Fundamentals of C++ Programming

DRAFT

Richard L. Halterman
School of Computing
Southern Adventist University

December 2, 2018

Copyright © 2008–2018 Richard L. Halterman. All rights reserved.

Contents

1 The Context of Software Development **1**

1.1 Software 2

1.2 Development Tools 2

1.3 Learning Programming with C++ 6

1.4 Exercises 6

2 Writing a C++ Program **7**

2.1 General Structure of a Simple C++ Program 7

2.2 Editing, Compiling, and Running the Program 8

2.3 Variations of our simple program 9

2.4 Template for simple C++ programs 12

2.5 Exercises 13

3 Values and Variables **15**

3.1 Integer Values 15

3.2 Variables and Assignment 18

3.3 Identifiers 21

3.4 Additional Integer Types 24

3.5 Floating-point Types 25

3.6 Constants 27

3.7 Other Numeric Types 28

3.8 Characters 28

3.9 Enumerated Types 31

3.10 Type Inference with auto 32

3.11 Exercises 33

4 Expressions and Arithmetic **37**

©2018 Richard L. Halterman

4.1 Expressions . . . 37
4.2 Mixed Type Expressions . . . 41
4.3 Operator Precedence and Associativity . . . 44
4.4 Comments . . . 46
4.5 Formatting . . . 47
4.6 Errors and Warnings . . . 50
4.6.1 Compile-time Errors . . . 50
4.6.2 Run-time Errors . . . 51
4.6.3 Logic Errors . . . 52
4.6.4 Compiler Warnings . . . 53
4.7 Arithmetic Examples . . . 55
4.8 Integers vs. Floating-point Numbers . . . 58
4.8.1 Integer Implementation . . . 59
4.8.2 Floating-point Implementation . . . 64
4.9 More Arithmetic Operators . . . 69
4.10 Bitwise Operators . . . 72
4.11 Algorithms . . . 76
4.12 Exercises . . . 78

5 Conditional Execution 85

5.1 Type bool . . . 85
5.2 Boolean Expressions . . . 86
5.3 The Simple if Statement . . . 88
5.4 Compound Statements . . . 91
5.5 The if/else Statement . . . 93
5.6 Compound Boolean Expressions . . . 97
5.7 Nested Conditionals . . . 100
5.8 Multi-way if/else Statements . . . 111
5.9 Errors in Conditional Statements . . . 116
5.10 Exercises . . . 117

6 Iteration 123

6.1 The while Statement . . . 123
6.2 Nested Loops . . . 133
6.3 Abnormal Loop Termination . . . 140

6.3.1 The break statement . . . 140
6.3.2 The goto Statement . . . 141
6.3.3 The continue Statement . . . 143
6.4 Infinite Loops . . . 144
6.5 Iteration Examples . . . 148
6.5.1 Drawing a Tree . . . 148
6.5.2 Printing Prime Numbers . . . 150
6.6 Exercises . . . 153

7 Other Conditional and Iterative Statements 159
7.1 The switch Statement . . . 159
7.2 The Conditional Operator . . . 164
7.3 The do/while Statement . . . 165
7.4 The for Statement . . . 167
7.5 Exercises . . . 173

8 Using Functions 179
8.1 Introduction to Using Functions . . . 181
8.2 Standard Math Functions . . . 186
8.3 Maximum and Minimum . . . 190
8.4 clock Function . . . 191
8.5 Character Functions . . . 192
8.6 Random Numbers . . . 193
8.7 Exercises . . . 197

9 Writing Functions 201
9.1 Function Basics . . . 202
9.2 Using Functions . . . 210
9.3 Pass by Value . . . 215
9.4 Function Examples . . . 217
9.4.1 Better Organized Prime Generator . . . 217
9.4.2 Command Interpreter . . . 219
9.4.3 Restricted Input . . . 220
9.4.4 Better Die Rolling Simulator . . . 222
9.4.5 Tree Drawing Function . . . 223

9.4.6 Floating-point Equality . . . 224
9.4.7 Multiplication Table with Functions . . . 227
9.5 Organizing Functions . . . 230
9.6 Commenting Functions . . . 233
9.7 Custom Functions vs. Standard Functions . . . 234
9.8 Exercises . . . 236

10 Managing Functions and Data 241
10.1 Global Variables . . . 241
10.2 Static Variables . . . 249
10.3 Overloaded Functions . . . 251
10.4 Default Arguments . . . 252
10.5 Recursion . . . 254
10.6 Making Functions Reusable . . . 260
10.7 Pointers . . . 266
10.8 Reference Variables . . . 271
10.9 Pass by Reference . . . 274
10.9.1 Pass by Reference via Pointers . . . 274
10.9.2 Pass by Reference via References . . . 276
10.10 Higher-order Functions . . . 277
10.11 Exercises . . . 280

11 Sequences 289
11.1 Vectors . . . 291
11.1.1 Declaring and Using Vectors . . . 291
11.1.2 Traversing a Vector . . . 295
11.1.3 Vector Methods . . . 299
11.1.4 Vectors and Functions . . . 302
11.1.5 Multidimensional Vectors . . . 307
11.2 Arrays . . . 311
11.2.1 Static Arrays . . . 311
11.2.2 Pointers and Arrays . . . 316
11.2.3 Dynamic Arrays . . . 322
11.2.4 Copying an Array . . . 326
11.2.5 Multidimensional Arrays . . . 330

11.2.6 C Strings . . . 333
11.2.7 Command-line Arguments . . . 336
11.3 Vectors vs. Arrays . . . 338
11.4 Prime Generation with a Vector . . . 342
11.5 Exercises . . . 345

12 Sorting and Searching 351
12.1 Sorting . . . 351
12.2 Flexible Sorting . . . 354
12.3 Search . . . 356
12.3.1 Linear Search . . . 356
12.3.2 Binary Search . . . 359
12.4 Vector Permutations . . . 369
12.5 Randomly Permuting a Vector . . . 375
12.6 Exercises . . . 381

13 Standard C++ Classes 383
13.1 String Objects . . . 384
13.2 Input/Output Streams . . . 388
13.3 File Streams . . . 391
13.4 Complex Numbers . . . 397
13.5 Better Pseudorandom Number Generation . . . 398
13.6 Exercises . . . 405

14 Custom Objects 407
14.1 Object Basics . . . 407
14.2 Instance Variables . . . 409
14.3 Member Functions . . . 415
14.4 Constructors . . . 422
14.5 Defining a New Numeric Type . . . 425
14.6 Encapsulation . . . 427
14.7 Exercises . . . 430

15 Fine Tuning Objects 437
15.1 Passing Object Parameters . . . 437
15.2 Pointers to Objects and Object Arrays . . . 439

15.3 The `this` Pointer . . . 442
15.4 `const` Methods . . . 445
15.5 Separating Method Declarations and Definitions . . . 446
15.6 Preventing Multiple Inclusion . . . 453
15.7 Overloaded Operators . . . 456
15.7.1 Operator Functions . . . 456
15.7.2 Operator Methods . . . 459
15.8 `static` Members . . . 460
15.9 Classes vs. structs . . . 464
15.10 Friends . . . 465
15.11 Exercises . . . 470

16 Building some Useful Classes 475

16.1 A Better Rational Number Class . . . 475
16.2 Stopwatch . . . 477
16.3 Sorting with Logging . . . 483
16.4 Automating Testing . . . 487
16.5 Convenient High-quality Pseudorandom Numbers . . . 491
16.6 Exercises . . . 493

17 Inheritance and Polymorphism 495

17.1 I/O Stream Inheritance . . . 495
17.2 Inheritance Mechanics . . . 497
17.3 Uses of Inheritance . . . 499
17.4 Polymorphism . . . 507
17.5 Alternative to Inheritance and Polymorphism . . . 513
17.6 Adapter Design Pattern . . . 520
17.7 Protected Members . . . 522
17.8 Fine Tuning Inheritance . . . 531
17.9 Exercises . . . 540

18 Memory Management 543

18.1 Memory Available to C++ Programs . . . 543
18.2 Manual Memory Management . . . 544
18.3 Linked Lists . . . 549

18.4 Resource Management 558
18.5 Rvalue References 578
18.6 Smart Pointers 590

19 Generic Programming 607
19.1 Function Templates 607
19.2 Class Templates 618
19.3 Exercises 631

20 The Standard Template Library 633
20.1 Containers 633
20.2 Iterators 640
20.3 Iterator Ranges 644
20.4 Lambda Functions 655
20.5 Algorithms in the Standard Library 661
20.6 Namespaces 679

21 Associative Containers 687
21.1 Associative Containers 687
21.2 The `std::set` Data Type 687
21.3 Tuples 693
21.4 The `std::map` Data Type 697
21.5 The `std::unordered_map` Data Type 702
21.6 Counting with Associative Containers 704
21.7 Grouping with Associative Containers 709
21.8 Memoization 712

22 Handling Exceptions 719
22.1 Motivation 719
22.2 Exception Examples 720
22.3 Custom Exceptions 728
22.4 Catching Multiple Exceptions 730
22.5 Exception Mechanics 733
22.6 Using Exceptions 736

Appendices 741

©2018 Richard L. Halterman

A Using Visual Studio 2015 to Develop C++ Programs **741**

B Command Line Development **747**

B.0.1 Visual Studio Command Line Tools 748

B.0.2 Developing C++ Programs with the GNU Tools 750

Bibliography **752**

Index **753**

Preface

Legal Notices and Information

Permission is hereby granted to make hardcopies and freely distribute the material herein under the following conditions:

- The copyright and this legal notice must appear in any copies of this document made in whole or in part.
- None of material herein can be sold or otherwise distributed for commercial purposes without written permission of the copyright holder.
- Instructors at any educational institution may freely use this document in their classes as a primary or optional textbook under the conditions specified above.

A local electronic copy of this document may be made under the terms specified for hard copies:

- The copyright and these terms of use must appear in any electronic representation of this document made in whole or in part.
- None of material herein can be sold or otherwise distributed in an electronic form for commercial purposes without written permission of the copyright holder.
- Instructors at any educational institution may freely store this document in electronic form on a local server as a primary or optional textbook under the conditions specified above.

Additionally, a hardcopy or a local electronic copy must contain the uniform resource locator (URL) providing a link to the original content so the reader can check for updated and corrected content. The current standard URL is `http://python.cs.southern.edu/cppbook/progcpp.pdf`.

If you are an instructor using this book in one or more of your courses, please let me know. Keeping track of how and where this book is used helps me justify to my employer that it is providing a useful service to the community and worthy of the time I spend working on it. Simply send a message to `halterman@southern.edu` with your name, your institution, and the course(s) in which you use it.

The source code for all labeled listings is available at

`https://github.com/halterman/CppBook-SourceCode`.

Chapter 1

The Context of Software Development

A computer program, from one perspective, is a sequence of instructions that dictate the flow of electrical impulses within a computer system. These impulses affect the computer's memory and interact with the display screen, keyboard, mouse, and perhaps even other computers across a network in such a way as to produce the "magic" that permits humans to perform useful tasks, solve high-level problems, and play games. One program allows a computer to assume the role of a financial calculator, while another transforms the machine into a worthy chess opponent. Note the two extremes here:

- at the lower, more concrete level electrical impulses alter the internal state of the computer, while
- at the higher, more abstract level computer users accomplish real-world work or derive actual pleasure.

So well is the higher-level illusion achieved that most computer users are oblivious to the lower-level activity (the machinery under the hood, so to speak). Surprisingly, perhaps, most programmers today write software at this higher, more abstract level also. An accomplished computer programmer can develop sophisticated software with little or no interest or knowledge of the actual computer system upon which it runs. Powerful software construction tools hide the lower-level details from programmers, allowing them to solve problems in higher-level terms.

The concepts of computer programming are logical and mathematical in nature. In theory, computer programs can be developed without the use of a computer. Programmers can discuss the viability of a program and reason about its correctness and efficiency by examining abstract symbols that correspond to the features of real-world programming languages but appear in no real-world programming language. While such exercises can be very valuable, in practice computer programmers are not isolated from their machines. Software is written to be used on real computer systems. Computing professionals known as *software engineers* develop software to drive particular systems. These systems are defined by their underlying hardware and operating system. Developers use concrete tools like compilers, debuggers, and profilers. This chapter examines the context of software development, including computer systems and tools.

©2018 Richard L. Halterman

1.1 Software

A computer program is an example of computer *software*. Software makes a computer a truly universal machine transforming it into the proper tool for the task at hand. One can refer to a program as a *piece* of software as if it were a tangible object, but software is actually quite intangible. It is stored on a *medium*. A hard drive, a CD, a DVD, and a USB pen drive are all examples of media upon which software can reside. The CD is not the software; the software is a pattern on the CD. In order to be used, software must be stored in the computer's memory. Typically computer programs are loaded into memory from a medium like the computer's hard disk. An electromagnetic pattern representing the program is stored on the computer's hard drive. This pattern of electronic symbols must be transferred to the computer's memory before the program can be executed. The program may have been installed on the hard disk from a CD or from the Internet. In any case, the essence that was transferred from medium to medium was a pattern of electronic symbols that direct the work of the computer system.

These patterns of electronic symbols are best represented as a sequence of zeroes and ones, digits from the binary (base 2) number system. An example of a binary program sequence is

10001011011000010001000001001110

To the underlying computer hardware, specifically the processor, a zero here and three ones there might mean that certain electrical signals should be sent to the graphics device so that it makes a certain part of the display screen red. Unfortunately, only a minuscule number of people in the world would be able to produce, by hand, the complete sequence of zeroes and ones that represent the program Microsoft Word for an Intel-based computer running the Windows 8 operating system. Further, almost none of those who could produce the binary sequence would claim to enjoy the task.

The Word program for older Mac OS X computers using a PowerPC processor works similarly to the Windows version and indeed is produced by the same company, but the program is expressed in a completely different sequence of zeroes and ones! The Intel Core i7 processor in the Windows machine accepts a completely different binary language than the PowerPC processor in the Mac. We say the processors have their own *machine language*.

1.2 Development Tools

If very few humans can (or want) to speak the machine language of the computers' processors and software is expressed in this language, how has so much software been developed over the years?

Software can be represented by printed words and symbols that are easier for humans to manage than binary sequences. Tools exist that automatically convert a higher-level description of what is to be done into the required lower-level code. Higher-level programming languages like C++ allow programmers to express solutions to programming problems in terms that are much closer to a natural language like English. Some examples of the more popular of the hundreds of higher-level programming languages that have been devised over the past 60 years include FORTRAN, COBOL, Lisp, Haskell, C, Perl, Python, Java, and C#. Most programmers today, especially those concerned with high-level applications, usually do not worry about the details of underlying hardware platform and its machine language.

One might think that ideally such a conversion tool would accept a description in a natural language, such as English, and produce the desired executable code. This is not possible today because natural languages are quite complex compared to computer programming languages. Programs called *compilers* that translate one computer language into another have been around for over 60 years, but natural language

processing is still an active area of artificial intelligence research. Natural languages, as they are used by most humans, are inherently ambiguous. To understand properly all but a very limited subset of a natural language, a human (or artificially intelligent computer system) requires a vast amount of background knowledge that is beyond the capabilities of today's software. Fortunately, programming languages provide a relatively simple structure with very strict rules for forming statements that can express a solution to any problem that can be solved by a computer.

Consider the following program fragment written in the C++ programming language:

```
subtotal = 25;
tax = 3;
total = subtotal + tax;
```

These three lines do not make up a complete C++ program; they are merely a piece of a program. The statements in this program fragment look similar to expressions in algebra. We see no sequence of binary digits. Three words, `subtotal`, `tax`, and `total`, called *variables*, are used to hold information. Mathematicians have used variables for hundreds of years before the first digital computer was built. In programming, a variable represents a value stored in the computer's memory. Familiar operators (= and +) are used instead of some cryptic binary digit sequence that instructs the processor to perform the operation. Since this program is expressed in the C++ language, not machine language, it cannot be executed directly on any processor. A C++ compiler is used to translate the C++ code into machine code.

The higher-level language code is called *source code*. The compiled machine language code is called the *target code*. The compiler translates the source code into the target machine language.

The beauty of higher-level languages is this: the same C++ source code can be compiled to different target platforms. The target platform must have a C++ compiler available. Minor changes in the source code may be required because of architectural differences in the platforms, but the work to move the program from one platform to another is far less than would be necessary if the program for the new platform had to be rewritten by hand in the new machine language. Just as importantly, when writing the program the human programmer is free to think about writing the solution to the problem in C++, not in a specific machine language.

Programmers have a variety of tools available to enhance the software development process. Some common tools include:

- **Editors**. An *editor* allows the user to enter the program source code and save it to files. Most programming editors increase programmer productivity by using colors to highlight language features. The *syntax* of a language refers to the way pieces of the language are arranged to make well-formed sentences. To illustrate, the sentence

  The tall boy runs quickly to the door.

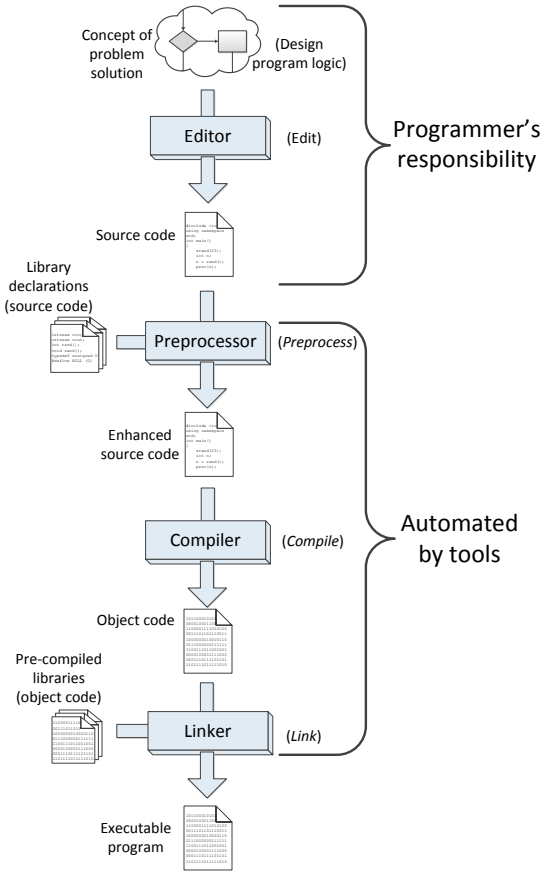
  uses proper English syntax. By comparison, the sentence

  Boy the tall runs door to quickly the.

  is not correct syntactically. It uses the same words as the original sentence, but their arrangement does not follow the rules of English.

  Similarly, programmers must follow strict syntax rules to create well-formed computer programs. Only well-formed programs are acceptable and can be compiled and executed. Some syntax-aware editors can use colors or other special annotations to alert programmers of syntax errors before the program is compiled.

Figure 1.1 Source code to target code sequence



- **Compilers**. A *compiler* translates the source code to target code. The target code may be the machine language for a particular platform or embedded device. The target code could be another source language; for example, the earliest C++ compiler translated C++ into C, another higher-level language. The resulting C code was then processed by a C compiler to produce an executable program. C++ compilers today translate C++ directly into machine language.

  The complete set of build tools for C++ includes a preprocessor, compiler, and linker:

  - **Preprocessor**—adds to or modifies the contents of the source file before the compiler begins processing the code. We use the services of the preprocessor mainly to `#include` information about library routines our programs use.
  - **Compiler**—translates C++ source code to machine code.
  - **Linker**—combines the compiler-generated machine code with precompiled library code or compiled code from other sources to make a complete executable program. Most compiled C++ code is incapable of running by itself and needs some additional machine code to make a complete executable program. The missing machine code has been precompiled and stored in a repository of code called a *library*. A program called a *linker* combines the programmer's compiled code and the library code to make a complete program.

  We generally do not think about the preprocessor, compiler, and linker working as three separate programs (although they do); the tools we use make it appear as only one process is taking place: translating our source code to an executable program.

- **Debuggers**. A *debugger* allows a programmer to more easily trace a program's execution in order to locate and correct errors in the program's implementation. With a debugger, a developer can simultaneously run a program and see which line in the source code is responsible for the program's current actions. The programmer can watch the values of variables and other program elements to see if their values change as expected. Debuggers are valuable for locating errors (also called *bugs*) and repairing programs that contain errors. (See Section 4.6 for more information about programming errors.)

- **Profilers**. A *profiler* collects statistics about a program's execution allowing developers to tune appropriate parts of the program to improve its overall performance. A profiler indicates how many times a portion of a program is executed during a particular run, and how long that portion takes to execute. Profilers also can be used for testing purposes to ensure all the code in a program is actually being used somewhere during testing. This is known as *coverage*. It is common for software to fail after its release because users exercise some part of the program that was not executed anytime during testing. The main purpose of profiling is to find the parts of a program that can be improved to make the program run faster.

The programming components of the development process are illustrated in Figure 1.1.

Many developers use integrated development environments (IDEs). An IDE includes editors, debuggers, and other programming aids in one comprehensive program. Examples of IDEs for C++ include Microsoft's Visual Studio 2015, the Eclipse Foundation's Eclipse CDT, and Apple's XCode.

Despite the plethora of tools (and tool vendors' claims), the programming process for all but trivial programs is not automatic. Good tools are valuable and certainly increase the productivity of developers, but they cannot write software. There are no substitutes for sound logical thinking, creativity, common sense, and, of course, programming experience.

©2018 Richard L. Halterman

1.3 Learning Programming with C++

Bjarne Stroustrup of AT&T Bell Labs created C++ in the mid 1980s. C++ is an extension of the programming language C, a product of AT&T Bell Labs from the early 1970s. C was developed to write the Unix operating system, and C is widely used for systems-level software and embedded systems development. C++ initially provided object-oriented programming features (see Chapter 13 and Chapter 14) and later added generic programming capabilities. C++'s close relationship to C allows C++ programs to utilize a large collection of code developed in C.

C++ is widely used in industry for commercial software development. It is an industrial strength programming language used for developing complex systems in business, science, and engineering. Examples of software written in C++ include Microsoft Windows 8, Microsoft Office, macOS, and Adobe Creative Suite.

In order to meet the needs of commercial software development and accomplish all that it does, C++ itself is complex. While experienced programmers can accomplish great things with C++, beginners sometimes have a difficult time with it. Professional software developers enjoy the flexible design options that C++ permits, but beginners need more structure and fewer options so they can master simpler concepts before moving on to more complex ones.

This book does not attempt to cover all the facets of the C++ programming language. Experienced programmers should look elsewhere for books that cover C++ in much more detail. The focus here is on introducing programming techniques and developing good habits. To that end, our approach avoids some of the more esoteric features of C++ and concentrates on the programming basics that transfer directly to other imperative programming languages such as Java, C#, and Python. We stick with the basics and explore more advanced features of C++ only when necessary to handle the problem at hand.

1.4 Exercises

1. What is a compiler?
2. How is compiled code different from source code?
3. What tool does a programmer use to produce C++ source code?
4. What tool(s) does a programmer use to convert C++ source code into executable machine code?
5. What does the linker do?
6. Does the linker deal with files containing source code or machine language code?
7. What does the preprocessor do to source code?
8. List several advantages developing software in a higher-level language has over developing software in machine language.
9. How can an IDE improve a programmer's productivity?
10. Name a popular C++ IDE is used by programmers developing for Microsoft Windows.
11. Name a popular C++ IDE is used by programmers developing for Apple macOS.

©2018 Richard L. Halterman

Chapter 2

Writing a C++ Program

Properly written C++ programs have a particular structure. The syntax must be correct, or the compiler will generate error messages and not produce executable machine language. This chapter introduces C++ by providing some simple example programs and associated fundamental concepts. Most of the concepts presented in this chapter are valid in many other programming languages as well. While other languages may implement the concepts using slightly different syntax, the ideas are directly transferable to other languages like C, Java, C#, and Ada.

2.1 General Structure of a Simple C++ Program

Listing 2.1 (simple.cpp) is one of the simplest C++ programs that does something:

Listing 2.1: simple.cpp

```
#include <iostream>

int main() {
    std::cout << "This is a simple C++ program!\n";
}
```

You can type the text as shown in Listing 2.1 (simple.cpp) into an editor and save it to a file named simple.cpp. The actual name of the file is irrelevant, but the name "simple" accurately describes the nature of this program. The extension .cpp is a common extension used for C++ source code.

After creating this file with a text editor and compiling it, you can run the program. The program prints the message

```
This is a simple C++ program!
```

Listing 2.1 (simple.cpp) contains four non-blank lines of code:

- `#include <iostream>`

  This line is a preprocessing directive. All preprocessing directives within C++ source code begin with a # symbol. This one directs the preprocessor to add some predefined source code to our existing

source code before the compiler begins to process it. This process is done automatically and is invisible to us.

Here we want to use an object from the `iostream` library, a collection precompiled C++ code that C++ programs (like ours) can use. The `iostream` library contains elements that handle input and output (I/O)—printing to the display, getting user input from the keyboard, and dealing with files.

One of the items used in Listing 2.1 (simple.cpp), `std::cout`, is not part of the C++ language itself. This item, along with other things related to input and output, were developed in C++, compiled, and stored in the `iostream` library. The compiler needs to be aware of these `iostream` items so it can compile our program. The `#include` directive specifies a file, called a *header*, that contains the specifications for the library code. The compiler checks how we use `std::cout` within our code against its specification in the `<iostream>` header to ensure that we are using the library code correctly.

Most of the programs we write use this `#include <iostream>` directive, and some programs we will write in the future will `#include` other headers as well.

- `int main() {`

  This specifies the real beginning of our program. Here we are declaring a function named `main`. All C++ programs must contain this function to be executable. Details about the meaning of `int` and the parentheses will appear in later chapters. More general information about functions appear in Chapter 8 and Chapter 9.

  The opening curly brace at the end of the line marks the beginning of the body of a function. The body of a function contains the statements the function is to execute.

- `std::cout << "This is a simple C++ program!\n";`

  The body of our `main` function contains only one statement. This statement directs the executing program to print the message *This is a simple C++ program!* on the screen. A statement is the fundamental unit of execution in a C++ program. Functions contain statements that the compiler translates into executable machine language instructions. C++ has a variety of different kinds of statements, and the chapters that follow explore these various kinds of statements. All statements in C++ end with a semicolon (;). A more detailed explanation of this statement appears below.

- `}`

  The closing curly brace marks the end of the body of a function. Both the open curly brace and close curly brace are required for every function definition.

Note which lines in the program end with a semicolon (;) and which do not. Do not put a semicolon after the `#include` preprocessor directive. Do not put a semicolon on the line containing `main`, and do not put semicolons after the curly braces.

2.2 Editing, Compiling, and Running the Program

C++ programmers have two options for C++ development environments. One option involves a command-line environment with a collection of independent tools. The other option is to use an IDE (see Section 1.2) which combines all the tools into a convenient package. Visual Studio is the dominant IDE on the Microsoft

Windows platform, and Apple Mac developers often use the XCode IDE. Appendix A provides an overview of how to use the Visual Studio 2015 IDE to develop a simple C++ program.

