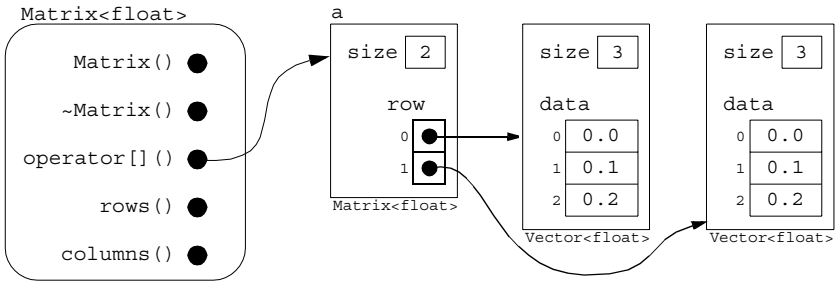
Here is a test driver and a sample run:

```
int main()
{ Matrix<float> a(2,3);
  a[0][0] = 0.0;  a[0][1] = 0.1;  a[0][2] = 0.2;
  a[1][0] = 1.0;  a[1][1] = 1.1;  a[1][2] = 1.2;
```

```
    cout << "The matrix a has " << a.rows() << " rows and "
         << a.columns() << " columns:\n";
    for (int i=0; i<2; i++)
    { for (int j=0; j<3; j++)
        cout << a[i][j] << "  ";
      cout << endl;
    }
}
```

```
The matrix a has 2 rows and 3 columns:
0  0.1  0.2
1  1.1  1.2
```

The matrix `a` can be visualized like this:



The diagram shows the situation during one of the subscript access calls `a[1][2]`.

Notice that the actual data values 0.2, 1.1, *etc*., are stored in two separate `Vector<float>` objects. The `Matrix<float>` object `m` only contains pointers to those objects.

Note that our `Matrix` class template used *composition* with the `Vector` class template, while our `Array` class template used *inheritance* with the `Vector` class template.

13.7 A CLASS TEMPLATE FOR LINKED LISTS

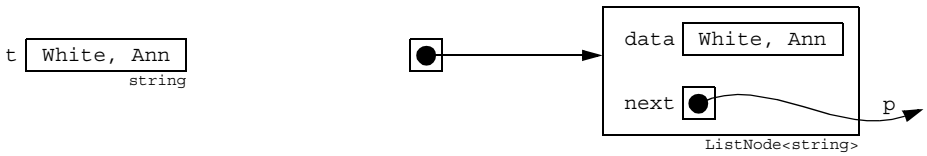
Linked lists were introduced in Example 10.13 on page 244. These data structures provide an alternative to vectors, with the advantage of dynamic storage. That is, unlike vectors, linked lists can grow and shrink dynamically according to how many data items are being stored. There is no wasted space for unused elements in the list.

EXAMPLE 13.7 A `List` Class Template

A list consists of a linked sequence of nodes. Each node contains one data item and a link to the next node. So we begin by defining a `ListNode` class template:

```
template<class T>
class ListNode
{   friend class List<T>;
  public:
    ListNode(T& t, ListNode<T>* p) : data(t), next(p) { }
  protected:
    T data;                 // data field
    ListNode* next;         // points to next node in list
};
```

The constructor creates a new node, assigning the `T` value `t` to its `data` field and the pointer `p` to its `next` field:



If `T` is a class (instead of an ordinary type), its constructor will be called by the declaration of `data`.

Note that the class `List<T>` is declared here to be a `friend` of the `ListNode` class. This will allow the member functions of the `List` class to access the protected members of the `Node` class. For this statement to compile, some compilers require the following *forward reference* to precede the `ListNode` template definition:

```
template<class T>
class List;
```

This simply tells the compiler that the identifier `List` will be defined later as a class template.

Here is the `List` class template interface, which follows the `ListNode` template definition:

```
template<class T>
class List
{ public:
    List() : first(0) { }
    ~List();
    void insert(T t);       // insert t at front of list
    int remove(T& t);       // remove first item t in list
    bool isEmpty() { return (first == 0); }
    void print();
  protected:
    ListNode<T>* first;
    ListNode<T>* newNode(T& t, ListNode<T>* p)
    { ListNode<T>* q = new ListNode<T>(t,p); return q; }
};
```

A `List` object contains only the pointer named `first`. This points to a `ListNode` object. The default constructor initializes the pointer to `NULL`. After items have been inserted into the list, the `first` pointer will point to the first item in the list.

The `newNode()` function invokes the `new` operator to obtain a new `ListNode` object by means of the `ListNode()` constructor. The new node will contain the `T` value `t` in its `data` field and the pointer `p` in its `next` field. The function returns a pointer to the new node. It is declared `protected` because it is a utility function that is used only by the other member functions.

The `List` destructor is responsible for deleting all the items in the list:

```
template<class T>
List<T>::~List()
{ ListNode<T>* temp;
  for (ListNode<T>* p = first; p; )          // traverses list
  { temp = p;
    p = p->next;
    delete temp;
  }
}
```

This has to be done in a loop that traverses the list. Each node is deleted by invoking the `delete` operator on a pointer to the node.

The `insert()` function creates a new node containing the `T` value `t` and then inserts this new node at the beginning of the list:

```
template<class T>
void List<T>::insert(T t)
{ ListNode<T>* p = newNode(t,first);
  first = p;
}
```

Since the new node will be made the first node in the list, its `next` pointer should point to the node that is currently first in the list. Passing the `first` pointer to the `NewNode` constructor does that. Then the `first` pointer is reset to point to the new node.

The `remove()` function removes the first item from the list, returning its `data` value by reference in the parameter `t`. The function's return value is 1 or 0 according to whether the operation succeeded:

```
template<class T>
int List<T>::remove(T& t)
{ if (isEmpty()) return 0; // flag to signal no removal
  t = first->data;         // data value returned by reference
  ListNode<T>* p = first;
  first = first->next;     // advance first pointer to remove node
  delete p;
  return 1;                // flag to signal successful removal
}
```

The `print()` function simply traverses the list, printing each node's `data` value:

```
template<class T>
void List<T>::print()
{ for (ListNode<T>* p=first; p; p=p->next)
    cout << p->data << " -> ";
  cout << "*\n";
}
```

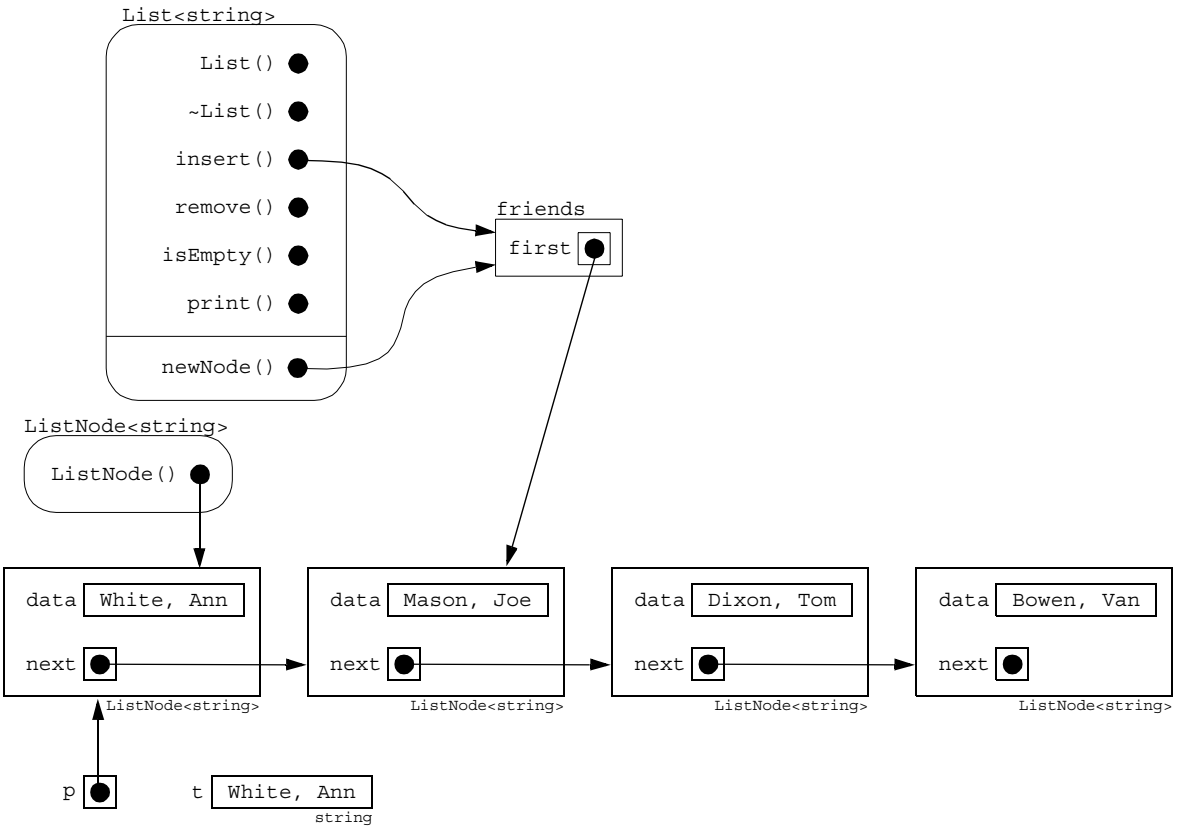
Here is a test driver and a sample run:

```
#include <iostream.h>
#include "List.h"

int main()
{ List<string> friends;
  friends.insert("Bowen, Van");
  friends.insert("Dixon, Tom");
  friends.insert("Mason, Joe");
  friends.insert("White, Ann");
  friends.print();
  string name;
  friends.remove(name);
  cout << "Removed: " << name << endl;
  friends.print();
}
```

```
White, Ann -> Mason, Joe -> Dixon, Tom -> Bowen, Van -> *
Removed: White, Ann
Mason, Joe -> Dixon, Tom -> Bowen, Van -> *
```

Notice that, since each item is inserted at the beginning of the list, they end up in the opposite order from their insertion.

This `friends` list can be visualized like this:



This shows the situation at the moment that the `insert()` function has invoked the `newNode()` function which has invoked the `ListNode()` constructor to create a new node for `"White, Ann"`.

13.8 ITERATOR CLASSES

A common activity performed on a container object is the *traversal* of the object. For example, to traverse a `List` object means to "travel" through the list, "visiting" each element. This was done by means of a `for` loop in both the destructor and the `print()` function in our `List` class template. (See Example 13.7 on page 309.)

An *iterator* is an object that has the ability to traverse through a container object. It acts like a pointer, locating one item in the container at a time. All iterators have the same basic functionality, regardless of the type of container to which they are attached. The five fundamental operations are:

- initialize the iterator at some initial position in the container;
- return the data value stored at the current position;
- change the data value stored at the current position;
- determine whether there actually is an item at the iterator's current position;
- advance to the next position in the container.

Since these five operations should be implemented by every iterator, it makes sense to declare an abstract base class with these functions. We actually need an abstract base class template because the container classes will be template instances:

```
template<class T>
class Iterator
{ public:
    virtual void reset() =0;          // initialize the iterator
    virtual T operator()() =0;        // read current value
    virtual void operator=(T t) =0;   // write current value
    virtual int operator!() =0;       // determine whether item exists
    virtual int operator++() =0;      // advance to next item
};
```

Recall that every pure virtual function prototype begins with the keyword "virtual" and ends with the code "`() =0`". The parentheses are required because it is a function, and the initializer "`=0`" makes it a pure virtual function. Also recall that an *abstract base class* is any class that contains at least one pure virtual function. (See Section 12.9 on page 286.)

Now we can use this abstract base class template to derive iterator templates for various container classes.

The `List` class template in Example 13.7 on page 309 had an obvious shortcoming: it allowed insertions and deletions only at the front of the list. A list iterator will solve this problem, as shown in the next example.

EXAMPLE 13.8 An Iterator Class Template for the `List` Class Template

```
#include "List.h"
#include "Iterator.h"

template<class T>
class ListIter : public Iterator<T>
{ public:
    ListIter(List<T>& l) : list(l) { reset(); }
    virtual void reset() { previous = NULL;  current = list.first; }
    virtual T operator()() { return current->data; }
    virtual void operator=(T t) { current->data = t; }
    virtual int operator!();     // determine whether current exists
    virtual int operator++();    // advance iterator to next item
    void insert(T t);            // insert t after current item
    void preInsert(T t);         // insert t before current item
    void remove();               // remove current item
  protected:
    ListNode<T>* current;        // points to current node
    ListNode<T>* previous;       // points to previous node
    List<T>& list;               // this is the list being traversed
};
```

In addition to a constructor and the five fundamental operations, we have added three other functions that will make lists much more useful. They allow the insertion and deletion of items anywhere in the list.

The `operator!()` function serves two purposes. First it resets the `current` pointer if necessary, and then it reports back whether that pointer is `NULL`. The first purpose is to "clean up" after a call to the `remove()` function which deletes the node to which `current` points.

```
template<class T>
int ListIter<T>::operator!()
{ if (current == NULL)                    // reset current pointer
    if (previous == NULL) current = list.first;
    else current = previous->next;
  return (current != NULL);      // returns TRUE if current exists
}
```

If the `current` and `previous` pointers are both `NULL`, then either the list is empty or it has only one item. So setting `current` equal to the list's `first` pointer will either make current `NULL` or leave it pointing to the first item in the list. If `current` is `NULL` but `previous` is pointing to a node, then we simply reset `current` to point to the item that follows that node. Finally, the function returns 0 or 1 according to whether `current` is `NULL`. This allows the function to be invoked in the form

```
if (!it) . . .
```

where `it` is an iterator. The expression `(!it)` is read "a current item exists," because the function will return 1 (*i.e.*, "true") if `current` is not `NULL`. We use this function to check the status of the `current` pointer before invoking an insertion or deletion function that requires using the pointer.

The `operator++()` "increments" the iterator by advancing its `current` pointer to the next item in the list after advancing its `previous` pointer. It precedes this action with the same resetting procedure that the `operator!()` function performed if it finds the `current` pointer `NULL`:

```
template<class T>
int ListIter<T>::operator++()
{ if (current == NULL)                // reset current pointer
    if (previous == NULL) current = list.first;
    else current = previous->next;
  else
  { previous = current;               // advance current pointer
    current = current->next;
  }
  return (current != NULL);           // returns TRUE if current exists
}
```

This operator allows for easy traversal of the list:

```
for (it.reset(); !it; ++it) . . .
```

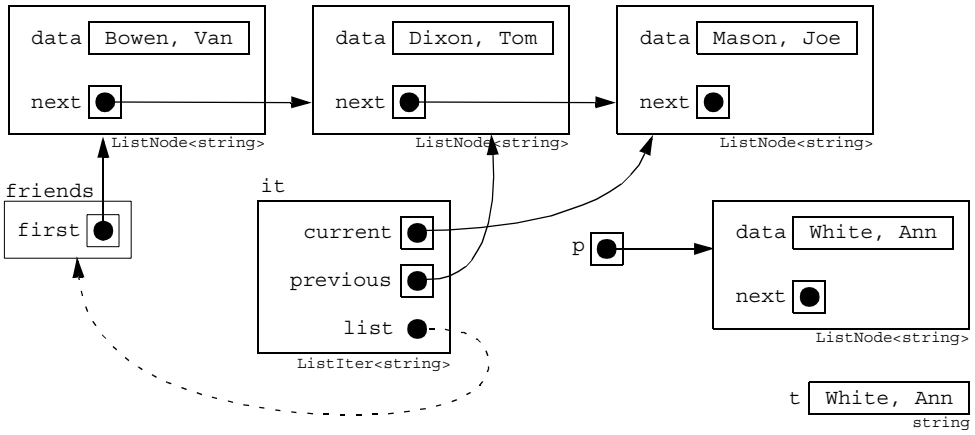
just like an ordinary `for` loop traversing an array. It resets the iterator to locate the first item in the list. Then after visiting that item, it increments the iterator to advance and visit the next item. The loop continues as long as `!it` returns "true", which means that there is still an item to be visited.

The `insert(t)` function creates a new node for `t` and then inserts that node immediately after the `current` node:

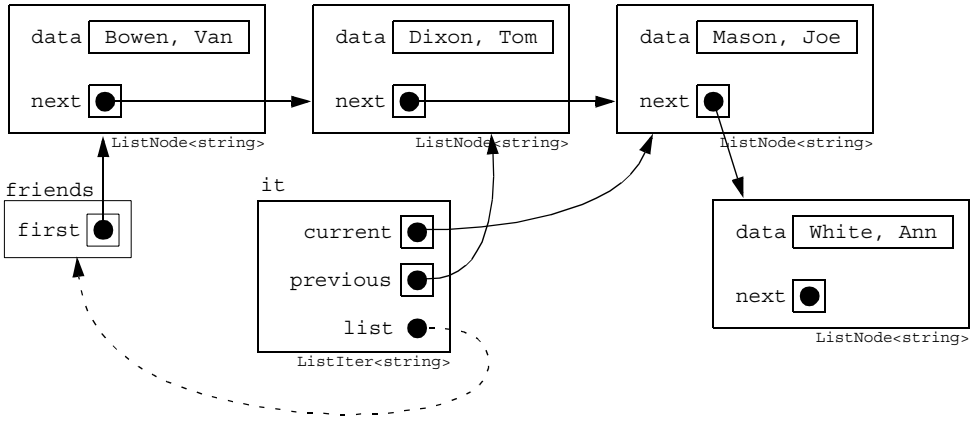
```
template<class T>
void ListIter<T>::insert(T t)
{ ListNode<T>* p = list.newNode(t,0);
  if (list.isEmpty()) list.first = p;
  else
  { p->next = current->next;
    current->next = p;
  }
}
```

The `insert` operation can be visualized like this:

***Before*:**



***After*:**



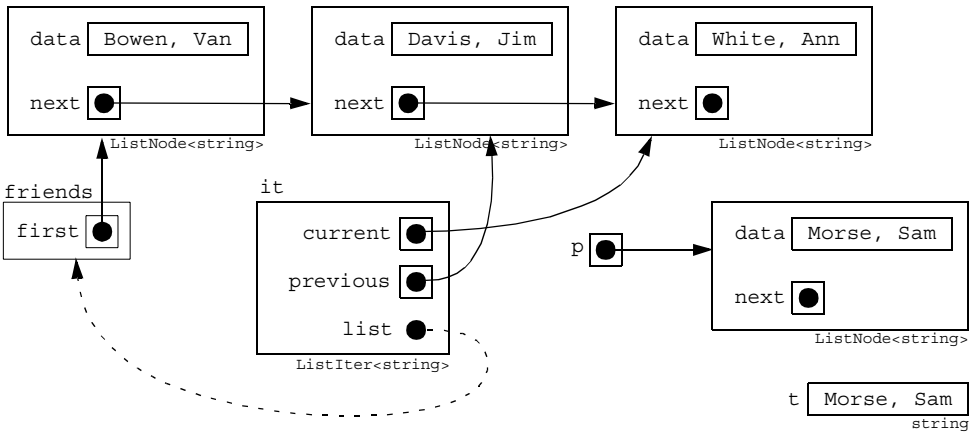
Note that the operation leaves the `current` and `previous` pointers unchanged.

The `preInsert()` function is similar to the `insert()` function, except that it inserts the new node in front of the `current` node:

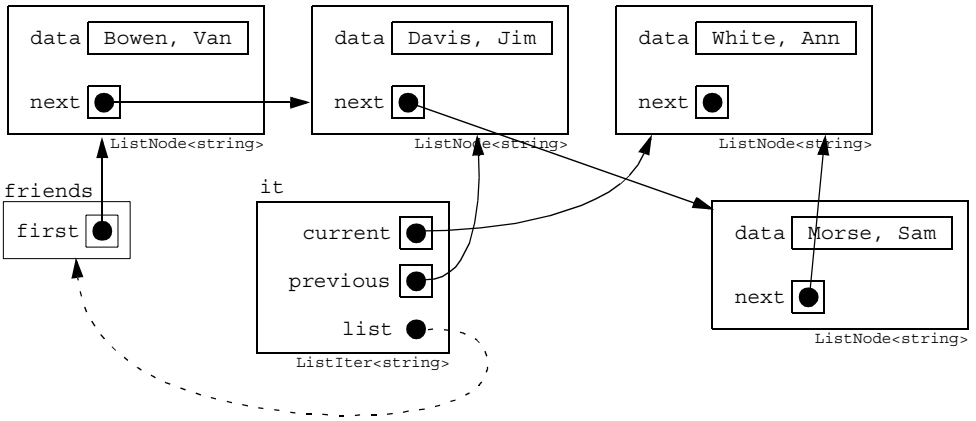
```
template<class T>
void ListIter<T>::preInsert(T t)
{ ListNode<T>* p = list.newNode(t,current);
  if (current == list.first) list.first = previous = p;
  else previous->next = p;
}
```

The `preInsert` operation can be visualized like this:

***Before*:**



***After*:**



Note that like `insert`, this operation also leaves the `current` and `previous` pointers unchanged.

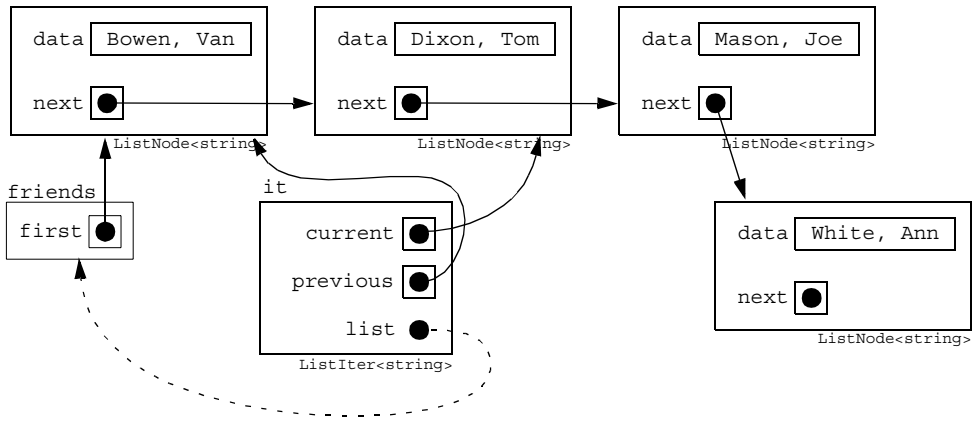
The `remove()` function deletes the `current` node:

```
template<class T>
void ListIter<T>::remove()
{ if (current == list.first) list.first = current->next;
  else previous->next = current->next;
  delete current;
  current = 0;
}
```

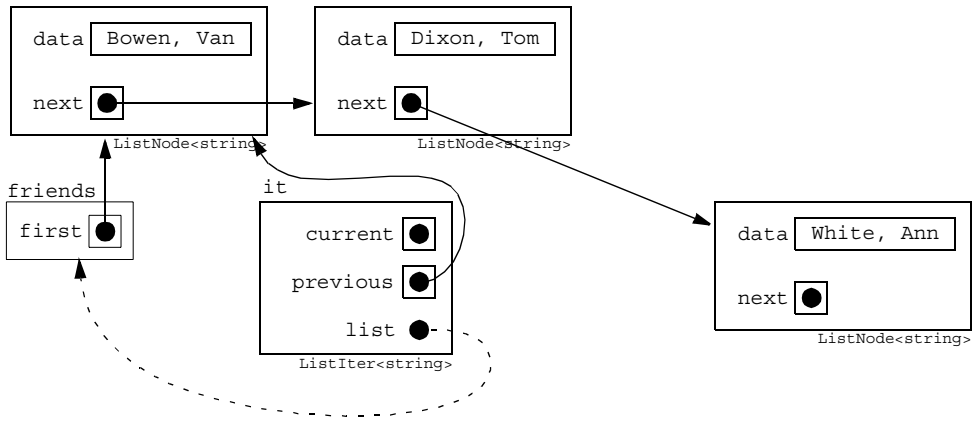
It leaves the `previous` pointer unchanged and the current pointer `NULL`.

The `remove` operation can be visualized like this:

***Before*:**



***After*:**



Here is a test driver for the list iterator:

```
#include "ListIter.h"
int main()
{ List<string> friends;
  ListIter<string> it(friends);
  it.insert("Bowen, Van");
  ++it;                              // sets current to first item
  it.insert("Dixon, Tom");
  ++it;                              // sets current to second item
  it.insert("Mason, Joe");
  ++it;                              // sets current to third item
  it.insert("White, Ann");
  ++it;                              // sets current to fourth item
```

```
  friends.print();
  it.reset();                          // sets current to first item
  ++it;                                // sets current to second item
  it = "Davis, Jim";                   // replace with new name
  ++it;                                // sets current to third item
  it.remove();                         // removes third item
  friends.print();
  if (!it) it.preInsert("Morse, Sam");
  friends.print();
  for (it.reset(); !it; ++it)          // traverses entire list
    it = "[" + it() + "]";
  friends.print();
}
```

```
Bowen, Van -> Dixon, Tom -> Mason, Joe -> White, Ann -> *
Bowen, Van -> Davis, Jim -> White, Ann -> *
Bowen, Van -> Davis, Jim -> Morse, Sam -> White, Ann -> *
[Bowen, Van] -> [Davis, Jim] -> [Morse, Sam] -> [White, Ann] -> *
```

The `for` loop changes each `data` value in the list by prepending a left bracket and appending a right bracket to each string. Note that the assignment `it = "[" + it() + "]"` calls the `operator()()` and `operator=()` functions of the `ListIter<string>` class as well as the constructor `string(const char*)` and `operator+=()` function defined in the `string` class.

To give `ListIter` objects the access to the `protected` members of `List` objects that they need to do their job, we need to declare the `ListIter` class a `friend` of the `List` class:

```
template<class T>
class List
{   friend class ListIter<T>;
  public:
    // other members
  protected:
    ListNode<T>* first;
    // other members
};
```

`List` iterators also need the access to the `protected` members of `ListNode` objects:

```
template<class T>
class ListNode
{   friend class List<T>;
    friend class ListIter<T>;
  public:
    ListNode(T& t, ListNode<T>* p) : data(t), next(p) { }
  protected:
    T data;                // data field
    ListNode* next;        // points to next node in list
};
```

An iterator acts like a window, allowing access to one item at a time in the container. Iterators are sometimes called *cursors* because they locate a specific element among the entire structure, the same way that a cursor on your computer screen locates one character location.

A structure may have more than one iterator. For example, one could declare three iterators on a list like this:

```
List<float> list;
ListIter<float> it1(list), it2(list), it3(list);
```

```
it1.insert(11.01);
++it1;
it1.insert(22.02);
++it1;
it1.insert(33.03);
for (it2.reset(); !it2; ++it2)
it2 = 10*it2;          // multiplies each stored number by 10
it3 = it1;                 // replaces 110.1 with 330.3 in first item
```

The iterators are independent of each other. While `it2` traverses the list, `it1` remains fixed on the third item.

Review Questions

13.1 What is the difference between a function template and a template function?

13.2 What is the difference between a class template and a template class?

13.3 What are the advantages and disadvantages of using a linked list instead of a vector?

13.4 How is an iterator like an array subscript?

Problems

13.1 Write and test a program that instantiates a function template that returns the minimum of two values.

13.2 Write and test a program that instantiates a function template that implements a binary search of a sorted array of objects.

13.3 Implement and test a template for generating `Queue` classes. A *queue* works like a stack, except that insertions are made at one end of the linear structure and removed from the other end. It simulates an ordinary waiting line.

13.4 Modify the `Vector` class template so that existing vectors can change their size.

13.5 Add a constructor to the `Vector` class template that replicates an ordinary array as a vector.

13.6 Derive an `Array<T,E>` class template from the `Vector<T>` class template, where the second template parameter `E` holds an enumeration type to be used for the array index.

Answers to Review Questions

13.1 A *function template* is a template that is used to generate functions. A *template function* is a function that is produced by a template. For example, `swap(T&, T&)` in Example 13.1 is a function template, but the call `swap(m, n)` generates the actual template function that is invoked by the call.

13.2 A *class template* is a template that is used to generate classes. A *template class* is a class that is produced by a template. For example, `Stack` in Example 13.3 is a class template, but the type `Stack<int>` used in the declarations is an actual template class.

13.3 Vectors have the advantage of *direct access* (also called "random access") to their components by means of the subscript operator. So if the elements are kept in order, we can locate them very quickly using the Binary Search Algorithm. Lists have the advantage of being dynamic, so that they never use more space than is currently needed, and they aren't restricted to a predetermined size (except for the size of the computer's memory). So vectors have a time advantage and lists have a space advantage.

13.4 Both iterators and array indexes act as locators into a data structure. The following code shows that they work the same way:

```
float a[100];              // an array of 100 floats
int i = 0;                 // an index for the array
a[i] = 3.14159;
for (i = 0; i < 100; i++) cout << a[i];
List<float> list;          // a list of floats
ListIter<float> it(list);  // an iterator for the list
it = 3.14159;
for (it.reset(); !it; ++it) cout << it();
```

Solutions to Problems

13.1 A minimum function should compare two objects of the same type and return the object whose value is smaller. The type should be the template parameter `T`:

```
template <class T>
T min(T x, T y)
{ return ( x < y ? x : y );
}
```

This implementation uses the conditional expression operator: `( x < y ? x : y )`. If `x` is less than `y`, the expression evaluates to `x`; otherwise it evaluates to `y`.

Here is the test driver and a sample run:

```
#include "Ratio.h"
int main()
{ cout << "min(22, 44) = " << min(22, 44) << endl;
  cout << "min(66.66, 33.33) = " << min(66.66, 33.33) << endl;
  Ratio x(22, 7), y(314, 100);
  cout << "min(x, y) = " << min(x, y) << endl;
}
```

```
min(22, 44) = 22
min(66.66, 33.33) = 33.33
min(x, y) = 314/100
```

13.2 A search function should be passed the array `a`, the object `key` to be found, and the bounds on the array index that define the scope of the search. If the object is found, its index in the array should be returned; otherwise, the function should return -1 to signal that the object was not found:

```
template<class T>
int search(T a[], T key, int first, int last)
{ while (first <= last)
  { int mid = (first + last)/2;
    if (key < a[mid]) last = mid - 1;
    else if (key > a[mid]) first = mid + 1;
    else return mid;
  }
  return -1;  // not found
}
```

Within the `while` loop, the subarray from `a[first]` to `a[last]` is bisected by `mid`. If `key < a[mid]` then `key` cannot be in the second half of the array, so `last` is reset to `mid-1` to reduce the scope of the search to the first half. Otherwise, if `key > a[mid]`, then `key` cannot be in the first half of the array, so `first` is reset to `mid+1` to reduce the scope of the search to the second half. If both conditions are false, then `key == a[mid]` and we can return.

Here is the test driver and a sample run:

```
template<class T> int search(T [], T, int, int);

string names[]
  = {"Adams", "Black", "Cohen", "Davis", "Evans", "Frost",
     "Green", "Healy", "Irwin", "Jones", "Kelly", "Lewis"};

int main()
{ string name;
  while (cin >> name)
  { int location = search(names, name, 0, 9);
    if (location == -1) cout << name << " is not in list.\n";
    else cout << name << " is in position " << location << endl;
  }
}
```

```
Green
Green is in position 6
Black
Black is in position 1
White
White is not in list.
Adams
Adams is in position 0
Jones
Jones is in position 9
Smith
Smith is not in list.
```

13.3 Like the implementation of the Stack template, this implementation uses an array `data` of `size` elements of type `T`. The location in the array where the next object will be inserted is always given by the value of `(front % size)`, and the location in the array that holds the next object to be removed is always given by the value of `(rear % size)`:

```
template<class T>
class Queue
{ public:
    Queue(int s = 100) : size(s+1), front(0), rear(0)
    { data = new T[size]; }
      ~Queue() { delete [] data;
    }
    void insert(const T& x) { data[rear++ % size] = x; }
    T remove() { return data[front++ % size]; }
    int isEmpty() const { return front == rear; }
    int isFull() const { return (rear + 1) % size == front; }
  private:
    int size, front, rear;
    T* data;
};
```

The test driver uses a queue that can hold at most 3 `chars`:

```
#include "Queue.h"
int main()
{ Queue<char> q(3);
  q.insert('A');
  q.insert('B');
```

```
  q.insert('C');
  if (q.isFull()) cout << "Queue is full.\n";
  else cout << "Queue is not full.\n";
  cout << q.remove() << endl;
  cout << q.remove() << endl;
  q.insert('D');
  q.insert('E');
  if (q.isFull()) cout << "Queue is full.\n";
  else cout << "Queue is not full.\n";
  cout << q.remove() << endl;
  cout << q.remove() << endl;
  cout << q.remove() << endl;
  if (q.isEmpty()) cout << "Queue is empty.\n";
  else cout << "Queue is not empty.\n";
}
```

```
Queue is full.
A
B
Queue is full.
C
D
E
Queue is empty.
```

13.4 We add two functions:

```
unsigned resize(unsigned n);
unsigned resize(unsigned n, T t);
```

Both functions transform the vector into one of size `n`. If `n < size`, then the last `size-n` elements are simply discarded. If `n == size`, then the vector is left unchanged. If `n > size`, then the first `size` elements of the transformed vector will be the same as those of the prior version; the last `n-size` are assigned the value `t` by the second `resize()` function and are left uninitialized by the first. Both functions return the new size:

```
template<class T>
unsigned Vector<T>::resize(unsigned n, T t)
{ T* new_data = new T[n];
  copy(v);
  for (i = size; i < n; i++)
    new_data[i] = t;
  delete [] data;
  size = n;
  data = new_data;
  return size;
  }
template<class T>
unsigned Vector<T>::resize(unsigned n)
{ T* new_data = new T[n];
  copy(v);
  delete [] data;
  size = n;
  data = new_data;
  return size;
}
```

13.5 The new constructor converts an array `a` whose elements have type `T`:

```
template<class T>
class Vector
{ public:
    Vector(T* a) : size(sizeof(a)), data(new T[size])
    { for (int i = 0; i < size; i++) data[i] = a[i]; }
    // other members
};
```

Here is a test driver for the new constructor:

```
int main()
{ int a[] = { 22, 44, 66, 88 };
  Vector<int> v(a);
  cout << v.size() << endl;
  for (int i = 0; i < 4; i++)
    cout << v[i] << "  ";
}
```

```
4
22  44  66  88
```

The advantage of this constructor is that we can initialize a vector now without having to assign each component separately.

13.6 The derived template has three member functions: two constructors and a new subscript operator:

```
template <class T, class E>
class Array : public Vector<T>
{ public:
    Array(E last) : Vector<T>(unsigned(last) + 1) { }
    Array(const Array<T,E>& a) : Vector<T>(a) { }
    T& operator[](E index) const
    { return Vector<T>::operator[](unsigned(index));
    }
};
```

The first constructor calls the default constructor defined in the parent class `Vector<T>`, passing to it the number of `E` values that are to be used for the index. The new copy constructor and subscript operator also invoke their equivalent in the parent class.

Here is a test driver for the `Array<T,E>` template:

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

int main()
{ Array<int,Days> customers(SAT);
  customers[MON] = 27;  customers[TUE] = 23;
  customers[WED] = 20;  customers[THU] = 23;
  customers[FRI] = 36;  customers[SAT] = customers[SUN] = 0;
  for (Days day = SUN; day <= SAT; day++)
    cout << customers[day] << "  ";
}
```

```
0  27  23  20  23  36  0
```

The enumeration type `Days` defines seven values for the type. Then the object `customers` is declared to be an array of `int`s indexed by these seven values. The rest of the program applies the subscript operator to initialize and then print the array.

Chapter 14

Standard C++ Vectors

14.1 INTRODUCTION

Although not as efficient, Standard C++ **`string`** objects are more robust than the classic C-strings. They are easier to use and they cause fewer run-time errors. In the same way, Standard C++ **`vector`** objects are more robust than ordinary arrays. So **`vector`** objects provide a good alternative to arrays. The **`vector`** class template is also the prototype for all the container classes in the Standard C++ Library. (See Chapter 15.)

The **`vector`** class template is defined in the `<vector>` header.

EXAMPLE 14.1 Using a `vector` of `strings`

This program creates a vector `v` of 8 strings and then calls a `load()` function and a `print()` function to load and print the vector.

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;
void load(vector<string>&);
void print(vector<string>);
const int SIZE=8;

int main()
{ vector<string> v(SIZE);
  load(v);
  print(v);
}

void load(vector<string>& v)
{ v[0] = "Japan";
  v[1] = "Italy";
  v[2] = "Spain";
  v[3] = "Egypt";
  v[4] = "Chile";
  v[5] = "Zaire";
  v[6] = "Nepal";
  v[7] = "Kenya";
}

void print(vector<string> v)
{ for (int i=0; i<SIZE; i++)
    cout << v[i] << endl;
  cout << endl;
}
```

```
Japan
Italy
Spain
Egypt
Chile
Zaire
Nepal
Kenya
```

Note that this program could have been written almost the same way using an array of strings:

```
string v[SIZE];
```

In particular, access by means of the subscript operator `v[i]` works the same with vectors and arrays.

EXAMPLE 14.2 Using the `push_back()` and `size()` Functions

This is the same program as in Example 14.1 except for the changes indicated in boldface: the type identifier `strings` is used in place of `vector<string>`, the `push_back()` function is used instead of assigning elements to `v[i]`, and the `size()` function is used instead of storing the constant `SIZE` as a global constant.

```
typedef vector<string> Strings;
void load(Strings&);
void print(Strings);

int main()
{ Strings v;
  load(v);
  print(v);
}

void load(Strings& v)
{ v.push_back("Japan");
  v.push_back("Italy");
  v.push_back("Spain");
  v.push_back("Egypt");
  v.push_back("Chile");
  v.push_back("Zaire");
  v.push_back("Nepal");
  v.push_back("Kenya");
}

void print(Strings v)
{ for (int i=0; i<v.size(); i++)
    cout << v[i] << endl;
  cout << endl;
}
```

Note that vector `v` has 0 elements when it is created. Each time the `push_back()` function is called, it appends the new element to the end of the vector and increments its size. So when the `load()` function returns, the size of `v` is 8.

The output here is the same as for the program as in Example 14.1.

14.2 ITERATORS ON VECTORS

EXAMPLE 14.3 Using `vector` Iterators

This program defines the type identifier `Sit` to stand for iterators on vectors of strings. It then uses such an iterator to traverse the vector in the `print()` function.

```
typedef vector<string> Strings;
typedef Strings::iterator Sit;
void load(Strings&);
void print(Strings);

int main()
{ Strings v;
  load(v);
  print(v);
}

void print(Strings v)
{ for (Sit it=v.begin(); it!=v.end(); it++)
    cout << *it << endl;
  cout << endl;
}
```

The `for` loop initializes the iterator `it` to the beginning of the vector `v`. The expression `*it` returns the element located by the iterator. The increment expression `it++` advances `it` to the next element in the vector. When `it == v.end()`, `it` has moved to the imaginary position that follows the last element of the vector. That signals that the traversal has finished and stops the loop.

The output here is the same as for the program as in Example 14.1.

EXAMPLE 14.4 Using the Generic `sort()` Algorithm

This uses the `sort()` function that is defined in the `<algorithm>` header. (See page 393.) The subsequent call `print(v)` shows that the strings are sorted alphabetically.

```
int main()
{ Strings v;
  load(v);
  sort(v.begin(),v.end());
  print(v);
}
```

```
Chile
Egypt
Italy
Japan
Kenya
Nepal
Spain
Zaire
```

The generic `sort()` algorithm requires two iterator arguments to indicate what part of the vector is to be sorted. The `begin()` and `end()` functions return iterators that locate the begining and ending locations of the vector, so passing these two iterators to `sort()` indicates that the entire vector is to be sorted.

14.3 ASSIGNING VECTORS

EXAMPLE 14.5 Using the Assignment Operator to Duplicate a `vector`

This program demonstrates that one vector can be assigned to another.

```
int main()
{ Strings v, w;
  load(v);
  w = v;
  sort(v.begin(),v.end());
  print(v);
  print(w);
}
```

```
Chile
Egypt
Italy
Japan
Kenya
Nepal
Spain
Zaire

Japan
Italy
Spain
Egypt
Chile
Zaire
Nepal
Kenya
```

The assignment `w = v` has the same effect as the call load(w) would have: it duplicates each of the 8 elements of v and loads them into w.

The fact that `w` is independent of `v` is evident from the output: `w` remains unchanged when `v` is sorted.

EXAMPLE 14.6 Using the `front()`, `back()`, and `pop_back()` Functions

The `front()` function returns the first element in the vector. The `back()` function returns the last element in the vector. The `pop_back()` function removes the last element in the vector.

```
int main()
{ Strings v;
  load(v);
  sort(v.begin(),v.end());
  print(v);
  cout << "v.front() = " << v.front() << endl;
  cout << "v.back()  = " << v.back() << endl;
  v.pop_back();
  cout << "v.back()  = " << v.back() << endl;
  v.pop_back();
  cout << "v.back()  = " << v.back() << endl;
  print(v);
}
```

The call `v.pop_back()` removes the string `Zaire` from the vector `v`.

```
Chile
Egypt
Italy
Japan
Kenya
Nepal
Spain
Zaire

v.front() = Chile
v.back()  = Zaire
v.back()  = Spain
v.back()  = Nepal
Chile
Egypt
Italy
Japan
Kenya
Nepal
```

14.4 THE `erase()` and `insert()` FUNCTIONS

EXAMPLE 14.7 Using the `erase()` Function

```
int main()
{ Strings v;
  load(v);
  sort(v.begin(),v.end());
  print(v);
  v.erase(v.begin()+2);  // removes Italy
  v.erase(v.end()-2);    // removes Spain
  print(v);
}
```

```
Chile
Egypt
Italy
Japan
Kenya
Nepal
Spain
Zaire

Chile
Egypt
Japan
Kenya
Nepal
Zaire
```

The call `v.erase(v.begin()+2)` removes the element `v[2]`. It is the element that follows the 2nd element (`Egypt`) from the beginning of the vector.

The call `v.erase(v.begin()-2)` removes the element `v[n-2]`, where `n` is the size of the vector. It is the element that follows the 2nd element (`Nepal`) from the end of the vector.

EXAMPLE 14.8 Using the `insert()` Function

This program illustrates the `insert()` function and the use of the `erase()` function to remove an entire segment of elements.

```
int main()
{ Strings v;
  load(v);
  sort(v.begin(),v.end());
  print(v);
  v.erase(v.begin()+2,v.end()-2);  // removes the segment Italy..Nepal
  print(v);
  v.insert(v.begin()+2,"India");
  print(v);
}
```

```
Chile
Egypt
Italy
Japan
Kenya
Nepal
Spain
Zaire

Chile
Egypt
Spain
Zaire

Chile
Egypt
India
Spain
Zaire
```

The call `v.erase(v.begin()+2,v.end()-2)` removes the segment `v[2..5]`.

The call `v.insert(v.begin()+2,"India")` inserts `India` immediately after the 2nd element (`Egypt`) from the beginning of the vector.

14.5 THE `find()` FUNCTION

The `find()` function is used to search for an element in a vector.

EXAMPLE 14.9 Using the `find()` Function

This program uses the `find()` function to obtain iterators that locate `Egypt` and `Malta` in the vector. Then it passes them to the `sort()` function to sort that segment within the vector.

```
int main()
{ Strings v;
  load(v);
  print(v);
  Sit egypt=find(v.begin(),v.end(),"Egypt");
  Sit malta=find(v.begin(),v.end(),"Malta");
```

```
  sort(egypt,malta);
  print(v);
}

void load(Strings& v)
{ v.push_back("Japan");
  v.push_back("Italy");
  v.push_back("Spain");
  v.push_back("Egypt");
  v.push_back("Chile");
  v.push_back("Zaire");
  v.push_back("Nepal");
  v.push_back("Kenya");
  v.push_back("India");
  v.push_back("China");
  v.push_back("Malta");
  v.push_back("Syria");
}
```

```
Japan
Italy
Spain
Egypt
Chile
Zaire
Nepal
Kenya
India
China
Malta
Syria

Japan
Italy
Spain
Chile
China
Egypt
India
Kenya
Nepal
Zaire
Malta
Syria
```

The two iterators `egypt` and `malta` are initialized by the `find()` function. Together, they delineate the segment `v[3..9]` consisting of the 7 elements {`Egypt`, `Chile`, `Zaire`, `Nepal`, `Kenya`, `India`, `China`}. The `sort()` function sorts that segment, leaving the other 5 elements unchanged.

Like the `sort()` function, the `find()` function is a generic algorithm that requires two iterators to specify what segment of the vector is to be processed. (See page 373.) If you want to search the entire vector, use the iterators that are returned by the `begin()` and `end()` functions, like this:

```
find(v.begin(), v.end(), x);
```

14.6 THE C++ STANDARD `vector` CLASS TEMPLATE

The interface for the `vector` class template is the prototype for all the Standard C++ container class templates. (See Chapter 15.) With only a few exceptions, each member function of the `vector` class corresponds to an equivalent member function for each of the other container classes (`stack`, `queue`, `list`, `set`, `map`, *etc.*).

Here is a simplified partial listing of the `vector` class template interface:

```
template <class T>
class vector
{   friend bool operator==(const vector&, const vector&);
    friend bool operator<(const vector&, const vector&);
  public:
    typedef T* iterator;
    vector();                           // default constructor
    vector(const vector&);              // copy constructor
    vector(int, const T&);              // auxiliary constructor
    vector(iterator, iterator);         // auxiliary constructor
    ~vector();                          // destructor
    vector& operator=(const vector&);   // assignment operator
    void assign(int, const T&);         // assigns a given value
    void assign(iterator, iterator);    // copies elements from object
    void resize(int);                   // changes size of vector
    void swap(vector&);                 // swaps elements with object
    bool empty() const;                 // returns true iff empty
    int size() const;                   // return number of elements
    iterator begin();                   // locates first element
    iterator end();                     // locates dummy element at end
    T& operator[](int);                 // subscript operator
    T& at(int);                         // range-checked access
    T& front();                         // accesses the first element
    T& back();                          // accesses the last element
    void push_back(const T&);           // inserts element at end
    void pop_back();                    // removes last element
    iterator insert(iterator, const T&);
    void insert(iterator, int, const T&);
    void insert(iterator, iterator, iterator);
    iterator erase(iterator);
    iterator erase(iterator, iterator);
    void clear();                       // removes all the elements
  private:
    //...
};
```

EXAMPLE 14.10 Using the Standard `vector<>` Class Template

Here is a complete C++ program that uses the Standard `vector<>` class template::

```
#include <iostream>
#include <vector>     // defines the Standard vector<T> class template
using namespace std;
typedef vector<double> Vec;
```

```
typedef vector<bool> Bits;

template <class T>
void copy(vector<T>& v, const T* x, int n)
{ vector<T> w;
  for (int i=0; i<n; i++)
    w.push_back(x[i]);
  v = w;
}

Vec projection(Vec& v, Bits& b)
{ int v_size = v.size();
  assert(b.size() >= v_size);
  Vec w;
  for (int i=0; i<v_size; i++)
    if (b[i]) w.push_back(v[i]);
  return w;
}

void print(Vec& v)
{ int v_size = v.size();
  for (int i=0; i<v_size; i++)
    cout << v[i] << "  ";
  cout << endl;
}

int main()
{ double x[8] = { 22.2, 33.3, 44.4, 55.5, 66.6, 77.7, 88.8, 99.9 };
  Vec v;
  copy(v, x, 8);
  bool y[8] = { false, true, false, true, true, true, false, true };
  Bits b;
  copy(b, y, 8);
  Vec w = projection(v, b);
  print(v);
  print(w);
}
22.2  33.3  44.4  55.5  66.6  77.7  88.8  99.9
33.3  55.5  66.6  77.7  99.9
```

This illustrates the `vector` class `push_back()` and `size()` member functions.