The myriad of features and configuration options in these powerful IDEs can be bewildering to those learning how to program. In a command-line environment the programmer needs only type a few simple commands into a console window to edit, compile, and execute programs. Some developers prefer the simplicity and flexibility of command-line build environments, especially for less complex projects.

One prominent command-line build system is the GNU Compiler Collection (`http://gcc.gnu.org`), or GCC for short. The GCC C++ compiler, called `g++`, is one of most C++ standards conforming compilers available. The GCC C++ compiler toolset is available for the Microsoft Windows, Apple Mac, and Linux platforms, and it is a free, open-source software project with a world-wide development team. Appendix B provides an overview of how to use the GCC C++ compiler.

Visual Studio and XCode offer *command line* development options as well. Appendix B provides an overview of the Visual Studio command line development process.

2.3 Variations of our simple program

Listing 2.2 (simple2.cpp) shows an alternative way of writing Listing 2.1 (simple.cpp).

Listing 2.2: `simple2.cpp`

```
#include <iostream>

using std::cout;

int main() {
    cout << "This is a simple C++ program!\n";
}
```

The `using` directive in Listing 2.2 (simple2.cpp) allows us to use a shorter name for the `std::cout` printing object. We can omit the `std::` prefix and use the shorter name, `cout`. This directive is optional, but if we omit it, we must use the longer name. The name `std` stands for "standard," and the `std` prefix indicates that `cout` is part of a collection of names called the *standard namespace*. The `std` namespace holds names for all the standard C++ types and functions that must be available to all standards-conforming C++ development environments. Components outside the standard library provided by third-party developers reside in their own separately-named namespaces. These include open-source projects and commercial libraries.

Listing 2.3 (simple3.cpp) shows another way to use the shorter name for `cout` within a C++ program.

Listing 2.3: `simple3.cpp`

```
#include <iostream>

using namespace std;

int main() {
    cout << "This is a simple C++ program!\n";
}
```

While Listing 2.2 (simple2.cpp) made the name `cout` known to the compiler via its focused `using` directive, Listing 2.3 (simple3.cpp) provides a blanket `using` directive that makes all names in the `std`

namespace available to the compiler. This approach offers some advantages for smaller programs, such as examples in books and online tutorials. This blanket `using` directive allows programmers to use shorter names as in the more more focused `using` directives, and it also can use fewer lines of code than the more focused `using` directives, especially when the program uses multiple elements from the `std` namespace.

Our choice of `using` directives (or not) makes no difference in our final product, the executable program. The compiler generates the same machine language code for all three versions—no `using`, focused `using`, and blanket `using`. We thus must select an approach that enhances our ability to write and manage our software projects.

It is important to note that while this blanket `using` approach has its place, its use generally is discouraged for more complex software projects. At this point we cannot fully appreciate the rationale for avoiding the `using namespace std` directive, but later, in Section 20.6, we will have enough experience to understand the disadvantages of the blanket `using namespace std` directive. We will strive for best practices from the start and avoid the blanket `using` statement. We generally will use the full names of the elements in the `std` namespace and use the more focused `using` directives in our code when it makes sense to do so.

The statement in the `main` function in any of the three versions of our program uses the services of an object called `std::cout`. The `std::cout` object prints text on the computer's screen. The text of the message as it appears in the C++ source code is called a *string*, for *string of characters*. Strings are enclosed within quotation marks (`"`). The symbols `<<` make up the *insertion operator*. You can think of the message to be printed as being "inserted" into the `cout` object. The `cout` object represents the output stream; that is, text that the program prints to the console window. The end of the message contains the symbol sequence `\n`. This known as a character *escape sequence*, and this combination of backslash and the letter `n` represents the *newline* character. It indicates that the printing on that line is complete, and any subsequent printing should occur on the next line. This newline character effectively causes the cursor to move down to the next line. If you read the statement from left to right, the `cout` object, which is responsible for displaying text on the screen, receives the text to print terminated with the newline character to move to the next line.

For simplicity, we'll refer to this type of statement as a *print statement*, even though the word *print* does not appear anywhere in the statement.

With minor exceptions, any statement in C++ must appear within a function definition. Our single print statement appears within the function named `main`.

Any function, including `main`, may contain multiple statements. In Listing 2.4 (arrow.cpp), six print statements draw an arrow on the screen:

Listing 2.4: `arrow.cpp`

```
#include <iostream>

int main() {
    std::cout << "    *    \n";
    std::cout << "   ***   \n";
    std::cout << "  *****  \n";
    std::cout << "    *    \n";
    std::cout << "    *    \n";
    std::cout << "    *    \n";
}
```

The output of Listing 2.4 (arrow.cpp) is

```
    *
```

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

Each print statement "draws" a horizontal slice of the arrow. The six statements

```
std::cout << "    *    \n";
std::cout << "   ***   \n";
std::cout << "  *****  \n";
std::cout << "    *    \n";
std::cout << "    *    \n";
std::cout << "    *    \n";
```

constitute the *body* of the `main` function. The body consists of all the statements between the open curly brace ({) and close curly brace (}). We say that the curly braces *delimit* the body of the function. The word *delimit* means to determine the boundaries or limits of something. The { symbol determines the beginning of the function's body, and the } symbol specifies the end of the function's body.

We can rewrite Listing 2.4 (arrow.cpp) to achieve the same effect with only one long print statement as Listing 2.5 (arrow2.cpp) shows.

Listing 2.5: `arrow2.cpp`

```
#include <iostream>

int main() {
    std::cout << "    *    \n"
              << "   ***   \n"
              << "  *****  \n"
              << "    *    \n"
              << "    *    \n"
              << "    *    \n";
}
```

At first, Listing 2.4 (arrow.cpp) and Listing 2.5 (arrow2.cpp) may appear to be identical, but upon closer inspection of this new program we see that `std::cout` appears only once within `main`, and only one semicolon (;) appears within `main`. Since semicolons in C++ terminate statements, there really is only one statement. Notice that a single statement can be spread out over several lines. The statement within `main` appearing as

```
std::cout << "    *    \n"
          << "   ***   \n"
          << "  *****  \n"
          << "    *    \n"
          << "    *    \n"
          << "    *    \n";
```

could have just as easily been written as

```
std::cout << "    *    \n" << "   ***   \n"
          << "  *****  \n" << "    *    \n"
          << "    *    \n" << "    *    \n";
```

but the first way of expressing it better portrays how the output will appear. Read this second version carefully to convince yourself that the printed pieces will indeed flow to the `std::cout` printing object in the proper sequence to produce the same picture of the arrow.

Consider the mistake of putting semicolons at the end of each of the lines in the "one statement" version:

```
std::cout << "    *    \n";
          << "   ***   \n";
          << "  *****  \n";
          << "    *    \n";
          << "    *    \n";
          << "    *    \n";
```

If we put this code fragment in `main`, the program will not compile. The reason is simple—the semicolon at the end of the first line terminates the statement on that line. The compiler expects a new statement on the next line, but

```
<< "   ***   \n";
```

is not a complete legal C++ statement since the `<<` operator is missing the `std::cout` object. The string `"   ***   \n"` has nothing to "flow into."

Listing 2.6 (empty.cpp) is even simpler than Listing 2.1 (simple.cpp).

Listing 2.6: `empty.cpp`

```
int main() {
}
```

Since Listing 2.6 (empty.cpp) does not use the `std::cout` object and so does not need the `#include` and `using` directives. While it is legal and sometimes even useful in C++ to write functions with empty bodies, such functions will do nothing when they execute. Listing 2.6 (empty.cpp) with its empty `main` function is, therefore, truly the simplest executable C++ program we can write, but it does nothing when we runs it!

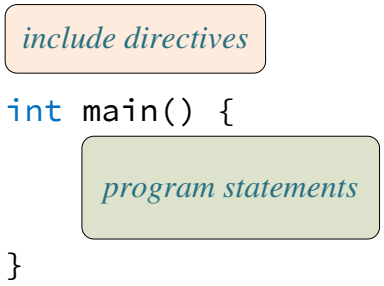
In general, a C++ program may contain multiple functions, but we defer such generality until Chapter 9. For now, we will restrict our attention to programs with only a `main` function.

2.4 Template for simple C++ programs

For our immediate purposes all the programs we write will have the form shown in Figure 2.1.

Our programs generally will print something, so we need the `#include` directive that brings the `std::cout` definition from `<iostream>` into our program. Depending on what we need our program to do, we may need additional `#include` directives. The `main` function definition is required for an executable program, and we will fill its body with statements that make our program do as we wish. Later, our programs will become more sophisticated, and we will need to augment this simple template.

Figure 2.1 The general structure of a very simple C++ program.



2.5 Exercises

1. What preprocessor directive is necessary to use statements with the `std::cout` printing stream object?
2. What statement allows the short name `cout` to be used instead of `std::cout`?
3. What does the name `std` stand for?
4. All C++ programs must have a function named what?
5. The body of `main` is enclosed within what symbols?
6. What operator directs information to the `std::cout` output stream?
7. Write a C++ program that prints your name in the console window.
8. Write a C++ program that prints your first and last name in the console window. Your first name should appear on one line, and your last name appear on the next line.
9. What other files must you distribute with your executable file so that your program will run on a Windows PC without Visual Studio installed?
10. Can a single statement in C++ span multiple lines in the source code?

©2018 Richard L. Halterman

Chapter 3

Values and Variables

In this chapter we explore some building blocks that are used to develop C++ programs. We experiment with the following concepts:

- numeric values
- variables
- declarations
- assignment
- identifiers
- reserved words

In the next chapter we will revisit some of these concepts in the context of other data types.

3.1 Integer Values

The number four (4) is an example of a *numeric* value. In mathematics, 4 is an *integer* value. Integers are whole numbers, which means they have no fractional parts, and an integer can be positive, negative, or zero. Examples of integers include 4, -19, 0, and -1005. In contrast, 4.5 is not an integer, since it is not a whole number.

C++ supports a number of numeric and non-numeric values. In particular, C++ programs can use integer values. It is easy to write a C++ program that prints the number four, as Listing 3.1 (number4.cpp) shows.

Listing 3.1: number4.cpp

```
#include <iostream>

int main() {
    std::cout << 4 << '\n';
}
```

Notice that unlike the programs we saw earlier, Listing 3.1 (number4.cpp) does not use quotation marks (`"`). The number 4 appears unadorned with no quotes. The expression `'\n'` represents a single newline character. Multiple characters comprising a string appear in double quotes (`"`), but, in C++, a single character represents a distinct type of data and is enclosed within single quotes (`'`). (Section 3.8 provides more information about C++ characters.) Compare Listing 3.1 (number4.cpp) to Listing 3.2 (number4-alt.cpp).

Listing 3.2: number4-alt.cpp

```
#include <iostream>

int main() {
    std::cout << "4\n";
}
```

Both programs behave identically, but Listing 3.1 (number4.cpp) prints the value of the number four, while Listing 3.2 (number4-alt.cpp) prints a message containing the digit four. The distinction here seems unimportant, but we will see in Section 3.2 that the presence or absence of the quotes can make a big difference in the output.

The statement

```
std::cout << "4\n";
```

sends one thing to the output stream, the string `"4\n"`. The statement

```
std::cout << 4 << '\n';
```

sends two things to the output stream, the integer value 4 and the newline character `'\n'`.

In published C++ code you sometimes will see a statement such as the following:

```
std::cout << 4 << std::endl;
```

This statement on the surface behaves exactly like the following statement:

```
std::cout << 4 << '\n';
```

but the two expressions `std::endl` and `'\n'` do not mean exactly the same thing. The `std::endl` expression does involve a newline character, but it also performs some additional work that normally is not necessary.

Programs that do significant printing may execute faster if they terminate their output lines with `'\n'` instead of `std::endl`. The difference in speed is negligible when printing to the console, but the different can be great when printing to files or other output streams. For most of the programs we consider, the difference in program execution speed between the two is imperceptible; nonetheless, we will prefer `'\n'` for printing newlines because it is a good habit to form (and it requires five fewer keystrokes when editing code).

The three major modern computing platforms are Microsoft Windows, Apple macOS, and Linux. Windows handles newlines differently from macOS and Linux. Historically, the character `'\n'` represents a *new line*, usually known as a *line feed* or *LF* for short, and the character `'\r'` means *carriage return*, or *CR* for short. The terminology comes from old-fashioned typewriters which feed a piece of paper into a roller on a carriage that moves to the left as the user types (so the imprinted symbols form left to right). At the end of a line, the user must advance the roller so as to move the paper up by one line (LF) and move the carriage back all the way to its left (CR). Windows uses the character sequence CR LF for newlines, while macOS and Linux use LF. This can be an issue when attempting to edit text files written with an editor on one platform with an editor on a different platform.

The good news is that the C++ standard guarantees that the `std::cout` output stream translates the `'\n'` character as it appears in C++ source code into the correct character sequence for the target platform. This means you can print `'\n'` via `std::cout`, and it will behave identically on all the major platforms.

In C++ source code, integers may not contain commas. This means we must write the number two thousand, four hundred sixty-eight as `2468`, not `2,468`. Modern C++ does support single quotes (`'`) as digit separators, as in `2'468`. Using digit separators can improve the human comprehension reading large numbers in C++ source code.

In mathematics, integers are unbounded; said another way, the set of mathematical integers is infinite. In C++ the range of integers is limited because all computers have a finite amount of memory. The exact range of integers supported depends on the computer system and particular C++ compiler. C++ on most 32-bit computer systems can represent integers in the range −2,147,483,648 to +2,147,483,647.

What happens if you exceed the range of C++ integers? Try Listing 3.3 (exceed.cpp) on your system.

Listing 3.3: `exceed.cpp`

```
#include <iostream>

int main() {
    std::cout << -3000000000 << '\n';
```

```
}
```

Negative three billion is too large for 32-bit integers, however, and the program's output is obviously wrong:

```
1294967296
```

The number printed was not even negative! Most C++ compilers will issue a warning about this statement. Section 4.6 explores errors vs. warnings in more detail. If the compiler finds an error in the source, it will not generate the executable code. A warning indicates a potential problem and does not stop the compiler from producing an executable program. Here we see that the programmer should heed this warning because the program's execution produces meaningless output.

This limited range of values is common among programming languages since each number is stored in a fixed amount of memory. Larger numbers require more storage in memory. In order to model the infinite set of mathematical integers an infinite amount of memory would be needed! As we will see later, C++ supports an integer type with a greater range. Section 4.8.1 provides some details about the implementation of C++ integers.

3.2 Variables and Assignment

In algebra, variables are used to represent numbers. The same is true in C++, except C++ variables also can represent values other than numbers. Listing 3.4 (variable.cpp) uses a variable to store an integer value and then prints the value of the variable.

Listing 3.4: `variable.cpp`

```
#include <iostream>

int main() {
    int x;
    x = 10;
    std::cout << x << '\n';
}
```

The `main` function in Listing 3.4 (variable.cpp) contains three statements:

- `int x;`

  This is a *declaration* statement. All variables in a C++ program must be declared. A declaration specifies the type of a variable. The word `int` indicates that the variable is an integer. The name of the integer variable is `x`. We say that variable `x` has type `int`. C++ supports types other than integers, and some types require more or less space in the computer's memory. The compiler uses the declaration to reserve the proper amount of memory to store the variable's value. The declaration enables the compiler to verify the programmer is using the variable properly within the program; for example, we will see that integers can be added together just like in mathematics. For some other data types, however, addition is not possible and so is not allowed. The compiler can ensure that a variable involved in an addition operation is compatible with addition. It can report an error if it is not.

  The compiler will issue an error if a programmer attempts to use an undeclared variable. The compiler cannot deduce the storage requirements and cannot verify the variable's proper usage if it not declared. Once declared, a particular variable cannot be redeclared in the same context. A variable may not change its type during its lifetime.

- `x = 10;`

  This is an *assignment* statement. An assignment statement associates a value with a variable. The key to an assignment statement is the symbol = which is known as the *assignment operator*. Here the value 10 is being assigned to the variable `x`. This means the value 10 will be stored in the memory location the compiler has reserved for the variable named `x`. We need not be concerned about where the variable is stored in memory; the compiler takes care of that detail.

  After we declare a variable we may assign and reassign it as often as necessary.

- `std::cout << x << '\n';`

  This statement prints the variable `x`'s current value.

Note that the lack of quotation marks here is very important. If `x` has the value 10, the statement

```
std::cout << x << '\n';
```

prints `10`, the value of the variable `x`, but the statement

```
std::cout << "x" << '\n';
```

prints `x`, the message containing the single letter *x*.

The meaning of the assignment operator (=) is different from equality in mathematics. In mathematics, = asserts that the expression on its left is equal to the expression on its right. In C++, = makes the variable on its left take on the value of the expression on its right. It is best to read `x = 5` as "`x` is assigned the value 5," or "`x` gets the value 5." This distinction is important since in mathematics equality is symmetric: if $x = 5$, we know $5 = x$. In C++, this symmetry does not exist; the statement

```
5 = x;
```

attempts to reassign the value of the literal integer value 5, but this cannot be done, because 5 is always 5 and cannot be changed. Such a statement will produce a compiler error:

error C2106: '=' : left operand must be l-value

Variables can be reassigned different values as needed, as Listing 3.5 (multipleassignment.cpp) shows.

Listing 3.5: `multipleassignment.cpp`

```
#include <iostream>

int main() {
    int x;
    x = 10;
    std::cout << x << '\n';
    x = 20;
    std::cout << x << '\n';
    x = 30;
    std::cout << x << '\n';
}
```

Observe the each print statement in Listing 3.5 (multipleassignment.cpp) is identical, but when the program runs the print statements produce different results.

A variable may be given a value at the time of its declaration; for example, Listing 3.6 (variable-init.cpp) is a variation of Listing 3.4 (variable.cpp).

Listing 3.6: variable-init.cpp

```
#include <iostream>

int main() {
    int x = 10;
    std::cout << x << '\n';
}
```

Notice that in Listing 3.6 (variable-init.cpp) the declaration and assignment of the variable x is performed in one statement instead of two. This combined declaration and immediate assignment is called *initialization*.

C++ supports another syntax for initializing variables as shown in Listing 3.7 (alt-variable-init.cpp).

Listing 3.7: alt-variable-init.cpp

```
#include <iostream>

int main() {
    int x{10};
    std::cout << x << '\n';
}
```

This alternate form is not commonly used for simple variables, but it necessary for initializing more complicated kinds of variables called *objects*. We introduce objects in Chapter 13 and Chapter 14.

Multiple variables of the same type can be declared and, if desired, initialized in a single statement. The following statements declare three variables in one declaration statement:

```
int x, y, z;
```

The following statement declares three integer variables and initializes two of them:

```
int x = 0, y, z = 5;
```

Here y's value is undefined. The declarations may be split up into multiple declaration statements:

```
int x = 0;
int y;
int z = 5;
```

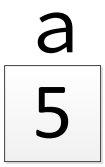
In the case of multiple declaration statements the type name (here `int`) must appear in each statement.

The compiler maps a variable to a location in the computer's memory. We can visualize a variable and its corresponding memory location as a box as shown in Figure 3.1.

We name the box with the variable's name. Figure 3.2 shows how the following sequence of C++ code affects memory.

```
int a, b;
a = 2;
```

Figure 3.1 Representing a variable and its memory location as a box




```
b = 5;
a = b;
b = 4;
```

Importantly, the statement

```
a = b;
```

does not mean a and b refer to the same box (memory location). After this statement a and b still refer to separate boxes (memory locations). It simply means the value stored in b's box (memory location) has been copied to a's box (memory location). a and b remain distinct boxes (memory locations). The original value found in a's box is overwritten when the contents of b's box are copied into a. After the assignment of b to a, the reassignment of b to 4 does not affect a.

3.3 Identifiers

While mathematicians are content with giving their variables one-letter names like x, programmers should use longer, more descriptive variable names. Names such as altitude, sum, and user_name are much better than the equally permissible a, s, and u. A variable's name should be related to its purpose within the program. Good variable names make programs more readable by humans. Since programs often contain many variables, well-chosen variable names can render an otherwise obscure collection of symbols more understandable.

C++ has strict rules for variable names. A variable name is one example of an *identifier*. An identifier is a word used to name things. One of the things an identifier can name is a variable. We will see in later chapters that identifiers name other things such as functions and classes. Identifiers have the following form:

- Identifiers must contain at least one character.
- The first character must be an alphabetic letter (upper or lower case) or the underscore

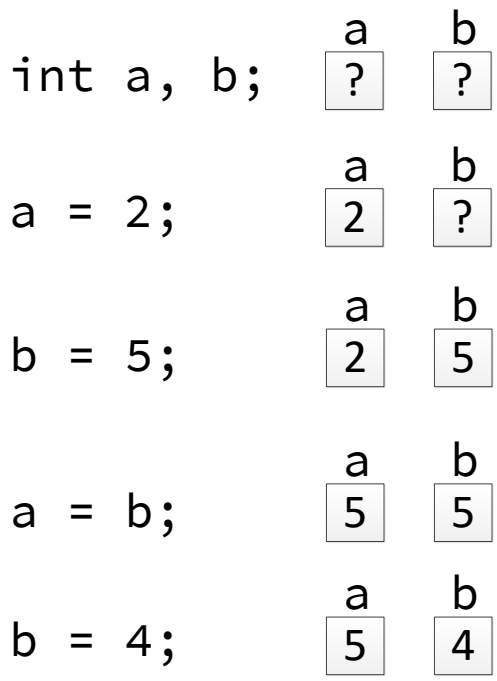
  ```
  ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz_
  ```

- The remaining characters (if any) may be alphabetic characters (upper or lower case), the underscore, or a digit

  ```
  ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz_0123456789
  ```

- No other characters (including spaces) are permitted in identifiers.

Figure 3.2 How memory changes during variable assignment



- A reserved word cannot be used as an identifier (see Table 3.1).

Here are some examples of valid and invalid identifiers:

- All of the following words are valid identifiers and so qualify as variable names: `x`, `x2`, `total`, `port_22`, and `FLAG`.
- None of the following words are valid identifiers: `sub-total` (dash is not a legal symbol in an identifier), `first entry` (space is not a legal symbol in an identifier), `4all` (begins with a digit), `#2` (pound sign is not a legal symbol in an identifier), and `class` (`class` is a reserved word).

C++ reserves a number of words for special use that could otherwise be used as identifiers. Called *reserved words* or *keywords*, these words are special and are used to define the structure of C++ programs and statements. Table 3.1 lists all the C++ reserved words.

The purposes of many of these reserved words are revealed throughout this book.

You may not use any of the reserved words in Table 3.1 as identifiers. Fortunately, if you accidentally attempt to use one of the reserved words in a program as a variable name, the compiler will issue an error (see Section 4.6 for more on compiler errors).

In Listing 2.1 (simple.cpp) we used several reserved words: `using`, `namespace`, and `int`. Notice that `include`, `cout`, and `main` are not reserved words.

Some programming languages do not require programmers to declare variables before they are used; the type of a variable is determined by how the variable is used. Some languages allow the same variable

alignas	decltype	namespace	struct
alignof	default	new	switch
and	delete	noexcept	template
and_eq	double	not	this
asm	do	not_eq	thread_local
auto	dynamic_cast	nullptr	throw
bitand	else	operator	true
bitor	enum	or	try
bool	explicit	or_eq	typedef
break	export	private	typeid
case	extern	protected	typename
catch	false	public	union
char	float	register	unsigned
char16_t	for	reinterpret_cast	using
char32_t	friend	return	virtual
class	goto	short	void
compl	if	signed	volatile
const	inline	sizeof	wchar_t
constexpr	int	static	while
const_cast	long	static_assert	xor
continue	mutable	static_cast	xor_eq

Table 3.1: C++ reserved words. C++ reserves these words for specific purposes in program construction. None of the words in this list may be used as an identifier; thus, you may not use any of these words to name a variable.

to assume different types as its use differs in different parts of a program. Such languages are known as *dynamically-typed languages*. C++ is a *statically-typed language*. In a statically-typed language, the type of a variable must be explicitly specified before it is used by statements in a program. While the requirement to declare all variables may initially seem like a minor annoyance, it offers several advantages:

- When variables must be declared, the compiler can catch typographical errors that dynamically-typed languages cannot detect. For example, consider the following section of code:

  ```
  int ZERO;
  ZER0 = 1;
  ```

  The identifier in the first line ends with a capital "Oh." In the second line, the identifier ends with the digit zero. The distinction may be difficult or impossible to see in a particular editor or printout of the code. A C++ compiler would immediately detect the typo in the second statement, since `ZER0` (last letter a zero) has not been declared. A dynamically-typed language would create two variables: `ZERO` and `ZER0`.

- When variables must be declared, the compiler can catch invalid operations. For example, a variable may be declared to be of type `int`, but the programmer may accidentally assign a non-numeric value to the variable. In a dynamically-typed language, the variable would silently change its type introducing an error into the program. In C++, the compiler would report the improper assignment as error, since once declared a C++ variable cannot change its type.

- Ideally, requiring the programmer to declare variables forces the programmer to plan ahead and think more carefully about the variables a program might require. The purpose of a variable is tied to its type, so the programmer must have a clear notion of the variable's purpose before declaring it. When

variable declarations are not required, a programmer can "make up" variables as needed as the code is written. The programmer need not do the simple double check of the variable's purpose that writing the variable's declaration requires. While declaring the type of a variable specifies its purpose in only a very limited way, any opportunity to catch such errors is beneficial.

- Statically-typed languages are generally more efficient than dynamically-typed languages. The compiler knows how much storage a variable requires based on its type. The space for that variable's value will not change over the life of the variable, since its type cannot change. In a dynamically typed language that allows a variable to change its type, if a variable's type changes during program execution, the storage it requires may change also, so memory for that variable must be allocated elsewhere to hold the different type. This memory reallocation at run time slows down the program's execution.

C++ is a case-sensitive language. This means that capitalization matters. `if` is a reserved word, but none of `If`, `IF`, or `iF` are reserved words. Identifiers are case sensitive also; the variable called `Name` is different from the variable called `name`.

Since it can be confusing to human readers, you should not distinguish variables merely by names that differ in capitalization. For the same reason, it is considered poor practice to give a variable the same name as a reserved word with one or more of its letters capitalized.

3.4 Additional Integer Types

C++ supports several other integer types. The type `short int`, which may be written as just `short`, represents integers that may occupy fewer bytes of memory than the `int` type. If the `short` type occupies less memory, it necessarily must represent a smaller range of integer values than the `int` type. The C++ standard does not require the `short` type to be smaller than the `int` type; in fact, they may represent the same set of integer values. The `long int` type, which may be written as just `long`, may occupy more storage than the `int` type and thus be able to represent a larger range of values. Again, the standard does not require the `long` type to be bigger then the `int` type. Finally, the `long long int` type, or just `long long`, may be larger than a `long`. The C++ standard guarantees the following relative ranges of values hold:

$$\texttt{short int} \leq \texttt{int} \leq \texttt{long int} \leq \texttt{long long int}$$

On a small embedded device, for example, all of these types may occupy the exact same amount of memory and, thus, there would be no advantage of using one type over another. On most systems, however, there will some differences in the ranges.