The purpose of the `projection(v, b)` function is to use the bit vector `b` as a *mask* to remove selected elements of the vector `v`. The resulting vector `w` is called the *projection* of `v` onto the subspace determined by `b`.

14.7 RANGE CHECKING

The `at()` member function of the standard `vector` class template automatically checks the value of the index variable to ensure that it is not out of range. This protection against program failure is not available for ordinary arrays.

Review Questions

14.1 What are the main differences between an array and a C++ `vector`?

14.2 How are `vector` iterators similar to array indexes?

Problems

14.1 Use the `find()` algorithm to implement and test the following function for `vector`s of `int`s:

```
int frequency(vector<int> v, int x);
// returns the number of occurrences of x in v;
```

14.2 Use the `find()` algorithm and the `erase()` function to implement and test the following function for `vector`s of `int`s:

```
void remove_duplicates(vector<int>& v);
// removes all duplicates in v;
```

14.3 Use the `sort()` algorithm to implement and test the following function for `vector`s of `float`s:

```
float median(vector<float>& v);
// returns the middle number among those stored in v;
```

14.4 Implement and test the following conversion functions:

```
int unsignedValue(BinaryCode bc);
// example: if bc has these bit values 1 0 1 0 1
//    unsignedValue(bc) returns 21
BinaryCode getUnsignedCode(unsigned n);
// returns shortest possible code for n
// example: if n = 15 returns the vector with elements 1 1 1 1
int signedValue(BinaryCode bc);
// example: if bc has these bit values  1 0 1 1 1 0
//    signedValue(bc) returns -30
BinaryCode getSignedCode(int n);
// returns shortest possible twosComplement code for n
// example: if n = 15 returns the vector with elements 0 1 1 1 1
//          if n = -15 returns the vector with elements 1 0 0 0 1
```

These use the following definitions:

```
typedef vector<int> BinaryCode;
typedef BinaryCode::iterator BCIterator;
```

Answers to Review Questions

14.1 Some of the main differences between arrays and C++ `vector`s are:

a. An array is declared as

```
string[8] a;  // a is an array of 8 strings
```

while a `vector` is declared as

```
vector<string> v(8);  // v is a vector of 8 strings
```

b. The assignment operator is defined for `vector`s but not for arrays:

```
v = w;  // assigns all the elements of the vector w to v
```

c. The comparison operators are defined for `vectors` but not for arrays:

```
if (v == w)  // true if the two vectors are equal
if (v < w)   // uses the lexicographic ordering of vectors
```

d. The `size()` member function is available for vectors but not for arrays:

```
int n = v.size();  // the number of elements in the vector v
```

e. The `at()` member function is available for vectors but not for arrays:

```
string v8 = v.at(8);  // the element at position 8
```

A range error exception is thrown if the element does not exist.

14.2 Some of the main similarities between arrays indexes and `vector` iterators are:

a. Both provide direct read-write access to the elements:

```
x = a[3];   // assigns to x element number 3
x = *it;    // assigns to x the element located by it
a[3] = 44;  // assigns 44 to element number 3
*it = 44;   // assigns 44 to the element located by it
```

b. Both can be incremented and decremented.

c. Both can be used as a basis for relative positions:

```
x = a[i+3];   // assigns to x the 3rd element after a[i]
x = *(it+3);  // assigns to x the 3rd element after *it
```

Solutions to Problems

14.1

```
int frequency(vector<int> v, int x)
{ int n=0;
  for (vector<int>::iterator it=v.begin(); ; it++)
  { it = find(it,v.end(),x);
    if (it==v.end()) return n;
    ++n;
  }
  return n;
}
```

14.2

```
void remove_duplicates(vector<int>& v)
{ for (vector<int>::iterator it=v.begin()+1; it!=v.end(); )
  { vector<int>::iterator jt=find(v.begin(),it,*it);
    if (jt == it) ++it;
    else it = v.erase(it);
  }
}
```

14.3

```
typedef vector<float> ScoreVector;
typedef ScoreVector::iterator ScoreVectorIterator;

float median( ScoreVector sv );
//precondition:  sv is not empty
// returns average of two sorted middle values in sv
// caller's argument remains unchanged

void  getScores( ScoreVector & sv );
void print( ScoreVectorIterator start, ScoreVectorIterator stop );
int main()
{ ScoreVector scores ;
  getScores( scores );
```

```
  print(  scores.begin() , scores.end() - 1 );
  cout << "median( scores ) = " << median( scores ) << endl;
}

float median( ScoreVector v  )
{   if (v.empty()) return 0.0;
    int n = v.size();
    sort( v.begin(), v.end() );
    return  ( v[ n/2 ] + v[ (n-1)/2 ] ) / 2.0;
}

void  getScores( ScoreVector & sv )
{  float nextScore;
   cout << "Enter next score or negative value to stop: ";
   cin >> nextScore;
   while ( nextScore >= 0.0 )
   { sv.push_back( nextScore );
     cout << "Enter next score or negative value to stop: ";
     cin >> nextScore;
   }
 }

void print( ScoreVectorIterator start, ScoreVectorIterator stop )
{  for(ScoreVectorIterator svIt = start; svIt <= stop; svIt++)
      cout << *svIt << endl;
}
```

14.4

```
typedef vector<int> BinaryCode;
typedef BinaryCode::iterator BCIterator;

int unsignedValue( BinaryCode bc );
// example if bc has these bit values 1 0 1 0 1
//    unsignedValue( bc ) returns 21

BinaryCode getUnsignedCode( unsigned n );
// returns shortest possible code for n
// example:  if n = 15 returns the vector with elements 1 1 1 1

int signedValue ( BinaryCode bc );
// example if bc has these bit values  1 0 1 1 1 0
//    signedValue( bc ) returns -30

BinaryCode getSignedCode( int n );
// returns shortest possible twosComplement code for n
// example: if n = 15 returns the vector with elements 0 1 1 1 1
//          if n = -15   "     "    "     "     "      1 0 0 0 1

void print( BinaryCode bc );
void testUnsigned();
void testSigned();
```

```
int main()
{ testUnsigned();
  testSigned();
}

void testUnsigned()
{ BinaryCode bc;
  for ( unsigned n = 0; n <= 11; n++ )
  {  bc  = getUnsignedCode( n );
     print( bc );
     cout << "  has unsigned value " << unsignedValue( bc )
          << " and signed value " << signedValue( bc ) << endl;
  }
}

int unsignedValue( BinaryCode bc )
{ int value = 0;
  for ( BCIterator bcIt = bc.begin(); bcIt != bc.end(); bcIt++ )
    value = value * 2 + *bcIt;
  return value;
}

BinaryCode getUnsignedCode( unsigned n )
{ BinaryCode answer;
  answer.push_back( n%2 );  // start with least sig bit
  n = n / 2;
  while ( n > 0 )
  { BCIterator bcIt = answer.begin();
    answer.insert( bcIt , n % 2 );
    n = n / 2;
  }
  return answer;
}

void print( BinaryCode bc )
{ for ( BCIterator bcIt = bc.begin(); bcIt != bc.end(); bcIt++ )
    cout << *bcIt << ' ';
}

int signedValue( BinaryCode bc )
{ int uvalue = unsignedValue( bc );
  if ( *bc.begin() == 0 ) return uvalue;  // not negative
  int modulus = (int) pow( 2 , bc.size() );
  return uvalue - modulus;
}

BinaryCode getSignedCode( int n )
{ BinaryCode answer;
  if ( n >= 0 )   // n not negative
  { answer = getUnsignedCode( n );
    BCIterator bcIt = answer.begin();
    answer.insert( bcIt , 0 );  // insert leading bit 0
```

```
  }
  else               // n is negative
  { int posN = -n;
    int modulus = 2;
    while ( posN > 0 )  // build modulus
    { posN /= 2;
      modulus *= 2;
    }
    answer = getUnsignedCode( modulus + n );
  }
  return answer;
}

void testSigned()
{ BinaryCode bcPos;
  BinaryCode bcNeg;
  for ( int n = 1; n <= 12; n++ )
  { bcPos = getSignedCode(  n );
    bcNeg = getSignedCode( -n );
    int decodePos = signedValue( bcPos );
    int decodeNeg = signedValue( bcNeg );
    cout << decodePos << ": ";
    print( bcPos );
    cout << "\tvs\t\t" << decodeNeg << ": ";
    print( bcNeg );
    cout << endl;
  }
}
```

Chapter 15

Container Classes

15.1 ANSI/ISO STANDARD C++

The standardization of C++ by the ANSI (American National Standards Institute) and the ISO (International Standards Organization) began in 1989. The final version was approved by those organizations in 1998. That approval defines *Standard C++*.

You can obtain a complete copy of the standard from ANSI at their website:

`http://www.ansi.org/`

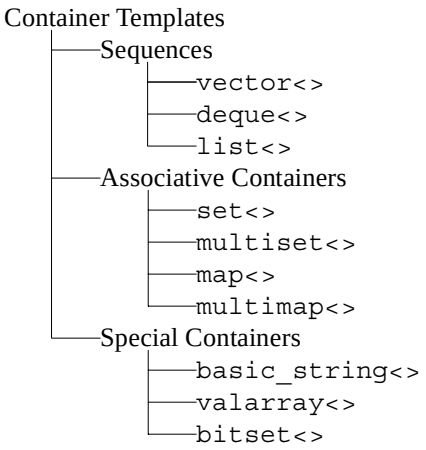
The title of the document is *Information Technology — Programming Languages — C++*.

15.2 THE STANDARD TEMPLATE LIBRARY

The standardization of C++ brought forth many changes, including namespaces and an official **`bool`** type. But the biggest improvement was the addition of the Standard Template Library (the STL). This is a collection of class templates and functions designed to facilitate the use of container objects such as strings, vectors, lists, stacks, queues, sets, and maps. Developed by a team led by Alexander Stepanov at Hewlett-Packard, the STL is now known simply as part of the Standard C++ Library. The classes that can be defined from these templates are called *container classes*.

15.3 STANDARD C++ CONTAINER CLASS TEMPLATES

The ten Standard C++ container class templates are organized as shown at right. The details of these class templates are given in Appendix C.



A *container* is data structure that contains other objects. The objects that it contains are called its *elements*. All the elements in a given container must have the same type.

A *sequence container* is a container whose elements are kept in an ordinal sequence, like an array. The position of each element is independent of its value. But the relative positions of the elements are guaranteed not to change unless they are intentionally moved. As the diagram shows, there are three general sequence containers: `vector`, `deque`, and `list`.

An *associative container* is a container whose elements are kept in sorted order. So the user has no control over where the elements are kept; their positions are completely determined by their values and those of the other elements in the container. So the order in which you insert the

elements doesn't matter. As the diagram shows, there are four general sequence containers: `set`, `multiset`, `map`, and `multimap`.

The Standard C++ Library also defines three specialized container class templates: `basic_string`, `valarray`, and `bitset`. These are not classified as general containers because their operations are not as general as the others.

The **`vector<>`** template is the prototype of all the container classes. It generalizes the direct access array, as described in Chapter 10. Most of its functions apply to the other templates.

The **`vector<>`** template is outlined in Chapter 14.

The **`deque<>`** template generalizes the stack and the queue containers. A *deque* (pronounced "deck") is a sequential container that allows insertions and deletions at both ends. Special adapters are provided that use this template to define the **`stack<>`** template and the **`queue<>`** template.

The **`list<>`** template generalizes the linked list structure which does not have indexed access but does have much faster insertion and deletion operations. A special adapter uses the list<> template to define the **`priority_queue<>`** template.

The **`set<>`** template provides containers that represent mathematical sets, using union and intersection operations.

The **`multiset<>`** template is the same as the **`set<>`** template except that its containers allow multiple copies elements.

The **`map<>`** template generalizes the look-up table structure. Maps are also called an *associative array.* The hash table data structure is a special kind of map.

The **`multimap<>`** template is the same as the **`map<>`** template except that its containers allow multiple copies elements.

The **`basic_string<>`** template generalizes the notion of a character string, allowing strings of any type. The common special cases are defined by **`typedef`**s:

```
typedef basic_string<char> string;
typedef basic_string<wchar_t> wstring;
```

The **`valarray<>`** template is intended for instantiating mathematical vectors and linear array processing.

The **`bitset<>`** template is used for processing bitstrings: objects whose values are usually in hexadecimal and which are operated upon by the logical operators `|`, `&`, `^`, `<<`, and `>>`.

15.4 STANDARD C++ GENERIC ALGORITHMS

The Standard C++ generic algorithms are non-member functions that apply to the Standard C++ container classes. They provide a consistent suite of tools that cover just about any application of containers. They also allow for easy transfer of elements from one type of container to another. The details of these functions are given in Appendix D.

Two of the most useful algorithms are the `find()` and `sort()` functions. These were illustrated in Chapter 14. (See Examples 14.4 and 14.9.) These are illustrated with other containers in the examples in this chapter.

15.5 HEADER FILES

The Standard C++ container templates and generic algorithms are defined in the following header files:

accumulate()	<numeric>
adjacent_difference()	<numeric>
adjacent_find()	<algorithm>
basic_string<>	<string>
binary_search()	<algorithm>
bitset<>	<bitset>
copy()	<algorithm>
copy_backward()	<algorithm>
count()	<algorithm>
count_if()	<algorithm>
deque<>	<deque>
equal()	<algorithm>
equal_find()	<algorithm>
fill()	<algorithm>
fill_n()	<algorithm>
find_end()	<algorithm>
find_first_of()	<algorithm>
find_if()	<algorithm>
for_each()	<algorithm>
generate()	<algorithm>
generate_n()	<algorithm>
includes()	<algorithm>
inner_product()	<numeric>
inplace_merge()	<algorithm>
iter_swap()	<algorithm>
lexicographic_compare()	<algorithm>
list<>	<list>
lower_bound()	<algorithm>
make_heap()	<algorithm>
map<>	<map>
max()	<algorithm>
max_element()	<algorithm>
merge()	<algorithm>
min()	<algorithm>
min_element()	<algorithm>
mismatch()	<algorithm>
multimap<>	<map>
multiset<>	<set>
next_permutation()	<algorithm>
nth_element()	<algorithm>
partial_sort()	<algorithm>
partial_sum()	<numeric>
partition()	<algorithm>
partition_sort_copy()	<algorithm>
pop_heap()	<algorithm>
prev_permutation()	<algorithm>
priority_queue<>	<queue>

push_heap()	<algorithm>
queue<>	<queue>
random_shuffle()	<algorithm>
remove_copy()	<algorithm>
remove_copy_if()	<algorithm>
remove_if()	<algorithm>
replace_()	<algorithm>
replace_copy()	<algorithm>
replace_copy_if()	<algorithm>
replace_if()	<algorithm>
reverse()	<algorithm>
reverse_copy()	<algorithm>
rotate()	<algorithm>
rotate_copy()	<algorithm>
search_n()	<algorithm>
set<>	<set>
set_difference()	<algorithm>
set_intersection()	<algorithm>
set_symmetric_difference()	<algorithm>
set_union()	<algorithm>
sort()	<algorithm>
sort_heap()	<algorithm>
stack<>	<stack>
string<>	<vector>
swap()	<algorithm>
transform()	<algorithm>
unique()	<algorithm>
unique_copy()	<algorithm>
upper_bound()	<algorithm>
valarray<>	<valarray>
vector<>	<vector>

For more information on the Standard C++ container classes and their generic algorithms, see the books **[Hubbard1]** and **[Hubbard2]** listed in Appendix H.

Appendix A

Character Codes

A.1 The ASCII Code

Each 8-bit character is stored as its ASCII[1] Code, which is an integer in the range 0 to 127. Note that the first 32 characters are *nonprinting characters*, so their symbols in the first column are indicated either with their control sequence or with their escape sequence. The *control sequence* of a nonprinting character is the combination of **Control** key and another key that is pressed on the keyboard to enter the character. For example, the *end-of-file character* (ASCII code 4) is entered with the `Ctrl-D` sequence. The *escape sequence* of a nonprinting character is the combination of the backslash character "`\`" (called the "control character") and a letter that is typed in C++ source code to indicate the character. For example, the *newline character* (ASCII code 10) is written "`\n`" in a C++ program.

Character	Description	Decimal	Octal	Hex	Binary
`Ctrl-@`	Null, end of string	0	000	0x0	0
`Ctrl-A`	Start of heading	1	001	0x1	1
`Ctrl-B`	Start of text	2	002	0x2	10
`Ctrl-C`	End of text	3	003	0x3	11
`Ctrl-D`	End of transmission, end of file	4	004	0x4	100
`Ctrl-E`	Enquiry	5	005	0x5	101
`Ctrl-F`	Acknowledge	6	006	0x6	110
`\a`	Bell, alert, system beep	7	007	0x7	111
`\b`	Backspace	8	010	0x8	1000
`\t`	Horizontal tab	9	011	0x9	1001
`\n`	Line feed, new line	10	012	0xa	1010
`\v`	Vertical tab	11	013	0xb	1011
`\f`	Form feed, new page	12	014	0xc	1100
`\r`	Carriage return	13	015	0xd	1101
`Ctrl-N`	Shift out	14	016	0xe	1110
`Ctrl-O`	Shift in	15	017	0xf	1111
`Ctrl-P`	Data link escape	16	020	0x10	10000
`Ctrl-Q`	Device control 1, resume scroll	17	021	0x11	10001
`Ctrl-R`	Device control 2	18	022	0x12	10010
`Ctrl-S`	Device control 3, stop scroll	19	023	0x13	10011

1. ASCII is an acronym for the American Standard Code for Information Interchange.

Character	Description	Decimal	Octal	Hex	Binary
Ctrl-T	Device control 4	20	024	0x14	10100
Ctrl-U	Negative acknowledgment	21	025	0x15	10101
Ctrl-V	Synchronous idle	22	026	0x16	10110
Ctrl-W	End transmission block	23	027	0x17	10111
Ctrl-X	Cancel	24	030	0x18	11000
Ctrl-Y	End of message, interrupt	25	031	0x19	11001
Ctrl-Z	Substitute, exit	26	032	0x1a	11010
Ctrl-[	Escape	27	033	0x1b	11011
Ctrl-/	File separator	28	034	0x1c	11100
Ctrl-]	Group separator	29	035	0x1d	11101
Ctrl-^	Record separator	30	036	0x1e	11110
Ctrl-_	Unit separator	31	037	0x1f	11111
	Blank, space	32	040	0x20	100000
!	Exclamation point	33	041	0x21	100001
"	Quotation mark, double quote	34	042	0x22	100010
#	Hash mark, number sign	35	043	0x23	100011
$	Dollar sign	36	044	0x24	100100
%	Percent sign	37	045	0x25	100101
&	Ampersand	38	046	0x26	100110
'	Apostrophe, single quote	39	047	0x27	100111
(	Left parenthesis	40	050	0x28	101000
)	Right parenthesis	41	051	0x29	101001
*	Asterisk, star, times	42	052	0x2a	101010
+	Plus	43	053	0x2b	101011
,	Comma	44	054	0x2c	101100
-	Dash, minus	45	055	0x2d	101101
.	Dot, period, decimal point	46	056	0x2e	101110
/	Slash	47	057	0x2f	101111
0	Digit zero	48	060	0x30	110000
1	Digit one	49	061	0x31	110001
2	Digit two	50	062	0x32	110010
3	Digit three	51	063	0x33	110011
4	Digit four	52	064	0x34	110100
5	Digit five	53	065	0x35	110101
6	Digit six	54	066	0x36	110110
7	Digit seven	55	067	0x37	110111
8	Digit eight	56	070	0x38	111000

Character	Description	Decimal	Octal	Hex	Binary
9	Digit nine	57	071	0x39	111001
:	Colon	58	072	0x3a	111010
;	Semicolon	59	073	0x3s	111011
<	Less than	60	074	0x3c	111100
=	Equal to	61	075	0x3d	111101
>	Greater than	62	076	0x3e	111110
?	Question mark	63	077	0x3f	111111
@	Commercial at sign	64	0100	0x40	1000000
A	Letter capital A	65	0101	0x41	1000001
B	Letter capital B	66	0102	0x42	1000010
C	Letter capital C	67	0103	0x43	1000011
D	Letter capital D	68	0104	0x44	1000100
E	Letter capital E	69	0105	0x45	1000101
F	Letter capital F	70	0106	0x46	1000110
G	Letter capital G	71	0107	0x47	1000111
H	Letter capital H	72	0110	0x48	1001000
I	Letter capital I	73	0111	0x49	1001001
J	Letter capital J	74	0112	0x4a	1001010
K	Letter capital K	75	0113	0x4b	1001011
L	Letter capital L	76	0114	04xc	1001100
M	Letter capital M	77	0115	0x4d	1001101
N	Letter capital N	78	0116	0x4e	1001110
O	Letter capital O	79	0117	0x4f	1001111
P	Letter capital P	80	0120	0x50	1010000
Q	Letter capital Q	81	0121	0x51	1010001
R	Letter capital R	82	1022	0x52	1010010
S	Letter capital S	83	0123	0x53	1010011
T	Letter capital T	84	0124	0x54	1010100
U	Letter capital U	85	0125	0x55	1010101
V	Letter capital V	86	0126	0x56	1010110
W	Letter capital W	87	0127	0x57	1010111
X	Letter capital X	88	0130	0x58	1011000
Y	Letter capital Y	89	0131	0x59	1011001
Z	Letter capital Z	90	0132	0x5a	1011010
[	Left bracket	91	0133	0x5b	1011011
\	Backslash	92	0134	0x5c	1011100
]	Right bracket	93	0135	0x5d	1011101

Character	Description	Decimal	Octal	Hex	Binary
^	Caret	94	0136	0x5e	1011110
_	Underscore	95	0137	0x5f	1011111
`	Accent grave	96	0140	0x60	1100000
a	Letter lowercase A	97	0141	0x61	1100001
b	Letter lowercase B	98	0142	0x62	1100010
c	Letter lowercase C	99	0143	0x63	1100011
d	Letter lowercase D	100	0144	0x64	1100100
e	Letter lowercase E	101	0145	0x65	1100101
f	Letter lowercase F	102	0146	0x66	1100110
g	Letter lowercase G	103	0147	0x67	1100111
h	Letter lowercase H	104	0150	0x68	1101000
i	Letter lowercase I	105	0151	0x69	1101001
j	Letter lowercase J	106	0152	0x6A	1101010
k	Letter lowercase K	107	0153	0x6B	1101011
l	Letter lowercase L	108	0154	0x6C	1101100
m	Letter lowercase M	109	0155	0x6D	1101101
n	Letter lowercase N	110	0156	0x6	1101110
o	Letter lowercase O	111	0157	0x6F	1101111
p	Letter lowercase P	112	0160	0x70	1110000
q	Letter lowercase Q	113	0161	0x71	1110001
r	Letter lowercase R	114	0162	0x72	1110010
s	Letter lowercase S	115	0163	0x73	1110011
t	Letter lowercase T	116	0164	0x74	1110100
u	Letter lowercase U	117	0165	0x75	1110101
v	Letter lowercase V	118	0166	0x76	1110110
w	Letter lowercase W	119	0167	0x77	1110111
x	Letter lowercase X	120	0170	0x78	1111000
y	Letter lowercase Y	121	0171	0x79	0111001
z	Letter lowercase Z	122	0172	0x7a	1111010
{	Left brace	123	0173	0x7b	1111011
|	Pipe	124	0174	0x7c	1111100
}	Right brace	125	0175	0x7d	1111101
~	Tilde	126	0176	0x7e	1111110
Delete	Delete, rub out	127	0177	0x7f	1111111

A.2 Unicode

Unicode is the international standardized character set that C++ uses for its 16-bit `wchar_t` (wide character) type. Each code is a 16-bit integer with unique value in the range 0 to 65,535. These values are usually expressed in hexadecimal form. (See Appendix G.) For example, the infinity symbol ∞ has the Unicode value 8734, which is `0x0000221e` in hexadecimal.