C++ provides integer-like types that exclude negative numbers. These types include the word *unsigned* in their names, meaning they do not allow a negative sign. The unsigned types come in various potential sizes in the same manner as the signed types. The C++ standard guarantees the following relative ranges of unsigned values:

$$\texttt{unsigned short} \leq \texttt{unsigned} \leq \texttt{unsigned long} \leq \texttt{unsigned long long}$$

Table 3.2 lists the differences among the signed and unsigned integer types in Visual C++. Notice that the corresponding signed and unsigned integer times occupy the same amount of memory. As a result, the unsigned types provide twice the range of positive values available to their signed counterparts. For applications that do not require negative numbers, the `unsigned` type may be a more appropriate option.

Type Name	Short Name	Storage	Smallest Magnitude	Largest Magnitude
`short int`	`short`	2 bytes	−32,768	32,767
`int`	`int`	4 bytes	−2,147,483,648	2,147,483,647
`long int`	`long`	4 bytes	−2,147,483,648	2,147,483,647
`long long int`	`long long`	8 bytes	−9,223,372,036,854,775,808	9,223,372,036,854,775,807
`unsigned short`	`unsigned short`	2 bytes	0	65,535
`unsigned int`	`unsigned`	4 bytes	0	4,294,967,295
`unsigned long int`	`unsigned long`	4 bytes	0	4,294,967,295
`unsigned long long int`	`unsigned long long`	8 bytes	0	18,446,744,073,709,551,615

Table 3.2: Characteristics of Visual C++ Integer Types

Within the source code, any unadorned numerical literal without a decimal point is interpreted as an `int` literal; for example, in the statement

```
int x = 4456;
```

the literal value `4456` is an `int`. In order to represent 4456 as an `long`, append an `L`, as in

```
long x = 4456L;
```

C++ also permits the lower-case *l* (*elle*), as in

```
long x = 4456l;
```

but you should avoid it since on many display and printer fonts it looks too much like the digit `1` (one). Use the `LL` suffix for `long long` literals. The suffixes for the unsigned integers are `u` (`unsigned`), `us` (`unsigned short`), `uL` (`unsigned long`), and `uLL` (`unsigned long long`). The capitalization is unimportant, although capital `L`s are preferred.

Within C++ source code all integer literals are `int` values unless an `L` or `l` is appended to the end of the number; for example, `2` is an `int` literal, while `2L` is a `long` literal.

3.5 Floating-point Types

Many computational tasks require numbers that have fractional parts. For example, the formula from mathematics to compute the area of a circle given the circle's radius, involves the value π, which is approximately 3.14159. C++ supports such non-integer numbers, and they are called *floating-point* numbers. The name comes from the fact that during mathematical calculations the decimal point can move or "float" to various positions within the number to maintain the proper number of significant digits. The types `float` and `double` represent different types of floating-point numbers. The type `double` is used more often, since it stands for "double-precision floating-point," and it can represent a wider range of values with more digits of precision. The `float` type represents single-precision floating-point values that are less precise. Table 3.3 provides some information about floating-point values as commonly implemented on 32-bit computer systems. Floating point numbers can be both positive and negative.

As you can see from Table 3.3, `double`s provide more precision at the cost of using more memory.

Listing 3.8 (pi-print.cpp) prints an approximation of the mathematical value π.

Type	Storage	Smallest Magnitude	Largest Magnitude	Minimum Precision
`float`	4 bytes	1.17549×10^{-38}	$3.40282 \times 10^{+38}$	6 digits
`double`	8 bytes	2.22507×10^{-308}	$1.79769 \times 10^{+308}$	15 digits
`long double`	8 bytes	2.22507×10^{-308}	$1.79769 \times 10^{+308}$	15 digits

Table 3.3: Characteristics of Floating-point Numbers on 32-bit Computer Systems

Listing 3.8: `pi-print.cpp`

```
#include <iostream>

int main() {
    double pi = 3.14159;
    std::cout << "Pi = " << pi << '\n';
    std::cout << "or " << 3.14 << " for short" << '\n';
}
```

The first line in Listing 3.8 (pi-print.cpp) declares a variable named `pi` and assigns it a value. The second line in Listing 3.8 (pi-print.cpp) prints the value of the variable `pi`, and the third line prints a literal value. Any literal numeric value with a decimal point in a C++ program automatically has the type `double`, so

```
3.14
```

has type `double`. To make a literal floating-point value a `float`, you must append an `f` or `F` to the number, as in

```
3.14f
```

(The `f` or `F` suffix is used with literal values only; you cannot change a `double` variable into a `float` variable by appending an `f`. Attempting to do so would change the name of the variable!)

All floating-point literals are `double` values unless an `f` or `F` is appended to the end of the number; for example, `2.0` is a `double` literal, while `2.0f` is a `float` literal.

Floating-point numbers are an approximation of mathematical real numbers. As in the case of the `int` data type, the range of floating-point numbers is limited, since each value requires a fixed amount of memory. In some ways, though, `int`s are very different from `double`s. Any integer within the range of the `int` data type can be represented exactly. This is not true for the floating-point types. Consider the real number π. Since π contains an infinite number of digits, a floating-point number with finite precision can only approximate its value. Since the number of digits available is limited, even numbers with a finite number of digits have no exact representation; for example, the number 23.3123400654033989 contains too many digits for the `double` type and must be approximated as 23.3023498654034. Section 4.8.2 contains more information about the consequences of the inexact nature of floating-point numbers.

We can express floating-point numbers in scientific notation. Since most programming editors do not provide superscripting and special symbols like $\times$, C++ slightly alters the normal scientific notation. The number 6.022×10^{23} is written `6.022e23`. The number to the left of the `e` (we can use capital `E` as well) is

the mantissa, and the number to the right of the e is the exponent of 10. As another example, -5.1×10^{-4} is expressed in C++ as `-5.1e-4`. Listing 3.9 (scientificnotation.cpp) prints some scientific constants using scientific notation.

Listing 3.9: `scientificnotation.cpp`

```
#include <iostream>

int main() {
    double avogadros_number = 6.022e23, c = 2.998e8;
    std::cout << "Avogadro's number = " << avogadros_number << '\n';
    std::cout << "Speed of light = " << c << '\n';
}
```

Section 4.8.2 provides some insight into the implementation of C++ floating-point values and explains how internally all floating-point numbers are stored in exponential notation with a mantissa and exponent.

3.6 Constants

In Listing 3.9 (scientificnotation.cpp), Avogadro's number and the speed of light are scientific constants; that is, to the degree of precision to which they have been measured and/or calculated, they do not vary. C++ supports named constants. Constants are declared like variables with the addition of the `const` keyword:

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

Once declared and initialized, a constant can be used like a variable in all but one way—a constant may not be reassigned. It is illegal for a constant to appear on the left side of the assignment operator (=) outside its declaration statement. A subsequent statement like

```
PI = 2.5;
```

would cause the compiler to issue an error message:

error C3892: 'PI' : you cannot assign to a variable that is const

and fail to compile the program. Since the scientific constants do not change, Listing 3.10 (const.cpp) is a better version of Listing 3.9 (scientificnotation.cpp).

Listing 3.10: `const.cpp`

```
#include <iostream>

int main() {
    const double avogadros_number = 6.022e23, c = 2.998e8;
    std::cout << "Avogadro's number = " << avogadros_number << '\n';
    std::cout << "Speed of light = " << c << '\n';
}
```

Since it is illegal to assign a constant outside of its declaration statement, all constants **must** initialized where they are declared.

By convention, C++ programmers generally express constant names in all capital letters; in this way, within the source code a human reader can distinguish a constant quickly from a variable.

3.7 Other Numeric Types

C++ supports several other numeric data types:

- `long int`—typically provides integers with a greater range than the `int` type; its abbreviated name is `long`. It is guaranteed to provide a range of integer values at least as large as the `int` type.

  An integer literal with a `L` suffix, as in `19L`, has type `long`. A lower case elle (`l`) is allowed as a suffix as well, but you should not use it because it is difficult for human readers to distinguish between `l` (lower case *elle*) and `1` (digit *one*). (The `L` suffix is used with literal values only; you cannot change an `int` variable into a `long` by appending an `L`. Attempting to do so would change the name of the variable!)

- `short int`—typically provides integers with a smaller range than the `int` type; its abbreviated name is `short`. It is guaranteed that the range of `int`s is at least as big as the range of `short`s.

- `unsigned int`—is restricted to nonnegative integers; its abbreviated name is `unsigned`. While the `unsigned` type is limited in nonnegative values, it can represent twice as many positive values as the `int` type. (The name `int` is actually the short name for `signed int` and `int` can be written as `signed`.)

- `long double`—can extend the range and precision of the `double` type.

While the C++ language standard specifies minimum ranges and precision for all the numeric data types, a particular C++ compiler may exceed the specified minimums.

C++ provides such a variety of numeric types for specialized purposes usually related to building highly efficient programs. We will have little need to use many of these types. Our examples will use mainly the numeric types `int` for integers, `double` for an approximation of real numbers, and, less frequently, `unsigned` when nonnegative integral values are needed.

3.8 Characters

The `char` data type is used to represent single characters: letters of the alphabet (both upper and lower case), digits, punctuation, and control characters (like newline and tab characters). Most systems support the American Standard Code for Information Interchange (ASCII) character set. Standard ASCII can represent 128 different characters. Table 3.4 lists the ASCII codes for various characters.

In C++ source code, characters are enclosed by single quotes (`'`), as in

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

Standard (double) quotes (`"`) are reserved for strings, which are composed of characters, but strings and `char`s are very different. C++ strings are covered in Section 11.2.6. The following statement would produce a compiler error message:

```
ch = "A";
```

since a string cannot be assigned to a character variable.

Internally, `char`s are stored as integer values, and C++ permits assigning numeric values to `char` variables and assigning characters to numeric variables. The statement

0	*null*	16		32	*space*	48	0	64	@	80	P	96	`	112	p
1		17		33	!	49	1	65	A	81	Q	97	a	113	q
2		18		34	"	50	2	66	B	82	R	98	b	114	r
3		19		35	#	51	3	67	C	83	S	99	c	115	s
4		20		36	$	52	4	68	D	84	T	100	d	116	t
5		21		37	%	53	5	69	E	85	U	101	e	117	u
6		22		38	&	54	6	70	F	86	V	102	f	118	v
7	*bell*	23		39	'	55	7	71	G	87	W	103	g	119	w
8	*backspace*	24		40	(	56	8	72	H	88	X	104	h	120	x
9	*tab*	25		41	)	57	9	73	I	89	Y	105	i	121	y
10	*newline*	26		42	*	58	:	74	J	90	Z	106	j	122	z
11		27		43	+	59	;	75	K	91	[	107	k	123	{
12	*form feed*	28		44	,	60	<	76	L	92	\	108	l	124	|
13	*return*	29		45	-	61	=	77	M	93	]	109	m	125	}
14		30		46	.	62	>	78	N	94	^	110	n	126	~
15		31		47	/	63	?	79	O	95	_	111	o	127	

Table 3.4: ASCII codes for characters

```
ch = 65;
```

assigns a number to a `char` variable to show that this perfectly legal. The value 65 is the ASCII code for the character *A*. If `ch` is printed, as in

```
ch = 65;
std::cout << ch;
```

the corresponding character, *A*, would be printed because `ch`'s declared type is `char`, not `int` or some other numeric type.

Listing 3.11 (charexample.cpp) shows how characters can be used within a program.

Listing 3.11: `charexample.cpp`

```
#include <iostream>

int main() {
    char ch1, ch2;
    ch1 = 65;
    ch2 = 'A';
    std::cout << ch1 << ", " << ch2 << ", " << 'A' << '\n';
}
```

The program displays

```
A, A, A
```

The first *A* is printed because the statement

```
ch1 = 65;
```

assigns the ASCII code for *A* to `ch1`. The second *A* is printed because the statement

```
ch2 = 'A';
```

assigns the literal character *A* to `ch2`. The third *A* is printed because the literal character `'A'` is sent directly to the output stream.

Integers and characters can be freely assigned to each other, but the range of `char`s is much smaller than the range of `int`s, so care must be taken when assigning an `int` value to a `char` variable.

Some characters are *non-printable* characters. The ASCII chart lists several common non-printable characters:

- `'\n'`—the newline character
- `'\r'`—the carriage return character
- `'\b'`—the backspace character
- `'\a'`—the "alert" character (causes a "beep" sound or other tone on some systems)
- `'\t'`—the tab character
- `'\f'`—the formfeed character
- `'\0'`—the *null* character (used in C strings, see Section 11.2.6)

These special non-printable characters begin with a backslash (\) symbol. The backslash is called an *escape* symbol, and it signifies that the symbol that follows has a special meaning and should not be interpreted literally. This means the literal backslash character must be represented as two backslashes: `'\\'`.

These special non-printable character codes can be embedded within strings. To embed a backslash within a string, you must escape it; for example, the statement

```
std::cout << "C:\\Dev\\cppcode" << '\n';
```

would print

```
C:\Dev\cppcode
```

See what this statement prints:

```
std::cout << "AB\bCD\aEF" << '\n';
```

The following two statements behave identically:

```
std::cout << "End of line" << '\n';
std::cout << "End of line\n";
```

On the Microsoft Windows platform, the character sequence `"\r\n"` (carriage return, line feed) appears at the end of lines in text files. Under Unix and Linux, lines in text files end with `'\n'` (line feed). On Apple Macintosh systems, text file lines end with the `'\r'` (carriage return) character. The compilers that adhere to the C++ standard will ensure that the `'\n'` character in a C++ program when sent to the output stream will produce the correct end-of-line character sequence for the given platform.

3.9 Enumerated Types

C++ allows a programmer to create a new, very simple type and list all the possible values of that type. Such a type is called an *enumerated type*, or an *enumeration type*. The `enum` keyword introduces an enumerated type. The following shows the simplest way to define an enumerated type:

```
enum Color { Red, Orange, Yellow, Green, Blue, Violet };
```

Here, the new type is named `Color`, and a variable of type `Color` may assume one of the values that appears in the list of values within the curly braces. The semicolon following the close curly brace is required. Sometimes the enumerated type definition is formatted as

```
enum Color {
    Red,
    Orange,
    Yellow,
    Green,
    Blue,
    Violet
};
```

but the formatting makes no different to the compiler.

The values listed with the curly braces constitute *all* the values that a variable of the enumerated type can attain. The name of each value of an enumerated type must be a valid C++ identifier (see Section 3.3).

Given the `Color` type defined as above, we can declare and use variables of the `enum` type as shown by the following code fragment:

```
Color myColor;
myColor = Orange;
```

Here the variable `myColor` has our custom type `Color`, and its value is `Orange`.

When declaring enumerated types in this manner it is illegal to reuse an enumerated value name within another enumerated type within the same program. In the following code, the enumerated value `Light` appears in both the `Shade` type and `Weight` type:

```
enum Shade { Dark, Dim, Light, Bright };
enum Weight { Light, Medium, Heavy };
```

These two enumerated types are incompatible because they share the value `Light`, and so the compiler will issue an error.

This style of enumerated type definition is known as an *unscoped enumeration*. C++ inherits this unscoped enumeration style from the C programming language. The C++ standards committee introduced relatively recently an enhanced way of defining enumerated types known as *scoped enumerations*, also known as *enumeration classes*. Scoped enumerations solve the problem of duplicate enumeration values in different types. The following definitions are legal within the same program:

```
enum class Shade { Dark, Dim, Light, Bright };
enum class Weight { Light, Medium, Heavy };
```

When referring to a value from a scoped enumeration we must prepend the name of its type (class), as in

```
Shade color = Shade::Light;
Weight mass = Weight::Light;
```

In this case `Shade` and `Weight` are the scoped enumeration types defined above. Prefixing the type name to the value with the `::` operator enables the compiler to distinguish between the two different values. Scoped enumerations require the type name prefix even if the program contains no other enumerated types. In modern C++ development, scoped enumerations are preferable to unscoped enumerations. You should be familiar with unscoped enumerations, though, as a lot of published C++ code and older C++ books use unscoped enumerations.

Whether scoped or unscoped, the value names within an `enum` type must be unique. The convention in C++ is to captialize the first letter of an `enum` type and its associated values, although the language does not enforce this convention.

An `enum` type is handy for representing a small number of discrete, non-numeric options. For example, consider a program that controls the movements made by a small robot. The allowed orientations are forward, backward, left, right, up, and down. The program could encode these movements as integers, where 0 means left, 1 means backward, etc. While that implementation will work, it is not ideal. Integers may assume many more values than just the six values expected. The compiler cannot ensure that an integer variable representing a robot move will stay in the range 0...5. What if the programmer makes a mistake and under certain rare circumstances assigns a value outside of the range 0...5? The program then will contain an error that may result in erratic behavior by the robot. With `enum` types, if the programmer uses only the named values of the `enum` type, the compiler will ensure that such a mistake cannot happen.

A particular enumerated type necessarily has far fewer values than a type such as `int`. Imagine making an integer `enum` type and having to list all of its values! (The standard 32-bit `int` type represents over four billion values.) Enumerated types, therefore, are practical only for types that have a relatively small range of values.

3.10 Type Inference with auto

C++ requires that a variable be declared before it is used. Ordinarily this means specifying the variable's type, as in

```
int count;
char ch;
double limit;
```

A variable may be initialized when it is declared:

```
int count = 0;
char ch = 'Z';
double limit = 100.0;
```

Each of the values has a type: `0` is an `int`, `'Z'` is a `char`, and `0.0` is a `double`. The `auto` keyword allows the compiler to automatically deduce the type of a variable if it is initialized when it is declared:

```
auto count = 0;
auto ch = 'Z';
auto limit = 100.0;
```

The `auto` keyword may **not** be used without an accompanying initialization; for example, the following declaration is illegal:

```
auto x;
```

because the compiler cannot deduce x's type.

> Automatic type inference is supported only by compilers that comply with the latest C++11 standard. Programmers using older compilers must specify a variable's exact type during the variable's declaration.

Automatic type deduction with `auto` is not useful to beginning C++ programmers. It is just as easy to specify the variable's type. The value of `auto` will become clearer when we consider some of the more advanced features of C++ (see Section 20.2).

3.11 Exercises

1. Will the following lines of code print the same thing? Explain why or why not.

```
std::cout << 6 << '\n';
std::cout << "6" << '\n';
```

2. Will the following lines of code print the same thing? Explain why or why not.

```
std::cout << x << '\n';
std::cout << "x" << '\n';
```

3. What is the largest `int` available on your system?
4. What is the smallest `int` available on your system?
5. What is the largest `double` available on your system?
6. What is the smallest `double` available on your system?
7. What C++ data type represents nonnegative integers?
8. What happens if you attempt to use a variable within a program, and that variable is not declared?
9. What is wrong with the following statement that attempts to assign the value ten to variable x?

```
10 = x;
```

10. Once a variable has been properly declared and initialized can its value be changed?
11. What is another way to write the following declaration and initialization?

```
int x = 10;
```

12. In C++ can you declare more than variable in the same declaration statement? If so, how?
13. In the declaration

```
int a;
int b;
```

do a and b represent the same memory location?

14. Classify each of the following as either a *legal* or *illegal* C++ identifier:

    (a) `fred`
    (b) `if`
    (c) `2x`
    (d) `-4`
    (e) `sum_total`
    (f) `sumTotal`
    (g) `sum-total`
    (h) `sum total`
    (i) `sumtotal`
    (j) `While`
    (k) `x2`
    (l) `Private`
    (m) `public`
    (n) `$16`
    (o) `xTwo`
    (p) `_static`
    (q) `_4`
    (r) `___`
    (s) `10%`
    (t) `a27834`
    (u) `wilma's`

15. What can you do if a variable name you would like to use is the same as a reserved word?

16. Why does C++ require programmers to declare a variable before using it? What are the advantages of declaring variables?

17. What is the difference between `float` and `double`?

18. How can a programmer force a floating-point literal to be a `float` instead of a `double`?

19. How is the value 2.45×10^{-5} expressed as a C++ literal?

20. How can you ensure that a variable's value can never be changed after its initialization?

21. How can you extend the range of `int` on some systems?

22. How can you extend the range and precision of `double` on some systems?

23. Write a program that prints the ASCII chart for all the values from 0 to 127.

24. Is `"i"` a string literal or character literal?

25. Is `'i'` a string literal or character literal?

26. Is it legal to assign a `char` value to an `int` variable?
27. Is it legal to assign an `int` value to a `char` variable?
28. What is printed by the following code fragment?

```
int x;
x = 'A';
std::cout << x << '\n';
```

29. What is the difference between the character `'n'` and the character `'\n'`?
30. Write a C++ program that simply emits a beep sound when run.
31. Create an unscoped enumeration type that represents the days of the week.
32. Create a scoped enumeration type that represents the days of the week.
33. Create an unscoped enumeration type that represents the months of the year.
34. Create a scoped enumeration type that represents the months of the year.
35. Determine the exact type of each of the following variables:
    (a) `auto a = 5;`
    (b) `auto b = false;`
    (c) `auto c = 9.3;`
    (d) `auto d = 5.1f;`
    (e) `auto e = 5L;`

Chapter 4

Expressions and Arithmetic

This chapter uses the C++ numeric types introduced in Chapter 3 to build expressions and perform arithmetic. Some other important concepts are covered—user input, source formatting, comments, and dealing with errors.

4.1 Expressions

A literal value like 34 and a properly declared variable like x are examples of simple *expressions*. We can use operators to combine values and variables and form more complex expressions. Listing 4.1 (adder.cpp) shows how the addition operator (+) is used to add two integers.

Listing 4.1: adder.cpp

```
#include <iostream>

int main() {
    int value1, value2, sum;
    std::cout << "Please enter two integer values: ";
    std::cin >> value1 >> value2;
    sum = value1 + value2;
    std::cout << value1 << " + " << value2 << " = " << sum << '\n';
}
```

In Listing 4.1 (adder.cpp):

- `int value1, value2, sum;`

  This statement declares three integer variables, but it does not initialize them. As we examine the rest of the program we will see that it would be superfluous to assign values to the variables here.

- `std::cout << "Please enter two integer values: ";`

  This statement prompts the user to enter some information. This statement is our usual print statement, but it is not terminated with the end-of-line marker `'\n'`. This is because we want the cursor to remain at the end of the printed line so when the user types in values they appear on the same line as the message prompting for the values. When the user presses the enter key to complete the input, the cursor will automatically move down to the next line.

- `std::cin >> value1 >> value2;`

  This statement causes the program's execution to stop until the user types two numbers on the keyboard and then presses enter. The first number entered will be assigned to `value1`, and the second number entered will be assigned to `value2`. Once the user presses the enter key, the value entered is assigned to the variable. The user may choose to type one number, press enter, type the second number, and press enter again. Instead, the user may enter both numbers separated by one of more spaces and then press enter only once. The program will not proceed until the user enters two numbers.

The `std::cin` input stream object can assign values to multiple variables in one statement, as shown here:

```
int num1, num2, num3;
std::cin >> num1 >> num2 >> num3;
```

A common beginner's mistake is use commas to separate the variables, as in

```
int num1, num2, num3;
std::cin >> num1, num2, num3;
```

The compiler will not generate an error message, because it is legal C++ code. The statement, however, will not assign the three variables from user input as desired. The comma operator in C++ has different meanings in different contexts, and here it is treated like a statement separator; thus, the variables `num2` and `num3` are not involved with the `std::cin` input stream object. We will have no need to use the comma operator in this way, but you should be aware of this potential pitfall.

  `std::cin` is a object that can be used to read input from the user. The `>>` operator—as used here in the context of the `std::cin` object—is known as the *extraction operator*. Notice that it is "backwards" from the `<<` operator used with the `std::cout` object. The `std::cin` object represents the input stream—information flowing into the program from user input from the keyboard. The `>>` operator extracts the data from the input stream `std::cin` and assigns the pieces of the data, in order, to the various variables on its right.

- `sum = value1 + value2;`

  This is an assignment statement because it contains the assignment operator (=). The variable `sum` appears to the left of the assignment operator, so `sum` will receive a value when this statement executes. To the right of the assignment operator is an arithmetic expression involving two variables and the addition operator. The expression is *evaluated* by adding together the values of the two variables. Once the expression's value has been determined, that value can be assigned to the `sum` variable.

All expressions have a value. The process of determining the expression's value is called *evaluation*. Evaluating simple expressions is easy. The literal value 54 evaluates to 54. The value of a variable named `x` is the value stored in the memory location reserved for `x`. The value of a more complex expression is found by evaluating the smaller expressions that make it up and combining them with operators to form potentially new values.

Table 4.1 lists the main C++ arithmetic operators. Table 4.1. The common arithmetic operations, addition, subtraction, and multiplication, behave in the expected way. All these operators are classified as *binary* operators because they operate on two operands. In the statement

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

Operator	Meaning
+	addition
-	subtraction
*	multiplication
/	division
%	modulus

Table 4.1: The simple C++ arithmetic operators

the right side is an addition expression `y + z`. The two operands of the + operator are `y` and `z`.

Two of the operators above, + and -, serve also as *unary* operators. A unary operator has only one operand. The - unary operator expects a single numeric expression (literal number, variable, or complex numeric expression within parentheses) immediately to its right; it computes the *additive inverse* of its operand. If the operand is positive (greater than zero), the result is a negative value of the same magnitude; if the operand is negative (less than zero), the result is a positive value of the same magnitude. Zero is unaffected. For example, the following code sequence

```
int x = 3;
int y = -4;
int z = 0;
std::cout << -x << " " << -y << " " -z << '\n';
```

within a program would print

```
-3 4 0
```

The following statement

```
std::cout << -(4 - 5) << '\n';
```

within a program would print

```
1
```

The unary + operator is present only for completeness; when applied to a numeric value, variable, or expression, the resulting value is no different from the original value of its operand. Omitting the unary + operator from the following statement

```
x = +y;
```

does not change the statement's behavior.

All the arithmetic operators are subject to the limitations of the data types on which they operate; for example, on a system in which the largest `int` is 2,147,483,647, the expression

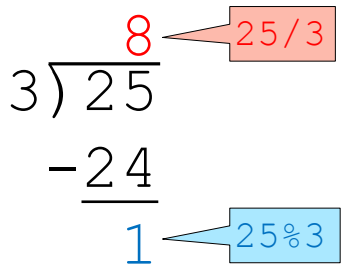
```
2147483647 + 1
```

will not evaluate to the correct answer since the correct answer falls outside the range of `int`s.

If you add, subtract, multiply, is divide two `int`s, the result is an integer. As long as the operation does not exceed the range of `int`s, the arithmetic works as expected. Division, however, is another matter. The statement

```
std::cout << 10/3 << " " << 3/10 << '\n';
```

Figure 4.1 Integer division vs. integer modulus. Integer division produces the quotient, and modulus produces the remainder. In this example, `25/3` is 8, and `25%3` is 1.



prints

```
3 0
```

because in the first case 10 divided by 3 is 3 with a remainder of 1, and in the second case 3 divided by 10 is 0 with a remainder of 3. Since integers are whole numbers, any fractional part of the answer must be discarded. The process of discarding the fractional part leaving only the whole number part is called *truncation*. 10 divided by 3 should be 3.3333..., but that value is truncated to 3. Truncation is not rounding; for example, 11 divided by 3 is 3.6666..., but it also truncates to 3.

Truncation simply removes any fractional part of the value. It does not round. Both 10.01 and 10.999 truncate to 10.

The modulus operator (%) computes the remainder of integer division; thus,

```
std::cout << 10%3 << " " << 3%10 << '\n';
```

prints

```
1 3
```

since 10 divided by 3 is 3 with a remainder of 1, and 3 divided by 10 is 0 with a remainder of 3. Figure 4.1 uses long division for a more hands on illustration of how the integer division and modulus operators work.

The modulus operator is more useful than it may first appear. Listing 4.11 (timeconv.cpp) shows how we can use it to convert a given number of seconds to hours, minutes, and seconds.

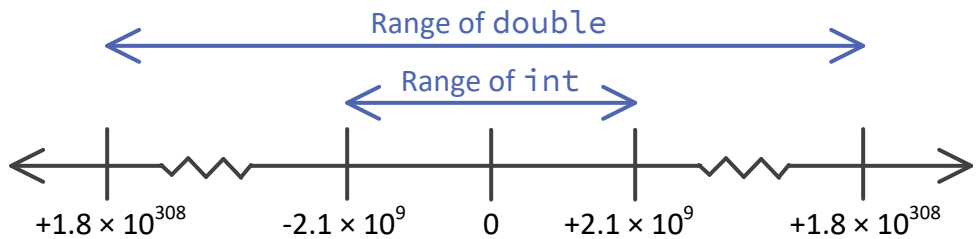
In contrast to integer arithmetic, floating-point arithmetic with `doubles` behaves as expected:

```
std::cout << 10.0/3.0 << " " << 3.0/10.0 << '\n';
```

prints

```
3.33333 0.3
```

Figure 4.2 Range of ints vs. range of doubles



Since a `char` is stored internally as a number (see Section 3.8), we can perform arithmetic on characters. We will have little need to apply mathematics to characters, but sometimes it is useful. As an example, the lower-case letters of the alphabet a–z occupy ASCII values 97–123, with a = 97, b = 98, etc. The upper-case letters A–Z are coded as 65–91, with A = 65, B = 66, etc. To capitalize any lower-case letter, you need only subtract 32, as in

```
char lower = 'd', upper = lower - 32;
std::cout << upper << '\n';
```

This section of code would print `D`. If you do not remember the offset of 32 between upper- and lower-case letter, you can compute it with the letters themselves:

```
upper = lower - ('a' - 'A');
```

In this case, if `lower` has been assigned any value in the range `'a'` to `'z'`, the statement will assign to `upper` the capitalized version of `lower`. On the other hand, if `lower`'s value is outside of that range, `upper` will not receive a meaningful value.

4.2 Mixed Type Expressions

Expressions may contain mixed elements; for example, the following program fragment

```
int x = 4;
double y = 10.2, sum;
sum = x + y;
```

adds an `int` to a `double`, and the result is being assigned to a `double`. How is the arithmetic performed?

As shown in Figure 4.2, the range of `int`s falls completely within the range of `double`s; thus, any `int` value can represented by a `double`. The `int` 4 also can be expressed as the `double` 4.0. In fact, since the largest `int` on most systems is 2,147,483,647, the minimum 15 digits of `double` precision are more than adequate to represent all integers exactly. This means that any `int` value can be represented by a `double`. The converse is not true, however. 2,200,000,000 can be represented by a `double` but it is too big for the `int` type. We say that the `double` type is *wider* than the `int` type and that the `int` type is *narrower* than the `double` type.

It would be reasonable, then, to be able to assign `int` values to `double` variables. The process is called *widening*, and it is always safe to widen an `int` to a `double`. The following code fragment

```
double d1;
int i1 = 500;
d1 = i1;
std::cout << "d1 = " << d1 << '\n';
```

is legal C++ code, and when part of a complete program it would display

```
d1 = 500
```

Assigning a `double` to an `int` variable is not always possible, however, since the `double` value may not be in the range of `int`s. Furthermore, if the `double` variable falls within the range of `int`s but is not a whole number, the `int` variable is unable to the manage fractional part. Consider the following code fragment:

```
double d = 1.6;
int i = d;
```

The second line assigns 1 to `i`. Truncation loses the 0.6 fractional part (see Section 4.1). Note that proper rounding is not done. The Visual C++ compiler will warn us of a potential problem:

warning C4244: '=' : conversion from 'double' to 'int', possible loss of data

This warning reminds us that some information may be lost in the assignment. While the compiler and linker will generate an executable program when warnings are present, you should carefully scrutinize all warnings. This warning is particularly useful, since it is easy for errors due to the truncation of floating-point numbers to creep into calculations.

Converting from a wider type to a narrower type (like `double` to `int`) is called *narrowing*. It often is necessary to assign a floating-point value to an integer variable. If we know the value to assign is within the range of `int`s, and the value has no fractional parts or its truncation would do no harm, the assignment is safe. To perform the assignment without a warning from the compiler, we use a procedure called a *cast*, also called a *type cast*. The cast forces the compiler to accept the assignment without a issuing a warning. The following statement convinces the compiler to accept the `double`-to-`int` assignment without a warning:

```
i = static_cast<int>(d);
```

The reserved word `static_cast` performs the narrowing conversion and silences the compiler warning. The item to convert (in this case the variable `d`) is placed in the parentheses, and the desired type (in this case the type `int`) appears in the angle brackets. The statement

```
i = static_cast<int>(d);
```

does not change the type of the variable `d`; `d` is declared to be a `double` and so must remain a `double` variable. The statement makes a copy of `d`'s value in a temporary memory location, converting it to its integer representation during the process.

We also can cast literal values and expressions:

```
i = static_cast<int>(1.6);
i = static_cast<int>(x + 2.1);
```

Narrowing a floating-point value to an integer discards any fractional part. Narrowing truncates; it does not round. For example, the `double` value 1.7 narrows to the `int` value 1.

The widening conversion is always safe, so a type cast is not required. Narrowing is a potentially dangerous operation, and using an explicit cast does not remove the danger—it simply silences the compiler. For example, consider Listing 4.2 (badnarrow.cpp).

Listing 4.2: `badnarrow.cpp`

```
#include <iostream>

int main() {
    double d = 2200000000.0;
    int i = d;
    std::cout << "d = " << d << ", i = " << i << '\n';
}
```

The Visual C++ compiler issues a warning about the possible loss of precision when assigning `d` to `i`. Silencing the warning with a type cast in this case is a bad idea; the program's output indicates that the warning should be heeded:

```
d = 2.2e+009, i = -2147483648
```

The printed values of `i` and `d` are not even close, nor can they be because it is impossible to represent the value 2,200,000,000 as an `int` on a system that uses 32-bit integers. When assigning a value of a wider type to a variable of a narrower type, the programmer must assume the responsibility to ensure that the actual value to be narrowed is indeed within the range of the narrower type. The compiler cannot ensure the safety of the assignment.

Casts should be used sparingly and with great care because a cast creates a spot in the program that is immune to the compiler's type checking. A careless assignment can produce a garbage result introducing an error into the program.

When we must perform mixed arithmetic—such as adding an `int` to a `double`—the compiler automatically produces machine language code that copies the `int` value to a temporary memory location and transforms it into its `double` equivalent. It then performs double-precision floating-point arithmetic to compute the result.

Integer arithmetic occurs only when both operands are `int`s. `1/3` thus evaluates to 0, but `1.0/3.0`, `1/3.0`, and `1.0/3` all evaluate to 0.33333.

Since `double` is wider than `int`, we say that `double` *dominates* `int`. In a mixed type arithmetic expression, the less dominant type is coerced into the more dominant type in order to perform the arithmetic operation.

Section 3.9 introduced enumerated types. Behind the scenes, the compiler translates enumerated values into integers. The first value in the enumeration is 0, the second value is 1, etc. Even though the underlying implementation of enumerated types is integer, the compiler does not allow the free exchange between integers and enumerated types. The following code will not compile:

```
enum class Color { Red, Orange, Yellow, Green, Blue, Violet };
std::cout << Color::Orange << " " << Color::Green << '\n';
```

The `std::cout` printing object knows how to print integers, but it does not know anything about our `Color` class and its values. If we really want to treat an enumerated type value as its underlying integer, we must use a type cast. Listing 4.3 (enumcast.cpp) shows how to extract the underlying integer value from an enumerated type.

Listing 4.3: `enumcast.cpp`

```
#include <iostream>

int main() {
    enum class Color { Red, Orange, Yellow, Green, Blue, Violet };
    std::cout << static_cast<int>(Color::Orange) << " "
              << static_cast<int>(Color::Green) << '\n';
}
```

Listing 4.3 (enumcast.cpp) prints prints

```
1 3
```

This is the expected output because `Color::Red` is 0, `Color::Orange` is 1, `Color::Yellow` is 2, `Color::Green` is 3, etc.

Even though enumerated types are encoded as integers internally, programmers may not perform arithmetic on enumerated types without involving casts. Such opportunities should be very rare; if you need to perform arithmetic on a variable, it really should be a numerical type, not an enumerated type.

4.3 Operator Precedence and Associativity

When different operators are used in the same expression, the normal rules of arithmetic apply. All C++ operators have a *precedence* and *associativity*:

- **Precedence**—when an expression contains two different kinds of operators, which should be applied first?
- **Associativity**—when an expression contains two operators with the same precedence, which should be applied first?

To see how precedence works, consider the expression

```
2 + 3 * 4
```

Should it be interpreted as

```
(2 + 3) * 4
```

(that is, 20), or rather is

```
2 + (3 * 4)
```

(that is, 14) the correct interpretation? As in normal arithmetic, in C++ multiplication and division have equal importance and are performed before addition and subtraction. We say multiplication and division have precedence over addition and subtraction. In the expression

```
2 + 3 * 4
```

the multiplication is performed before addition, since multiplication has precedence over addition. The result is 14. The multiplicative operators (`*`, `/`, and `%`) have equal precedence with each other, and the additive operators (binary `+` and `-`) have equal precedence with each other. The multiplicative operators have precedence over the additive operators.

As in standard arithmetic, in C++ if the addition is to be performed first, parentheses can override the precedence rules. The expression

```
(2 + 3) * 4
```

evaluates to 20. Multiple sets of parentheses can be arranged and nested in any ways that are acceptable in standard arithmetic.

To see how associativity works, consider the expression

```
2 - 3 - 4
```

The two operators are the same, so they have equal precedence. Should the first subtraction operator be applied before the second, as in

```
(2 - 3) - 4
```

(that is, -5), or rather is

```
2 - (3 - 4)
```

(that is, 3) the correct interpretation? The former (-5) is the correct interpretation. We say that the subtraction operator is *left associative*, and the evaluation is left to right. This interpretation agrees with standard arithmetic rules. All binary operators except assignment are left associative. Assignment is an exception; it is *right associative*. To see why associativity is an issue with assignment, consider the statement

```
w = x = y = z;
```

This is legal C++ and is called *chained assignment*. Assignment can be used as both a statement and an expression. The *statement*

```
x = 2;
```

assigns the value 2 to the variable `x`. The *expression*

```
x = 2
```

assigns the value 2 to the variable `x` and evaluates to the value that was assigned; that is, 2. Since assignment is right associative, the compiler would interpret the chained assignment example as if it were written as

```
w = (x = (y = z));
```

which behaves as follows:

- The expression `y = z` is evaluated first. `z`'s value is assigned to `y`, and the value of the expression `y = z` is `z`'s value.

Arity	Operators	Associativity
Unary	+, −	
Binary	*, /, %	Left
Binary	+, −	Left
Binary	=	Right

Table 4.2: Operator precedence and associativity. The operators in each row have a higher precedence than the operators below it. Operators within a row have the same precedence.

- The expression `x = (y = z)` is evaluated. The value of `y = z`, that is `z`, is assigned to `x`. The overall value of the expression `x = y = z` is thus the value of `z`. Now the values of `x`, `y`, and `z` are all equal (to `z`).
- The expression `w = (x = y = z)` is evaluated. The value of the expression `x = y = z` is equal to `z`'s value, so `z`'s value is assigned to `w`. The overall value of the expression `w = x = y = z` is equal to `z`, and the variables `w`, `x`, `y`, and `z` are all equal (to `z`).

As in the case of precedence, we can use parentheses to override the natural associativity within an expression.

The unary operators have a higher precedence than the binary operators, and the unary operators are right associative. This means the statements

```
std::cout << -3 + 2 << '\n';
std::cout << -(3 + 2) << '\n';
```

which display

```
-1
-5
```

behave as expected.

Table 4.2 shows the precedence and associativity rules for some C++ operators. The * operator also has a unary form that has nothing to do with mathematics; it is covered in Section 10.7.

4.4 Comments

Good programmers annotate their code by inserting remarks that explain the purpose of a section of code or why they chose to write a section of code the way they did. These notes are meant for human readers, not the compiler. It is common in industry for programs to be reviewed for correctness by other programmers or technical managers. Well-chosen identifiers (see Section 3.3) and comments can aid this assessment process. Also, in practice, teams of programmers develop software. A different programmer may be required to finish or fix a part of the program written by someone else. Well-written comments can help others understand new code quicker and increase their productivity modifying old or unfinished code. While it may seem difficult to believe, even the same programmer working on her own code months later can have a difficult time remembering what various parts do. Comments can help greatly.

Any text contained within comments is ignored by the compiler. C++ supports two types of comments: *single line comments* and *block comments*:

- **Single line comment**—the first type of comment is useful for writing a single line remark:

```
// Compute the average of the values
avg = sum / number;
```

The first line here is a comment that comment explains what the statement that follows it is supposed to do. The comment begins with the double forward slash symbols (`//`) and continues until the end of that line. The compiler will ignore the `//` symbols and the contents of the rest of the line. This type of comment is also useful for appending a short comment to the end of a statement:

```
avg = sum / number;  // Compute the average of the values
```

Here, an executable statement and the comment appear on the same line. The compiler will read the assignment statement here, but it will ignore the comment. The compiler generates the same machine code for this example as it does for the preceding example, but this example uses one line of source code instead of two.

- **Block comment**—the second type of comment begins with the symbols `/*` and is in effect until the `*/` symbols are encountered. The `/* . . . */` symbols delimit the comment like parentheses delimit a parenthetical expression. Unlike parentheses, however, these block comments cannot be nested within other block comments.

  The block comment is handy for multi-line comments:

```
/*  After the computation is completed
    the result is displayed.  */
std::cout << result << '\n';
```

What should be commented? Avoid making a remark about the obvious; for example:

```
result = 0;  // Assign the value zero to the variable named result
```

The effect of this statement is clear to anyone with even minimal C++ programming experience. Thus, the audience of the comments should be taken into account; generally, "routine" activities require no remarks. Even though the *effect* of the above statement is clear, its *purpose* may need a comment. For example:

```
result = 0;  // Ensures 'result' has a well-defined minimum value
```

This remark may be crucial for readers to completely understand how a particular part of a program works. In general, programmers are not prone to providing too many comments. When in doubt, add a remark. The extra time it takes to write good comments is well worth the effort.

4.5 Formatting

Program comments are helpful to human readers but ignored by the compiler. Another aspect of source code that is largely irrelevant to the compiler but that people find valuable is its formatting. Imagine the difficulty of reading a book in which its text has no indentation or spacing to separate one paragraph from another. In comparison to the source code for a computer program, a book's organization is quite simple. Over decades of software construction programmers have established a small collection of source code formatting styles that the industry finds acceptable.

The compiler allows a lot of leeway for source code formatting. Consider Listing 4.4 (reformattedvariable.cpp) which is a reformatted version of Listing 3.4 (variable.cpp).

Listing 4.4: reformattedvariable.cpp

```
#include <iostream>
int
main
(
)
{
int
x
;
x
=
10
;
std
::
cout
<<
x
<<
'\n'
;
}
```

Listing 4.5 (reformattedvariable2.cpp) is another reformatted version of Listing 3.4 (variable.cpp).

Listing 4.5: reformattedvariable2.cpp

```
#include <iostream>
int main(){int x;x=10;std::cout<<x<<'\n';}
```

Both reformatted programs are valid C++ and compile to the same machine language code as the original version. Most would argue that the original version is easier to read and understand more quickly than either of the reformatted versions. The elements in Listing 3.4 (variable.cpp) are organized better. Experienced C++ programmers would find both Listing 4.4 (reformattedvariable.cpp) and Listing 4.5 (reformattedvariable2.cpp) visually painful.

What are some distinguishing characteristics of Listing 3.4 (variable.cpp)?

- Each statement appears on its own line. A statement is not unnecessarily split between two lines of text. Visually, one line of text implies one action (statement) to perform.
- The close curly brace aligns vertically with the line above that contains the corresponding open curly brace. This makes it easier to determine if the curly braces match and nest properly. It also better portrays the logical structure of the program. The ability to accurately communicate the logical structure of a program becomes very important as write more complex programs. Programs with complex logic frequently use multiple nested curly braces (for example, see Listing 5.11 (troubleshoot.cpp)). Without a consistent, organized arrangement of curly braces it can difficult to determine which opening brace goes with a particular closing brace.
- The statements that constitute the body of `main` are indented several spaces. This visually emphasizes the fact that the elements are indeed logically enclosed. As with curly brace alignment, indentation to emphasize logical enclosure becomes more important as more complex programs are considered.

- Spaces are used to spread out statements and group pieces of the statement. Space around the operators (=) makes it easier to visually separate the operands from the operators and comprehend the details of the expression. Most people find the statement

  ```
  total_sale = subtotal + tax;
  ```

  much easier to read than

  ```
  total_sale=subtotal+tax;
  ```

  since the lack of space in the second version makes it more difficult to pick out the individual pieces of the statement. In the first version with extra space, it is clearer where operators and variable names begin and end.

  In a natural language like English, a book is divided into distinct chapters, and chapters are composed of paragraphs. One paragraph can be distinguished from another because the first line is indented or an extra space appears between two paragraphs. Space is used to separate words in each sentence. Consider how hard it would be to read a book if all the sentences were printed like this one:

  > Theboyranquicklytothetreetoseethestrandedcat.

  Judiciously placed open space in a C++ program can greatly enhance its readability.

C++ gives the programmer a large amount of freedom in formatting source code. The compiler reads the characters that make up the source code one symbol at a time left to right within a line before moving to the next line. While extra space helps readability, spaces are not allowed in some places:

- Variable names and reserved words must appear as unbroken units.
- Multi-symbol operators like `<<` cannot be separated (`< <` is illegal).

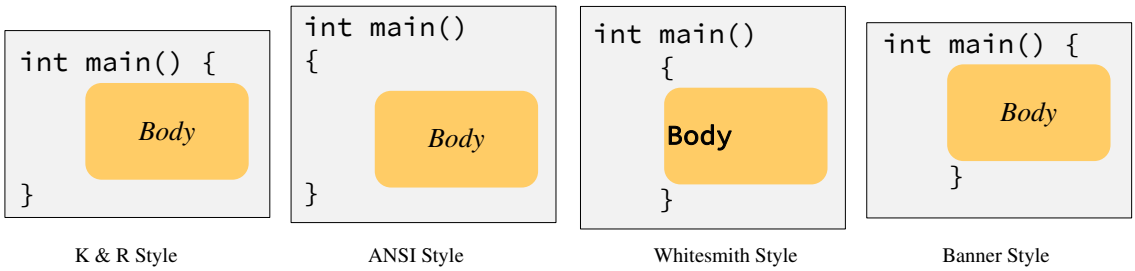
One common coding convention that is universal in C++ programming is demonstrated in Listing 3.10 (const.cpp). While programmers usually use lower-case letters in variable names, they usually express constant names with all capital letters; for example, `PI` is used for the mathematical constant π instead of `pi`. C++ does not require constants to be capitalized, but capitalizing them aids humans reading the source code so they can quickly distinguish between variables and constants.

Figure 4.3 shows the four most common ways programmers use indentation and place curly braces in C++ source code.

The K&R and ANSI styles are the most popular in published C++ source code. The Whitesmith and Banner styles appear much less frequently. `http://en.wikipedia.org/wiki/Indent_style` reviews the various ways to format C++ code. Observe that all the accepted formatting styles indent the block of statements contained in the `main` function.

Most software development organizations adopt a set of *style guidelines*, sometimes called *code conventions*. These guidelines dictate where to indent and by how many spaces, where to place curly braces, how to assign names to identifiers, etc. Programmers working for the organization are required to follow these style guidelines for the code they produce. This better enables any member of the development team to read and understand more quickly code written by someone else. This is necessary when code is reviewed for correctness or when code must be repaired or extended, and the original programmer is no longer with the development team.

Figure 4.3 The most common C++ coding styles.



K & R Style ANSI Style Whitesmith Style Banner Style

Even if you are not forced to use a particular style, it is important to use a consistent style throughout the code you write. As our programs become more complex we will need to use additional curly braces and various levels of indentation to organize the code we write. A consistent style (especially one of the standard styles shown in Figure 4.3) makes it easier to read and verify that the code actually expresses our intent. It also makes it easier to find and fix errors. Said another way, haphazard formatting increases the time it takes to develop correct software because programmer's mistakes hide better in poorly formatted code.

Good software development tools can boost programmer productivity, and many programming editors have the ability to automatically format source code according to a standard style. Some of these editors can correct the code's style as the programmer types in the text. A standalone program known as a *pretty printer* can transform an arbitrarily formatted C++ source file into a properly formatted one.

4.6 Errors and Warnings

Beginning programmers make mistakes writing programs because of inexperience in programming in general or because of unfamiliarity with a programming language. Seasoned programmers make mistakes due to carelessness or because the proposed solution to a problem is faulty and the correct implementation of an incorrect solution will not produce a correct program. Regardless of the reason, a programming error falls under one of three categories:

- compile-time error
- run-time error
- logic error

4.6.1 Compile-time Errors

A *compile-time error* results from the programmer's misuse of the language. A *syntax error* is a common compile-time error. For example, in English one can say

> The boy walks quickly.

This sentence uses correct syntax. However, the sentence

The boy walk quickly.

is not correct syntactically: the number of the subject (singular form) disagrees with the number of the verb (plural form). It contains a syntax error. It violates a grammatical rule of the English language. Similarly, the C++ statement

```
x = y + 2;
```

is syntactically correct because it obeys the rules for the structure of an assignment statement described in Section 3.2. However, consider replacing this assignment statement with a slightly modified version:

```
y + 2 = x;
```

If a statement like this one appears in a program and the variables `x` and `y` have been properly declared, the compiler will issue an error message; for example, the Visual C++ compiler reports (among other things):

error C2106: '=' : left operand must be l-value

The syntax of C++ does not allow an expression like `y + 2` to appear on the left side of the assignment operator.

(The term *l-value* in the error message refers to the left side of the assignment operator; the `l` is an "elle," not a "one.".)

The compiler may generate an error for a syntactically correct statement like

```
x = y + 2;
```

if either of the variables `x` or `y` has not been declared; for example, if `y` has not been declared, Visual C++ reports:

error C2065: 'y' : undeclared identifier

Other common compile-time errors include missing semicolons at the end of statements, mismatched curly braces and parentheses, and simple typographical errors.

Compile-time errors usually are the easiest to repair. The compiler pinpoints the exact location of the problem, and the error does not depend on the circumstances under which the program executes. The exact error can be reproduced by simply recompiling the same source code.

Compilers have the reputation for generating cryptic error messages. They seem to provide little help as far as novice programmers are concerned. Sometimes a combination of errors can lead to messages that indicate errors on lines that follow the line that contains the actual error. Once you encounter the same error several times and the compiler messages become more familiar, you become better able to deduce the actual problem from the reported message. Unfortunately C++ is such a complex language that sometimes a simple compile-time error can result in a message that is incomprehensible to beginning C++ programmers.

4.6.2 Run-time Errors

The compiler ensures that the structural rules of the C++ language are not violated. It can detect, for example, the malformed assignment statement and the use of a variable before its declaration. Some violations of the language cannot be detected at compile time, however. A program may not run to completion but instead terminate with an error. We commonly say the program "crashed." Consider Listing 4.6 (dividedanger.cpp) which under certain circumstances will crash.

Listing 4.6: dividedanger.cpp

```
// File dividedanger.cpp

#include <iostream>

int main() {
    int dividend, divisor;

    // Get two integers from the user
    std::cout << "Please enter two integers to divide:";
    std::cin >> dividend >> divisor;
    // Divide them and report the result
    std::cout << dividend << "/" << divisor << " = "
              << dividend/divisor << '\n';
}
```

The expression

```
dividend/divisor
```

is potentially dangerous. If the user enters, for example, 32 and 4, the program works nicely

```
Please enter two integers to divide: 32 4
32/4 = 8
```

and displays the answer of 8. If the user instead types the numbers 32 and 0, the program reports an error and terminates. Division by zero is undefined in mathematics, and integer division by zero in C++ is illegal. When the program attempts the division at run time, the system detects the attempt and terminates the program.

This particular program can fail in other ways as well; for example, outside of the C++ world, 32.0 looks like a respectable integer. If the user types in 32.0 and 8, however, the program crashes because 32.0 is not a valid way to represent an integer in C++. When the compiler compiles the source line

```
std::cin >> dividend >> divisor;
```

given that `dividend` has been declared to be an `int`, it generates slightly different machine language code than it would if `dividend` has been declared to be a `double` instead. The compiled code expects the text entered by the user to be digits with no extra decoration. Any deviation from this expectation results in a run-time error. Similar results occur if the user enters text that does not represent an integer, like *fred*.

Observe that in either case—entry of a valid but inappropriate integer (zero) or entry of a non-integer (32.0 or *fred*)—it is impossible for the compiler to check for these problems at compile time. The compiler cannot predict what the user will enter when the program is run. This means it is up to the programmer to write code that can handle bad input that the user may provide. As we continue our exploration of programming in C++, we will discover ways to make our programs more robust against user input (see Listing 5.2 (betterdivision.cpp) in Chapter 5, for example). The solution involves changing the way the program runs depending on the actual input provided by the user.

4.6.3 Logic Errors

Consider the effects of replacing the expression

```
dividend/divisor;
```

in Listing 4.6 (dividedanger.cpp) with the expression:

```
divisor/dividend;
```

The program compiles with no errors. It runs, and unless a value of zero is entered for the dividend, no run-time errors arise. However, the answer it computes is not correct in general. The only time the correct answer is printed is when `dividend = divisor`. The program contains an error, but neither the compiler nor the run-time system is able detect the problem. An error of this type is known as a *logic error*.