In C++, the character literal whose Unicode is `0x0000hhhh` in hexadecimal is denoted `L'\xhhhh'`. For example, the infinity symbol is expressed as `L'\x221e'`, like this:

```
wchar_t infinity = L'\x221e';
```

The first 127 Unicode values encode the same characters as the ASCII Code.

The following table summarizes the various alphabets and their Unicodes.

You can obtain more information from the Unicode Consortium website

```
http://www.unicode.org/
```

Also, see the book **[Unicode]** listed in Appendix H.

Range (Hexadecimal)	Alphabet
\u0000 – \u024F	Latin Alphabets
\u0370 – \u03FF	Greek
\u0400 – \u04FF	Cyrillic
\u0530 – \u058F	Armenian
\u0590 – \u05FF	Hebrew
\u0600 – \u06FF	Arabic
\u0900 – \u097F	Devanagari
\u0980 – \u09FF	Bengali
\u0A00 – \u0A7F	Gurmukhi
\u0A80 – \u0AFF	Gujarati
\u0B00 – \u0B7F	Oriya
\u0B80 – \u0BFF	Tamil
\u0C00 – \u0C7F	Teluga
\u0C80 – \u0CFF	Kannada
\u0D00 – \u0D7F	Malayam
\u0E00 – \u0E7F	Thai
\u0E80 – \u0EFF	Lao
\u0F00 – \u0FBF	Tibetan
\u10A0 – \u10FF	Georgian
\u1100 – \u11FF	Hangul Jamo
\u2000 – \u206F	Punctuation
\u2070 – \u209F	Superscripts and subscripts
\u20A0 – \u20CF	Currency symbols
\u20D0 – \u20FF	Diacritical marks
\u2100 – \u214F	Letterlike symbols

Range (Hexadecimal)	Alphabet
\u2150 – \u218F	Numeral forms
\u2190 – \u21FF	Arrows
\u2200 – \u22FF	Mathematical symbols
\u2300 – \u23FF	Miscellaneous technical symbols
\u2400 – \u243F	Control pictures
\u2440 – \u245F	Optical Character Recognition symbols
\u2460 – \u24FF	Enclosed alphanumerics
\u2500 – \u257F	Box drawing
\u2580 – \u259F	Block elements
\u25A0 – \u25FF	Geometric shapes
\u2700 – \u27BF	Dingbats
\u3040 – \u309F	Hiragana
\u30A0 – \u30FF	Katakana
\u3100 – \u312F	Bopomofo
\u3130 – \u318F	Jamo
\u3190 – \u319F	Kanbun
\u3200 – \u32FF	Enclosed CJK letters and months
\u4E00 – \u9FFF	CJK Ideographs

Appendix B

Standard C++ Keywords

Keyword	Description	Example
and	A synonym for the AND operator &&	`(x>0 and x<8)`
and_eq	A synonym for the bitwise AND assignment operator &=	`b1 and_eq b2;`
asm	Allows information to be passed to the assembler directly	`asm ("check");`
auto	Storage class for objects that exist only within their own block	`auto int n;`
bitand	A synonym for the bitwise AND operator &	`b0 = b1 bitand b2;`
bitor	A synonym for the bitwise OR operator |	`b0 = b1 bitor b2;`
bool	A boolean type	`bool flag;`
break	Terminates a loop or `a switch` statement	`break;`
case	Used in `a switch` statement to specify control expression	`switch (n/10)`
catch	Specifies actions to take when an exception occurs	`catch(error)`
char	An integer type	`char c;`
class	Specifies a class declaration	`class X { ... };`
compl	A synonym for the bitwise NOT operator ~	`b0 = compl b1;`
const	Specifies a constant definition	`const int s = 32;`
const_cast	Used to change objects from within immutable member functions	`pp = const_cast<T*>(p)`
continue	Jumps to beginning of next iteration in a loop	`continue;`
default	The "otherwise" case in `a switch` statement	`default: sum = 0;`
delete	Deallocates memory allocated by `a new` statement	`delete a;`
do	Specifies `a do..while` loop	`do {...} while ...`
double	A real number type	`double x;`
dynamic_cast	Returns a `T*` pointer for a given pointer	`pp = dynamic_cast<T*>p`
else	Specifies alternative in an `if` statement	`else n = 0;`
enum	Used to declare an enumeration type	`enum bool { ... };`
explicit	Used to prevent a constructor from being invoked implicitly	`explicit X(int n);`
export	Allows access from another compilation unit	`export template<class T>`
extern	Storage class for objects declared outside the local block	`extern int max;`

Keyword	Description	Example
false	One of the two literals for the `bool` type	`bool flag=false;`
float	A real number type	`float x;`
for	Specifies a `for` loop	`for ( ; ; ) ...`
friend	Specifies a `friend` function in a class	`friend int f();`
goto	Causes execution to jump to a labeled statement	`goto error;`
if	Specifies an `if` statement	`if (n > 0) ...`
inline	Declares a function whose text is to be substituted for its call	`inline int f();`
int	An integer type	`int n;`
long	Used to define integer and real types	`long double x;`
mutable	Allows immutable functions to change the field	`mutable string ssn;`
namespace	Allows the identification of scope blocks	`namespace Best { int num; }`
new	Allocates memory	`int* p = new int;`
not	A synonym for the NOT operator `!`	`(not(x==0))`
not_eq	A synonym for the inequality operator `!=`	`(x not_eq 0)`
operator	Used to declare an operator overload	`X operator++();`
or	A synonym for the OR operator `||`	`(x>0 or x<8)`
or_eq	A synonym for the bitwise OR assignment operator `|=`	`b1 or_eq b2;`
private	Specifies `private` declarations in a class	`private: int n;`
protected	Specifies `protected` declarations in a class	`protected: int n;`
public	Specifies `public` declarations in a class	`public: int n;`
register	Storage class specifier for objects stored in registers	`register int i;`
reinterpret_cast	Returns an object with given value and type	`pp = reinterpret_cast<T*>(p)`
return	Statement that terminates a function and returns a value	`return 0;`
short	An integer type	`short n;`
signed	Used to define integer types	`signed char c;`
sizeof	Operator that returns the number of bytes used to store an object	`n = sizeof(float);`

Keyword	Description	Example
static	Storage class of objects that exist for the duration of the program	`static int n;`
static_cast	Returns a `T*` pointer for a given pointer	`pp = static_cast<T*>p`
struct	Specifies a structure definition	`struct X { ... };`
switch	Specifies a `switch` statement	`switch (n) { ... }`
template	Specifies a `template` class	`template <class T>`
this	Pointer that points to the current object	`return *this;`
throw	Used to generate an exception	`throw X();`
true	One of the two literals for the `bool` type	`bool flag=true;`
try	Specifies a block that contains exception handlers	`try { ... }`
typedef	Declares a synonym for an existing type	`typedef int Num;`
typeid	Returns an object that represents an expression's type	`cout << typeid(x).name();`
typename	A synonym for the keyword `class`	`typename X { ... };`
using	Directive that allows omission of namespace prefix	`using namespace std;`
union	Specifies a structure whose elements occupy the same storage	`union z { ... };`
unsigned	Used to define integer types	`unsigned int b;`
virtual	Declares a member function that is defined in a subclass	`virtual int f();`
void	Designates the absence of a type	`void f();`
volatile	Declares objects that can be modified outside of program control	`int volatile n;`
wchar_t	Wide (16-bit) character type	`wchar_t province;`
while	Specifies a `while` loop	`while (n > 0) ...`
xor	A synonym for the bitwise exclusive OR operator ^	`b0 = b1 xor b2;`
xor_eq	A synonym for the bitwise exclusive OR assignment operator ^=	`b1 xor_eq b2;`

Appendix C

Standard C++ Operators

This table lists all the operators in C++, grouping them by order of precedence. The higher-level precedence operators are evaluated before the lower-level precedence operators. For example, in the expression `(a - b*c)`, the `*` operator will be evaluated first and the `-` operator second, because `*` has precedence level 13 which is higher than the level 12 precedence of `-`. The column labeled "Assoc." tells whether an operator is right associative or left associative. For example, the expression `(a - b - c)` is evaluated as `((a - b) - c)` because `-` is left associative. The column labeled "Arity" tells whether an operator operates on one, two, or three operands (unary, binary, or ternary). The column labeled "Ovrldbl." tells whether an operator is overloadable. (See Chapter 8.)

Op.	Name	Prec.	Assoc.	Arity	Ovrldbl.	Example
`::`	Global scope resolution	17	Right	Unary	No	`::x`
`::`	Class scope resolution	17	Left	Binary	No	`X::x`
`.`	Direct member selection	16	Left	Binary	No	`s.len`
`->`	Indirect member selection	16	Left	Binary	Yes	`p->len`
`[]`	Subscript	16	Left	Binary	Yes	`a[i]`
`()`	Function call	16	Left	n/a	Yes	`rand()`
`()`	Type construction	16	Left	n/a	Yes	`int(ch)`
`++`	Post-increment	16	Right	Unary	Yes	`n++`
`--`	Post-decrement	16	Right	Unary	Yes	`n--`
`sizeof`	Size of object or type	15	Right	Unary	No	`sizeof(a)`
`++`	Pre-increment	15	Right	Unary	Yes	`++n`
`--`	Pre-decrement	15	Right	Unary	Yes	`--n`
`~`	Bitwise complement	15	Right	Unary	Yes	`~s`
`!`	Logical NOT	15	Right	Unary	Yes	`!p`
`+`	Unary plus	15	Right	Unary	Yes	`+n`
`-`	Unary minus	15	Right	Unary	Yes	`-n`
`*`	Dereference	15	Right	Unary	Yes	`*p`
`&`	Address	15	Right	Unary	Yes	`&x`
`new`	Allocation	15	Right	Unary	Yes	`new p`
`delete`	Deallocation	15	Right	Unary	Yes	`delete p`
`()`	Type conversion	15	Right	Binary	Yes	`int(ch)`
`.*`	Direct member selection	14	Left	Binary	No	`x.*q`
`->*`	Indirect member selection	14	Left	Binary	Yes	`p->q`
`*`	Multiplication	13	Left	Binary	Yes	`m*n`
`/`	Division	13	Left	Binary	Yes	`m/n`
`%`	Remainder	13	Left	Binary	Yes	`m%n`

Op.	Name	Prec.	Assoc.	Arity	Ovrldbl.	Example
`+`	Unary plus	15	Right	Unary	Yes	`+n`
`-`	Unary minus	15	Right	Unary	Yes	`-n`
`*`	Dereference	15	Right	Unary	Yes	`*p`
`&`	Address	15	Right	Unary	Yes	`&x`
`new`	Allocation	15	Right	Unary	Yes	`new p`
`delete`	Deallocation	15	Right	Unary	Yes	`delete p`
`()`	Type conversion	15	Right	Binary	Yes	`int(ch)`
`.*`	Direct member selection	14	Left	Binary	No	`x.*q`
`->*`	Indirect member selection	14	Left	Binary	Yes	`p->q`
`*`	Multiplication	13	Left	Binary	Yes	`m*n`
`/`	Division	13	Left	Binary	Yes	`m/n`
`%`	Remainder	13	Left	Binary	Yes	`m%n`
`+`	Addition	12	Left	Binary	Yes	`m + n`
`-`	Subtraction	12	Left	Binary	Yes	`m - n`
`<<`	Bit shift left	11	Left	Binary	Yes	`cout << n`
`>>`	Bit shift right	11	Left	Binary	Yes	`cin >> n`
`<`	Less than	10	Left	Binary	Yes	`x < y`
`<=`	Less than or equal to	10	Left	Binary	Yes	`x <= y`
`>`	Greater than	10	Left	Binary	Yes	`x > y`
`>=`	Greater than or equal to	10	Left	Binary	Yes	`x >= y`
`==`	Equal to	9	Left	Binary	Yes	`x == y`
`!=`	Not equal to	9	Left	Binary	Yes	`x != y`
`&`	Bitwise AND	8	Left	Binary	Yes	`s&t`
`^`	Bitwise XOR	7	Left	Binary	Yes	`s^t`
`|`	Bitwise OR	6	Left	Binary	Yes	`s|t`
`&&`	Logical AND	5	Left	Binary	Yes	`u && v`
`||`	Logical OR	4	Left	Binary	Yes	`u || v`
`?:`	Conditional expression	3	Left	Ternary	No	`u ? x : y`
`=`	Assignment	2	Right	Binary	Yes	`n = 22`
`+=`	Addition assignment	2	Right	Binary	Yes	`n += 8`
`-=`	Subtraction assignment	2	Right	Binary	Yes	`n -= 4`
`*=`	Multiplication assignment	2	Right	Binary	Yes	`n *= -1`
`/=`	Division assignment	2	Right	Binary	Yes	`n /= 10`
`%=`	Remainder assignment	2	Right	Binary	Yes	`n %= 10`
`&=`	Bitwise AND assignment	2	Right	Binary	Yes	`s &= mask`
`^=`	Bitwise XOR assignment	2	Right	Binary	Yes	`s ^= mask`
`|=`	Bitwise OR assignment	2	Right	Binary	Yes	`s |= mask`
`<<=`	Bit shift left assignment	2	Right	Binary	Yes	`s <<= 1`
`>>=`	Bit shift right assignment	2	Right	Binary	Yes	`s >>= 1`
`,`	Comma	0	Left	Binary	Yes	`++m,--n`

Appendix D

Standard C++ Container Classes

This appendix summarizes the standard C++ container class templates and their most widely used member functions. This is the part of standard C++ that used to be called the Standard Template Library (STL).

D.1 THE `vector` CLASS TEMPLATE

A `vector` object acts like an array with index range checking (using its `at()` member function). As an object, it has the additional advantages over an array of being able to be assigned, passed by value, and returned by value. The `vector` class template is defined in the `<vector>` header. See Example D.1 on page 355.

```
vector();
// default constructor: creates an empty vector;

vector(const vector& v);
// copy constructor: creates a copy of the vector v;
// postcondition: *this == v;

vector(unsigned n, const T& x=T());
// constructor: creates a vector containing n copies of the element x;
// precondition: n >= 0;
// postcondition: size() == n;

~vector();
// destructor: destroys this vector;

vector& operator=(const vector& v);
// assignment operator: assigns v to this vector, making it a duplicate;
// postcondition: *this == v;

unsigned size() const;
// returns the number of elements in this vector;

unsigned capacity() const;
// returns the maximum number of elements that this vector can have
// without being reallocated;

void reserve(unsigned n);
// reallocates this vector to a capacity of n elements;
// precondition: capacity() <= n;
// postcondition: capacity() == n;
```

```
bool empty() const;
// returns true iff size() == 0;

void assign(unsigned n, const T& x=T());
// clears this vector and then inserts n copies of the element x;
// precondition: n >= 0;
// postcondition: size() == n;

T& operator[](unsigned i);
// returns element number i;
// precondition: 0 <= i < size();
// result is unpredictable if precondition is false;

T& at(unsigned i);
// returns element number i;
// precondition: 0 <= i < size();
// exception is thrown is precondition is false;

T& front();
// returns the first element of this vector;

T& back();
// returns the last element of this vector;

iterator begin();
// returns an iterator pointing to the first element of this vector;

iterator end();
// returns an iterator pointing to the dummy element that follows
// the last element of this vector;

reverse_iterator rbegin();
// returns a reverse iterator pointing to the last element of this vector;

reverse_iterator rend();
// returns a reverse iterator pointing to the dummy element that precedes
// the first element of this vector;

void push_back(const T& x);
// appends a copy of the element x to the back of this vector;
// postcondition: back() == x;
// postcondition: size() has been incremented;

void pop_back();
// removes the last element of this vector;
// precondition: size() > 0;
// postcondition: size() has been decremented;
```

```
iterator insert(iterator p, const T& x);
// inserts a copy of the element x at position p; returns p;
// precondition: begin() <= p <= end();
// postcondition: size() has been incremented;

iterator erase(iterator p);
// removes the element at position p; returns p
// precondition: begin() <= p <= end();
// postcondition: size() has been decremented;

iterator erase(iterator p1, iterator p2);
// removes the elements from position p1 to the position before p2;
// returns p1;
// precondition: begin() <= p1 <= p2 <= end();
// postcondition: size() has been decreased by int(p2-p1);

void clear();
// removes all the elements from this vector;
// postcondition: size() == 0;
```

EXAMPLE D.1 Using an Iterator on a `vector` Object

```
#include <iostream>
#include <vector>
using namespace std;
typedef vector<int>::iterator It;

int main()
{ vector<int> v(4);
  for (int i=0; i<4; i++)
    v[i] = 222*i + 333;
  cout << "Using the iterator it in a for loop:\n";
  for (It it=v.begin(); it!=v.end(); it++)
    cout << "\t*it=" << *it << "\n";
  cout << "Using the iterator p in a while loop:\n";
  It p=v.begin();
  while(p!=v.end())
    cout << "\t*p++=" << *p++ << "\n";
}
```

```
Using the iterator it in a for loop:
        *it=333
        *it=555
        *it=777
        *it=999
Using the iterator p in a while loop:
        *p++=333
        *p++=555
        *p++=777
        *p++=999
```

The vector `v` has 4 elements: 333, 555, 777, and 999. The second `for` loop uses the iterator `it` to traverse the vector `v` from beginning to end, accessing each of its elements with `*it`. The `while` loop has the same effect using `*p`.

EXAMPLE D.2 Using a Reverse Iterator on a `vector` Object

```
#include <iostream>
#include <vector>
using namespace std;
typedef vector<int>::reverse_iterator RIt;

int main()
{ vector<int> v(4);
  for (int i=0; i<4; i++)
    v[i] = 222*i + 333;
  cout << "Using the reverse iterator rit in a for loop:\n";
  for (RIt rit=v.rbegin(); rit!=v.rend(); rit++)
    cout << "\t*rit=" << *rit << "\n";
  cout << "Using the reverse iterator rp in a while loop:\n";
  RIt rp=v.rbegin();
  while(rp!=v.rend())
    cout << "\t*rp++=" << *rp++ << "\n";
}
```

```
Using the reverse iterator rit in a for loop:
        *rit=999
        *rit=777
        *rit=555
        *rit=333
Using the reverse iterator rp in a while loop:
        *rp++=999
        *rp++=777
        *rp++=555
        *rp++=333
```

The vector `v` has 4 elements: 333, 555, 777, and 999 (the same as in Example D.1). The second `for` loop uses the reverse iterator `rit` to traverse the vector `v` backwards, accessing each of its elements with `*rit`. The `while` loop has the same effect using `*rp`.

EXAMPLE D.3 Using the `insert()` Function on a `vector` Object

```
#include <iostream>
#include <vector>
using namespace std;
typedef vector<int> Vector;
typedef Vector::iterator It;
void print(const Vector&);

int main()
{ Vector v(4);
  for (int i=0; i<4; i++)
    v[i] = 222*i + 333;
  print(v);
  It it = v.insert(v.begin()+2,666);
  print(v);
  cout << "*it=" << *it << "\n";
}
```

```
void print(const Vector& v)
{ cout << "size=" << v.size() << ": (" << v[0];
  for (int i=1; i<v.size(); i++)
    cout << "," << v[i];
  cout << ")\n";
}
size=4: (333,555,777,999)
size=5: (333,555,666,777,999)
*it=666
```

The vector `v` has 4 elements: 333, 555, 777, and 999 (the same as in Example D.1). The second `for` loop uses the reverse iterator `rit` to traverse the vector `v` backwards, accessing each of its elements with `*rit`. The `while` loop has the same effect using `*rp`.

EXAMPLE D.4 Using Some Generic Algorithms on a `vector` Object

```
#include <iostream>
#include <vector>
using namespace std;
typedef vector<int> Vector;
typedef Vector::iterator It;
void print(const Vector&);

int main()
{ Vector v(9);
  for (int i=0; i<9; i++)
    v[i] = 111*i + 111;
  print(v);
  It it=v.begin();
  fill(it+2,it+5,400);  // replaces v[2:5] with 400
  print(v);
  reverse(it+4,it+7);   //
  print(v);
  iter_swap(it+6,it+8);
  print(v);
  sort(it+4,it+9);
  print(v);
}

void print(const Vector& v)
{ cout << "size=" << v.size() << ": (" << v[0];
  for (int i=1; i<v.size(); i++)
    cout << "," << v[i];
  cout << ")\n";
}
size=9: (111,222,333,444,555,666,777,888,999)
size=9: (111,222,400,400,400,666,777,888,999)
size=9: (111,222,400,400,777,666,400,888,999)
size=9: (111,222,400,400,777,666,999,888,400)
size=9: (111,222,400,400,400,666,777,888,999)
```

EXAMPLE D.5 Using Some More Generic Algorithms on a `vector` Object

```
#include <iostream>
#include <vector>
using namespace std;
typedef vector<int> Vector;
typedef Vector::iterator It;
void print(const Vector&);

int main()
{ Vector v1(9);
  for (int i=0; i<9; i++)
    v1[i] = 111*i + 111;
  print(v1);
  Vector v2(9);
  print(v2);
  It p1=v1.begin(), p2=v2.begin();
  copy(p1+3,p1+8,p2+3);
  print(v2);
  It p = min_element(p1+4,p1+8);
  cout << "*p=" << *p << "\n";
  p = max_element(p1+4,p1+8);
  cout << "*p=" << *p << "\n";
  p = find(p1,p1+9,444);
  if (p != p1+9) cout << "*p=" << *p << "\n";
}

void print(const Vector& v)
{ cout << "size=" << v.size() << ": (" << v[0];
  for (int i=1; i<v.size(); i++)
    cout << "," << v[i];
  cout << ")\n";
}
size=9: (111,222,333,444,555,666,777,888,999)
size=9: (0,0,0,0,0,0,0,0,0)
size=9: (0,0,0,444,555,666,777,888,0)
*p=555
*p=888
*p=444
```

D.2 THE `deque` CLASS TEMPLATE

A `deque` (pronounced “deck”) object is a double-ended queue, intended to provide efficient insertion and deletion at both its beginning and its end. It has the following two member functions in addition to all the member functions that a `vector` class has (except the `capacity()` and `reserve()` functions). The `deque` class template is defined in the `<deque>` header.

```
void push_front(const T& x);
// inserts a copy of the element x at the front of this deque;
// postcondition: front() == x;
// postcondition: size() has been incremented;

void pop_front();
// removes the first element of this vector;
// precondition: size() > 0;
// postcondition: size() has been decremented;
```

D.3 THE `stack` CLASS TEMPLATE

A `stack` object is a sequential container that allows insertions and deletions only at one end, called its *top*. In the standard C++ library, the `stack` class template is adapted from the `deque` class template. This means that `stack` member functions are implemented with `deque` member functions, as shown below. The `stack` class template is defined in the `<stack>` header.

```
template <class T> class stack
{ public:
    unsigned size() const { return _d.size(); }
    bool empty() const    { return _d.empty(); }
    T& top()              { return _d.back(); }
    void push(const T& x) { _d.push_back(x); }
    void pop()            { _d.pop_back(); }
  protected:
    deque<T> _d;
};
```

D.4 THE `queue` CLASS TEMPLATE

A `queue` object is a sequential container that allows insertions only at one end and deletions only at the other end. Like the `stack` class template, the `queue` class template is adapted from the `deque` class template in the standard C++ library. This means that `queue` member functions are implemented with `deque` member functions, as shown below. The `queue` class template is defined in the `<queue>` header.

```
template <class T> class stack
{ public:
    unsigned size() const { return _d.size(); }
    bool empty() const    { return _d.empty(); }
    T& front()            { return _d.front(); }
    T& back()             { return _d.back(); }
    void push(const T& x) { _d.push_back(x); }
    void pop()            { _d.pop_front(); }
  protected:
    deque<T> _d;
};
```

D.5 THE priority_queue CLASS TEMPLATE

A priority_queue object is a container that acts like a queue except that the order in which the elements are popped is determined by their priorities. This means that the operator<() function must be defined for the element type T. The priority_queue class template is defined in the <queue> header. See Example D.6 on page 360.

```
vector();
// constructs an empty vector;

vector(const vector& v);
// constructs a copy of the vector v;
// postcondition: *this == v;
```

EXAMPLE D.6 Using a priority_queue Object

```
#include <iostream>
#include <queue>
using namespace std;
int main()
{ priority_queue<string> pq;
  pq.push("Japan");
  pq.push("Japan");
  pq.push("Korea");
  pq.push("China");
  pq.push("India");
  pq.push("Nepal");
  pq.push("Qatar");
  pq.push("Yemen");
  pq.push("Egypt");
  pq.push("Zaire");
  pq.push("Libya");
  pq.push("Italy");
  pq.push("Spain");
  pq.push("Chile");
  while (!pq.empty())
  { cout << pq.top() << "\n";
    pq.pop();
  }
}
```

```
Zaire
Yemen
Spain
Qatar
Nepal
Libya
Korea
Japan
Japan
Italy
India
```

```
India
Egypt
China
Chile
```

The priority queue always maintains its highest priority element at the top (*i.e.*, the front) of the queue. Using the standard lexicographic ordering (*i.e.*, the dictionary ordering) of strings, that results in the names being accessed in reverse alphabetical order.

Note that `priority_queue` objects store duplicate elements.

D.6 THE `list` CLASS TEMPLATE

A `list` object is a sequential container that allows efficient insertion and deletion at any position in the sequence. It has the following member functions in addition to all the member functions that the `deque` class has (except the `operator[]()` and `at()` functions). The `list` class template is defined in the `<list>` header.

```
void splice(iterator p, list& l, iterator p1);
// moves the element from l at position p1 to this list at position p;
// precondition: p is a valid iterator on this list;
// precondition: p1 is a valid iterator on list l;

void splice(iterator p, list& l, iterator p1, iterator p2);
// moves the elements from l at positions [p1:p2-1] to this list
// beginning at position p;
// precondition: p is a valid iterator on this list;
// precondition: p1 and p2 are valid iterators on list l;
// precondition p1 < p2;

void remove(const T& x);
// removes from this list all elements that are equal to x;
// invariant: the order of all elements that are not removed;
// invariant: all iterators pointing to elements that are not removed;

void unique();
// removes from this list all duplicate elements;
// invariant: the order of all elements that are not removed;
// invariant: all iterators pointing to elements that are not removed;

void merge(list& l);
// merges all elements of list l into this list;
// precondition: both list l and this list are sorted;
// postcondition: size() in increased by l.size();
// postcondition: l.size() == 0;
// complexity: O(n);

void reverse();
// reverses the order of the elements of this list;
// invariant: size();
// complexity: O(n);
```

```
void sort();
// sorts the elements of this list;
// postcondition: this list is sorted;
// invariant: size();
// complexity: O(n*log(n));
```

EXAMPLE D.7 Sorting and Reversing a `list` Object

```
#include <iostream>
#include <list>
using namespace std;
typedef list<string> List;
typedef List::iterator It;
void print(List&);

int main()
{ List l;
  l.push_back("Kenya");
  l.push_back("Sudan");
  l.push_back("Egypt");
  l.push_back("Zaire");
  l.push_back("Libya");
  l.push_back("Congo");
  l.push_back("Ghana");
  print(l);
  l.sort();
  print(l);
  l.reverse();
  print(l);
}

void print(List& l)
{ cout << "\n";
  for (It it=l.begin(); it != l.end(); it++)
    cout << *it << "\n";
}
```

```
Kenya
Sudan
Egypt
Zaire
Libya
Congo
Ghana

Congo
Egypt
Ghana
Kenya
Libya
Sudan
Zaire
```

```
Zaire
Sudan
Libya
Kenya
Ghana
Egypt
Congo
```

D.7 THE map CLASS TEMPLATE

A `map` object (also called a *dictionary*, a *table*, or an *associative array*) acts like an array whose index can be any type that implements the `<` operator. A map is like a mathematical function that gives a unique *y*-value for each *x*-value. The *x*-value, called the *key* value, is the index. The *y*-value is the stored object that the key identifies.

An English language dictionary is an example of a map object. The key value is the word and its associated object is the dictionary's definition of the word.

Another standard example would be a database table of student records. The key value is the student identification number (*e.g.*, Social Security number), and its associated object is the data record for that student.

The `map` class template is defined in the `<map>` header. It has the same member functions as the `vector` class template.

EXAMPLE D.8 Using a map Object

```
#include <iostream>
#include <map.h>
using namespace std;
struct Country
{ friend ostream& operator<<(ostream&, const Country&);
  Country();
  Country(string, string, string, int, int);
  string abbr, capital, language;
  int population, area;
};
typedef map<string,Country> Map;
typedef Map::iterator It;
typedef pair<const string,Country> Pair;
void load(Map&);
void print(Map&);
void find(Map&, const string&);

int main()
{ Map map;
  load(map);
  print(map);
  find(map,"Cuba");
  find(map,"Iran");
  find(map,"Oman");
}
```

```
ostream& operator<<(ostream& ostr, const Country& c)
{ return ostr << c.abbr << ", " << c.capital << ", " << c.language
              << ", pop=" << c.population << ", area=" << c.area;
}

Country::Country()
  : abbr(""), capital(""), language(""), population(0), area(0) { }

Country::Country(string ab, string c, string l, int p, int ar)
  : abbr(ab), capital(c), language(l), population(p), area(ar) { }

void load(Map& m)
{ m["Iran"] = Country("IR","Tehran","Persian",68959931,632457);
  m["Iran"] = Country("IR","Tehran","Farsi",68959931,632457);
  m["Peru"] = Country("PE","Lima","Spanish",26111110,496223);
  m["Iraq"] = Country("IQ","Baghdad","Arabic",21722287,167975);
  m.insert(Pair("Togo",Country("TG","Lome","French",4905824,21927)));
  m.insert(Pair("Fiji",Country("FJ","Suva","English",802611,7054)));
  m.insert(Pair("Fiji",Country("FJ","Suva","Fijian",802611,7054)));
}

void print(Map& m)
{ for (It it=m.begin(); it != m.end(); it++)
    cout << it->first << ":\t" << it->second << "\n";
  cout << "size=" << m.size() << "\n";
}

void find(Map& m, const string& s)
{ cout << s;
  It it = m.find(s);
  if (it == m.end()) cout << " was not found.\n";
  else cout << ":\t" << it->second << "\n";
}
```

```
Fiji:   FJ, Suva, English, pop=802611, area=7054
Iran:   IR, Tehran, Farsi, pop=68959931, area=632457
Iraq:   IQ, Baghdad, Arabic, pop=21722287, area=167975
Peru:   PE, Lima, Spanish, pop=26111110, area=496223
Togo:   TG, Lome, French, pop=4905824, area=21927
size=5
Cuba was not found.
Iran:   IR, Tehran, Farsi, pop=68959931, area=632457
Oman was not found.
```

The program creates a `map` whose keys are four-letter names of countries and whose mapped values are `Country` objects, where `Country` is a class defined to have five fields: `abbr`, `capital`, `language`, `population`, and `area`. It uses a separate function to load the data into the `map`.

The `load()` function illustrates two different ways to insert a `pair` element into a `map`. The first four lines use the subscript operator and the last three lines use the `insert()` function. The subscript operator works the same way on a `map` container as with other container classes: just like an array, except that with a `map` the index need not be an integer. In this example it is a string.

The `insert()` function takes a single `pair` argument, where the two component types must be the same as for the `map` itself, except that the first component (the *key field*) must be `const`.

The `map` class does not allow duplicate keys. Note that the subscript operator replaces existing elements when a duplicate key is inserted, so that the last pair inserted is the one that remains. But the `insert()` function does not replace existing elements when a duplicate key is inserted, so the first pair inserted is the one that remains.

The `print()` function uses the iterator `it` to traverse the map. On each iteration of the `for` loop, it points to a `pair` object whose `first` component is the key value and whose `second` component is the data object. These two componenets are accessed by the expressions `it->first` and `it->second`. The `first` component is a `string`, the four-letter name of the country. The `second` component is a `Country` object which can be passed to the output operator since it is overloaded in the `Country` class definition. Note that the pairs are sorted automatically by their key values.

The `find()` function uses the `find` member function of the `map` class. The call `m.find(s)` returns an iterator that points to the `map` element whose `first` component equals `s`. If no such element is found, then the returned pointer points to `m.end()`, which is the dummy element that follows the last element of the map container.

D.8 THE `set` CLASS TEMPLATE

A `set` object acts like a `map` object with only the keys stored.

The `set` class template is defined in the `<set>` header.

EXAMPLE D.9 Using `set` Functions

The program defines overloaded operators `+`, `*`, and `-` to perform set-theoretic union, intersection, and relative complement operations. These are implemented using the `insert()` and `erase()` member functions and the the `set_intersection()` and `set_difference()` generic algorithms (nonmember functions). This example illustrates the distinctions between the `set` generic algorithms (`set_union()`, `set_difference()`, and `set_difference()`) and the corresponding set-theoretic operations (union, intersection, and complement).

```
#include <iostream>
#include <set>
#include <string>
using namespace std;
typedef set<string> Set;
typedef set<string>::iterator It;
void print(Set);
Set operator+(Set&,Set&);  // union
Set operator*(Set&,Set&);  // intersection
Set operator-(Set&,Set&);  // relative complement

int main()
{ string str1[] = { "A", "B", "C", "D", "E", "F", "G" };
  string str2[] = { "A", "E", "I", "O", "U" };
  Set s1(str1,str1+7);
  Set s2(str2,str2+5);
  print(s1);
  print(s2);
  print(s1+s2);
  print(s1*s2);
  print(s1-s2);
}
```

```
Set operator+(Set& s1, Set& s2)
{ Set s(s1);
  s.insert(s2.begin(),s2.end());
  return s;
}

Set operator*(Set& s1, Set& s2)
{ Set s(s1);
  It it = set_intersection(s1.begin(),s1.end(),
                           s2.begin(),s2.end(),s.begin());
  s.erase(it,s.end());
  return s;
}
Set operator-(Set& s1, Set& s2)
{ Set s(s1);
  It it = set_difference(s1.begin(),s1.end(),
                         s2.begin(),s2.end(),s.begin());
  s.erase(it,s.end());
  return s;
}

void print(Set s)
{ cout << "size=" << s.size() << ": {";
  for (It it=s.begin(); it != s.end(); it++)
    if (it == s.begin()) cout << *it;
    else cout << "," << *it;
  cout << "}\n";
}
size=7: {A,B,C,D,E,F,G}
size=5: {A,E,I,O,U}
size=10: {A,B,C,D,E,F,G,I,O,U}
size=2: {A,E}
size=5: {B,C,D,F,G}
```

The set objects `s1` and `s2` are constructed from the string arrays `str1` and `str2` using the expressions `str1`, `str1+7`, `str2`, and `str2+7` as iterators.

The elements of a `set` object are always stored in sorted order. That allows the union function (`operator+()`) to be implemented with the `set::insert()` function.

The main reason why the `set` generic algorithms do not produce directly the expected set-theoretic operations is that they leave the size of the target set unchanged. Thus we use the `erase()` member function together with the `set_intersection()` and `set_difference()` generic algorithms to implement the `operator*()` and `operator-()` functions.

Appendix E

Standard C++ Generic Algorithms

The *generic algorithms* in standard C++ are the 70 nonmember function templates that apply to container objects. There are 66 listed here alphabetically. We use the symbol `[p,q[` to represents the segment of elements from `*p` to `*(q-1)` (*i.e.*, including the element `*p` but excluding the element `*q`). The parameters are

```
iterator p, q;  // used to describe the segment [p,q[
iterator r;     // p <= r <= q
unsigned n;     // used as a counter
T& x, y;        // values of the sequence's element type
class p;        // a predicate class, with boolean operator()()
```

The parameter list `(p,q,pp)` is used frequently; it means that the elements from the segment `[p,q[` are to be copied into the segment `[pp,pp+n[` where `n` is the number of elements in `[p,q[`, namely `q-p`.

For simplicity, we use arrays instead of general container objects. In that context, pointers serve as iterators. Recall that if `a` is an array and `k` is an `int` then `a+k` represents the subarray that starts with `a[k]`, and `*(a+k) = a[k]`. Also, if `l` is the length of the array, then `a+l` points to the (imaginary) element that follows the last element of the array.

The following `print()` function is used to display the `n` element `a[0],...,a[n-1]` of an array `a`:

```
void print(int* a, int n)
{ cout << "n=" << n << ": {" << a[0];
  for (int i=1; i<n; i++)
    cout << "," << a[i];
  cout << "}\n";
}
```

The 66 algorithms listed here naturally fall into 8 groups, summarized in the following tables:

Searching and Sorting Algorithms in `<algorithm>`

`binary_search()`	Determines whether a given value is an element in the segment.
`inplace_merge()`	Merges two adjacent sorted segments into one sorted segment.
`lower_bound()`	Finds the first element in the segment that has a given value.
`merge()`	Merges two sorted segments into a third sorted segment.
`nth_element()`	Finds the first occurrence of a given value.
`partial_sort()`	Sorts the first *n* elements of the segment.
`partial_sort_copy()`	Copies the smallest *n* elements of the segment into another sorted segment.
`partition()`	Partitions the segment so that $P(x)$ is true for the elements in the first part.
`sort()`	Sorts the segment.
`upper_bound()`	Finds the last element in the segment that has a given value.

Nonmodifying Algorithms on Sequences in `<algorithm>`

`adjacent_find()`	Finds the first adjacent pair in the segment.
`count()`	Counts the number of elements that have a given value.
`count_if()`	Counts the number of elements that satisfy a given predicate.
`equal()`	Determines whether two segments have the same value in the same order.
`find()`	Finds the first element that has a given value.
`find_end()`	Finds the location of the last occurrence of a given substring.
`find_first_of()`	Finds the location of the first occurrence of any element of a given segment.
`find_if()`	Finds the first element that satisfies a given predicate.
`for_each()`	Applies a given function to each element.
`mismatch()`	Finds the first positions where two segments do not match.
`search()`	Searches for a given subsequence.
`search_n()`	Searches for a subsequence of *n* consecutive elements that have a given value.

Modifying Algorithms on Sequences in `<algorithm>`

`copy()`	Copies the segment to a new location.
`copy_backward()`	Copies the segment to a new location.
`fill()`	Replaces each element in the segment to a given value.
`fill_n()`	Replaces *n* elements in the segment to a given value.
`generate()`	Assigns the output from successive calls to $f(x)$ to elements of the segment.
`generate_n()`	Assigns the output from *n* successive calls to $f(x)$ to elements of the segment.
`iter_swap()`	Swaps the elements at the positions of the given iterators.
`random_shuffle()`	Shuffles the elements in the segment.
`remove()`	Shifts to the left all elements that do not have a given value.
`remove_copy()`	Copies all elements into another segment that do not have a given value.
`remove_copy_if()`	Copies all elements into another segment for which $P(x)$ is false.
`remove_if()`	Shifts to the left all elements for which $P(x)$ is false.
`replace()`	Changes the value of each element in the segment from x to y.
`replace_copy()`	Copies each element to another segment changing each x to y.
`replace_copy_if()`	Copies each element to another segment changing x to y where $P(x)$ is true.
`replace_if()`	Changes those elements in the segment from x to y where $P(x)$ is true.
`reverse()`	Reverses the elements in the segment.
`reverse_copy()`	Copies the elements to a new segment in reverse order.
`rotate()`	Shifts the elements to the left, wrapping around the end of the segment.
`rotate_copy()`	Copies elements to another segment, shifting to the left and wrapping.
`swap()`	Swaps the two given elements.
`transform()`	Applies $f(x)$ to each element, storing the results in another segment.
`unique()`	Shifts one of each occurring value to the left.
`unique_copy()`	Copies the nonduplicate elements to another segment.

Comparison Algorithms in `<algorithm>`

`lexicographical_compare()`	Returns true iff first segment is lexicographically less than second.
`max()`	Returns the largest element in the segment.
`max_element()`	Returns the position of largest element in the segment.
`min()`	Returns the smallest element in the segment.
`min_element()`	Returns the position of smallest element in the segment.

Algorithms on Sets in `<algorithm>`

`includes()`	Returns true iff every element of the second segment is in the first.
`set_difference()`	Copies to a third segment the relative complement of two sets.
`set_intersection()`	Copies to a third segment the intersection of two sets.
`set_symmetric_difference()`	Copies to a third segment the symmetric difference of two sets.
`set_union()`	Copies to a third segment the union of two sets.

Algorithms on Heaps in `<algorithm>`

`make_heap()`	Rearranges the elements of the segment into a heap.
`pop_heap()`	Moves first element to end and then `make_heap()` on rest.
`push_heap()`	Shifts last element to left to make segment a heap.
`sort_heap()`	Applies `pop_heap()` *n* times to sort the segment.

Permutation Algorithms in `<algorithm>`

`next_permutation()`	Permutes the segment; *n*! calls produce *n*! distinct permutations.
`prev_permutation()`	Permutes the segment; *n*! calls produce *n*! distinct permutations.

Numeric Algorithms in `<numeric>`

`accumulate()`	Adds the elements of the segment; returns *x* + *sum*.
`adjacent_difference()`	Loads second segment with the differences of adjacent elements.
`inner_product()`	Returns the inner product of two segments.
`partial_sum()`	Loads second segment with the partial sums from first.

Algorithms that search for an element always return an iterator that locates it or one that locates the dummy end element that follows the last element of the sequence.

Algorithms that use predicates are illustrated with the following predicate class:

```
class Odd
{ public:
    bool operator()(int n) { return n%2 ? true : false; }
};
```

This class is passed as a function, like this: `Odd()`. (See Example E.8 on page 372.)