Listing 4.20 (faultytempconv.cpp) is an example of a program that contains a logic error. Listing 4.20 (faultytempconv.cpp) compiles and does not generate any run-time errors, but it produces incorrect results.

Beginning programmers tend to struggle early on with compile-time errors due to their unfamiliarity with the language. The compiler and its error messages are actually the programmer's best friend. As the programmer gains experience with the language and the programs written become more complicated, the number of compile-time errors decrease or are trivially fixed and the number of logic errors increase. Unfortunately, both the compiler and run-time environment are powerless to provide any insight into the nature and sometimes location of logic errors. Logic errors, therefore, tend to be the most difficult to find and repair. Tools such as debuggers are frequently used to help locate and fix logic errors, but these tools are far from automatic in their operation.

Errors that escape compiler detection (run-time errors and logic errors) are commonly called *bugs*. Since the compiler is unable to detect these problems, such bugs are the major source of frustration for developers. The frustration often arises because in complex programs the bugs sometimes only reveal themselves in certain situations that are difficult to reproduce exactly during testing. You will discover this frustration as your programs become more complicated. The good news is that programming experience and the disciplined application of good programming techniques can help reduce the number logic errors. The bad news is that since software development in an inherently human intellectual pursuit, logic errors are inevitable. Accidentally introducing and later finding and eliminating logic errors is an integral part of the programming process.

4.6.4 Compiler Warnings

A warning issued by the compiler does mark a violation of the rules in the C++ language, but it is an notification to the programmer that the program contains a construct that is a potential problem. In Listing 4.10 (tempconv.cpp) the programmer is attempting to print the value of a variable before it has been given a known value.

Listing 4.7: `uninitialized.cpp`

```cpp
// uninitialized.cpp

#include <iostream>

int main() {
    int n;
    std::cout << n << '\n';
}
```

An attempt to build Listing 4.7 (uninitialized.cpp) yields the following message from the Visual C++ compiler:

warning C4700: uninitialized local variable 'n' used

The compiler issued a warning but still generated the executable file. When run, the program produces a random result because it prints the value in memory associated with the variable, but the program does not initialize that memory location.

Listing 4.8 (narrow.cpp) assigns a `double` value to an `int` variable, which we know from Section 4.1 truncates the result.

Listing 4.8: `narrow.cpp`

```
#include <iostream>

int main() {
    int n;
    double d = 1.6;
    n = d;
    std::cout << n << '\n';
}
```

When compiled we see

warning C4244: '=' : conversion from 'double' to 'int', possible loss of data

Since it is a warning and not an error, the compiler generates the executable, but the warning should prompt us to stop and reflect about the correctness of the code. The enhanced warning level prevents the programmer from being oblivious to the situation.

The default Visual C++ warning level is 3 when compiling in the IDE and level 1 on the command line (that is why we use the `/W3` option on the command line); the highest warning level is 4. You can reduce the level to 1 or 2 or disable warnings altogether, but that is not recommended. The only reason you might want to reduce the warning level is to compile older existing C++ source code that does meet newer C++ standards. When developing new code, higher warning levels are preferred since they provide more help to the programmer. Unless otherwise noted, all the complete program examples in this book compile cleanly under Visual C++ set at warning level 3. Level 3 is helpful for detecting many common logic errors.

We can avoid most warnings by a simple addition to the code. Section 4.2 showed how we can use `static_cast` to coerce a wider type to a narrower type. At Visual C++ warning Level 3, the compiler issues a warning if the cast is not used. The little code that must be added should cause the programmer to stop and reflect about the correctness of the construct. The enhanced warning level prevents the programmer from being oblivious to the situation.

Use the strongest level of warnings available to your compiler. Treat all warnings as problems that must be corrected. Do not accept as completed a program that compiles with warnings.

We may assign a `double` literal to a `float` variable without any special type casting. The compiler automatically narrows the `double` to a `float` as Listing 4.9 (assignfloat.cpp) shows:

Listing 4.9: assignfloat.cpp

```
#include <iostream>

int main() {
    float number;
    number = 10.0;  // OK, double literal assignable to a float
    std::cout << "number = " << number << '\n';
}
```

The statement

```
number = 10.0;
```

assigns a `double` literal (10.0) to a `float` variable. You instead may explicitly use a `float` literal as:

```
number = 10.0f;
```

4.7 Arithmetic Examples

The kind of arithmetic to perform in a complex expression is determined on an operator by operator basis. For example, consider Listing 4.10 (tempconv.cpp) that attempts to convert a temperature from degrees Fahrenheit to degrees Celsius using the formula

$$^{\circ}C = \frac{5}{9} \times (^{\circ}F - 32)$$

Listing 4.10: tempconv.cpp

```
// File tempconv.cpp

#include <iostream>

int main() {
    double degreesF, degreesC;
    // Prompt user for temperature to convert
    std::cout << "Enter the temperature in degrees F: ";
    // Read in the user's input
    std::cin >> degreesF;
    // Perform the conversion
    degreesC = 5/9*(degreesF - 32);
    // Report the result
    std::cout << degreesC << '\n';
}
```

Listing 4.10 (tempconv.cpp) contains comments that document each step explaining the code's purpose. An initial test is promising:

```
Enter the temperature in degrees F: 32
Degrees C = 0
```

Water freezes at 32 degrees Fahrenheit and 0 degrees Celsius, so the program's behavior is correct for this test. Several other attempts are less favorable—consider

```
Enter the temperature in degrees F: 212
Degrees C = 0
```

Water boils at 212 degrees Fahrenheit which is 100 degrees Celsius, so this answer is not correct.

```
Enter the temperature in degrees F: -40
Degrees C = 0
```

The value -40 is the point where the Fahrenheit and Celsius curves cross, so the result should be -40, not zero. The first test was only *coincidentally correct*.

Unfortunately, the printed result is always zero regardless of the input. The problem is the division `5/9` in the statement

```
degreesC = 5/9*(degreesF - 32);
```

Division and multiplication have equal precedence, and both are left associative; therefore, the division is performed first. Since both operands are integers, integer division is performed and the quotient is zero (5 divided by 9 is 0, remainder 5). Of course zero times any number is zero, thus the result. The fact that a floating-point value is involved in the expression (`degreesF`) and the overall result is being assigned to a floating-point variable, is irrelevant. The decision about the exact type of operation to perform is made on an operator-by-operator basis, not globally over the entire expression. Since the division is performed first and it involves two integer values, integer division is used before the other floating-point pieces become involved.

One solution simply uses a floating-point literal for either the five or the nine, as in

```
degreesC = 5.0/9*(degreesF - 32);
```

This forces a double-precision floating-point division (recall that the literal `5.0` is a `double`). The correct result, subject to rounding instead of truncation, is finally computed.

Listing 4.11 (timeconv.cpp) uses integer division and modulus to split up a given number of seconds to hours, minutes, and seconds.

Listing 4.11: `timeconv.cpp`

```
// File timeconv.cpp

#include <iostream>

int main() {
    int hours, minutes, seconds;
    std::cout << "Please enter the number of seconds:";
    std::cin >> seconds;
    // First, compute the number of hours in the given number
    // of seconds
    hours = seconds / 3600;  // 3600 seconds = 1 hours
    // Compute the remaining seconds after the hours are
    // accounted for
    seconds = seconds % 3600;
    // Next, compute the number of minutes in the remaining
    // number of seconds
    minutes = seconds / 60;  // 60 seconds = 1 minute
    // Compute the remaining seconds after the minutes are
```

```
    // accounted for
    seconds = seconds % 60;
    // Report the results
    std::cout << hours << " hr, " << minutes << " min, "
              << seconds << " sec\n";
}
```

If the user enters `10000`, the program prints `2 hr, 46 min, 40 sec`. Notice the assignments to the `seconds` variable, such as

```
seconds = seconds % 3600
```

The right side of the assignment operator (`=`) is first evaluated. The remainder of `seconds` divided by 3,600 is assigned back to `seconds`. This statement can alter the value of `seconds` if the current value of `seconds` is greater than 3,600. A similar statement that occurs frequently in programs is one like

```
x = x + 1;
```

This statement increments the variable `x` to make it one bigger. A statement like this one provides further evidence that the C++ assignment operator does not mean mathematical equality. The following statement from mathematics

$$x = x + 1$$

is surely never true; a number cannot be equal to one more than itself. If that were the case, I would deposit one dollar in the bank and then insist that I really had two dollars in the bank, since a number is equal to one more than itself. That two dollars would become 3.00, then 4.00, etc., and soon I would be rich. In C++, however, this statement simply means "add one to `x` and assign the result back to `x`."

A variation on Listing 4.11 (timeconv.cpp), Listing 4.12 (enhancedtimeconv.cpp) performs the same logic to compute the time pieces (hours, minutes, and seconds), but it uses more simple arithmetic to produce a slightly different output—instead of printing 11,045 seconds as `3 hr, 4 min, 5 sec`, Listing 4.12 (enhancedtimeconv.cpp) displays it as `3:04:05`. It is trivial to modify Listing 4.11 (timeconv.cpp) so that it would print `3:4:5`, but Listing 4.12 (enhancedtimeconv.cpp) includes some extra arithmetic to put leading zeroes in front of single-digit values for minutes and seconds as is done on digital clock displays.

Listing 4.12: `enhancedtimeconv.cpp`

```
// File enhancedtimeconv.cpp

#include <iostream>

int main() {
    int hours, minutes, seconds;
    std::cout << "Please enter the number of seconds:";
    std::cin >> seconds;
    // First, compute the number of hours in the given number
    // of seconds
    hours = seconds / 3600;  // 3600 seconds = 1 hours
    // Compute the remaining seconds after the hours are
    // accounted for
    seconds = seconds % 3600;
    // Next, compute the number of minutes in the remaining
    // number of seconds
    minutes = seconds / 60;  // 60 seconds = 1 minute
    // Compute the remaining seconds after the minutes are
```

```
    // accounted for
    seconds = seconds % 60;
    // Report the results
    std::cout << hours << ":";
    // Compute tens digit of minutes
    int tens = minutes / 10;
    std::cout << tens;
    // Compute ones digit of minutes
    int ones = minutes % 10;
    std::cout << ones << ":";
    // Compute tens digit of seconds
    tens = seconds / 10;
    std::cout << tens;
    // Compute ones digit of seconds
    ones = seconds % 10;
    std::cout << ones << '\n';
}
```

Listing 4.12 (enhancedtimeconv.cpp) uses the fact that if `x` is a one- or two-digit number, `x / 10` is the tens digit of `x`. If `x / 10` is zero, `x` is necessarily a one-digit number.

4.8 Integers vs. Floating-point Numbers

Floating-point numbers offer some distinct advantages over integers. Floating-point numbers, especially `double`s have a much greater range of values than any integer type. Floating-point numbers can have fractional parts and integers cannot. Integers, however, offer one big advantage that floating-point numbers cannot—exactness. To see why integers are exact and floating-point numbers are not, we will explore the way computers store and manipulate the integer and floating-point types.

Computers store all data internally in binary form. The binary (base 2) number system is much simpler than the familiar decimal (base 10) number system because it uses only two digits: 0 and 1. The decimal system uses 10 digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. Despite the lack of digits, every decimal integer has an equivalent binary representation. Binary numbers use a place value system not unlike the decimal system. Figure 4.4 shows how the familiar base 10 place value system works.

Figure 4.4 The base 10 place value system

$$\begin{array}{cccccc} \cdots & \boxed{4} & \boxed{7} & \boxed{3} & \boxed{4} & \boxed{0} & \boxed{6} \\ \cdots & 10^5 & 10^4 & 10^3 & 10^2 & 10^1 & 10^0 \\ \cdots & 100{,}000 & 10{,}000 & 1{,}000 & 100 & 10 & 1 \end{array}$$

$$\begin{aligned} 473,406 &= 4\times10^5+7\times10^4+3\times10^3+4\times10^2+0\times10^1+6\times10^0 \\ &= 400,000+70,000+3,000+400+0+6 \\ &= 473,406 \end{aligned}$$

With 10 digits to work with, the decimal number system distinguishes place values with powers of 10. Compare the base 10 system to the base 2 place value system shown in Figure 4.5.

Figure 4.5 The base 2 place value system

$$
\begin{array}{cccccc}
\cdots & \boxed{1} & \boxed{0} & \boxed{0} & \boxed{1} & \boxed{1} & \boxed{1} \\
\cdots & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \\
\cdots & 32 & 16 & 8 & 4 & 2 & 1
\end{array}
$$

$$
\begin{aligned}
100111_2 &= 1\times 2^5 + 0\times 2^4 + 0\times 2^3 + 1\times 2^2 + 1\times 2^1 + 1\times 2^0 \\
&= 32+0+0+4+2+1 \\
&= 39
\end{aligned}
$$

With only two digits to work with, the binary number system distinguishes place values by powers of two. Since both binary and decimal numbers share the digits 0 and 1, we will use the subscript 2 to indicate a binary number; therefore, 100 represents the decimal value *one hundred*, while 100_2 is the binary number *four*. Sometimes to be very clear we will attach a subscript of 10 to a decimal number, as in 100_{10}.

In the decimal system, it is easy to add $3+5$:

$$
\begin{array}{r}
3 \\
+\ \ 5 \\
\hline
8
\end{array}
$$

The sum $3+9$ is a little more complicated, as early elementary students soon discover:

$$
\begin{array}{r}
3 \\
+\ \ 9 \\
\hline
\end{array}
$$

The answer, of course, is 12, but there is no single *digit* that means 12—it takes two digits, 1 and 2. The sum is

$$
\begin{array}{r}
1 \\
03 \\
+\ \ 09 \\
\hline
12
\end{array}
$$

We can say $3+9$ is 2, *carry the* 1. The rules for adding binary numbers are shorter and simpler than decimal numbers:

$$
\begin{aligned}
0_2+0_2 &= 0_2 \\
0_2+1_2 &= 1_2 \\
1_2+0_2 &= 1_2 \\
1_2+1_2 &= 10_2
\end{aligned}
$$

We can say the sum 1_2+1_2 is 0_2, *carry the* 1_2. A typical larger sum would be

$$
\begin{array}{rcr}
 & & 11 \\
9_{10} & = & 1001_2 \\
+\quad 3_{10} & = & 11_2 \\
\hline
12_{10} & = & 1100_2
\end{array}
$$

4.8.1 Integer Implementation

Mathematical integers are whole numbers (no fractional parts), both positive and negative. Standard C++ supports multiple integer types: `int`, `short`, `long`, and `long long`, `unsigned`, `unsigned short`,

`unsigned long`, and `unsigned long long`. These are distinguished by the number of bits required to store the type, and, consequently, the range of values they can represent. Mathematical integers are infinite, but all of C++'s integer types correspond to finite subsets of the mathematical integers. The most commonly used integer type in C++ is `int`. All `int`s, regardless of their values, occupy the same amount of memory and, therefore use the same number of bits. The exact number of bits in an `int` is processor specific. A 32-bit processor, for example, is built to manipulate 32-bit integers very efficiently. A C++ compiler for such a system most likely would use 32-bit `int`s, while a compiler for a 64-bit machine might represent `int`s with 64 bits. On a 32-bit computer, the numbers 4 and 1,320,002,912 both occupy 32 bits of memory.

For simplicity, we will focus on unsigned integers, particularly the `unsigned` type. The `unsigned` type in Visual C++ occupies 32 bits. With 32 bits we can represent $4,294,967,296$ different values, and so Visual C++'s `unsigned` type represents the integers $0\ldots4,294,967,295$. The hardware in many computer systems in the 1990s provided only 16-bit integer types, so it was common then for C++ compilers to support 16-bit `unsigned` values with a range $0\ldots65,535$. To simplify our exploration into the properties of computer-based integers, we will consider an even smaller, mythical unsigned integer type that we will call **`unsigned tiny`**. C++ has no such **`unsigned tiny`** type as it has a very small range of values—too small to be useful as an actual type in real programs. Our **`unsigned tiny`** type uses only five bits of storage, and Table 4.3 shows all the values that a variable of type **`unsigned tiny`** can assume.

Binary Bit String	Decimal Value
00000	0
00001	1
00010	2
00011	3
00100	4
00101	5
00110	6
00111	7
01000	8
01001	9
01010	10
01011	11
01100	12
01101	13
01110	14
01111	15
10000	16
10001	17
10010	18
10011	19
10100	20
10101	21
10110	22
10111	23
11000	24
11001	25
11010	26
11011	27
11100	28
11101	29
11110	30
11111	31

Table 4.3: The **`unsigned tiny`** values

Table 4.3 shows that the **`unsigned tiny`** type uses all the combinations of 0s and 1s in five bits. We can derive the decimal number 6 directly from its bit pattern:

$$00110 \Longrightarrow \frac{0}{16}\frac{0}{8}\frac{1}{4}\frac{1}{2}\frac{0}{1} \Longrightarrow 0\times16+0\times8+1\times4+1\times2+0\times1=6$$

To see that arithmetic works, try adding $4+3$:

$$\begin{array}{rrcl} & 4_{10} & = & 00100_2 \\ + & 3_{10} & = & 00011_2 \\ \hline & 7_{10} & = & 00111_2 \end{array}$$

That was easy since involved no carries. Next we will try $3+1$:

$$\begin{array}{rrcl} & & & 11 \\ & 3_{10} & = & 00011_2 \\ + & 1_{10} & = & 00001_2 \\ \hline & 4_{10} & = & 00100_2 \end{array}$$

In the ones column (rightmost column), $1_2+1_2=10_2$, so write a 0 and carry the 1 to the top of the next column to the left (that is, the twos column). In the twos column, $1_2+1_2+0_2=10_2$, so we must carry a 1 into the fours column as well.

The next example illustrates a limitation of our finite representation. Consider the sum $8+28$:

$$\begin{array}{rrcr} & & & 11 \\ & 8_{10} & = & 01000_2 \\ + & 28_{10} & = & 11100_2 \\ \hline & 4_{10} & = & 1\ 00100_2 \end{array}$$

In the this sum we have a carry of 1 from the eights column to the 16s column, and we have a carry from the 16s column to nowhere. We need a sixth column (a 32s column), another place value, but our **`unsigned tiny`** type is limited to five bits. That carry out from the 16s place is lost. The largest **`unsigned tiny`** value is 31, but $28+8=36$. It is not possible to store the value 36 in an **`unsigned tiny`** just as it is impossible to store the value $5,000,000,000$ in a C++ `unsigned` variable.

Consider exceeding the capacity of the **`unsigned tiny`** type by just one:

$$\begin{array}{rrcr} & & & 11111 \\ & 31_{10} & = & 11111_2 \\ + & 1_{10} & = & 00001_2 \\ \hline & 0_{10} & = & 1\ 00000_2 \end{array}$$

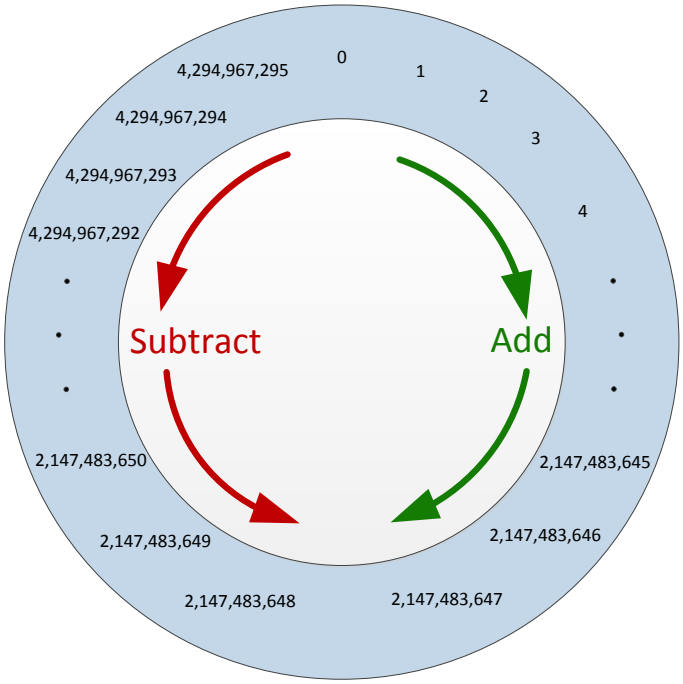
Adding one to the largest possible **`unsigned tiny`**, 31, results in the smallest possible value, 0! This mirrors the behavior of the actual C++ `unsigned` type, as Listing 4.13 (unsignedoverflow.cpp) demonstrates.

Listing 4.13: `unsignedoverflow.cpp`

```
#include <iostream>

int main() {
    unsigned x = 4294967293;  // Almost the largest possible unsigned value
    std::cout << x << " + 1 = " << x + 1 << '\n';
    std::cout << x << " + 2 = " << x + 2 << '\n';
    std::cout << x << " + 3 = " << x + 3 << '\n';

}
```

Listing 4.13 (unsignedoverflow.cpp) prints

Figure 4.6 The cyclic nature of 32-bit unsigned integers. Adding 1 to 4,294,967,295 produces 0, one position clockwise from 4,294,967295. Subtracting 4 from 2 yields 4,294,967,294, four places counterclockwise from 2.




```
4294967293 + 1 = 4294967294
4294967293 + 2 = 4294967295
4294967293 + 3 = 0
```

In fact, Visual C++'s 32-bit `unsigned`s follow the cyclic pattern shown in Figure 4.6.

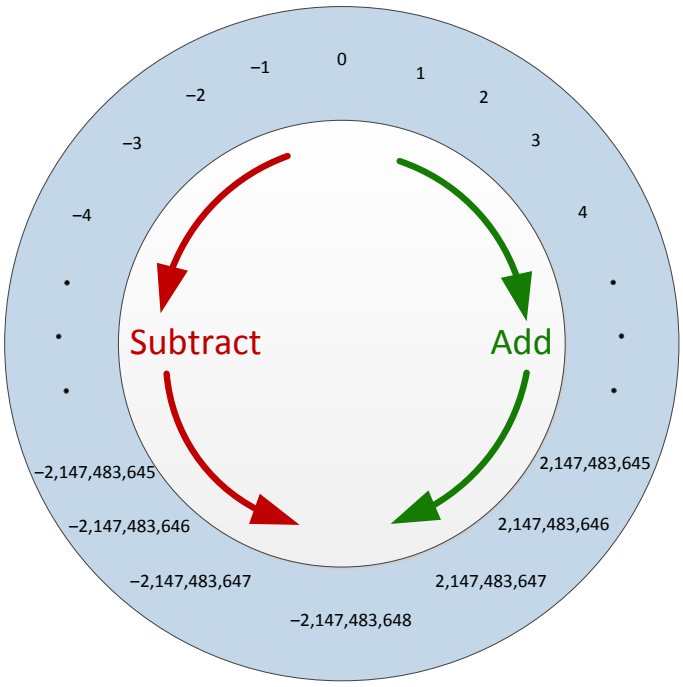
In the figure, an addition moves a value clockwise around the circle, while a subtraction moves a value counterclockwise around the circle. When the numeric limit is reached, the value rolls over like an automobile odometer. Signed integers exibit a similar cyclic pattern as shown in Figure 4.7.

In the case of signed integers, as Figure 4.7 shows, adding one to the largest representable value produces the smallest negative value. Listing 4.14 (integeroverflow.cpp) demonstrates.

Listing 4.14: `integeroverflow.cpp`

```
#include <iostream>

int main() {
    int x = 2147483645; // Almost the largest possible int value
    std::cout << x << " + 1 = " << x + 1 << '\n';
    std::cout << x << " + 2 = " << x + 2 << '\n';
    std::cout << x << " + 3 = " << x + 3 << '\n';

}
```

Figure 4.7 The cyclic nature of 32-bit signed integers. Adding 1 to $2,147,483,647$ produces $-2,147,483,648$, one position clockwise from $2,147,483,647$. Subtracting 5 from $-2,147,483,645$ yields $2,147,483,646$, five places counterclockwise from $-2,147,483,645$.



Listing 4.14 (integeroverflow.cpp) prints

```
2147483645 + 1 = 2147483646
2147483645 + 2 = 2147483647
2147483645 + 3 = -2147483648
```

Attempting to exceed the maximum limit of a numeric type results in *overflow*, and attempting to exceed the minimum limit is called *underflow*. Integer arithmetic that overflow or underflow produces a valid, yet incorrect integer result. The compiler does not check that a computation will result in exceeding the limit of a type because it is impossible to do so in general (consider adding two integer variables whose values are determined at run time). Also significantly, an overflow or underflow situation does not generate a run-time error. It is, therefore, a logic error if a program performs an integral computation that, either as a final result or an intermediate value, is outside the range of the integer type being used.

4.8.2 Floating-point Implementation

The standard C++ floating point types consist of `float`, `double`, and `long double`. Floating point numbers can have fractional parts (decimal places), and the term floating point refers to the fact the decimal point in a number can float left or right as necessary as the result of a calculation (for example, $2.5 \times 3.3 = 8.25$, two one-decimal place values produce a two-decimal place result). As with the integer types, the different floating-point types may be distinguished by the number of bits of storage required and corresponding range of values. The type `float` stands for *single-precision floating-point*, and `double` stands for *double-precision floating-point*. Floating point numbers serve as rough approximations of mathematical *real numbers*, but as we shall see, they have some severe limitations compared to actual real numbers.

On most modern computer systems floating-point numbers are stored internally in exponential form according to the standard adopted by the Institute for Electrical and Electronic Engineers (IEEE 754). In the decimal system, *scientific notation* is the most familiar form of exponential notation:

One mole contains 6.023×10^{23} molecules.

Here 6.023 is called the mantissa, and 23 is the exponent.

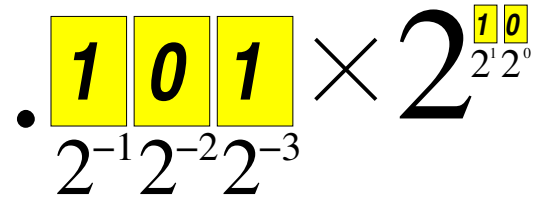
The IEEE 754 standard uses binary exponential notation; that is, the mantissa and exponent are binary numbers. Single-precision floating-point numbers (type `float`) occupy 32 bits, distributed as follows:

Mantissa	24 bits
Exponent	7 bits
Sign	1 bit
Total	32 bits

Double-precision floating-point numbers (type `double`) require 64 bits:

Mantissa	52 bits
Exponent	11 bits
Sign	1 bit
Total	64 bits

Figure 4.8 A **`tiny float`** simplified binary exponential value



The details of the IEEE 754 implementation are beyond the scope of this book, but a simplified example serves to highlight the limitations of floating-point types in general. Recall the fractional place values in the decimal system. The place values, from left to right, are

...	10,000	1,000	100	10	1	•	$\frac{1}{10}$	$\frac{1}{100}$	$\frac{1}{1000}$	$\frac{1}{10,000}$	...
...	10^4	10^3	10^2	10^1	10^0	•	10^{-1}	10^{-2}	10^{-3}	10^{-4}	...