Note that the modifying algorithms do not change the length of the segment `[p,q[`. Instead, they return an iterator that points to the element that follows the modified part.

```
accumulate(p,q,x);
// returns x plus the sum of the elements in the segment [p,q[;
// invariant: [p,q[ is left unchanged;
```

EXAMPLE E.1 Testing the `accumulate()` Algorithm

```
int main()
{ int a[] = {0,1,1,2,3,5,8,13,21,34};
  int sum = accumulate(a,a+10,1000);
  cout << "sum=" << sum << '\n';
}
sum=1088
```

```
adjacent_difference(p,q,pp);
// loads the segment a[pp,pp+p-q[ with b[i] = a[i]-a[i-1];
// invariant: [p,q[ is left unchanged;
```

EXAMPLE E.2 Testing the `adjacent_difference()` Algorithm

```
int main()
{ int a[] = {0,1,1,2,3,5,8,13,21,34};
  print(a,10);
  int b[10];
  adjacent_difference(a,a+10,b);
  print(b,10);
}
n=10: {0,1,1,2,3,5,8,13,21,34}
n=10: {0,1,0,1,1,2,3,5,8,13}
```

The `adjacent_difference()` algorithm is the inverse of the `partial_sum()` algorithm (Example E.36 on page 382).

```
adjacent_find(p,q);
// returns the location of the first element in the segment a[p,q[
// that has the same value as its successor;
// invariant: [p,q[ is left unchanged;
```

EXAMPLE E.3 Testing the `adjacent_find()` Algorithm

```
int main()
{ int a[] = {0,1,0,1,1,1,0,1,1,0};
  print(a,10);
  int* r = adjacent_find(a,a+10);
  cout << "*r=" << *r << '\n';    // this is the element a[i]
  cout << "r-a=" << r-a << '\n';  // this is the index i
}
n=10: {0,1,0,1,1,1,0,1,1,0}
*r=1
r-a=3
```

```
binary_search(p,q,x);
// returns true iff x is in the segment [p,q[;
// precondition: the segment [p,q) must be sorted;
// invariant: [p,q[ is left unchanged;
```

EXAMPLE E.4 Testing the `binary_search()` Algorithm

```
int main()
{ int a[] = {0,1,1,2,3,5,8,13,21,34};
  print(a,10);
  bool found = binary_search(a,a+10,21);
  cout << "found=" << found << '\n';
  found = binary_search(a+2,a+7,21);
  cout << "found=" << found << '\n';
}
n=10: {0,1,1,2,3,5,8,13,21,34}
found=1
found=0
```

```
copy(p,q,pp);
// copies the segment [p,q[ to [pp,pp+n[ where n=q-p;
// invariant: [p,q[ is left unchanged;
```

EXAMPLE E.5 Testing the `copy()` Algorithm

```
int main()
{ int a[] = {100,111,122,133,144,155,166,177,188,199};
  print(a,10);
  copy(a+7,a+10,a+2);
  print(a,10);
  int b[3];
  copy(a+7,a+10,b);
  print(b,3);
}
n=10: {100,111,122,133,144,155,166,177,188,199}
n=10: {100,111,177,188,199,155,166,177,188,199}
n=3: {177,188,199}
```

```
copy_backward(p,q,pp);
// copies the segment [p,q[ to [qq-n,qq[ where n=q-p;
// invariant: [p,q[ is left unchanged;
```

EXAMPLE E.6 Testing the `copy_backward()` Algorithm

```
int main()
{ int a[] = {100,111,122,133,144,155,166,177,188,199};
  print(a,10);
  copy_backward(a+7,a+10,a+5);
  print(a,10);
  int b[3];
  copy_backward(a+7,a+10,b+3);
```

```
  print(b,3);
}
n=10: {100,111,122,133,144,155,166,177,188,199}
n=10: {100,111,177,188,199,155,166,177,188,199}
n=3: {177,188,199}
```

```
count(p,q,x);
// returns the number of occurrences of x in the segment [p,q[;
// invariant: [p,q[ is left unchanged;
```

EXAMPLE E.7 Testing the `count()` Algorithm

```
int main()
{ int a[] = {0,1,0,1,1,1,0,1,1,0};
  print(a,10);
  int n = count(a,a+10,1);
  cout << "n=" << n << '\n';
}
n=10: {0,1,0,1,1,1,0,1,1,0}
n=6
```

```
count_if(p,q,P());
// returns the number of occurrences where P(x) in the segment [p,q[;
// invariant: [p,q[ is left unchanged;
```

EXAMPLE E.8 Testing the `count_if()` Algorithm

```
int main()
{ int a[] = {0,1,0,1,1,1,0,1,1,0};
  print(a,10);
  int n = count_if(a,a+10,Odd());
  cout << "n=" << n << '\n';
}
n=10: {0,1,0,1,1,1,0,1,1,0}
n=6
```

```
equal(p,q,pp);
// returns true iff the segment [p,q) matches [pp,pp+n[, where n = q-p;
// invariant: [p,q[ and [pp,qq+n[ are left unchanged;
```

EXAMPLE E.9 Testing the `equal()` Algorithm

```
int main()
{ int a[] = {0,1,0,1,1,1,0,1,1,0};
  int b[] = {0,1,0,0,1,1,0,1,0,0};
  print(a,10);
  print(b,10);
  cout << "equal(a,a+10,b)=" << equal(a,a+10,b) << '\n';
  cout << "equal(a+1,a+4,a+5)=" << equal(a+1,a+4,a+5) << '\n';
```

```
}
n=10: {0,1,0,1,1,1,0,1,1,0}
n=10: {0,1,0,0,1,1,0,1,0,0}
equal(a,a+10,b)=0
equal(a+1,a+4,a+5)=1
```

```
fill(p,q,x);
// replaces each element in the segment [p,q[ with x;
```

EXAMPLE E.10 Testing the `fill()` Algorithm

```
int main()
{ int a[] = {0,1,1,2,3,5,8,13,21,34};
  print(a,10);
  fill(a+6,a+9,0);
  print(a,10);
}
n=10: {0,1,1,2,3,5,8,13,21,34}
n=10: {0,1,1,2,3,5,0,0,0,34}
```

```
fill_n(p,n,x);
// replaces each element in the segment [p,p+n[ with x;
```

EXAMPLE E.11 Testing the `fill_n()` Algorithm

```
int main()
{ int a[] = {0,1,1,2,3,5,8,13,21,34};
  print(a,10);
  fill_n(a+6,3,0);
  print(a,10);
}
n=10: {0,1,1,2,3,5,8,13,21,34}
n=10: {0,1,1,2,3,5,0,0,0,34}
```

```
find(p,q,x);
// returns the first location of x in the segment [p,q[;
// invariant: [p,q[ is left unchanged;
```

EXAMPLE E.12 Testing the `find()` Algorithm

```
int main()
{ int a[] = {0,1,1,2,3,5,8,13,21,34};
  print(a,10);
  int* r = find(a,a+10,13);
  cout << "*r=" << *r << '\n';    // this is the element a[i]
  cout << "r-a=" << r-a << '\n';  // this is the index i
  r = find(a,a+6,13);
  cout << "*r=" << *r << '\n';    // this is the element a[i]
  cout << "r-a=" << r-a << '\n';  // this is the index i
```

```
}
n=10: {0,1,1,2,3,5,8,13,21,34}
*r=13
r-a=7
*r=8
r-a=6
```

```
find_end(p,q,pp,qq);
// returns the location of the last occurrence of the the segment [pp,qq[
// within the segment [p,q[;
// invariant: [p,q[ and [pp,qq[ are left unchanged;
```

EXAMPLE E.13 Testing the `find_end()` Algorithm

```
int main()
{ int a[] = {0,1,0,1,1,1,0,1,1,0};
  int b[] = {1,0,1,1,1};
  int* r = find_end(a,a+10,b,b+5);  // search for 10111 in a
  cout << "*r=" << *r << '\n';      // this is the element a[i]
  cout << "r-a=" << r-a << '\n';    // this is the index i
  r = find_end(a,a+10,b,b+4);       // search for 1011 in a
  cout << "*r=" << *r << '\n';
  cout << "r-a=" << r-a << '\n';
}
*r=1
r-a=1
*r=1
r-a=5
```

```
find_first_of(p,q,pp,qq);
// returns the position in [p,q[ of the first element found that is also in
// [pp,qq[;
// invariant: [p,q[ and [pp,qq[ are left unchanged;
```

EXAMPLE E.14 Testing the `find_first_of()` Algorithm

```
int main()
{ int a[] = {0,1,1,2,3,5,8,13,21,34};
  int b[] = {6,7,8,9,10,11,12,13,14,15};
  int* r = find_first_of(a,a+10,b,b+10);
  cout << "*r=" << *r << '\n';    // this is the element a[i]
  cout << "r-a=" << r-a << '\n';  // this is the index i
}
*r=8
r-a=6
```

```
find_if(p,q,P());
// returns the first location of where P(x) in the segment [p,q[;
// invariant: [p,q[ is left unchanged;
```

EXAMPLE E.15 Testing the `find_if()` Algorithm

```
int main()
{ int a[] = {2,4,8,16,32,64,128,256,333,512};
  int* r = find_if(a,a+10,Odd());
  cout << "*r=" << *r << '\n';    // this is the element a[i]
  cout << "r-a=" << r-a << '\n';  // this is the index i
  r = find_if(a,a+5,Odd());
  cout << "*r=" << *r << '\n';    // this is the element a[i]
  cout << "r-a=" << r-a << '\n';  // this is the index i
}
```

```
*r=333
r-a=8
*r=64
r-a=5
```

```
for_each(p,q,f);
// applies the function f(x) to each x in the segment [p,q[;
```

EXAMPLE E.16 Testing the `for_each()` Algorithm

```
void print(int);

int main()
{ int a[] = {0,1,1,2,3,5,8,13,21,34};
  for_each(a,a+10,print);
}

void print(int x)
{ cout << x << " ";
}
```

```
0 1 1 2 3 5 8 13 21 34
```

```
generate(p,q,f);
// assigns to [p,q[ the outputs of successive calls to f(x);
```

EXAMPLE E.17 Testing the `generate()` Algorithm

```
long fibonacci();

int main()
{ int a[10]={0};
  generate(a,a+10,fibonacci);
  print(a,10);
}

long fibonacci()
{ static int f1=0, f2=1;
  int f0=f1;
  f1 = f2;
```

```
  f2 += f0;
  return f0;
}
n=10: {0,1,1,2,3,5,8,13,21,34}
```

```
generate_n(p,n,f);
// assigns the outputs of successive calls f(x) to each x in [p,p+n[;
```

EXAMPLE E.18 Testing the `generate_n()` Algorithm

```
long fibonacci();

int main()
{ int a[10]={0};
  generate_n(a,10,fibonacci);
  print(a,10);
}

long fibonacci()
{ static int f1=0, f2=1;
  int f0=f1;
  f1 = f2;
  f2 += f0;
  return f0;
}
n=10: {0,1,1,2,3,5,8,13,21,34}
```

```
includes(p,q,pp,qq);
// returns true iff every element of [pp,qq[ is found in [p,q[;
// precondition: both segments must be sorted;
// invariant: [p,q[ and [pp,qq[ are left unchanged;
```

EXAMPLE E.19 Testing the `includes()` Algorithm

```
int main()
{ int a[] = {0,1,1,2,3,5,8,13,21,34};
  int b[] = {0,1,2,3,4};
  bool found = includes(a,a+10,b,b+5);
  cout << "found=" << found << '\n';
  found = includes(a,a+10,b,b+4);
  cout << "found=" << found << '\n';
}
found=0
found=1
```

```
inner_product(p,q,pp,x)
// returns the sum of x and the inner product of [p,q[ with [pp,pp+n[,
// where n = q-p;
// invariant: [p,q[ and [pp,qq[ are left unchanged;
```

EXAMPLE E.20 Testing the `inner_product()` Algorithm

```
int main()
{ int a[] = {1,3,5,7,9};
  int b[] = {4,3,2,1,0};
  int dot = inner_product(a,a+4,b,1000);
  cout << "dot=" << dot << '\n';
}
sum=1030
```

```
inplace_merge(p,r,q);
// merges the segments [p,r[ and [r,q[;
// precondition: the two segments must be contiguous and sorted;
// postcondition: the segment [p,r[ is sorted;
```

EXAMPLE E.21 Testing the `inplace_merge()` Algorithm

```
int main()
{ int a[] = {22,55,66,88,11,33,44,77,99};
  print(a,9);
  inplace_merge(a,a+4,a+9);
  print(a,9);
}
n=9: {22,55,66,88,11,33,44,77,99}
n=9: {11,22,33,44,55,66,77,88,99}
```

```
iter_swap(p,q);
// swaps the elements *p and *q;
```

EXAMPLE E.22 Testing the `iter_swap()` Algorithm

```
int main()
{ int a[] = {11,22,33,44,55,66,77,88,99};
  int b[] = {10,20,30,40,50,60,70,80,90};
  print(a,9);
  print(b,9);
  iter_swap(a+4,b+7);
  print(a,9);
  print(b,9);
}
n=9: {11,22,33,44,55,66,77,88,99}
n=9: {10,20,30,40,50,60,70,80,90}
n=9: {11,22,33,44,80,66,77,88,99}
n=9: {10,20,30,40,50,60,70,55,90}
```

```
lexicographical_compare(p,q,pp,qq);
// compares the two segments [pp,qq[ and [p,q[ lexicographically;
// returns true iff the first precedes the second;
// invariant: [p,q[ and [pp,qq[ are left unchanged;
```

EXAMPLE E.23 Testing the `lexicographical_compare()` Algorithm

```
void test(char*,int,char*,int);

int main()
{ char* s1="COMPUTER";
  char* s2="COMPUTABLE";
  char* s3="COMPUTE";
  test(s1,3,s2,3);
  test(s1,8,s2,10);
  test(s1,8,s3,7);
  test(s2,10,s3,7);
  test(s1,7,s3,7);
}

char* sub(char*,int);

void test(char* s1, int n1, char* s2, int n2)
{ bool lt=lexicographical_compare(s1,s1+n1,s2,s2+n2);
  bool gt=lexicographical_compare(s2,s2+n2,s1,s1+n1);
  if (lt) cout << sub(s1,n1) << " < " << sub(s2,n2) << "\n";
  else if (gt) cout << sub(s1,n1) << " > " << sub(s2,n2) << "\n";
  else cout << sub(s1,n1) << " == " << sub(s2,n2) << "\n";
}

char* sub(char* s, int n)
{ char* buffer = new char(n+1);
  strncpy(buffer,s,n);
  buffer[n] = 0;
  return buffer;
}
```

```
COM == COM
COMPUTER > COMPUTABLE
COMPUTER > COMPUTE
COMPUTABLE < COMPUTE
COMPUTE == COMPUTE
```

```
lower_bound(p,q,x);
// returns the position of the first occurrence of x in [p,q[;
// precondition: the segment must be sorted;
// invariant: [p,q[ is left unchanged;
```

EXAMPLE E.24 Testing the `lower_bound()` Algorithm

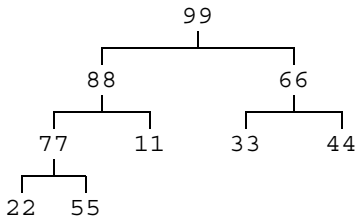
```
int main()
{ int a[] = {11,22,22,33,44,44,44,55,66};
  int* p = lower_bound(a,a+9,44);
  cout << "*p=" << *p << '\n';
  cout << "p-a=" << p-a << '\n';
}
```

```
*p=44
p-a=4
```

```
make_heap(p,q);
// rearranges the elements of [p,q[ into a heap;
// postcondition: [p,q[ is a heap;
```

EXAMPLE E.25 Testing the `make_heap()` Algorithm

```
int main()
{ int a[] = {44,88,33,77,11,99,66,22,55};
  print(a,9);
  make_heap(a,a+9);
  print(a,9);
}
n=9: {44,88,33,77,11,99,66,22,55}
n=9: {99,88,66,77,11,33,44,22,55}
```




```
max(x,y);
// returns the maximum of x and y;
```

EXAMPLE E.26 Testing the `max()` Algorithm

```
int main()
{ cout << "max(48,84)=" << max(48,84) << '\n';
}
max(48,84)=84
```

```
max_element(p,q);
// returns the position of the maximum element in the segment [pp,qq[;
// invariant: [p,q[ is left unchanged;
```

EXAMPLE E.27 Testing the `max_element()` Algorithm

```
int main()
{ int a[] = {77,22,99,55,11,88,44,33,66};
  const int* p = max_element(a,a+9);
  cout << "*p=" << *p << '\n';
  cout << "p-a=" << p-a << '\n';
}
*p=99
p-a=2
```

```
merge(p,q,pp,qq,ppp);
// merges the segments [p,q[ and [pp,qq[ into [ppp,ppp+n[,
// where n = q - p + qq - pp;
// precondition: [p,q[ and [pp,qq[  must be sorted;
// postcondition: the segment [ppp,ppp+n[ is sorted;
// invariant: [p,q[ and [pp,qq[ are left unchanged;
```

EXAMPLE E.28 Testing the `merge()` Algorithm

```
int main()
{ int a[] = {22,55,66,88};
  int b[] = {11,33,44,77,99};
  int c[9];
  merge(a,a+4,b,b+5,c);
  print(c,9);
}
n=9: {11,22,33,44,55,66,77,88,99}
```

`min(x,y);`

```
// returns the minimum of x and y;
```

EXAMPLE E.29 Testing the `min()` Algorithm

```
int main()
{ cout << "min(48,84)=" << min(48,84) << '\n';
}
min(48,84)=48
```

`min_element(p,q);`

```
// returns the position of the minimum element in the segment [p,q[;
// invariant: [p,q[ is left unchanged;
```

EXAMPLE E.30 Testing the `min_element()` Algorithm

```
int main()
{ int a[] = {77,22,99,55,11,88,44,33,66};
  const int* p = min_element(a,a+9);
  cout << "*p=" << *p << '\n';
  cout << "p-a=" << p-a << '\n';
}
*p=11
p-a=4
```

`mismatch(p,q,pp);`

```
// returns a pair of iterators giving the positions in [p,q[ and
// in [pp,qq[ where the first mismatch of elements occurs;
// if the two segments match entirely, then their ends are returned;
// invariant: [p,q[ and [pp,qq[ are left unchanged;
```

EXAMPLE E.31 Testing the `mismatch()` Algorithm

```
int main()
{ char* s1="Aphrodite, Apollo, Ares, Artemis, Athena";
  char* s2="Aphrodite, Apallo, Ares, Artimis, Athens";
  int n=strlen(s1);
  cout << "n=" << n << '\n';
```

```
  pair<char*,char*> x = mismatch(s1,s1+n,s2);
  char* p1 = x.first;
  char* p2 = x.second;
  cout << "*p1=" << *p1 << ", *p2=" << *p2 << '\n';
  cout << "p1-s1=" << p1-s1 << '\n';
}
n=40
*p1=o, *p2=a
p1-s1=13
```

```
next_permutation(p,q);
// permutes the elements of [p,q[; n! calls will cycle through all n!
// permutations of the n elements, where n = q-p;
```

EXAMPLE E.32 Testing the `next_permutation()` Algorithm

```
int main()
{ char* s="ABCD";
  for (int i=0; i<24; i++)
  { next_permutation(s,s+4);
    cout << (i%8?'\t':'\n') << s;
  }
}
```

```
ABDC    ACBD    ACDB    ADBC    ADCB    BACD    BADC    BCAD
BCDA    BDAC    BDCA    CABD    CADB    CBAD    CBDA    CDAB
CDBA    DABC    DACB    DBAC    DBCA    DCAB    DCBA    ABCD
```

The `next_permutation()` algorithm is the inverse of the `prev_permutation()` algorithm (Example E.39 on page 383).

```
nth_element(p,r,q);
// rearranges the elements of [p,q[ so that *r partitions it into the two
// subsegments [p,r1[ and [r1+2,q], where r1 is the new location of *r,
// all the elements of [p,r1] are <= to *r, and all the elements of
// [r1+2,q] are >= to *r; *r is called the pivot element;
```

EXAMPLE E.33 Testing the `nth_element()` Algorithm

```
int main()
{ int a[] = {77,22,99,55,44,88,11,33,66};
  print(a,9);
  nth_element(a,a+3,a+9);
  print(a,9);
}
```

```
n=9: {77,22,99,55,44,88,11,33,66}
n=9: {11,22,33,44,55,88,66,99,77}
```

```
partial_sort(p,r,q);
// sorts the first r-p elements of [p,q[, placing them in [p,r[ and
// shifting the remaining q-r elements down to [r,q[;
```

EXAMPLE E.34 Testing the `partial_sort()` Algorithm

```
int main()
{ int a[] = {77,22,99,55,44,88,11,33,66};
  print(a,9);
  partial_sort(a,a+3,a+9);
  print(a,9);
}
n=9: {77,22,99,55,44,88,11,33,66}
n=9: {11,22,33,99,77,88,55,44,66}
```

```
partial_sort_copy(p,q,pp,qq);
// copies the qq-pp smallest elements of [p,q[ into [pp,qq[ in sorted
// order; then copies the remaining n elements into [qq,qq+n[,
// where n = q-p+pp-qq;
// invariant: [p,q[ is left unchanged;
```

EXAMPLE E.35 Testing the `partial_sort_copy()` Algorithm

```
int main()
{ int a[] = {77,22,99,55,44,88,11,33,66};
  print(a,9);
  int b[3];
  partial_sort_copy(a,a+9,b,b+3);
  print(a,9);
  print(b,3);
}
n=9: {77,22,99,55,44,88,11,33,66}
n=9: {77,22,99,55,44,88,11,33,66}
n=3: {11,22,33}
```

```
partial_sum(p,q,pp);
// invariant: a[p,q[ is left unchanged;
// postcondition b[i] == a[0]+...+a[i] for each b[i] in [pp,pp+q-p[;
```

EXAMPLE E.36 Testing the `partial_sum()` Algorithm

```
int main()
{ int a[] = {0,1,1,2,3,5,8,13,21,34};
  int b[10];
  partial_sum(a,a+10,b);
  print(a,10);
  print(b,10);
}
n=10: {0,1,1,2,3,5,8,13,21,34}
n=10: {0,1,2,4,7,12,20,33,54,88}
```

The `partial_sum()` algorithm is the inverse of the `adjacent_difference()` algorithm (Example E.2 on page 370).

```
partition(p,q,P());
// partitions [p,q[ into [p,r[ and [r,q[ so that
// x is in [p,r[ iff P(x) is true;
```

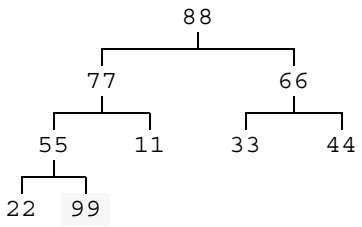
EXAMPLE E.37 Testing the `partition()` Algorithm

```
int main()
{ int a[] = {0,1,1,2,3,5,8,13,21,34};
  print(a,10);
  partition(a,a+10,Odd());
  print(a,10);
}
n=10: {0,1,1,2,3,5,8,13,21,34}
n=10: {21,1,1,13,3,5,8,2,0,34}
```

```
pop_heap(p,q);
// moves *p into temp, then shifts elements to the left so that the
// remaining elements form a heap in [p,q-1[ into a heap, then copies
// temp into *(q-1);
// precondition: [p,q[ must be a heap;
// postcondition: [p,q-1[ is a heap;
```

EXAMPLE E.38 Testing the `pop_heap()` Algorithm

```
int main()
{ int a[] = {44,88,33,77,11,99,66,22,55};
  print(a,9);
  make_heap(a,a+9);
  print(a,9);
  pop_heap(a,a+9);
  print(a,9);
  print(a,8);
}
n=9: {44,88,33,77,11,99,66,22,55}
n=9: {99,88,66,77,11,33,44,22,55}
n=9: {88,77,66,55,11,33,44,22,99}
n=8: {88,77,66,55,11,33,44,22}
```



See Example E.25 on page 379 and Example E.38 on page 383.

```
prev_permutation(p,q);
// permutes the elements of [p,q[; n! calls will cycle backward through
// all n! permutations of the n elements, where n = q-p;
```

EXAMPLE E.39 Testing the `prev_permutation()` Algorithm

```
int main()
{ char* s="ABCD";
  for (int i=0; i<24; i++)
  { prev_permutation(s,s+4);
    cout << (i%8?'\t':'\n') << s;
```

```
  }
}
```

```
DCBA    DCAB    DBCA    DBAC    DACB    DABC    CDBA    CDAB
CBDA    CBAD    CADB    CABD    BDCA    BDAC    BCDA    BCAD
BADC    BACD    ADCB    ADBC    ACDB    ACBD    ABDC    ABCD
```

The `prev_permutation()` algorithm is the inverse of the `next_permutation()` algorithm (Example E.32 on page 381).

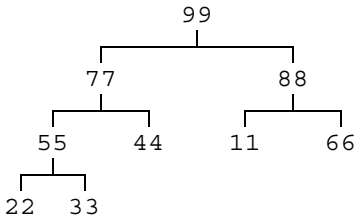
`push_heap(p,q);`

```
// adds the element at *(q-1) to those in [p,q-1[ so that [p,q[ is a heap;
// precondition: [p,q-1[ must be a heap;
// postcondition: [p,q[ is a heap;
```

EXAMPLE E.40 Testing the `push_heap()` Algorithm

```
int main()
{ int a[] = {66,44,88,33,55,11,99,22,77};
  print(a,8);
  make_heap(a,a+8);
  print(a,8);
  print(a,9);
  push_heap(a,a+9);
  print(a,9);
}
```

```
n=8: {66,44,88,33,55,11,99,22}
n=8: {99,55,88,33,44,11,66,22}
n=9: {99,55,88,33,44,11,66,22,77}
n=9: {99,77,88,55,44,11,66,22,33}
```



The `push_heap()` algorithm reverses the effect of `pop_heap()`. (See Example E.38.)