Each place value is one-tenth the place value to its left. Move to the right, divide by ten; move to the left, multiply by ten. In the binary system, the factor is two instead of ten:

...	16	8	4	2	1	•	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{16}$	...
...	2^4	2^3	2^2	2^1	2^0	•	2^{-1}	2^{-2}	2^{-3}	2^{-4}	...

As in our **`unsigned tiny`** example (see Section 4.8.1), consider a a binary exponential number that consists of only five bits—far fewer bits than either `float`s or `double`s in C++. We will call our mythical floating-point type **`tiny float`**. The first three bits of our 5-bit **`tiny float`** type will represent the mantissa, and the remaining two bits store the exponent. The three bits of the mantissa all appear to the right of the binary point. The base of the 2-bit exponent is, of course, two. Figure 4.8 illustrates such a value.

To simplify matters even more, neither the mantissa nor the exponent can be negative. Thus, with three bits, the mantissa may assume one of eight possible values. Since two bits constitute the exponent of **`tiny float`**s, the exponent may assume one of four possible values. Table 4.4 lists all the possible values that **`tiny float`** mantissas and exponents may assume. The number shown in Figure 4.8 is thus

$$\begin{aligned}(1\times 2^{-1} + 0\times 2^{-2} + 1\times 2^{-3})\times 2^{(1\times 2^1+0\times 2^0)} &= \left(\frac{1}{2}+\frac{1}{8}\right)\times 2^2 \\ &= \frac{5}{8}\times 4 \\ &= 2.5\end{aligned}$$

Table 4.5 combines the mantissas and exponents to reveal all possible **`tiny float`** values that we can represent with the 32 different bit strings made up of five bits. The results are interesting.

The range of our **`tiny float`** numbers is $0\ldots 7$. Just stating the range is misleading, however, since it might give the impression that we may represent any value in between 0 and 7 down to the $\frac{1}{8}$th place. This, in fact, is not true. We *can* represent 2.5 with this scheme, but we have no way of expressing 2.25. Figure 4.9 plots all the possible **`tiny float`** values on the real number line.

3-bit Mantissas		
Bit String	Binary Value	Decimal Value
`000`	0.000_2	$\frac{0}{2}+\frac{0}{4}+\frac{0}{8} = 0.000$
`001`	0.001_2	$\frac{0}{2}+\frac{0}{4}+\frac{1}{8} = 0.125$
`010`	0.010_2	$\frac{0}{2}+\frac{1}{4}+\frac{0}{8} = 0.250$
`011`	0.011_2	$\frac{0}{2}+\frac{1}{4}+\frac{1}{8} = 0.375$
`100`	0.100_2	$\frac{1}{2}+\frac{0}{4}+\frac{0}{8} = 0.500$
`101`	0.101_2	$\frac{1}{2}+\frac{0}{4}+\frac{1}{8} = 0.625$
`110`	0.110_2	$\frac{1}{2}+\frac{1}{4}+\frac{0}{8} = 0.750$
`111`	0.111_2	$\frac{1}{2}+\frac{1}{4}+\frac{1}{8} = 0.875$

2-bit Exponents		
Bit String	Binary Value	Decimal Value
`00`	2^{00_2}	$2^0 = 1$
`01`	2^{01_2}	$2^1 = 2$
`10`	2^{10_2}	$2^2 = 4$
`11`	2^{11_2}	$2^3 = 8$

Table 4.4: The eight possible mantissas and four possible exponents that make up all **`tiny float`** values

Figure 4.9 A plot of all the possible **`tiny float`** numbers on the real number line. Note that the numbers are more dense near zero and become more sparse moving to the right. The precision in the range $0\ldots1$ is one-eighth. The precision in the range $1\ldots2$ is only one-fourth, and over the range $2\ldots4$ it drops to one-half. In the range $4\ldots7$ our **`tiny float`** type can represent only whole numbers.

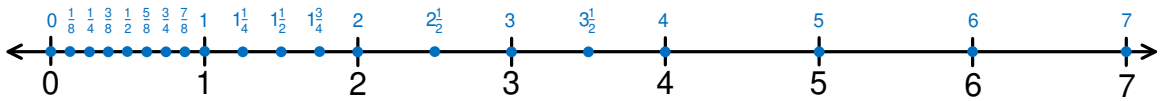


Table 4.5 and Figure 4.9 reveal several troubling issues about our **`tiny float`** type:

1. There are many gaps; for example, the value 2.4 is missing and thus cannot be represented exactly (2.5 is the closest approximation). As another example, 0.75 and 1.75 both appear, but 2.75 is missing.

2. The scheme duplicates some numbers; for example, three different bit patterns represent the decimal value 0.5:
$$0.100 \times 2^{00} \;=\; 0.010 \times 2^{01} \;=\; 0.001 \times 2^{10} \;=\; 0.5_{10}$$
This duplication limits the number of different values that can be represented by a given number of bits. In our **`tiny float`** example 12 of the 32 bit strings (37.5%) are redundant.

3. The numbers are not uniformly dense. There are more values nearer to zero, and the numbers become more sparse farther away from zero.

Our **`unsigned tiny`** type discussed in Section 4.8.1 exhibits none of these weaknesses. All integers in a given range $(0\ldots31)$ are present, no two bit strings represent the same value, and the integers are uniformly distributed across their specified range. While the standard integer types provided by C++ have much greater ranges than our **`unsigned tiny`** type, they all share these same qualities: all values in their ranges are present, and all bit strings represent unique integer values. The standard floating-point types provided by C++ use many more bits than our **`tiny float`** type, yet they exhibit the same problems

Bit String	Interpretation	Decimal Equivalent	Value
00000	$.000_2 \times 2^{00_2}$	0.000×1	0.000
00001	$.000_2 \times 2^{01_2}$	0.000×2	0.000
00010	$.000_2 \times 2^{10_2}$	0.000×4	0.000
00011	$.000_2 \times 2^{11_2}$	0.000×8	0.000
00100	$.001_2 \times 2^{00_2}$	0.125×1	0.125
00101	$.001_2 \times 2^{01_2}$	0.125×2	0.250
00110	$.001_2 \times 2^{10_2}$	0.125×4	0.500
00111	$.001_2 \times 2^{11_2}$	0.125×8	1.000
01000	$.010_2 \times 2^{00_2}$	0.250×1	0.250
01001	$.010_2 \times 2^{01_2}$	0.250×2	0.500
01010	$.010_2 \times 2^{10_2}$	0.250×4	1.000
01011	$.010_2 \times 2^{11_2}$	0.250×8	2.000
01100	$.011_2 \times 2^{00_2}$	0.375×1	0.375
01101	$.011_2 \times 2^{01_2}$	0.375×2	0.750
01110	$.011_2 \times 2^{10_2}$	0.375×4	1.500
01111	$.011_2 \times 2^{11_2}$	0.375×8	3.000
10000	$.100_2 \times 2^{00_2}$	0.500×1	0.500
10001	$.100_2 \times 2^{01_2}$	0.500×2	1.000
10010	$.100_2 \times 2^{10_2}$	0.500×4	2.000
10011	$.100_2 \times 2^{11_2}$	0.500×8	4.000
10100	$.101_2 \times 2^{00_2}$	0.625×1	0.625
10101	$.101_2 \times 2^{01_2}$	0.625×2	1.250
10110	$.101_2 \times 2^{10_2}$	0.625×4	2.500
10111	$.101_2 \times 2^{11_2}$	0.625×8	5.000
11000	$.110_2 \times 2^{00_2}$	0.750×1	0.750
11001	$.110_2 \times 2^{01_2}$	0.750×2	1.500
11010	$.110_2 \times 2^{10_2}$	0.750×4	3.000
11011	$.110_2 \times 2^{11_2}$	0.750×8	6.000
11100	$.111_2 \times 2^{00_2}$	0.875×1	0.875
11101	$.111_2 \times 2^{01_2}$	0.875×2	1.750
11110	$.111_2 \times 2^{10_2}$	0.875×4	3.500
11111	$.111_2 \times 2^{11_2}$	0.875×8	7.000

Table 4.5: The **`tiny float`** values. The first three bits of the bit string constitute the mantissa, and the last two bits represent the exponent. Given five bits we can produce 32 different bit strings. Notice that due to the ways different mantissas and exponents can combine to produce identical values, the 32 different bit strings yield only 20 unique **`tiny float`** values.

shown to a much smaller degree: missing values, multiple bit patterns representing the same values, and uneven distribution of values across their ranges. This is not solely a problem of C++'s implementation of floating-point numbers; all computer languages and hardware that adhere to the IEEE 754 standard exhibit these problems. To overcome these problems and truly represent and compute with mathematical real numbers we would need a computer with an infinite amount of memory along with an infinitely fast processor.

Listing 4.15 (imprecisedifference.cpp) demonstrates the inexactness of floating-point arithmetic.

Listing 4.15: `imprecisedifference.cpp`

```
#include <iostream>
#include <iomanip>

int main() {
    double d1 = 2000.5;
    double d2 = 2000.0;
    std::cout << std::setprecision(16) << (d1 - d2) << '\n';
    double d3 = 2000.58;
    double d4 = 2000.0;
    std::cout << std::setprecision(16) << (d3 - d4) << '\n';
}
```

The output of Listing 4.15 (imprecisedifference.cpp) is:

```
0.5
0.5799999999999272
```

The program uses an additional `#include` directive:

```
#include <iomanip>
```

This preprocessor directive allows us to use the `std::setprecision` output stream manipulator that directs the `std::cout` output stream object to print more decimal places in floating-point values. During the program's execution, the first subtraction yields the correct answer. We now know that some floating-point numbers (like 0.5) have exact internal representations while others are only approximations. The exact answer for the second subtraction should be 0.58, and if we round the reported result to 12 decimal places, the answer matches. Floating-point arithmetic often produces results that are close approximations of the true answer.

Listing 4.16 (precise8th.cpp) computes zero in a roundabout way:

$$1-\frac{1}{8}-\frac{1}{8}-\frac{1}{8}-\frac{1}{8}-\frac{1}{8}-\frac{1}{8}-\frac{1}{8}-\frac{1}{8} = 0$$

Listing 4.16: `precise8th.cpp`

```
#include <iostream>

int main() {
    double one = 1.0,
           one_eighth = 1.0/8.0,
           zero = one - one_eighth - one_eighth - one_eighth
                      - one_eighth - one_eighth - one_eighth
                      - one_eighth - one_eighth;

    std::cout << "one = " << one << ", one_eighth = " << one_eighth
              << ", zero = " << zero << '\n';

}
```

Listing 4.16 (precise8th.cpp) prints

```
one = 1, one_eighth = 0.125, zero = 0
```

The number $\frac{1}{8}$ has an exact decimal representation, 0.625. It also has an exact binary representation, 0.001_2. Consider, however, $\frac{1}{5}$. While $\frac{1}{5} = 0.2$ has a finite representation in base 10, it has no finite representation in base 2:

$$\frac{1}{5} = 0.2 = 0.0011001100110\overline{0011}_2$$

In the binary representation the 0011_2 bit sequence repeats without end. This means $\frac{1}{5}$ does not have an exact floating-point representation. Listing 4.17 (imprecise5th.cpp) illustrates with arithmetic involving $\frac{1}{5}$.

Listing 4.17: imprecise5th.cpp

```
#include <iostream>

int main() {
    double one = 1.0,
           one_fifth = 1.0/5.0,
           zero = one - one_fifth - one_fifth - one_fifth
                      - one_fifth - one_fifth;

    std::cout << "one = " << one << ", one_fifth = " << one_fifth
              << ", zero = " << zero << '\n';

}
```

```
one = 1, one_fifth = 0.2, zero = 5.55112e-017
```

Surely the reported answer ($5.551122 \times 10^{-17} = 0.00000000000000005551122$) is close to the correct answer (zero). If you round it to the one-quadrillionth place (15 places behind the decimal point), it is correct.

What are the ramifications for programmers of this inexactness of floating-point numbers? Section 9.4.6 shows how the misuse of floating-point values can lead to logic errors in programs.

Being careful to avoid overflow and underflow, integer arithmetic is exact and, on most computer systems, faster than floating-point arithmetic. If an application demands the absolute correct answer and integers are appropriate for the computation, you should choose integers. For example, in financial calculations it is important to keep track of every cent. The exact nature of integer arithmetic makes integers an attractive option. When dealing with numbers, an integer type should be the first choice of programmers.

The limitations of floating-point numbers are unavoidable since computers have finite resources. Compromise is inevitable even when we do our best to approximate values with infinite characteristics in a finite way. Despite their inexactness, double-precision floating-point numbers are used every day throughout the world to solve sophisticated scientific and engineering problems; for example, the appropriate use of floating-point numbers have enabled space probes to reach distant planets. In the example C++ programs above that demonstrate the inexactness of floating-point numbers, the problems largely go away if we agree that we must compute with the most digits possible and then round the result to fewer digits. Floating-point numbers provide a good trade-off of precision for practicality.

4.9 More Arithmetic Operators

As Listing 4.12 (enhancedtimeconv.cpp) demonstrates, an executing program can alter a variable's value by performing some arithmetic on its current value. A variable may increase by one or decrease by five. The statement

```
x = x + 1;
```

increments x by one, making it one bigger than it was before this statement was executed. C++ has a shorter statement that accomplishes the same effect:

```
x++;
```

This is the *increment* statement. A similar *decrement* statement is available:

```
x--;    // Same as x = x - 1;
```

These statements are more precisely *post-increment* and *post-decrement* operators. There are also *pre-increment* and *pre-decrement* forms, as in

```
--x;    // Same as x = x - 1;
++y;    // Same as y = y + 1;
```

When they appear alone in a statement, the pre- and post- versions of the increment and decrement operators work identically. Their behavior is different when they are embedded within a more complex statement. Listing 4.18 (prevspost.cpp) demonstrates how the pre- and post- increment operators work slightly differently.

Listing 4.18: prevspost.cpp

```
#include <iostream>

int main() {
    int x1 = 1, y1 = 10, x2 = 100, y2 = 1000;
    std::cout << "x1=" << x1 << ", y1=" << y1
              << ", x2=" << x2 << ", y2=" << y2 << '\n';
    y1 = x1++;
    std::cout << "x1=" << x1 << ", y1=" << y1
              << ", x2=" << x2 << ", y2=" << y2 << '\n';
    y2 = ++x2;
    std::cout << "x1=" << x1 << ", y1=" << y1
              << ", x2=" << x2 << ", y2=" << y2 << '\n';
}
```

Listing 4.18 (prevspost.cpp) prints

```
x1=1, y1=10, x2=100, y2=1000
x1=2, y1=1, x2=100, y2=1000
x1=2, y1=1, x2=101, y2=101
```

If `x1` has the value 1 just before the statement

```
y1 = x1++;
```

then immediately after the statement executes `x1` is 2 and `y1` is 1.

If `x1` has the value 1 just before the statement

```
y1 = ++x1;
```

then immediately after the statement executes `x1` is 2 and `y1` is also 2.

As you can see, the pre-increment operator uses the new value of the incremented variable when evaluating the overall expression. In contrast, the post-increment operator uses the original value of the incremented variable when evaluating the overall expression. The pre- and post-decrement operator behaves similarly.

For beginning programmers it is best to avoid using the increment and decrement operators within more complex expressions. We will use them frequently as standalone statements since there is no danger of misinterpreting their behavior when they are not part of a more complex expression.

C++ provides a more general way of simplifying a statement that modifies a variable through simple arithmetic. For example, the statement

```
x = x + 5;
```

can be shorted to

```
x += 5;
```

This statement means "increase x by five." Any statement of the form

x *op*= *exp*;

where

- x is a variable.
- *op*= is an arithmetic operator combined with the assignment operator; for our purposes, the ones most useful to us are +=, −=, ⋆=, /=, and %=.
- *exp* is an expression compatible with the variable x.

Arithmetic reassignment statements of this form are equivalent to

x = x *op exp*;

This means the statement

```
x *= y + z;
```

is equivalent to

```
x = x * (y + z);
```

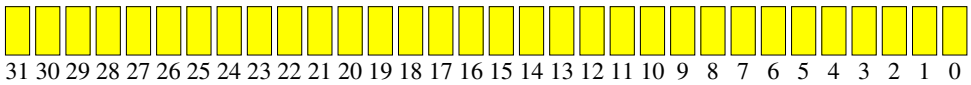
The version using the arithmetic assignment does not require parentheses. The arithmetic assignment is especially handy if a variable with a long name is to be modified; consider

```
temporary_filename_length = temporary_filename_length / (y + z);
```

versus

```
temporary_filename_length /= y + z;
```

Figure 4.10 The bit positions of a 32-bit C++ unsigned integer



Do not accidentally reverse the order of the symbols for the arithmetic assignment operators, like in the statement

```
x =+ 5;
```

Notice that the + and = symbols have been reversed. The compiler interprets this statement as if it had been written

```
x = +5;
```

that is, assignment and the unary operator. This assigns x to exactly five instead of increasing it by five.

Similarly,

```
x =- 3;
```

would assign −3 to x instead of decreasing x by three.

Section 4.10 examines some additional operators available in C++.

4.10 Bitwise Operators

In addition to the common arithmetic operators introduced in Section 4.1, C++ provides a few other special-purpose arithmetic operators. These special operators allow programmers to examine or manipulate the individual bits that make up data values. They are known as the *bitwise operators*. These operators consist of &, |, ^, ∼ , >>, and <<. Applications programmers generally do not need to use bitwise operators very often, but bit manipulation is essential in many systems programming tasks.

Consider 32-bit unsigned integers. The bit positions usually are numbered right to left, starting with zero. Figure 4.10 shows how the individual bit positions often are numbered.

The bitwise *and* operator, &, takes two integer subexpressions and computes an integer result. The expression e_1 & e_2 is evaluated as follows:

If bit 0 in both e_1 and e_2 is 1, then bit 0 in the result is 1; otherwise, bit 0 in the result is 0.

If bit 1 in both e_1 and e_2 is 1, then bit 1 in the result is 1; otherwise, bit 1 in the result is 0.

If bit 2 in both e_1 and e_2 is 1, then bit 2 in the result is 1; otherwise, bit 2 in the result is 0.

⋮

If bit 31 in both e_1 and e_2 is 1, then bit 31 in the result is 1; otherwise, bit 31 in the result is 0.

For example, the expression `13 & 14` evaluates to 12, since:

$$\begin{array}{rrcl} & 13_{10} & = & 00000000000000000000000000001101_2 \\ \& & 14_{10} & = & 00000000000000000000000000001110_2 \\ \hline & 12_{10} & = & 00000000000000000000000000001100_2 \end{array}$$

Bits 2 and 3 are one for both 13 and 14; thus, bits 2 and 3 in the result must be one.

The bitwise *or* operator, `|`, takes two integer subexpressions and computes an integer result. The expression e_1 `|` e_2 is evaluated as follows:

If bit 0 in both e_1 and e_2 is 0, then bit 0 in the result is 0; otherwise, bit 0 in the result is 1.

If bit 1 in both e_1 and e_2 is 0, then bit 1 in the result is 0; otherwise, bit 1 in the result is 1.

If bit 2 in both e_1 and e_2 is 0, then bit 2 in the result is 0; otherwise, bit 2 in the result is 1.

$\vdots$

If bit 31 in both e_1 and e_2 is 0, then bit 31 in the result is 0; otherwise, bit 31 in the result is 1.

For example, the expression `13 | 14` evaluates to 15, since:

$$\begin{array}{rrcl} & 13_{10} & = & 00000000000000000000000000001101_2 \\ | & 14_{10} & = & 00000000000000000000000000001110_2 \\ \hline & 15_{10} & = & 00000000000000000000000000001111_2 \end{array}$$

Bits 4–31 are zero in both 13 and 14. In bits 0–3 either 13 has a one or 14 has a one; therefore, the result has ones in bits 0–3 and zeroes everywhere else.

The bitwise *exclusive or* (often refered to as *xor*) operator (`^`) takes two integer subexpressions and computes an integer result. The expression e_1 `^` e_2 is evaluated as follows:

If bit 0 in e_1 is the same as bit 0 in e_2, then bit 0 in the result is 0; otherwise, bit 0 in the result is 1.

If bit 1 in e_1 is the same as bit 1 in e_2, then bit 1 in the result is 0; otherwise, bit 1 in the result is 1.

If bit 2 in e_1 is the same as bit 2 in e_2, then bit 2 in the result is 0; otherwise, bit 2 in the result is 1.

$\vdots$

If bit 31 in e_1 is the same as bit 31 in e_2, then bit 31 in the result is 0; otherwise, bit 31 in the result is 1.

For example, the expression `13 ^ 14` evaluates to 3, since:

$$\begin{array}{rrcl} & 13_{10} & = & 00000000000000000000000000001101_2 \\ \hat{} & 14_{10} & = & 00000000000000000000000000001110_2 \\ \hline & 3_{10} & = & 00000000000000000000000000000011_2 \end{array}$$

Bits 0 and 1 differ in 13 and 14, so these bits are one in the result. The bits match in all the other positions, so these positions must be set to zero in the result.

The bitwise *negation* operator ($\sim$) is a unary operator that inverts all the bits of its expression. The expression $\sim e$ is evaluated as follows:

If bit 0 in e is 0, then bit 0 in the result is 1; otherwise, bit 0 in the result is 0.

If bit 1 in e is 0, then bit 1 in the result is 1; otherwise, bit 1 in the result is 0.

If bit 2 in e is 0, then bit 2 in the result is 1; otherwise, bit 2 in the result is 0.

$$\vdots$$

If bit 31 in e is 0, then bit 31 in the result is 1; otherwise, bit 31 in the result is 0.

For example, the `unsigned` expression `~13u` evaluates to 4,294,967,282, since

$$\begin{array}{rcl} 13_{10} & = & 00000000000000000000000000001101_2 \\ & & \text{negate} \downarrow \\ \hline -14_{10} & = & 11111111111111111111111111110010_2 \end{array}$$

For signed integers the 31st bit stores the number's sign. Signed integers use a representation called *two's complement* binary, a slight variation of the standard binary layout. Suffice it to say that the `int` expression `~13` evaluates to the same bit pattern as `~13u`, but as a signed integer it represents -14.

The shift operators move all the bits in an integer to the left or right:

- **Shift left** (`<<`). The expression x `<<` y, where x and y are integer types, shifts all the bits in `x` to the left y places. Zeros fill vacated positions. The bits shifted off the left side are discarded. The expression `5 << 2` evaluates to 20, since $5_{10} = 101_2$ shifted two places to the left yields $10100_2 = 20_{10}$. Observe that x `<<` y is equal to $x \times 2^y$.

- **Shift right** (`>>`). The expression x `>>` y, where x and y are integer types, shifts all the bits in `x` to the right y places. What fills the vacated bits on the left depends on whether the integer is signed or unsigned (for example, `int` vs. `unsigned`):

  - For signed values the vacated bit positions are filled with the sign bit (the original leftmost bit).
  - For unsigned values the vacated bit positions are filled with zeros.

  The bits shifted off the right side are discarded. The expression `5 >> 2` evaluates to 1, since $5_{10} = 101_2$ shifted two places to the left yields $001_2 = 20_{10}$ (the original bits in positions 1 and 0 are shifted off the end and lost). Observe that x `>>` y is equal to $x \div 2^y$.

Do not confuse the left shift operator (`<<`) with the output stream insertion operator (`<<`). The operators are identical, but the context differentiates them. If the left operand is an integer type, `<<` means left shift; if the left operand is a stream output object like `std::cout`, `<<` means send the right-hand operand to the output stream object for display.
Similarly, the input stream object `std::cin` uses the `>>` for a different purpose from the right shift operator used with integers.

Listing 4.19 (bitwiseoperators.cpp) experiments with bitwise operators.

Listing 4.19: bitwiseoperators.cpp

```
#include <iostream>

int main() {
    int x, y;
    std::cout << "Please enter two integers: ";
    std::cin >> x >> y;
    std::cout << x << " & " << y << " = " << (x & y) << '\n';
    std::cout << x << " | " << y << " = " << (x | y) << '\n';
    std::cout << x << " ^ " << y << " = " << (x ^ y) << '\n';
    std::cout << "~" << x << " = " << ~x << '\n';
    std::cout << x << " << " << 2 << " = " << (x << 2) << '\n';
    std::cout << x << " >> " << 2 << " = " << (x >> 2) << '\n';
}
```

Developers use bitwise operations for a variety of systems-level programming tasks. For example, in a graphical user interface (GUI), the user generates *events* by interacting with an application using the mouse and a keyboard. One event might be clicking a mouse button over a particular graphical element (like a button) within a window. Multiple pieces of information about this event can be stored in a single integer. For example, bit 0 may indicate whether or not the ↑ Shift key was held down when the mouse button was clicked. Bit 1 may be responsible for the Alt key, bit 3 for the Ctrl key, etc. Thus the integer value 5, which in binary is

`00000000000000000000000000000101`

means that when the mouse button was clicked both the ↑ Shift and Alt keys were being held down. This might require a different action on the part of the program than if some other combination of keys (or none) were being pressed. For example, suppose the `int` variable `key_status` holds information about which keys the user was depressing during the most recent mouse click. Consider the expression `key_status & 1`. The bit string

`00000000000000000000000000000001`

represents the expression 1, and the value of `key_status` is unknown, so `key_status & 1` is

$$\begin{array}{rrcl} & \texttt{key_status} & = & \texttt{????????????????????????????????}_2 \\ \& & 1_{10} & = & \texttt{00000000000000000000000000000001}_2 \\ \hline & 0_{10} \text{ or } 1_{10} & = & \texttt{0000000000000000000000000000000?}_2 \end{array}$$

If the answer is zero, this means bit 0 is 0 in `key_status`, and so the ↑ Shift key is not depressed. On the other hand, if the answer is one, this means bit 0 is 1, and so the ↑ Shift key *is* depressed. In the expression

```
key_status & 1
```

the 1 is called a *mask*, since it serves to mask off, or "remove" the first 31 bits of `key_status`.

Usually the GUI library will define constants that help programmers examine or modify particular bits; for example, given the following constants:

```
const int SHIFT_DOWN = 1;                // This is 0...0001
const int CTRL_DOWN = SHIFT_DOWN << 1;   // This is 0...0010
const int ALT_DOWN = CTRL_DOWN << 1;     // This is 0...0100
```

the expression

```
key_status & 1
```

is better written

```
key_status & SHIFT_DOWN
```

The expression

```
key_status & SHIFT_DOWN | ALT_DOWN
```

can test for both the ↑ Shift and Alt keys being down during the mouse event. Do you see how the expression `SHIFT_DOWN | ALT_DOWN` means both keys are down simultaneously?