`random_shuffle(p,q);`

```
// performs a random (but deterministic) shuffle on [pp,qq[
```

EXAMPLE E.41 Testing the `random_shuffle()` Algorithm

```
int main()
{ char* s="ABCDEFGHIJ";
  cout << s << '\n';
  for (int i=0; i<4; i++)
  { random_shuffle(s,s+10);
    cout << s << '\n';
  }
}
```

```
ABCDEFGHIJ
CIJDBEAHGF
CFBDEIGAHJ
IDJABEFGHC
DBJIFEGACH
```

```
remove(p,q,x);
// removes all occurrences of x from [p,q[, shifting (copying) the
// remaining elements to the left;
// invariant: the length of the segment remains unchanged;
```

EXAMPLE E.42 Testing the `remove()` Algorithm

```
int main()
{ char* s="All is flux, nothing is stationary.";  // Heraclitus
  int l = strlen(s);
  int n = count(s,s+l,' ');
  cout << "l=" << l << '\n';
  cout << "n=" << n << '\n';
  remove(s,s+l,' ');
  cout << s << '\n';
  s[l-n] = 0;  // truncate s
  cout << s << '\n';
}
```

```
l=35
n=5
Allisflux,nothingisstationaryonary.
Allisflux,nothingisstationary.
```

Since 5 blanks were removed, the last 5 letters remain after their copies were shifted left.

```
remove_copy(p,q,pp,x);
// copies all elements of [p,q[ that do not match x to [pp,pp+n[,
// where n is the number of nonmatching elements;
// returns pp+n;
// invariant: [p,q[ remains unchanged;
```

EXAMPLE E.43 Testing the `remove_copy()` Algorithm

```
int main()
{ char* s="All is flux, nothing is stationary.";  // Heraclitus
  char buffer[80];
  int l = strlen(s);
  int n = count(s,s+l,' ');
  cout << "l=" << l << '\n';
  cout << "n=" << n << '\n';
  char* ss = remove_copy(s,s+l,buffer,' ');
  *ss = 0;  // truncate buffer
  cout << s << '\n';
  cout << buffer << '\n';
  cout << ss-buffer << '\n';
}
```

```
l=35
n=5
All is flux, nothing is stationary.
Allisflux,nothingisstationary.
30
```

```
remove_copy_if(p,q,pp,P());
// copies all elements x of [p,q[ for which !P(x) to [pp,pp+n[,
// where n is the number of nonmatching elements;
// returns pp+n;
// invariant: [p,q[ remains unchanged;
```

EXAMPLE E.44 Testing the `remove_copy_if()` Algorithm

```
class Blank
{ public:
    bool operator()(char c) { return c == ' '; }
};

int main()
{ char* s="All is flux, nothing is stationary.";  // Heraclitus
  char buffer[80];
  int l = strlen(s);
  int n = count(s,s+l,' ');
  cout << "l=" << l << '\n';
  cout << "n=" << n << '\n';
  char* ss = remove_copy_if(s,s+l,buffer,Blank());
  *ss = 0;  // truncate buffer
  cout << s << '\n';
  cout << buffer << '\n';
  cout << ss-buffer << '\n';
}
```

```
l=35
n=5
All is flux, nothing is stationary.
Allisflux,nothingisstationary.
30
```

This is the same as Example E.43 except that a predicate is used.

```
remove_if(p,q,P());
// removes all x from [p,q[ for which !P(x), shifting (copying) the
// remaining elements to the left;
```

EXAMPLE E.45 Testing the `remove_if()` Algorithm

```
class Blank
{ public:
    bool operator()(char c) { return c == ' '; }
};

int main()
{ char* s="All is flux, nothing is stationary.";  // Heraclitus
  int l = strlen(s);
  int n = count(s,s+l,' ');
  cout << "l=" << l << '\n';
  cout << "n=" << n << '\n';
  remove_if(s,s+l,Blank());
  cout << s << '\n';
  s[l-n] = 0;
```

```
  cout << s << '\n';
}
l=35
n=5
Allisflux,nothingisstationaryonary.
Allisflux,nothingisstationary.
```

This is the same as Example E.42 except that a predicate is used.

```
replace(p,q,x,y);
// replaces all occurrences of x with y in [p,q[;
// invariant: the length of the segment remains unchanged;
```

EXAMPLE E.46 Testing the `replace()` Algorithm

```
int main()
{ char* s="All is flux, nothing is stationary.";  // Heraclitus
  int l = strlen(s);
  cout << s << '\n';
  replace(s,s+l,' ','!');
  cout << s << '\n';
}
All is flux, nothing is stationary.
All!is!flux,!nothing!is!stationary.
```

```
replace_copy(p,q,pp,x,y);
// copies all elements of [p,q[ to [pp,pp+n[, replacing each occurrence
// of x with y, where n = q-p;
// returns pp+n;
// invariant: [p,q[ remains unchanged;
```

EXAMPLE E.47 Testing the `replace_copy()` Algorithm

```
int main()
{ char* s="All is flux, nothing is stationary.";  // Heraclitus
  cout << s << '\n';
  int l = strlen(s);
  char buffer[80];
  char* ss = replace_copy(s,s+l,buffer,'n','N');
  *ss = 0;  // truncate buffer for printing
  cout << s << '\n';
  cout << buffer << '\n';
}
All is flux, nothing is stationary.
All is flux, nothing is stationary.
All is flux, NothiNg is statioNary.
```

```
replace_copy_if(p,q,pp,P(),y);
// copies all elements of [p,q[ to [pp,pp+n[, replacing each x for
// which P(x) with y, where n = q-p;
```

```
// returns pp+n;
// invariant: [p,q[ remains unchanged;
```

EXAMPLE E.48 Testing the `replace_copy_if()` Algorithm

```
class Blank
{ public:
    bool operator()(char c) { return c == ' '; }
};

int main()
{ char* s="All is flux, nothing is stationary.";  // Heraclitus
  int l = strlen(s);
  char buffer[80];
  cout << s << '\n';
  char* ss = replace_copy_if(s,s+l,buffer,Blank(),'!');
  *ss = 0;  // truncate buffer
  cout << s << '\n';
  cout << buffer << '\n';
}
All is flux, nothing is stationary.
All is flux, nothing is stationary.
All!is!flux,!nothing!is!stationary.
```

This is the same as Example E.47 except that a predicate is used.

```
replace_if(p,q,P(),y);
// replaces each x for which P(x) with y in [p,q[;
```

EXAMPLE E.49 Testing the `replace_if()` Algorithm

```
class Blank
{ public:
    bool operator()(char c) { return c == ' '; }
};

int main()
{ char* s="All is flux, nothing is stationary.";  // Heraclitus
  int l = strlen(s);
  cout << s << '\n';
  replace_if(s,s+l,Blank(),'!');
  cout << s << '\n';
}
All is flux, nothing is stationary.
All!is!flux,!nothing!is!stationary.
```

This is the same as Example E.46 except that a predicate is used.

```
reverse(p,q);
// reverses the segment [p,q[;
```

EXAMPLE E.50 Testing the `reverse()` Algorithm

```
int main()
{ char* s="ABCDEFGHIJKLMNOPQRSTUVWXYZ";
  cout << s << '\n';
  reverse(s,s+26);
  cout << s << '\n';
}
```

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ZYXWVUTSRQpONMLKJIHGFEDCBA
```

```
reverse_copy(p,q,pp);
// copies the segment [p,q[ into [pp,pp+n[ in reverse order,
// where n = q-p;
// returns pp+n
// invariant: [p,q[ remains unchanged;
```

EXAMPLE E.51 Testing the `reverse_copy()` Algorithm

```
int main()
{ char* s="ABCDEFGHIJKLMNOPQRSTUVWXYZ";
  cout << s << '\n';
  char buffer[80];
  char* ss = reverse_copy(s,s+26,buffer);
  *ss = 0;  // truncate buffer for printing
  cout << s << '\n';
  cout << buffer << '\n';
}
```

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ZYXWVUTSRQpONMLKJIHGFEDCBA
```

```
rotate(p,r,q);
// shifts [r,q[ to the left by r positions into [p,p+q-r[,
// and wraps [p,r[ around to the right end into [p+q-r,q[;
```

EXAMPLE E.52 Testing the `rotate()` Algorithm

```
int main()
{ char* s="ABCDEFGHIJKLMNOPQRSTUVWXYZ";
  cout << s << '\n';
  rotate(s,s+4,s+26);
  cout << s << '\n';
}
```

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
EFGHIJKLMNOPQRSTUVWXYZABCD
```

```
rotate_copy(p,r,q,pp);
// copies the segment [r,q[ into [pp,pp+m[, where m = q-r,
// and copies the segment [p,r[ into [pp+m,pp+n[, where n = q-p;
// returns pp+m+n;
// invariant: [p,q[ remains unchanged;
```

EXAMPLE E.53 Testing the `rotate_copy()` Algorithm

```
int main()
{ char* s="ABCDEFGHIJKLMNOPQRSTUVWXYZ";
  cout << s << '\n';
  char buffer[80];
  char* ss = rotate_copy(s,s+4,s+26,buffer);
  *ss = 0;  // truncate buffer for printing
  cout << s << '\n';
  cout << buffer << '\n';
}
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
EFGHIJKLMNOPQRSTUVWXYZABCD
```

```
search(p,q,pp,qq);
// searches for the subsequence [pp,qq[ in [p,q[;
// if found, the position r of its first occurrence is returned;
// otherwise, q is returned;
// postcondition: either r = q or [r,r+n[ = [pp,qq[, where n = qq-pp;
// invariant: [p,q[ is left unchanged;
```

EXAMPLE E.54 Testing the `search()` Algorithm

```
int main()
{ char* p="ABCDEFGHIJKLABCDEFGHIJKL";
  char* pp="HIJK";
  char* r = search(p,p+24,pp,pp+4);
  int n = r-p;  // number of characters before pp in p
  cout << "n=r-p=" << n << '\n';
  cout << "*r=" << *r << '\n';
  cout << p << '\n';
  cout << string(n,'-') << pp << string(20-n,'-') << '\n';
  pp = "LMNOp";
  r = search(p,p+24,pp,pp+5);
  n = r-p;
  cout << "n=r-p=" << n << '\n';
  cout << p << '\n';
  cout << string(n,'-') << '\n';
}
n=r-p=7
*r=H
ABCDEFGHIJKLABCDEFGHIJKL
-------HIJK-------------
n=r-p=24
ABCDEFGHIJKLABCDEFGHIJKL
------------------------
```

```
search_n(p,q,n,x);
// searches for the subsequence of n consecutive copies of x in [p,q[;
// if found, the position r of its first occurrence is returned;
// otherwise, q is returned;
// postcondition: either r = q or [r,r+n[ = [pp,qq[, where n = qq-pp;
// invariant: [p,q[ is left unchanged;
```

EXAMPLE E.55 Testing the `search_n()` Algorithm

```
int main()
{ char* p="0010111001111110";
  char* r = search_n(p,p+16,3,'1');
  int m = r-p;  // number of characters before the substring in p
  cout << "m=r-p=" << m << '\n';
  cout << p << '\n';
  cout << string(m,'-') << string(3,'1') << string(13-m,'-') << '\n';
  r = search_n(p,p+16,4,'1');
  m = r-p;  // number of characters before substring in p
  cout << "m=r-p=" << m << '\n';
  cout << p << '\n';
  cout << string(m,'-') << string(4,'1') << string(12-m,'-') << '\n';
}
```

```
m=r-p=4
0010111001111110
----111---------
m=r-p=9
0010111001111110
---------1111---
```

```
set_difference(p,q,pp,qq,ppp);
// copies into [ppp,ppp+n[ the elements in [p,q[ that are not in [pp,qq[;
// returns ppp+n, where n is the number of elements copied;
// invariant: [p,q[ and [pp,qq[ are left unchanged;
```

EXAMPLE E.56 Testing the `set_difference()` Algorithm

```
int main()
{ char* p="ABCDEFGHIJ";
  char* pp="AEIOUXYZ";
  char ppp[16];
  char* qqq = set_difference(p,p+10,pp,pp+8,ppp);
  cout << p << '\n';
  cout << pp << '\n';
  *qqq = 0;  // terminates the ppp string
  cout << ppp << '\n';
}
```

```
ABCDEFGHIJ
AEIOUXYZ
BCDFGHJ
```

```
set_intersection(p,q,pp,qq,ppp);
// copies into [ppp,ppp+n[ the elements in [p,q[ that are also in [pp,qq[;
// returns ppp+n, where n is the number of elements copied;
// invariant: [p,q[ and [pp,qq[ are left unchanged;
```

EXAMPLE E.57 Testing the `set_intersection()` Algorithm

```
int main()
{ char* p="ABCDEFGHIJ";
  char* pp="AEIOUXYZ";
  char ppp[16];
  char* r = set_intersection(p,p+10,pp,pp+8,ppp);
  cout << p << '\n';
  cout << pp << '\n';
  *r = 0;  // terminates the ppp string
  cout << ppp << '\n';
}
ABCDEFGHIJ
AEIOUXYZ
AEI
```

```
set_symmetric_difference(p,q,pp,qq,ppp);
// copies into [ppp,ppp+n[ the elements in [p,q[ that are not in [pp,qq[
// and those that are in [pp,qq[ but not in [p,q[;
// returns ppp+n, where n is the number of elements copied;
// invariant: [p,q[ and [pp,qq[ are left unchanged;
```

EXAMPLE E.58 Testing the `set_symmetric_difference()` Algorithm

```
int main()
{ char* p="ABCDEFGHIJ";
  char* pp="AEIOUXYZ";
  char ppp[16];
  char* qqq = set_symmetric_difference(p,p+10,pp,pp+8,ppp);
  cout << p << '\n';
  cout << pp << '\n';
  *qqq = 0;  // terminates the ppp string
  cout << ppp << '\n';
}
ABCDEFGHIJ
AEIOUXYZ
BCDFGHJOUXYZ
```

```
set_union(p,q,pp,qq,ppp);
// copies into [ppp,ppp+n[ all the elements in [p,q[ and all the elements
// in [pp,qq[ without duplicates;
// returns ppp+n, where n is the number of elements copied;
// invariant: [p,q[ and [pp,qq[ are left unchanged;
```

EXAMPLE E.59 Testing the `set_union()` Algorithm

```
int main()
{ char* p="ABCDEFGHIJ";
  char* pp="AEIOUXYZ";
  char ppp[16];
  char* r = set_union(p,p+10,pp,pp+8,ppp);
  cout << p << '\n';
  cout << pp << '\n';
  *r = 0;  // terminates the ppp string
  cout << ppp << '\n';
}
```

```
ABCDEFGHIJ
AEIOUXYZ
ABCDEFGHIJOUXYZ
```

`sort(p,q);`
`// sorts [p,q[;`

EXAMPLE E.60 Testing the `sort()` Algorithm

```
int main()
{ char* p="GAJBHCHDIEFAGDHC";
  cout << p << '\n';
  sort(p,p+16);
  cout << p << '\n';
}
```

```
GAJBHCHDIEFAGDHC
AABCCDDEFGGHHHIJ
```

`sort_heap(p,q);`
`// sorts [p,q[;`

EXAMPLE E.61 Testing the `sort_heap()` Algorithm

```
int main()
{ int a[] = {66,88,44,77,33,55,11,99,22};
  print(a,9);
  make_heap(a,a+9);
  print(a,9);
  sort_heap(a,a+9);
  print(a,9);
}
```

```
n=9: {66,88,44,77,33,55,11,99,22}
n=9: {99,88,55,77,33,44,11,66,22}
n=9: {11,22,33,44,55,66,77,88,99}
```

`swap(x,y);`
`// swaps the two elements x and y;`

EXAMPLE E.62 Testing the `swap()` Algorithm

```
int main()
{ char* p="ABCDEFGHIJ";
  cout << p << '\n';
  swap(p[2],p[8]);
  cout << p << '\n';
}
ABCDEFGHIJ
ABIDEFGHCJ
```

```
transform(p,q,pp,f);
// applies the function f(x) to each x in [p,q[ and copies the result
// into [pp,pp+n[, where n = q-p;
// invariant: [p,q[ remains unchanged;
```

EXAMPLE E.63 Testing the `transform()` Algorithm

```
char capital(char);

int main()
{ char* s="All is flux, nothing is stationary.";  // Heraclitus
  int len = strlen(s);
  char buffer[80];
  char* ss = transform(s,s+len,buffer,capital);
  *ss = 0;  // truncate buffer
  cout << s << '\n';
  cout << buffer << '\n';
}

char capital(char c)
{ return (isalpha(c) ? toupper(c) : c);
}
All is flux, nothing is stationary.
ALL IS FLUX, NOTHING IS STATIONARY.
```

```
unique(p,q);
// removes all adjacent duplicates in [p,q[ shifting their suffixes left;
// returns the position that follows the last shifted element;
```

EXAMPLE E.64 Testing the `unique()` Algorithm

```
int main()
{ char* s="All is flux, nothing is stationary.";  // Heraclitus
  int len = strlen(s);
  cout << s << '\n';
  sort(s,s+len);
  cout << s << '\n';
  char* ss = unique(s,s+len);
  cout << s << '\n';
  *ss = 0;  // truncate buffer
```

```
  cout << s << '\n';
}
All is flux, nothing is stationary.
     ,.Aaafghiiiilllnnnoorsssttttuxy
 ,.Aafghilnorstuxyllnnnoorsssttttuxy
 ,.Aafghilnorstuxy
```

```
unique_copy(p,q,pp);
// copies the nonduplicate elements of [p,q[ into [pp,pp+n[,
// where n is the number of unique elements in [p,q[;;
// returns pp+n;
// invariant: [p,q[ is left unchanged;
```

EXAMPLE E.65 Testing the `unique_copy()` Algorithm

```
int main()
{ char* s="All is flux, nothing is stationary.";  // Heraclitus
  int len = strlen(s);
  cout << s << '\n';
  sort(s,s+len);
  cout << s << '\n';
  char buffer[80];
  char* ss = unique_copy(s,s+len,buffer);
  *ss = 0;  // truncate buffer for printing
  cout << s << '\n';
  cout << buffer << '\n';
}
All is flux, nothing is stationary.
     ,.Aaafghiiiilllnnnoorsssttttuxy
     ,.Aaafghiiiilllnnnoorsssttttuxy
 ,.Aafghilnorstuxy
```

```
upper_bound(p,q,x);
// returns the position that immediately follows the last occurrence
// of x in [pp,qq[;
// precondition: [p,q[ must be sorted;
// invariant: [p,q[ is left unchanged;
```

EXAMPLE E.66 Testing the `upper_bound()` Algorithm