We can use masks to ensure that certain bits are on or off. To see how, consider the mask 5, which is

```
00000000000000000000000000000101
```

If `x` is a 32-bit integer variable, we can selectively turn on its bits 0 and 2 with the statement

```
x = x | 5;
```

Next, consider the unsigned value `4294967290u`, which is $\sim$ `5u`, or

```
11111111111111111111111111111010
```

If `x` is a 32-bit integer variable, we can selectively turn off its bits 0 and 2 with the statement

```
x = x & 4294967290u;
```

We cannot fully appreciate the utility of using bitwise operators for masking purposes until we consider conditional execution in Chapter 5. Even then, since we concentrate on applications programming instead of systems programming in this book, we will have little use for the bitwise operators except for a few isolated situations. It is good to be aware of their presence, though, since their accidental use may lead to difficult to diagnose compiler messages and logic errors.

Like the other C++ arithmetic operators that work on two operands, we may combine the bitwise binary operators `&`, `|`, `^`, `<<`, and `>>` with assignment to simplify the modification of a variable; for example, the following statement

```
x = x | y;  // Turn on bits in x determined by
```

may be written as

```
x |= y;  // Turn on bits in x determined by
```

Table 4.6 lists the possibilities.

4.11 Algorithms

An *algorithm* is a finite sequence of steps, each step taking a finite length of time, that solves a problem or computes a result. A computer program is one example of an algorithm, as is a recipe to make lasagna. In

Assignment	Short Cut
`x = x & y;`	`x &= y;`
`x = x | y;`	`x |= y;`
`x = x ^ y;`	`x ^= y;`
`x = x << y;`	`x <<= y;`
`x = x >> y;`	`x >>= y;`

Table 4.6: The bitwise assignment operators

both of these examples, the order of the steps matter. In the case of lasagna, the noodles must be cooked in boiling water before they are layered into the filling to be baked. It would be inappropriate to place the raw noodles into the pan with all the other ingredients, bake it, and then later remove the already baked noodles to cook them in boiling water separately. In the same way, the ordering of steps is very important in a computer program. While this point may be obvious, consider the following sound argument:

1. The relationship between degrees Celsius and degrees Fahrenheit can be expressed as

$$^{\circ}C = \frac{5}{9} \times (^{\circ}F - 32)$$

2. Given a temperature in degrees Fahrenheit, the corresponding temperature in degrees Celsius can be computed.

Armed with this knowledge, Listing 4.20 (faultytempconv.cpp) follows directly.

Listing 4.20: `faultytempconv.cpp`

```
// File faultytempconv.cpp

#include <iostream>

int main() {
    double degreesF = 0, degreesC = 0;
    // Define the relationship between F and C
    degreesC = 5.0/9*(degreesF - 32);
    // Prompt user for degrees F
    std::cout << "Enter the temperature in degrees F: ";
    // Read in the user's input
    std::cin >> degreesF;
    // Report the result
    std::cout << degreesC << '\n';
}
```

Unfortunately, the executing program always displays

```
-17.7778
```

regardless of the input provided. The English description provided above is correct. No integer division problems lurk, as in Listing 4.10 (tempconv.cpp). The problem lies simply in statement ordering. The statement

```
degreesC = 5.0/9*(degreesF - 32);
```

is an *assignment* statement, not a definition of a relationship that exists throughout the program. At the point of the assignment, `degreesF` has the value of zero. The executing program computes and assigns the `degreesC` variable *before* receiving `degreesF`'s value from the user.

As another example, suppose `x` and `y` are two integer variables in some program. How would we interchange the values of the two variables? We want `x` to have `y`'s original value and `y` to have `x`'s original value. This code may seem reasonable:

```
x = y;
y = x;
```

The problem with this section of code is that after the first statement is executed, `x` and `y` both have the same value (`y`'s original value). The second assignment is superfluous and does nothing to change the values of `x` or `y`. The solution requires a third variable to remember the original value of one the variables before it is reassigned. The correct code to swap the values is

```
temp = x;
x = y;
y = temp;
```

This small example emphasizes the fact that algorithms must be specified precisely. Informal notions about how to solve a problem can be valuable in the early stages of program design, but the coded program requires a correct detailed description of the solution.

The algorithms we have seen so far have been simple. Statement 1, followed by Statement 2, etc. until every statement in the program has been executed. Chapter 5 and Chapter 6 introduce some language constructs that permit optional and repetitive execution of some statements. These constructs allow us to build programs that do much more interesting things, but more complex algorithms are required to make it happen. We must not lose sight of the fact that a complicated algorithm that is 99% correct is *not* correct. An algorithm's design and implementation can be derailed by inattention to the smallest of details.

4.12 Exercises

1. Is the literal `4` a valid C++ expression?
2. Is the variable `x` a valid C++ expression?
3. Is `x + 4` a valid C++ expression?
4. What affect does the unary `+` operator have when applied to a numeric expression?
5. Sort the following binary operators in order of high to low precedence: `+`, `-`, `*`, `/`, `%`, `=`.
6. Write a C++ program that receives two integer values from the user. The program then should print the sum (addition), difference (subtraction), product (multiplication), quotient (division), and remainder after division (modulus). Your program must use only integers.

   A sample program run would look like (the user enters the 10 and the 2 after the colons, and the program prints the rest):

   ```
   Please enter the first number: 10
   Please enter the second number: 2
   10 + 2 = 12
   10 - 2 = 8
   ```

```
10 * 2 = 20
10 / 2 = 5
10 % 2 = 0
```

Can you explain the results it produces for all of these operations?

7. Write a C++ program that receives two double-precision floating-point values from the user. The program then should print the sum (addition), difference (subtraction), product (multiplication), and quotient (division). Your program should use only integers.

   A sample program run would look like (the user enters the 10 and the 2.5 after the colons, and the program prints the rest):

```
Please enter the first number: 10
Please enter the second number: 2.5
10 + 2.5 = 12.5
10 - 2.5 = 7.5
10 * 2.5 = 25
10 / 2.5 = 4
```

   Can you explain the results it produces for all these operations? What happens if you attempt to compute the remainder after division (modulus) with double-precision floating-point values?

8. Given the following declaration:

```
int x = 2;
```

   Indicate what each of the following C++ statements would print.

   (a) `std::cout << "x"<< '\n';`
   (b) `std::cout << 'x'<< '\n';`
   (c) `std::cout << x << '\n';`
   (d) `std::cout << "x + 1"<< '\n';`
   (e) `std::cout << 'x'+ 1 << '\n';`
   (f) `std::cout << x + 1 << '\n';`

9. Sort the following types in order from narrowest to widest: `int`, `double`, `float`, `long`, `char`.

10. Given the following declarations:

```
int i1 = 2, i2 = 5, i3 = -3;
double d1 = 2.0, d2 = 5.0, d3 = -0.5;
```

   Evaluate each of the following C++ expressions.

   (a) `i1 + i2`
   (b) `i1 / i2`
   (c) `i2 / i1`
   (d) `i1 * i3`
   (e) `d1 + d2`
   (f) `d1 / d2`
   (g) `d2 / d1`

(h) `d3 * d1`
(i) `d1 + i2`
(j) `i1 / d2`
(k) `d2 / i1`
(l) `i2 / d1`
(m) `i1/i2*d1`
(n) `d1*i1/i2`
(o) `d1/d2*i1`
(p) `i1*d1/d2`
(q) `i2/i1*d1`
(r) `d1*i2/i1`
(s) `d2/d1*i1`
(t) `i1*d2/d1`

11. What is printed by the following statement:

```
std::cout << /* 5 */ 3 << '\n';
```

12. Given the following declarations:

```
int i1 = 2, i2 = 5, i3 = -3;
double d1 = 2.0, d2 = 5.0, d3 = -0.5;
```

Evaluate each of the following C++ expressions.

(a) `i1 + (i2 * i3)`
(b) `i1 * (i2 + i3)`
(c) `i1 / (i2 + i3)`
(d) `i1 / i2 + i3`
(e) `3 + 4 + 5 / 3`
(f) `(3 + 4 + 5) / 3`
(g) `d1 + (d2 * d3)`
(h) `d1 + d2 * d3`
(i) `d1 / d2 - d3`
(j) `d1 / (d2 - d3)`
(k) `d1 + d2 + d3 / 3`
(l) `(d1 + d2 + d3) / 3`
(m) `d1 + d2 + (d3 / 3)`
(n) `3 * (d1 + d2) * (d1 - d3)`

13. How are single-line comments different from block comments?

14. Can block comments be nested?

15. Which is better, too many comments or too few comments?

16. What is the purpose of comments?

17. The programs in Listing 3.4 (variable.cpp), Listing 4.4 (reformattedvariable.cpp), and Listing 4.5 (reformattedvariable2.cpp) compile to the same machine code and behave exactly the same. What makes one of the programs clearly better than the others?

18. Why is human readability such an important consideration?

19. Consider the following program which contains some errors. You may assume that the comments within the program accurately describe the program's intended behavior.

```
#include <iostream>

int main() {
    int n1, n2, d1;                        // 1
    // Get two numbers from the user
    cin << n1 << n2;                       // 2
    // Compute sum of the two numbers
    std::cout << n1 + n2 << '\n';          // 3
    // Compute average of the two numbers
    std::cout << n1+n2/2 << '\n';          // 4
    // Assign some variables
    d1 = d2 = 0;                           // 5
    // Compute a quotient
    std::cout << n1/d1 << '\n';            // 6
    // Compute a product
    n1*n2 = d1;                            // 7
    // Print result
    std::cout << d1 << '\n';               // 8
}
```

For each line listed in the comments, indicate whether or not a compile-time, run-time, or logic error is present. Not all lines contain an error.

20. What distinguishes a compiler warning from a compiler error? Should you be concerned about warnings? Why or why not?

21. What are the advantages to enhancing the warning reporting capabilities of the compiler?

22. Write the shortest way to express each of the following statements.

    (a) `x = x + 1;`

    (b) `x = x / 2;`

    (c) `x = x - 1;`

    (d) `x = x + y;`

    (e) `x = x - (y + 7);`

    (f) `x = 2*x;`

    (g) `number_of_closed_cases = number_of_closed_cases + 2*ncc;`

23. What is printed by the following code fragment?

```
int x1 = 2, y1, x2 = 2, y2;
y1 = ++x1;
y2 = x2++;
std::cout << x1 << " " << x2 << '\n';
std::cout << y1 << " " << y2 << '\n';
```

Why does the output appear as it does?

24. Consider the following program that attempts to compute the circumference of a circle given the radius entered by the user. Given a circle's radius, r, the circle's circumference, C is given by the formula:

$$C = 2\pi r$$

```
#include <iostream>

int main() {
    double C, r;
    const double PI = 3.14159;
    // Formula for the area of a circle given its radius
    C = 2*PI*r;
    // Get the radius from the user
    cout >> "Please enter the circle's radius: ";
    cin << r;
    // Print the circumference
    std::cout << "Circumference is " << C << '\n';
}
```

(a) The compiler issues a warning. What is the warning?

(b) The program does not produce the intended result. Why?

(c) How can it be repaired so that it not only eliminates the warning but also removes the logic error?

25. In mathematics, the midpoint between the two points (x_1, y_1) and (x_2, y_2) is computed by the formula

$$\left(\frac{x_1 + x_2}{2}, \frac{y_1 + y_2}{2}\right)$$

Write a C++ program that receives two mathematical points from the user and computes and prints their midpoint.

A sample run of the program produces

```
Please enter the first point: (0,0)
Please enter the second point: (1,1)
The midpoint of (0,0) and (1,1) is (0.5,0.5)
```

The user literally enters "(0,0)" and "(1,1)" with the parentheses and commas as shown. To see how to do this, suppose you want to allow a user to enter the point $(2.3, 9)$, assigning the x component of the point to a variable named `x` and the y component to a variable named `y`. You can add the following code fragment to your program to achieve the desired effect:

Food	Calories
Bean burrito	357
Salad w/dressing	185
Milkshake	388

Table 4.7: Calorie content of several fast food items

```
double x, y;
char left_paren, comma, right_paren;
std::cin >> left_paren >> x >> comma >> y >> right_paren;
```

If the user literally types `(2.3,9)`, the `std::cin` statement will assign the `(` character to the variable `left_paren`. It next will assign 2.3 to the variable `x`. It assigns the `,` character to the variable named `comma`, the value 9 to the `y` variable, and the `)` character to the `right_paren` variable. The `left_paren`, `comma`, and `right_paren` variables are just placeholders for the user's input and are not used elsewhere within the program. In reality, the user can type in other characters in place of the parentheses and comma as long as the numbers are in the proper location relative to the characters; for example, the user can type `*2.3:9#`, and the program will interpret the input as the point $(2.3, 9)$.

26. Table 4.7 lists the Calorie contents of several foods. Running or walking burns off about 100 Calories per mile. Write a C++ program that requests three values from the user: the number of bean burritos, salads, and shakes consumed (in that order). The program should then display the number of miles that must be run or walked to burn off the Calories represented in that food. The program should run as follows (the user types in the 3 2 1):

```
Number of bean burritos, bowls of salad, and milkshakes eaten?   3 2 1

You ingested 1829 Calories
You will have to run 18.29 miles to expend that much energy
```

Observe that the result is a floating-point value, so you should use floating-point arithmetic to compute the answers for this problem.

Chapter 5

Conditional Execution

All the programs in the preceding chapters execute exactly the same statements regardless of the input, if any, provided to them. They follow a linear sequence: *Statement* 1, *Statement* 2, etc. until the last statement is executed and the program terminates. Linear programs like these are very limited in the problems they can solve. This chapter introduces constructs that allow program statements to be optionally executed, depending on the context (input) of the program's execution.

5.1 Type bool

Arithmetic expressions evaluate to numeric values; a *Boolean* expression, evaluates to `true` or `false`. While Boolean expressions may appear very limited on the surface, they are essential for building more interesting and useful programs.

C++ supports the non-numeric data type `bool`, which stands for Boolean. The term Boolean comes from the name of the British mathematician George Boole. A branch of discrete mathematics called Boolean algebra is dedicated to the study of the properties and the manipulation of logical expressions. Compared to the numeric types, the `bool` type is very simple in that it can represent only two values: `true` or `false`. Listing 5.1 (boolvars.cpp) is a simple program demonstrating the use of Boolean variables.

Listing 5.1: `boolvars.cpp`

```
#include <iostream>

int main() {
    // Declare some Boolean variables
    bool a = true, b = false;
    std::cout << "a = " << a << ", b = " << b << '\n';
    // Reassign a
    a = false;
    std::cout << "a = " << a << ", b = " << b << '\n';
    // Mix integers and Booleans
    a = 1;
    b = 1;
    std::cout << "a = " << a << ", b = " << b << '\n';
    // Assign Boolean value to an integer
```

Operator	Meaning
==	Equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
!=	Not equal to

Table 5.1: C++ Relational operators

```
    int x = a, y = true;
    std::cout << "a = " << a << ", b = " << b
              << ", x = " << x << ", y = " << y << '\n';
    // More mixing
    a = 1725;   // Warning issued
    b = -19;    // Warning issued
    std::cout << "a = " << a << ", b = " << b << '\n';
}
```

As you can see from running Listing 5.1 (boolvars.cpp), the Boolean values `false` and `true` are represented as integer 0 and integer 1. More precisely, zero represents the `bool` value `false`, and any non-zero integer (positive or negative) means `true`. The direct assignment to a `bool` variable of an integer other than 0 or 1 may result in a warning (Visual C++ reports truncation of 'int' to 'bool'), but the variable is still interpreted as `true`. The data type `bool` is basically a convenience for programmers; any C++ program that uses `bool` variables can be rewritten using integers instead to achieve the same results. While Boolean values and variables are freely compatible and interchangeable with integers, the `bool` type is convenient and should be used when the context involves truth values instead of numbers.

It is important to note that the Visual C++ compiler issues warnings for the last two assignment statements in Listing 5.1 (boolvars.cpp). Even though any non-zero value is considered `true`, 1 is the preferred integer equivalent to `true` (as you can see when you attempt to print the literal value `true`). Since the need to assign to a Boolean variable a value other than `true` or `false` or the equivalent 1 or 0 should be extremely rare, the compiler's message alerts the programmer to check to make sure the assignment is not a mistake.

5.2 Boolean Expressions

The simplest Boolean expressions are `false` and `true`, the Boolean literals. A Boolean variable is also a Boolean expression. An expression comparing numeric expressions for equality or inequality is also a Boolean expression. The simplest kinds of Boolean expressions use *relational operators* to compare two expressions. Table 5.1 lists the relational operators available in C++.

Table 5.2 shows some simple Boolean expressions with their associated values. An expression like `10 < 20` is legal but of little use, since the expression `true` is equivalent, simpler, and less likely to confuse human readers. Boolean expressions are extremely useful when their truth values depend on the values of one or more variables.

The relational operators are binary operators and are all left associative. They all have a lower precedence than any of the arithmetic operators; therefore, the expression

Expression	Value
`10 < 20`	always true
`10 >= 20`	always false
`x == 10`	true only if `x` has the value 10
`X != y`	true unless `x` and `y` have the same values

Table 5.2: Relational operator examples

```
x + 2 < y / 10
```

is evaluated as if parentheses were placed as so:

```
(x + 2) < (y / 10)
```

C++ allows statements to be simple expressions; for example, the statement

```
x == 15;
```

may look like an attempt to assign the value 15 to the variable `x`, but it is not. The `=` operator performs assignment, but the `==` operator checks for relational equality. If you make a mistake and use `==` as shown here, Visual C++ will issue a warning that includes the message

warning C4553: '==' : operator has no effect; did you intend '='?

Recall from Section 4.6.4 that a compiler warning does not indicate a violation of the rules of the language; rather it alerts the programmer to a possible trouble spot in the code.

Another example of an expression used as a statement is

```
x + 15;
```

This statement is a legal (but useless) C++ statement, and the compiler notifies us accordingly:

warning C4552: '+' : operator has no effect; expected operator with side-effect

Why are expressions allowed as statements? Some simple expressions have side effects that do alter the behavior of the program. One example of such an expression is `x++`. Listing 4.18 (prevspost.cpp) showed how `x++` behaves both as a standalone statement and as an expression within a larger statement. A more common example is the use of a function call (which is an expression) as standalone a statement. (We introduce functions in Chapter 8.) In order to keep the structure of the language as uniform as possible, C++ tolerates useless expressions as statements to enable programmers to use the more useful expression-statements. Fortunately, most compilers issue informative warnings about the useless expression-statements to keep developers on track.

5.3 The Simple if Statement

The Boolean expressions described in Section 5.2 at first may seem arcane and of little use in practical programs. In reality, Boolean expressions are essential for a program to be able to adapt its behavior at run time. Most truly useful and practical programs would be impossible without the availability of Boolean expressions.

The run-time exceptions mentioned in Section 4.6 arise from logic errors. One way that Listing 4.6 (dividedanger.cpp) can fail is when the user enters a zero for the divisor. Fortunately, programmers can take steps to ensure that division by zero does not occur. Listing 5.2 (betterdivision.cpp) shows how it might be done.

Listing 5.2: betterdivision.cpp

```
#include <iostream>

int main() {
    int dividend, divisor;

    // Get two integers from the user
    std::cout << "Please enter two integers to divide:";
    std::cin >> dividend >> divisor;
    // If possible, divide them and report the result
    if (divisor != 0)
        std::cout << dividend << "/" << divisor << " = "
                  << dividend/divisor << '\n';
}
```

The second `std::cout` statement may not always be executed. In the following run

```
Please enter two integers to divide: 32 8
32/8 = 4
```

it is executed, but if the user enters a zero as the second number:

```
Please enter two integers to divide: 32 0
```

the program prints nothing after the user enters the values.

The last statement in Listing 5.2 (betterdivision.cpp) begins with the reserved word `if`. The `if` statement allows code to be optionally executed. In this case, the printing statement is executed only if the variable `divisor`'s value is not zero.

The Boolean expression

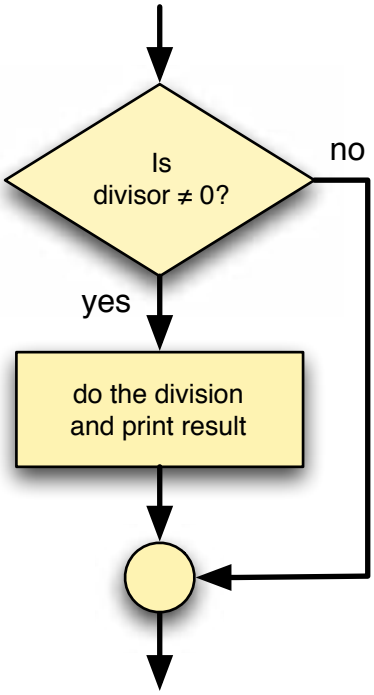
```
divisor != 0
```

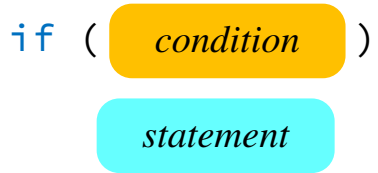
determines if the single statement that follows the right parenthesis is executed. If `divisor` is not zero, the message is printed; otherwise, the program prints nothing.

Figure 5.1 shows how program execution flows through the `if` statement. of Listing 5.2 (betterdivision.cpp).

The general form of a simple `if` statement is

Figure 5.1 if flowchart





- The reserved word `if` begins the `if` statement.
- The Boolean expression *condition* determines whether or not the body will be executed. The Boolean expression *must* be enclosed within parentheses as shown.
- The *statement* is the statement to be executed if the Boolean expression is true. The statement makes up the *body* of the `if` statement. Section 5.4 shows how the body can be composed of multiple statements.

Good coding style dictates we should indent the body to emphasize the optional execution and improve the program's readability. The compiler does not require the indentation. Sometimes programmers will place a one-statement body on the same line as the `if`; for example, the following `if` statement optionally assigns `y`:

```
if (x < 10)
    y = x;
```

and could be written as

```
if (x < 10) y = x;
```

but should *not* be written as

```
if (x < 10)
y = x;
```

because the lack of indentation hides the fact that the program optionally executes the assignment statement. The compiler will accept it, but it is misleading to human readers accustomed to the indentation convention. The compiler, of course, will accept the code written as

```
if(x<10)y=x;
```

but the lack of spaces makes it difficult for humans to read.

When the `if` statement is written the preferred way using two lines of source code, it is important **not** to put a semicolon at the end of the first line:

```
if (x < 10);  // No! Don't do this!
    y = x;
```

Here, the semicolon terminates the `if` statement, but the indentation implies that the second line is intended to be the body of the `if` statement. The compiler, however, interprets the badly formatted `if` statement as if it were written as

```
if (x < 10)
    ;  // This is what is really going on.
y = x;
```

This is legal in C++; it means the `if` statement has an *empty* body. In which case the assignment is not part of the body. The assignment statement is after the body and always will be executed regardless of the truth value of the Boolean expression.

When checking for equality, as in

```
if (x == 10)
    std::cout << "ten";
```

be sure to use the relational equality operator (==), not the assignment operator (=). Since an assignment statement has a value (the value that is assigned, see Section 4.3), C++ allows = within the conditional expression. It is, however, almost always a mistake when beginning programmers use = in this context. Visual C++ at warning Level 4 checks for the use of assignment within a conditional expression; the default Level 3 does not.

5.4 Compound Statements

Sometimes you need to optionally execute more than one statement based on a particular condition. Listing 5.3 (alternatedivision.cpp) shows how you must use curly braces to group multiple statements together into one *compound statement*.

Listing 5.3: `alternatedivision.cpp`

```
#include <iostream>

int main() {
    int dividend, divisor, quotient;

    // Get two integers from the user
    std::cout << "Please enter two integers to divide:";
    std::cin >> dividend >> divisor;
    // If possible, divide them and report the result
    if (divisor != 0) {
```

```
        quotient = dividend / divisor;
        std::cout << dividend << " divided by " << divisor << " is "
                  << quotient << '\n';
    }
}
```

The assignment statement and printing statement are both a part of the body of the `if` statement. Given the truth value of the Boolean expression `divisor != 0` during a particular program run, either both statements will be executed or neither statement will be executed.

A compound statement consists of zero or more statements grouped within curly braces. We say the curly braces define a *block* of statements. As a matter of style many programmers always use curly braces to delimit the body of an `if` statement even if the body contains only one statement:

```
if (x < 10) {
    y = x;
}
```

They do this because it is easy to introduce a logic error if additional statements are added to the body later and the programmer forgets to add then required curly braces.

The format of the following code

```
if (x < 10)
    y = x;
    z = x + 5;
```

implies that both assignments are part of the body of the `if` statement. Since multiple statements making up the body must be in a compound statement within curly braces, the compiler interprets the code fragment as if it had been written

```
if (x < 10)
    y = x;
z = x + 5;
```

Such code will optionally execute the first assignment statement and *always* execute the second assignment statement.

The programmer probably meant to write it as

```
if (x < 10) {
    y = x;
    z = x + 5;
}
```

The curly braces are optional if the body consists of a single statement. If the body consists of only one statement and curly braces are not used, then the semicolon that terminates the statement in the body also terminates the `if` statement. If curly braces are used to delimit the body, a semicolon is not required after the body's close curly brace.

An empty pair of curly braces represents an empty block. An empty block is a valid compound statement.

5.5 The if/else Statement

One undesirable aspect of Listing 5.2 (betterdivision.cpp) is if the user enters a zero divisor, the program prints nothing. It may be better to provide some feedback to the user to indicate that the divisor provided cannot be used. The `if` statement has an optional `else` clause that is executed only if the Boolean expression is false. Listing 5.4 (betterfeedback.cpp) uses the `if`/`else` statement to provide the desired effect.

Listing 5.4: `betterfeedback.cpp`

```
#include <iostream>

int main() {
    int dividend, divisor;

    // Get two integers from the user
    std::cout << "Please enter two integers to divide:";
    std::cin >> dividend >> divisor;
    // If possible, divide them and report the result
    if (divisor != 0)
        std::cout << dividend << "/" << divisor << " = "
                  << dividend/divisor << '\n';
    else
        std::cout << "Division by zero is not allowed\n";
}
```

A given program run will execute exactly one of either the `if` body or the `else` body. Unlike in Listing 5.2 (betterdivision.cpp), a message is always displayed.

```
Please enter two integers to divide: 32 0
Division by zero is not allowed
```

The `else` clause contains an alternate body that is executed if the condition is false. The program's flow of execution is shown in Figure 5.2.

Listing 5.4 (betterfeedback.cpp) avoids the division by zero run-time error that causes the program to terminate prematurely, but it still alerts the user that there is a problem. Another application may handle the situation in a different way; for example, it may substitute some default value for `divisor` instead of zero.