```
int main()
{ int a[] = {11,22,22,33,44,44,44,55,66};
  int* p = upper_bound(a,a+9,44);
  cout << "*p=" << *p << '\n';
  cout << "p-a=" << p-a << '\n';
}
*p=55
p-a=7
```

Appendix F

The Standard C Library

This appendix describes the pre-defined functions provided in the Standard C Library. Each entry lists the function name, its prototype, a brief description of what it does, and the header file where it is declared.

Function	Prototype and Description	Header File
`abort()`	`void abort();` Aborts the program.	`<cstdlib>`
`abs()`	`int abs(int n);` Returns the absolute value of `n`.	`<cstdlib>`
`acos()`	`double acos(double x);` Returns the inverse cosine (arccosine) of `x`.	`<cmath>`
`asin()`	`double asin(double x);` Returns the inverse sine (arcsine) of `x`.	`<cmath>`
`atan()`	`double atan(double x);` Returns the inverse tangent (arctangent) of `x`.	`<cmath>`
`atof()`	`double atof(const char* s);` Returns the number represented literally in the string `s`.	`<cstdlib>`
`atoi()`	`int atoi(const char* s);` Returns the integer represented literally in the string `s`.	`<cstdlib>`
`atol()`	`long atol(const char* s);` Returns the integer represented literally in the string `s`.	`<cstdlib>`
`bad()`	`int ios::bad();` Returns nonzero if `badbit` is set; returns 0 otherwise.	`<iostream>`

`bsearch()`	`void* bsearch(const void* x, void* a,`
	`size_t n, size_t s,`
	`int (*cmp)(const void, *const void*));`
	Implements the Binary Search Algorithm to search for `x` in the sorted array `a` of `n` elements each of size `s` using the function `*cmp` to compare any two such elements. If found, a pointer to the element is returned; otherwise, the `NULL` pointer is returned.
		`<cstdlib>`

`ceil()`	`double ceil(double x);` Returns `x` rounded up to the next whole number.	`<cmath>`
`clear()`	`void ios::clear(int n=0);` Changes stream state to `n`.	`<iostream>`
`clearerr()`	`void clearerr(FILE* p);` Clears the end-of-file and error flags for the file `*p`.	`<cstdio>`
`close()`	`void fstreambase::close();` Closes the file attached to the owner object.	`<fstream>`

cos()	double cos(double x); Returns the inverse cosine of x.	<cmath>
cosh()	double cosh(double x); Returns the hyperbolic cosine of x: $(e^x + e^{-x})/2$.	<cmath>
difftime()	double difftime(time_t t1, time_t t0); Returns time elapsed (in seconds) from time t0 to time t1.	<ctime>
eof()	int ios::eof(); Returns nonzero if eofbit is set; returns 0 otherwise.	<iostream>
exit()	void exit(int n); Terminates the program and returns n to the invoking process.	<cstdlib>
exp()	double exp(double x); Returns the exponential of x: e^x.	<cmath>
fabs()	double fabs(double x); Returns the absolute value of x .	<cmath>
fail()	int ios::fail(); Returns nonzero if failbit is set; returns 0 otherwise.	<iostream>
fclose()	int fclose(FILE* p); Closes the file *p and flushes all buffers. Returns 0 if successful; returns EOF otherwise.	<cstdio>
fgetc()	int fgetc(FILE* p); Reads and returns the next character from the file *p if possible; returns EOF otherwise.	<cstdio>
fgets()	char* fgets(char* s, int n, FILE* p); Reads the next line from the file *p and stores it in *s. The "next line" means either the next n-1 characters or all the characters up to the next endline character, whichever comes first. The NUL character is appended to the characters stored in s. Returns s if successful; returns NULL otherwise.	<cstdio>

fill()	char ios::fill();
	Returns the current fill character.
	char ios::fill(char c);
	Changes the current fill character to c and returns the previous fill character.
		<iostream>

flags()	long ios::flags();
	Returns the current format flags.
	long ios::flags(long n);
	Changes the current format flags to n; returns previous flags.
		<iostream>

floor()	double floor(double x); Returns x rounded down to the next whole number.	<cmath>
flush()	ostream& ostream::flush(); Flushes the output buffer and returns the updates stream.	<iostream>

fopen()	FILE* fopen(const char* p, const char* s); Opens the file *p and returns the address of the structure that represents the file if successful; returns NULL otherwise. The string s determines the file's *mode*: "r" for *read*, "w" for *write*, "a" for *append*, "r+" for reading and writing an existing file, "w+" for reading and writing an existing file, and "a+" for reading and appending an existing file.	<cstdio>
fprintf()	int fprintf(FILE* p, const char* s, ...); Writes formatted output to the file *p. Returns the number of characters printed if successful; otherwise it returns a negative number.	<cstdio>
fputc()	int fputc(int c, FILE* p); Writes character c to the file *p. Returns the character written or EOF if unsuccessful.	<cstdio>
fputs()	int fputs(const char* s, FILE* p); Writes string s to the file *p. Returns a nonnegative integer if successful; otherwise it returns EOF.	<cstdio>
fread()	size_t fread(void* s, size_t k, size_t n, FILE* p); Reads up to n items each of size k from the file *p and stores them at location s in memory. Returns the number of items read.	<cstdio>
fscanf()	int fscanf(FILE* p, const char* s, ...); Reads formatted input from the file *p and stores them at location s in memory. Returns EOF if end of file is reached; otherwise it returns the number of items read into memory.	<cstdio>
fseek()	int fseek(FILE* p, long k, int base); Repositions the position marker of the file *p k bytes from its base, where base should be SEEK_SET for the beginning of the file, SEEK_CUR for the current position of the file marker, or SEEK_END for the end of the file. Returns 0 if successful.	<cstdio>
ftell()	long ftell(FILE* p); Returns location of the position marker in file *p or returns -1.	<cstdio>
fwrite()	size_t fwrite(void* s, size_t k, size_t n, FILE* p); Writes n items each of size k to the file *p and returns the number written.	<cstdio>
gcount()	int istream::gcount(); Returns the number of characters most recently read.	<iostream>

get()	int istream::get();
	istream& istream::get(signed char& c);
	istream& istream::get(unsigned char& c);
	istream& istream::get(signed char* b, int n,
	char e='\n');
	istream& istream::get(unsigned char* b, int n,
	char e='\n');
	Reads the next character c from the istream. The first version returns c or EOF. The last two versions read up to n characters into b, stopping when e is encountered.
		<iostream>

`getc()`	`int getc(FILE* p);` Same as `fgetc()` except implemented as a macro.	`<cstdio>`
`getchar()`	`int getchar();` Returns the next character from standard input or returns `EOF`.	`<cstdio>`
`gets()`	`char* gets(char* s);` Reads next line from standard input and stores it in `s`. Returns `s` or `NULL` if no characters are read.	`<cstdio>`
`good()`	`int ios::good();` Returns nonzero if stream `state` is zero; returns zero otherwise.	`<iostream>`
`ignore()`	`istream& ignore(int n=1, int e=EOF);` Extracts up to `n` characters from stream, or up to character `e`, whichever comes first. Returns the stream.	`<iostream>`
`isalnum()`	`int isalnum(int c);` Returns nonzero if `c` is an alphabetic or numeric character; returns 0 otherwise.	`<cctype>`
`isalpha()`	`int isalpha(int c);` Returns nonzero if `c` is an alphabetic character; otherwise returns 0.	`<cctype>`
`iscntrl()`	`int iscntrl(int c);` Returns nonzero if `c` is a control character; otherwise returns 0.	`<cctype>`
`isdigit()`	`int isdigit(int c);` Returns nonzero if `c` is a digit character; otherwise returns 0.	`<cctype>`
`isgraph()`	`int isgraph(int c);` Returns nonzero if `c` is any non-blank printing character; otherwise returns 0.	`<cctype>`
`islower()`	`int islower(int c);` Returns nonzero if `c` is a lowercase alphabetic character; otherwise returns 0.	`<cctype>`
`isprint()`	`int isprint(int c);` Returns nonzero if `c` is any printing character; otherwise returns 0.	`<cctype>`
`ispunct()`	`int ispunct(int c);` Returns nonzero if `c` is any punctuation mark, except the alphabetic characters, the numeric characters, and the blank character; otherwise 0 is returned.	`<cctype>`
`isspace()`	`int isspace(int c);` Returns nonzero if `c` is any white-space character, including the blank `' '`, the form feed `'\f'`, the newline `'\n'`, the carriage return `'\r'`, the horizontal tab `'\t'`, and the vertical tab `'\v'`; otherwise returns 0.	`<cctype>`
`isupper()`	`int isupper(int c);` Returns nonzero if `c` is an uppercase alphabetic character; otherwise returns 0.	`<cctype>`
`isxdigit()`	`int isxdigit(int c);` Returns nonzero if `c` is one of the 10 digit characters or one of the 12 hexadecimal digit letters: `'a'`, `'b'`, `'c'`, `'d'`, `'e'`, `'f'`, `'A'`, `'B'`, `'C'`, `'D'`, `'E'`, or `'F'`; otherwise returns 0.	`<cctype>`

`labs()`	`long labs(long n);` Returns absolute value of `n`.	`<cstdlib>`
`log()`	`double log(double x);` Returns the natural logarithm (base e) of `x`.	`<cmath>`
`log10()`	`double log10(double x);` Returns the common logarithm (base 10) of `x`.	`<cmath>`
`memchr()`	`void* memchr(const void* s, int c, size_t k);` Searches the `k` bytes of memory beginning at `s` for character `c`. If found, the address of its first occurrence is returned. Returns `NULL` otherwise.	`<string>`
`memcmp()`	`int memcmp(const void* s1, const void* s2, size_t k);` Compares the `k` bytes of memory beginning at `s1` with the `k` bytes of memory beginning at `s2` and returns a negative, zero, or a positive integer according to whether the first string is lexicographically less than, equal to, or greater than the second string.	`<string>`
`memcpy()`	`void* memcpy(const void* s1, const void* s2, size_t k);` Copies the `k` bytes of memory beginning at `s2` into memory location `s1` and returns `s1`.	`<string>`
`memmove()`	`int memmove(const void* s1, const void* s2, size_t k);` Same as `memcpy()` except strings may overlap.	`<string>`

`open()`	`void fstream::open(const char* f, int m, int p=filebuf::openprot);`
	`void ifstream::open(const char* f, int m=ios::in, int p=filebuf::openprot);`
	`void ofstream::open(const char* f, int m=ios::out, int p=filebuf::openprot);`
	Opens the file `f` in mode `m` with protection `p`.
		`<fstream>`

`peek()`	`int istream:: peek();` Returns next character (or `EOF`) from stream without extracting it.	`<iostream>`
`pow()`	`double pow(double x, double y);` Returns `x` raised to the power `y` (x^y).	`<cmath>`

`precision()`	`int ios::precision();`
	`int ios::precision(int k);`
	Returns the current precision for the stream. The second version changes the current precision to `k` and returns the old precision.
		`<iostream>`

`tolower()`	`int tolower(int c);` Returns the lowercase version of `c` if `c` is an uppercase alphabetic character; otherwise returns `c`.	`<cctype>`
`toupper()`	`int toupper(int c);` Returns the uppercase version of `c` if `c` is a lowercase alphabetic character; otherwise returns `c`.	`<cctype>`

Appendix G

Hexadecimal Numbers

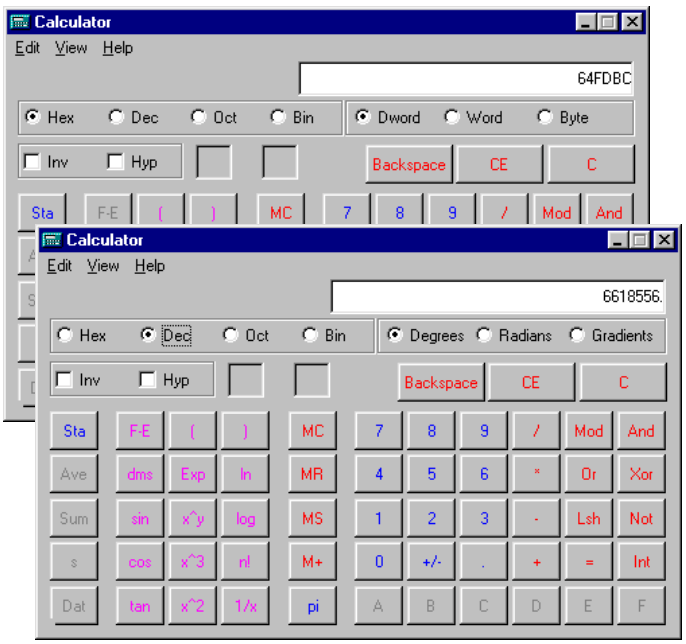
Humans normally use the base 10 numbering system. This is called the *decimal* system for the Greek word *deka* for "ten." Our ancient ancestors learned it by counting with their 10 fingers.

Computers have only 2 fingers (*i.e.*, there are only 2 possible values for each bit), so the *binary* system works well for computers. But the trouble with binary numbers is that their representations require long strings of bits. For example, 1996 is represented as `11111001100` in binary. Most humans find long strings like that difficult to process.

Binary numbers are easy to convert to other bases if the base is a power of 2. For example, conversion between binary and octal (base 8 = 2^3) merely requires grouping the binary bits into groups of 3 and interpreting each triplet as an octal digit. For example, to convert the binary numeral `11111001100` write `11,111,001,100 = 3714`. Here, `11` converts to `3`, `111` converts to `7`, `001` converts to `1`, and `100` converts to `4`. Conversion from octal back to binary is just as simple. For example, `2650` converts to `10110101000`, which is 1448 in decimal. Note that octal numerals use only the first 8 decimal digits: `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`.

After 8, the next power of 2 is 16. Using that base makes the numerals even shorter. This is called the *hexadecimal* system (from the Greek *hex* + *deka* for "six" + "ten"). Conversion between binary and hexadecimal is just as simple as it is between binary and octal. For example, to convert the binary numeral `10111010100` to hexadecimal, group the bits into groups of 4 (from right to left) and then translate each group into the corresponding hexadecimal digit: `101,1101,0100 = 5d4`. Here, `101` converts to `5`, `1101` converts to 11, and `0100` converts to `4`. The hexadecimal digits `10`, `11`, `12`, `13`, `14`, and `15` are denoted by the first six letters of the alphabet: `a`, `b`, `c`, `d`, `e`, `f`.

Most operating systems provide a calculator utility that converts number representations between hexadecimal, decimal, octal, and binary. For example, the Calculator utility in Microsoft Windows is located in Start > Programs > Accessories. In that application, to convert from hexadecimal to decimal, select Scientific from its View menu, select the Hex radio button, enter the hexadecimal representation of the number, and then select the Dec radio buttons. The example here shows that 0x0064fdbc is hexadecimal notation for 6,618,556.



The *output manipulators* `dec`, `hex`, and `oct` are used for converting different bases, as the next example illustrates.

EXAMPLE G.1 Using Output Manipulators

This shows how both the value and the address of a variable can be printed:

```
int main()
{ int n = 1492;  //  base 10
  cout << "Base  8: n = " << oct << n << endl;
  cout << "Base 10: n = " << n << endl;
  cout << "Base 16: n = " << hex << n << endl;
}
```

```
Base  8: n = 2724
Base 10: n = 1492
Base 16: n = 5d4
```

Here the manipulator `oct` is used to convert the next output to octal form. Note that the output reverts back to decimal until the `hex` manipulator is used.

The next example shows how to input integers in octal and hexadecimal. Octal numerals are denoted with a `0` prefix, and hexadecimal numerals are denoted with a `0x` prefix.

EXAMPLE G.2 Using Input Manipulators

This shows how both the value and the address of a variable can be printed:

```
int main()
{ int n;
  cout << "Enter an octal numeral (use 0 prefix): ";
  cin >> oct >> n;
  cout << "Base  8: n = " << oct << n << endl;
  cout << "Base 10: n = " << dec << n << endl;
  cout << "Base 16: n = " << hex << n << endl;
  cout << "Enter a decimal numeral: ";
  cin >> dec >> n;
  cout << "Base  8: n = " << oct << n << endl;
  cout << "Base 10: n = " << dec << n << endl;
  cout << "Base 16: n = " << hex << n << endl;
  cout << "Enter a hexadecimal numeral (use 0x prefix): ";
  cin >> hex >> n;
  cout << "Base  8: n = " << oct << n << endl;
  cout << "Base 10: n = " << dec << n << endl;
  cout << "Base 16: n = " << hex << n << endl;
}
```

```
Enter an octal numeral (use 0 prefix): 0777
Base  8: n = 777
Base 10: n = 511
Base 16: n = 1ff
Enter a decimal numeral: 511
Base  8: n = 777
Base 10: n = 511
```

```
Base 16: n = 1ff
Enter a hexadecimal numeral (use 0x prefix): 0x1ff
Base  8: n = 777
Base 10: n = 511
Base 16: n = 1ff
```

Algorithm G.1 Decimal Integer to Hexadecimal

To convert the integer x into its equivalent hexadecimal numeral:

1. Assert $x > 0$.
2. Set $k = 0$.
3. Divide x by 16, setting x equal to the (integer) quotient.
4. Set h_k equal to the remainder from the previous division. Use one of the 16 *hexadecimal digits* 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f, representing the numbers 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, for h_k.
5. Add 1 to k.
6. If $x > 0$, repeat steps 3–6.
7. Return $h_k \cdots h_2 h_1 h_0$ (*i.e.*, the hexadecimal numeral whose jth hex symbol is h_j)

EXAMPLE G.3 Converting the Decimal Numeral 100,000 to Hexadecimal

Applying Algorithm G.1 to the decimal number 100,000 yields $100000_{10} = h_4 h_3 h_2 h_1 h_0 = 186a0_{16}$:

k	x	h_k
0	100000	
1	6250	0
2	390	a
3	24	6
4	1	8
8	0	1

Algorithm G.2 Hexadecimal Integer to Decimal

To convert the hexadecimal integer $h_k \cdots h_2 h_1 h_0$ into its equivalent decimal numeral:

1. Set $x = 0$.
2. Set $j = k + 1$ (the actual number of bits in the hexadecimal string).
3. Subtract 1 from j.
4. Multiply x by 16.
5. Add h_j to x.
6. If $j > 0$, repeat steps 3–6.
7. Return x.

EXAMPLE G.4 Converting the Hexadecimal Numeral `f4d9` to Decimal

Convert `f4d9` to decimal:

j	h_j	$x = 2x + h_j$	
4			0
3	f	16·0 + f =	15
2	4	16·15 + 4 =	244
1	d	16·244 + 13 =	3917
0	9	16·3917 + 9 =	62,681

So $\texttt{f4d9}_{16} = \texttt{62,681}_{10}$.

EXAMPLE G.5 Converting the Hexadecimal Numeral `543ab` to Decimal

Converting `543ab` to decimal:

j	h_j	$x = 2x + h_j$	
5			0
4	5	16·0 + 5 =	5
3	4	16·5 + 4 =	84
2	3	16·84 + 3 =	1347
1	a	16·1347 + a =	21,562
0	b	16·21,562 + b =	345,003

So $543ab_{16} = 345{,}003_{10}$.

Appendix H

References

[Adams]
C++ An Introduction to Computing, by Joel Adams, Sanford Leestma, and Larry Nyhoff.
Prentice Hall, Englewood Cliffs, NJ (1995) 0-02-369402-5.

[Barton]
Scientific and Engineering C++, by John J. Barton and Lee R. Nackman.
Addison-Wesley Publishing Company, Reading, MA (1994) 0-201-53393-6.

[Bergin]
Data Abstraction, the Object-Oriented Approach Using C++, by Joseph Bergin.
McGraw-Hill, Inc., New York, NY (1994) 0-07-911691-4.

[Bronson]
A First Book of C++, by Gary J. Bronson.
West Publishing Company, St. Paul, MN (1995) 0-314-04236-9.

[Budd]
Classic Data Structures in C++, by Timothy A. Budd.
Addison-Wesley Publishing Company, Reading, MA (1994) 0-201-50889-3.

[Capper]
Introducing C++ for Scientists, Engineers and Mathematicians, by D. M. Capper.
Springer-Verlag, London (1994) 3-540-19847-4.

[Cargill]
C++ Programming Style, by Tom Cargill.
Addison-Wesley Publishing Company, Reading, MA (1992) 0-201-56365-7.

[Carrano]
Data Abstraction and Problem Solving with C++, by Frank M. Carrano.
Benjamin/Cummings Publishing Company, Redwood City, CA (1993) 0-8053-1226-9.

[Carroll]
Designing and Coding Reusable C++, by Martin D. Carroll and Margaret A. Ellis.
Addison-Wesley Publishing Company, Reading, MA (1995) 0-201-51284-X.

[Cline]
C++ FAQs, Second Edition, by Marshall Cline, Greg Lomow, and Mike Girou.
Addison-Wesley Publishing Company, Reading, MA (1999) 0-201-30983-1.

[Coplien]
Advanced C++, Programming Styles and Idioms, by James O. Coplien.
Addison-Wesley Publishing Company, Reading, MA (1992) 0-201-54855-0.

[Deitel]
C++ How to Program, Second Edition by H. M. Deitel and P. J. Deitel.
Prentice Hall, Englewood Cliffs, NJ (1998) 0-13-528910-6.

[Dewhurst]
Programming in C++, Second Edition, by Stephen C. Dewhurst and Kathy T. Stark.
Prentice Hall, Englewood Cliffs, NJ (1995) 0-13-182718-9.

[Ellis]
The Annotated C++ Reference Manual, by Margaret A. Ellis and Bjarne Stroustrup.
Addison-Wesley Publishing Company, Reading, MA (1992) 0-201-51459-1.

[Friedman]
Problem Solving, Abstraction, and Design Using C++, by F. L. Friedman and E. B. Koffman.
Addison-Wesley Publishing Company, Reading, MA (1994) 0-201-52649-2.

[Hansen]
The C++ Answer Book, by Tony L. Hansen.
Addison-Wesley Publishing Company, Reading, MA (1990) 0-201-11497-6.

[Headington]
Data Abstraction and Structures Using C++, by Mark R. Headington and David D. Riley.
D. C. Heath and Company, Lexington, MA (1994) 0-669-29220-6.

[Horowitz]
Fundamentals of Data Structures in C++, by Ellis Horowitz, Sartaj Sahni, and Dinesh Mehta.
W. H. Freeman and Company, New York, NY (1995) 0-7167-8292-8.

[Hubbard1]
Fundamentals of Computing with C++, by John R. Hubbard.
McGraw-Hill, Inc, New York, NY (1998) 0-07-030868-3.

[Hubbard2]
Data Structures with C++, by John R. Hubbard.
McGraw-Hill, Inc, New York, NY (1999) 0-07-135345-3.

[Hughes]
Mastering the Standard C++ Classes, by Cameron Hughes and Tracey Hughes.
John Wiley & Sons, Inc, New York, NY (1999) 0-471-32893-6.

[Johnsonbaugh]
Object-Oriented Programming in C++, by Richard Johnsonbaugh and Martin Kalin.
Prentice Hall, Englewood Cliffs, NJ (1995) 0-02-360682-7.

[Josuttis]
The C++ Standard Library, by Nicolai M. Josuttis.
Addison-Wesley Publishing Company, Reading, MA, 1999, 0-201-37926-0.

[Knuth1]
The Art of Computer Programming, Vol. 1: *Fundamental Algorithms*, Third Edition, by Donald E. Knuth.
Addison-Wesley Publishing Company, Reading, MA, 1997, 0-201-89683-4.

[Knuth2]
The Art of Computer Programming, Vol. 2: *Seminumerical Algorithms*, Third Edition, by D. E. Knuth.
Addison-Wesley Publishing Company, Reading, MA, 1998, 0-201-89684-2.

[Knuth3]
The Art of Computer Programming, Vol. 3: *Sorting and Searching*, Second Edition, by Donald E. Knuth.
Addison-Wesley Publishing Company, Reading, MA, 1998, 0-201-89685-0.

[Ladd]
C++ Templates and Tools, by Scott Robert Ladd.
M&T Books, New York, NY (1995) 0-55851-437-6.

[Lippman]
The C++ Primer, Third Edition, by Stanley B. Lippman and Josee Lajoie.
Addison-Wesley Publishing Company, Reading, MA (1998) 0-201-82470-1.

[Meyers]
More Effective C++, by Scott Meyers.
Addison-Wesley Publishing Company, Reading, MA (1996) 0-201-63371-X.

[Model]
Data Structures, Data Abstraction: A Contemporary Introduction Using C++, by M. L. Model.
Prentice Hall, Englewood Cliffs, NJ (1994) 0-13-088782-X.

[Murray]
C++ Strategies and Tactics, by Robert B. Murray.
Addison-Wesley Publishing Company, Reading, MA (1993) 0-201-56382-7.

[Nelson]
C++ Programmers Guide to the Standard Template Library, by Mark Nelson.
IDG Books Worldwide, Inc., Foster City, CA (1995) 0-56884-314-3.

[Oualline]
Practical C++ Programming, by Steve Oualline.
O'Reilly & Associates, Sebastopol, CA (1995) 1-56592-139-9.

[Perry]
An Introduction to Object-Oriented Design in C++, by Jo Ellen Perry and Harold D. Levin.
Addison-Wesley Publishing Company, Reading, MA (1996) 0-201-76564-0.

[Plauger1]
The Standard C Library, by P. J. Plauger.
Prentice Hall, Englewood Cliffs, NJ (1992) 0-13-131509-9.

[Plauger2]
The Draft Standard C++ Library, by P. J. Plauger.
Prentice Hall, Englewood Cliffs, NJ (1995) 0-13-117003-1.

[Pohl.1]
Object-Oriented Programming Using C++, by Ira Pohl.
The Benjamin/Cummings Publishing Company, Inc, Redwood City, CA (1993) 0-8053-5384-4.

[Pohl.2]
C++ for Pascal Programmers, Second Edition, by Ira Pohl.
The Benjamin/Cummings Publishing Company, Inc, Redwood City, CA (1994) 0-8053-3158-1.

[Prata]
C++ Primer Plus, by Stephen Prata.
The Waite Group, Corte Madera, CS (1998) 1-57169-131-6.

[Ranade & Zamir]
C++ Primer for C Programmers, by Jay Ranade and Saba Zamir.
McGraw-Hill, Inc., New York, NY (1994) 0-07-051487-9.

[Satir]
C++: The Core Language, by Gregory Satir and Doug Brown.
O'Reilly & Associates, Sebastopol, CA (1995) 0-56592-116-X.

[Savitch]
Problem Solving with C++, by Walter Savitch.
Addison-Wesley Publishing Company, Reading, MA (1996) 0-8053-7440-X.

[Sedgewick]
Algorithms in C++ Parts 1-4, Third Edition, by Robert Sedgewick.
Addison-Wesley Publishing Company, Reading, MA (1998) 0-201-35088-2.

[Sengupta]
C++ Object-Oriented Data Structures, by Saumyendra Sengupta and Carl Phillip Korobkin.
Springer-Verlag, New York, NY (1994) 0-387-94194-0

[Shammas]
Advanced C++, by Namir Clement Shammas.
SAMS Publishing, Carmel, IN (1992) 0-672-30158-X.

[Stepanov]
"The Standard Template Library," *Technical Report HPL-94-34*, by A. A. Stepanov and M. Lee.
Hewlett-Packard Laboratories, April 1994.

[Stroustrup1]
The C++ Programming Language, Special Edition, by Bjarne Stroustrup.
Addison-Wesley Publishing Company, Reading, MA (2000) 0-201-70073-5.

[Stroustrup2]
The Design and Evolution of C++, by Bjarne Stroustrup.
Addison-Wesley Publishing Company, Reading, MA (1994) 0-201-54330-3.

[Teale]
C++ IOStreams, by Steve Teale.
Addison-Wesley Publishing Company, Reading, MA (1993) 0-201-59641-5.

[Trudeau]
Mastering CodeWarrior for Windows 95/NT, The Official Guide, by Jim Trudeau.
SYBEX, Alameda, CA (`http://www.sybex.com/`), 1997, 1-7821-2057-1.

[Unicode]
The Unicode Standard, Version 2.0, by The Unicode Consortium.
Addison-Wesley, Reading, MA (`http://www2.awl.com/corp/`), 1996, 0-201-48345-9.

[Wang]
C++ with Object-Oriented Programming, by Paul S. Wang.
PWS Publishing Company, Boston, MA (1994) 0-534-19644-6.

[Weiss]
Data Structures and Algorithm Analysis in C++, by Mark Allen Weiss.
Benjamin/Cummings Publishing Company, Redwood City, CA (1994) 0-8053-5443-3.

[Winston]
On to C++, by Patrick Henry Winston.
Addison-Wesley Publishing Company, Reading, MA (1994) 0-201-58043-8.

Index