The general form of an `if`/`else` statement is

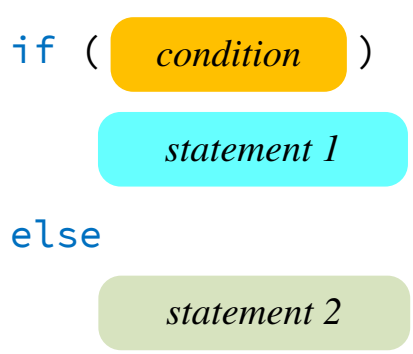
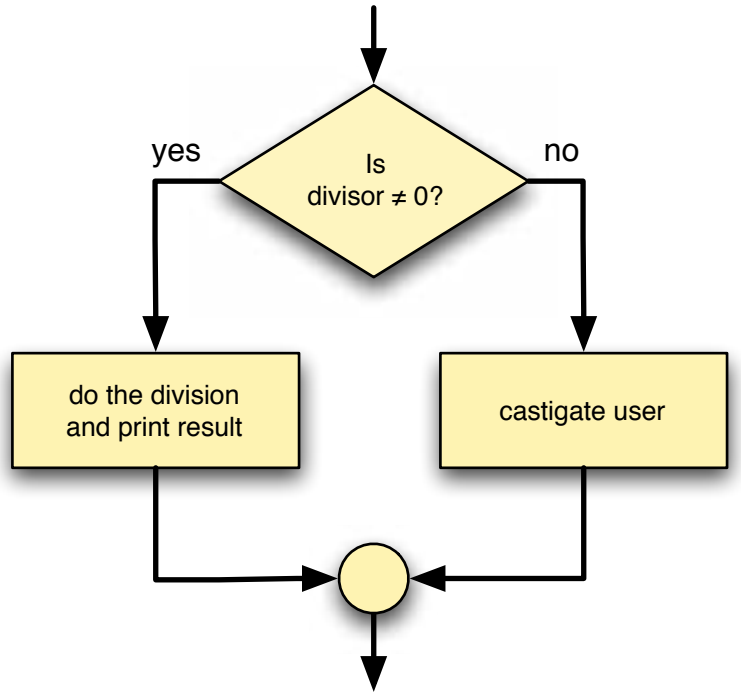


Figure 5.2 if/else flowchart



- The reserved word `if` begins the `if`/`else` statement.
- The *condition* is a Boolean expression that determines whether the running program will execute *statement 1* or *statement 2*. As with the simple `if` statement, the condition must appear within parentheses.
- The program executes *statement 1* if the condition is true. To make the `if`/`else` statement more readable, indent *statement 1* more spaces than the `if` line. This part of the `if` statement is sometimes called the body of the `if`.
- The reserved word `else` begins the second part of the `if`/`else` statement.
- The program executes *statement 2* if the condition is false. To make the `if`/`else` statement more readable, indent *statement 2* more spaces than the `else` line. This part of the `if`/`else` statement is sometimes called the body of the `else`.

The body of the `else` clause of an `if`/`else` statement may be a compound statement:

```
if (x == y)
    std::cout << x;
else {
    x = 0;
    std::cout << y;
}
```

or the `if` body alone may be a compound statement:

```
if (x == y) {
    std::cout << x;
    x = 0;
}
else
    std::cout << y;
```

or both parts may be compound:

```
if (x == y) {
    std::cout << x;
    x = 0;
}
else {
    std::cout << y;
    y = 0;
}
```

or, as in Listing 5.4 (betterfeedback.cpp), both the `if` body and the `else` body can be simple statements.

Remember, if you wish to associate more than one statement with the body of the `if` or `else`, you must use a compound statement. Compound statements are enclosed within curly braces (`{}`).

If you ever attempt to use an `if/else` statement and discover that you need to leave the `else` clause empty, as in

```
if (x == 2)
    std::cout << "x = " << x << '\n';
else
    ;   // Nothing to do otherwise
```

or, using a slightly different syntax, as

```
if (x == 2)
    std::cout << "x = " << x << '\n';
else {
}   // Nothing to do otherwise
```

you instead should use a simple `if` statement:

```
if (x == 2)
    std::cout << "x = " << x << '\n';
```

The empty `else` clauses shown above do work, but they complicate the code and make it more difficult for humans to read.

Due to the imprecise representation of floating-point numbers (see Listing 4.17 (imprecise5th.cpp) in Section 4.1), programmers must use caution when using the equality operator (==) by itself to compare floating-point expressions. Listing 5.5 (samedifferent.cpp) uses an `if/else` statement to demonstrate the perils of using the equality operator with floating-point quantities.

Listing 5.5: samedifferent.cpp

```
#include <iostream>
#include <iomanip>

int main() {
    double d1 = 1.11 - 1.10,
           d2 = 2.11 - 2.10;
    std::cout << "d1 = " << d1 << '\n';
    std::cout << "d2 = " << d2 << '\n';
    if (d1 == d2)
        std::cout << "Same\n";
    else
        std::cout << "Different\n";
    std::cout << "d1 = " << std::setprecision(20) << d1 << '\n';
    std::cout << "d2 = " << std::setprecision(20) << d2 << '\n';
}
```

In Listing 5.5 (samedifferent.cpp) the displayed values of `d1` and `d2` are rounded so they appear equivalent, but internally the exact representations are slightly different. By including the header `iomanip` we can use the `std::setprecision` stream manipulator to force `std::cout` to display more decimal places in the floating-point number it prints. Observe from the output of Listing 5.5 (samedifferent.cpp) that the two quantities that should be identically 0.01 are actually slightly different.

```
d1 = 0.01
d2 = 0.01
Different
```

e_1	e_2	e_1 && e_2	e_1 || e_2	!e_1
false	false	false	false	true
false	true	false	true	true
true	false	false	true	false
true	true	true	true	false

Table 5.3: Logical operators—e_1 and e_2 are Boolean expressions

```
d1 = 0.010000000000000009
d2 = 0.0099999999999997868
```

This result should not discourage you from using floating-point numbers where they truly are needed. In Section 9.4.6 we will see how to handle floating-point comparisons properly.

5.6 Compound Boolean Expressions

Simple Boolean expressions, each involving one relational operator, can be combined into more complex Boolean expressions using the logical operators && (and), || (or), and ! (not). A combination of two or more Boolean expressions using logical operators is called a *compound Boolean expression*.

To introduce compound Boolean expressions, consider a computer science degree that requires, among other computing courses, *Operating Systems* **and** *Programming Languages*. If we isolate those two courses, we can say a student must successfully complete both *Operating Systems* and *Programming Languages* to qualify for the degree. A student that passes *Operating Systems* but not *Programming Languages* will not have met the requirements. Similarly, *Programming Languages* without *Operating Systems* is insufficient, and a student completing neither *Operating Systems* nor *Programming Languages* surely does not qualify.

Logical **AND** works in exactly the same way. If e_1 and e_2 are two Boolean expressions, e_1 && e_2 is true only if e_1 and e_2 are both true; if either one is false or both are false, the compound expression is false.

To illustrate logical **OR**, consider two mathematics courses, *Differential Equations* and *Linear Algebra*. A computer science degree requires one of those two courses. A student who successfully completes *Differential Equations* but does not take *Linear Algebra* meets the requirement. Similarly, a student may take *Linear Algebra* but not *Differential Equations*. It is important to note the a student may elect to take both *Differential Equations* and *Linear Algebra* (perhaps on the way to a mathematics minor), but the requirement is no less fulfilled.

Logical **OR** works in a similar fashion. Given our Boolean expressions e_1 and e_2, the compound expression e_1 || e_2 is false only if e_1 and e_2 are both false; if either one is true or both are true, the compound expression is true. Note that logical **OR** is an *inclusive or*, not an *exclusive or*. In informal conversion we often imply *exclusive or* in a statement like "Would you like cake **or** ice cream for dessert?" The implication is one or the other, not both. In computer programming the *or* is inclusive; if both subexpressions in an *or* expression are true, the *or* expression is true.

Logical **NOT** simply reverses the truth value of the expression to which it is applied. If e is a true Boolean expression, !e is false; if e is false, !e is true.

Table 5.3 is called a *truth table*. It shows all the combinations of truth values for two simple expressions and the values of compound Boolean expressions built from applying the &&, ||, and ! C++ logical operators.

Both && and || are binary operators; that is, they require two operands, both of which must be Boolean

expressions. Logical *not* (`!`) is a unary operator (see Section 4.1); it requires a single Boolean operand immediately to its right.

Operator `!` has higher precedence than both `&&` and `||`. `&&` has higher precedence than `||`. `&&` and `||` are left associative; `!` is right associative. `&&` and `||` have lower precedence than any other binary operator except assignment. This means the expression

```
x <= y && x <= z
```

is evaluated

```
(x <= y) && (x <= z)
```

Some programmers prefer to use the parentheses as shown here even though they are not required. The parentheses improve the readability of complex expressions, and the compiled code is no less efficient.

The relational operators such as `<` compare two operands. The result of the comparison is a Boolean value, which is freely convertible to an integer. The misapplication of relational operators can lead to surprising results; consider, for example, the expression

```
1 <= x <= 10
```

This expression is always true, regardless of the value of `x`! If the programmer's intent is to represent the mathematical notion of `x` falling within the range 1...10 inclusive, as in $1 \leq x \leq 10$, the above C++ expression is not equivalent.

The expression

```
1 <= x <= 10
```

is evaluated as

```
(1 <= x) <= 10
```

If `x` is greater than or equal to one, the subexpression `1` $\sim$ `<=` $\sim$ `x` evaluates to true, or integer 1. Integer 1, however, is always less than 10, so the overall expression is true. If instead `x` is less than one, the subexpression `1` $\sim$ `<=` $\sim$ `x` evaluates to false, or integer 0. Integer 0 is always less than 10, so the overall expression is true. The problem is due to the fact that C++ does not strictly distinguish between Boolean and integer values.

A correct way to represent the mathematical notion of $1 \leq x \leq 10$ is

```
1 <= x && x <= 10
```

In this case `x` must simultaneously be greater than or equal to 1 **and** less than or equal to 10. The revised Boolean expression is a little more verbose than the mathematical representation, but it is the correct formulation for C++.

The following section of code assigns the indicated values to a `bool`:

```
bool b;
int x = 10;
int y = 20;
```

```
b = (x == 10);  // assigns true to b
b = (x != 10);  // assigns false to b
b = (x == 10 && y == 20);  // assigns true to b
b = (x != 10 && y == 20);  // assigns false to b
b = (x == 10 && y != 20);  // assigns false to b
b = (x != 10 && y != 20);  // assigns false to b
b = (x == 10 || y == 20);  // assigns true to b
b = (x != 10 || y == 20);  // assigns true to b
b = (x == 10 || y != 20);  // assigns true to b
b = (x != 10 || y != 20);  // assigns false to b
```

Convince yourself that the following expressions are equivalent:

```
(x != y)
!(x == y)
(x < y || x > y)
```

In the expression e_1 && e_2 both subexpressions e_1 and e_2 must be true for the overall expression to be true. Since the && operator evaluates left to right, this means that if e_1 is false, there is no need to evaluate e_2. If e_1 is false, no value of e_2 can make the expression e_1 && e_2 true. The logical *and* operator first tests the expression to its left. If it finds the expression to be false, it does not bother to check the right expression. This approach is called *short-circuit evaluation*. In a similar fashion, in the expression e_1 || e_2, if e_1 is true, then it does not matter what value e_2 has—a logical *or* expression is true unless both subexpressions are false. The || operator uses short-circuit evaluation also.

Why is short-circuit evaluation important? Two situations show why it is important to consider:

- The order of the subexpressions can affect performance. When a program is running, complex expressions require more time for the computer to evaluate than simpler expressions. We classify an expression that takes a relatively long time to evaluate as an *expensive* expression. If a compound Boolean expression is made up of an expensive Boolean subexpression and an less expensive Boolean subexpression, and the order of evaluation of the two expressions does not affect the behavior of the program, then place the more expensive Boolean expression second. If the first subexpression is false and && is being used, then the expensive second subexpression is not evaluated; if the first subexpression is true and || is being used, then, again, the expensive second subexpression is avoided.

- Subexpressions can be ordered to prevent run-time errors. This is especially true when one of the subexpressions depends on the other in some way. Consider the following expression:

  ```
  (x != 0) && (z/x > 1)
  ```

  Here, if x is zero, the division by zero is avoided. If the subexpressions were switched, a run-time error would result if x is zero.

Arity	Operators	Associativity
unary	(post) ++, (post) --, static_cast	
unary	(pre) ++, (pre) --, !, +, -	
binary	*, /, %	left
binary	+, -	left
binary	<<, >>	left
binary	>, <, >=, <=	left
binary	==, !=	left
binary	&&	left
binary	||	left
binary	=, +=, -=, *=, /=, %=	right

Table 5.4: Precedence of C++ Operators (High to Low)

Suppose you wish to print the word "OK" if a variable x is 1, 2, or 3. An informal translation from English might yield:

```
if (x == 1 || 2 || 3)
    std::cout << "OK\n";
```

Unfortunately, x's value is irrelevant; the code always prints the word "OK." Since the == operator has lower precedence than ||, the expression

```
x == 1 || 2 || 3
```

is interpreted as

```
(x == 1) || 2 || 3
```

The expression x == 1 is either true or false, but integer 2 is always interpreted as true, and integer 3 is interpreted as true is as well.

The correct statement would be

```
if (x == 1 || x == 2 || x == 3)
    std::cout << "OK\n";
```

The revised Boolean expression is more verbose and less similar to the English rendition, but it is the correct formulation for C++.

Our current list of C++ operators is shown in Table 5.4.

5.7 Nested Conditionals

The statements in the body of the if or the else may be any C++ statements, including other if/else statements. We can use nested if statements to build arbitrarily complex control flow logic. Consider Listing 5.6 (checkrange.cpp) that determines if a number is between 0 and 10, inclusive.

Listing 5.6: checkrange.cpp

```
#include <iostream>
```

```
int main() {
    int value;
    std::cout << "Please enter an integer value in the range 0...10: ";
    std::cin >> value;
    if (value >= 0)   // First check
        if (value <= 10)   // Second check
            std::cout << "In range";
    std::cout << "Done\n";
}
```

Listing 5.6 (checkrange.cpp) behaves as follows:

- The program checks the `value >= 0` condition first. If `value` is less than zero, the executing program does not evaluate the second condition and does not print *In range*, but it immediately executes the print statement following the outer `if` statement which prints *Done*.
- If the executing program finds `value` to be greater than or equal to zero, it checks the second condition. If the second condition is met, it displays the *In range* message; otherwise, it is not. Regardless, the program prints *Done* before it terminates.

For the program to display the message *In range* both conditions of this nested `if` must be met. Said another way, the first condition *and* the second condition must be met for the *In range* message to be printed. From this perspective, we can rewrite the program to behave the same way with only *one* `if` statement, as Listing 5.7 (newcheckrange.cpp) shows.

Listing 5.7: `newcheckrange.cpp`

```
#include <iostream>

int main() {
    int value;
    std::cout << "Please enter an integer value in the range 0...10: ";
    std::cin >> value;
    if (value >= 0 && value <= 10)
        std::cout << "In range\n";
}
```

Listing 5.7 (newcheckrange.cpp) uses a logical && to check both conditions at the same time. Its logic is simpler, using only one `if` statement, at the expense of a slightly more complex Boolean expression in its condition. The second version is preferable here because simpler logic is usually a desirable goal.

Sometimes a program's logic cannot be simplified as in Listing 5.7 (newcheckrange.cpp). In Listing 5.8 (enhancedcheckrange.cpp) one `if` statement alone is insufficient to implement the necessary behavior.

Listing 5.8: `enhancedcheckrange.cpp`

```
#include <iostream>

int main() {
    int value;
    std::cout << "Please enter an integer value in the range 0...10: ";
    std::cin >> value;
    if (value >= 0)  // First check
        if (value <= 10)  // Second check
```

```
            std::cout << value << " is acceptable\n";
        else
            std::cout << value << " is too large\n";
    else
        std::cout << value << " is too small\n";
}
```

Listing 5.8 (enhancedcheckrange.cpp) provides a more specific message instead of a simple notification of acceptance. The program prints exactly one of three messages based on the value of the variable. A single `if` or `if/else` statement cannot choose from among more than two different execution paths.

Listing 5.9 (binaryconversion.cpp) uses a series of `if` statements to print a 10-bit binary string representing the binary equivalent of a decimal integer supplied by the user. (Section 4.8 provides some background information about the binary number system.) We use `if/else` statements to print the individual digits left to right, essentially assembling the sequence of bits that represents the binary number.

Listing 5.9: `binaryconversion.cpp`

```
#include <iostream>

int main() {
    int value;
    // Get number from the user
    std::cout << "Please enter an integer value in the range 0...1023: ";
    std::cin >> value;
    // Integer must be less than 1024
    if (0 <= value &&  value < 1024)  {
        if (value >= 512)  {
            std::cout << 1;
            value %= 512;
        }
        else
            std::cout << 0;
        if (value >= 256)  {
            std::cout << 1;
            value %= 256;
        }
        else
            std::cout << 0;
        if (value >= 128)  {
            std::cout << 1;
            value %= 128;
        }
        else
            std::cout << 0;
        if (value >= 64)  {
            std::cout << 1;
            value %= 64;
        }
        else
            std::cout << 0;
        if (value >= 32)  {
            std::cout << 1;
            value %= 32;
        } else
```

```
            std::cout << 0;
        if (value >= 16)  {
            std::cout << 1;
            value %= 16;
        }
        else
            std::cout << 0;
        if (value >= 8)  {
            std::cout << 1;
            value %= 8;
        }
        else
            std::cout << 0;
        if (value >= 4)  {
            std::cout << 1;
            value %= 4;
        }
        else
            std::cout << 0;
        if (value >= 2)  {
            std::cout << 1;
            value %= 2;
        }
        else
            std::cout << 0;
        std::cout << value << '\n';
    }
}
```

In Listing 5.9 (binaryconversion.cpp):

- The outer `if` checks to see if the value the use provides is in the proper range. The program works only for nonnegative integer values less than 1,024, so the range is 0-1023.
- Each inner `if` compares the user-supplied entered integer against decreasing powers of two. If the number is large enough, the program:
  - prints the digit 1 to the console, and
  - removes via the remainder operator that power of two's contribution to the value.

  If the number is not at least as big as the given power of two, the program prints a 0 instead and moves on without modifying the input value.
- For the ones place at the end no check is necessary—the remaining value will be 0 or 1 and so the program prints whatever remains.

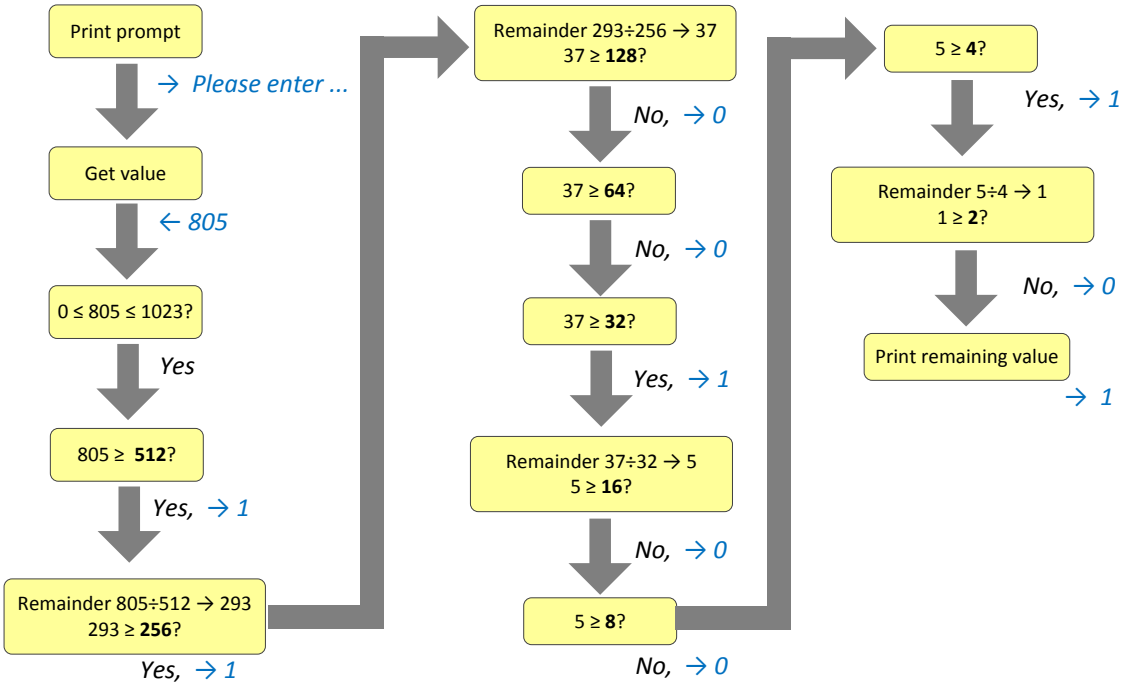
The following shows a sample run of Listing 5.9 (binaryconversion.cpp):

```
Please enter an integer value in the range 0...1023: 805
1100100101
```

Figure 5.3 illustrates the execution of Listing 5.9 (binaryconversion.cpp) when the user enters 805.

Listing 5.10 (simplerbinaryconversion.cpp) simplifies the logic of Listing 5.9 (binaryconversion.cpp) at the expense of some additional arithmetic. It uses only one `if` statement.

Figure 5.3 The process of the binary number conversion program when the user supplies 805 as the input value.



Listing 5.10: simplerbinaryconversion.cpp

```
#include <iostream>

int main() {
    int value;
    // Get number from the user
    std::cout << "Please enter an integer value in the range 0...1023: ";
    std::cin >> value;
    // Integer must be less than 1024
    if (0 <= value &&  value < 1024)  {
        std::cout << value/512;
        value %= 512;
        std::cout << value/256;
        value %= 256;
        std::cout << value/128;
        value %= 128;
        std::cout << value/64;
        value %= 64;
        std::cout << value/32;
        value %= 32;
        std::cout << value/16;
        value %= 16;
        std::cout << value/8;
        value %= 8;
        std::cout << value/4;
        value %= 4;
        std::cout << value/2;
        value %= 2;
        std::cout << value << '\n';
    }
}
```

The sole `if` statement in Listing 5.10 (simplerbinaryconversion.cpp) ensures that the user provides an integer in the proper range. The other `if` statements that originally appeared in Listing 5.9 (binaryconversion.cpp) are gone. A clever sequence of integer arithmetic operations replace the original conditional logic. The two programs—binaryconversion.cpp and simplerbinaryconversion.cpp—behave identically but simplerbinaryconversion.cpp's logic is simpler.

Listing 5.11 (troubleshoot.cpp) implements a very simple troubleshooting program that (an equally simple) computer technician might use to diagnose an ailing computer.

Listing 5.11: troubleshoot.cpp

```
#include <iostream>

int main() {
    std::cout << "Help!  My computer doesn't work!\n";
    char choice;
    std::cout << "Does the computer make any sounds "
              << "(fans, etc.) or show any lights? (y/n):";
    std::cin >> choice;
    // The troubleshooting control logic
    if (choice == 'n') { // The computer does not have power
        std::cout << "Is it plugged in? (y/n):";
        std::cin >> choice;
```

```
        if (choice == 'n') { // It is not plugged in, plug it in
            std::cout << "Plug it in.  If the problem persists, "
                      << "please run this program again.\n";
        }
        else { // It is plugged in
            std::cout << "Is the switch in the \"on\" position? (y/n):";
            std::cin >> choice;
            if (choice == 'n') { // The switch is off, turn it on!
                std::cout << "Turn it on.  If the problem persists, "
                          << "please run this program again.\n";
            }
            else { // The switch is on
                std::cout << "Does the computer have a fuse?  (y/n):";
                std::cin >> choice;
                if (choice == 'n') { // No fuse
                    std::cout << "Is the outlet OK? (y/n):";
                    std::cin >> choice;
                    if (choice == 'n') { // Fix outlet
                        std::cout << "Check the outlet's circuit "
                                  << "breaker or fuse.  Move to a "
                                  << "new outlet, if necessary. "
                                  << "If the problem persists, "
                                  << "please run this program again.\n";
                    }
                    else { // Beats me!
                        std::cout << "Please consult a service "
                                  << "technician.\n";
                    }
                }
                else { // Check fuse
                    std::cout << "Check the fuse. Replace if "
                              << "necessary.  If the problem "
                              << "persists, then "
                              << "please run this program again.\n";
                }
            }
        }
    }
    else { // The computer has power
        std::cout << "Please consult a service technician.\n";
    }
}
```

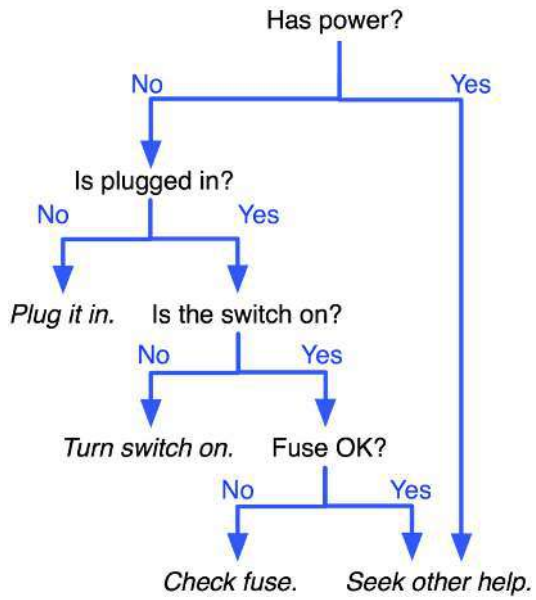
This very simple troubleshooting program attempts to diagnose why a computer does not work. The potential for enhancement is unlimited, but this version only deals with power issues that have simple fixes. Notice that if the computer has power (fan or disk drive makes sounds or lights are visible), the program directs the user to seek help elsewhere! The decision tree capturing the basic logic of the program is shown in Figure 5.4.

The steps performed are:

1. Is it plugged in? This simple fix is sometimes overlooked.

2. Is the switch in the *on* position? This is another simple fix.

Figure 5.4 Decision tree for troubleshooting a computer system



3. If applicable, is the fuse blown? Some computer systems have a user-serviceable fuse that can blow out during a power surge. (Most newer computers have power supplies that can handle power surges and have no user-serviceable fuses.)

4. Is there power at the receptacle? Perhaps the outlet's circuit breaker or fuse has a problem.

The program directs the user to make the easier checks first. It progressively introduces more difficult checks as it continues. Based on your experience with troubleshooting computers that do not run properly, you may be able to think of many enhancements to this simple program.

Note that in Listing 5.11 (troubleshoot.cpp) curly braces are used in many places where they strictly are not necessary. Their inclusion in Listing 5.11 (troubleshoot.cpp) improves the readability of the program and makes the logic easier to understand. Even if you do not subscribe to the philosophy of using curly braces for every `if/else` body, it is a good idea to use them in situations that improve the code's readability.

Recall the time conversion program in Listing 4.11 (timeconv.cpp). If the user enters `10000`, the program runs as follows:

```
Please enter the number of seconds:10000
2 hr 46 min 40 sec
```

and if the user enters `9961`, the program prints:

```
Please enter the number of seconds:9961
2 hr 46 min 1 sec
```

Suppose we wish to improve the English presentation by not using abbreviations. If we spell out *hours*, *minutes*, and *seconds*, we must be careful to use the singular form *hour*, *minute*, or *second* when the corresponding value is one. Listing 5.12 (timeconvcond1.cpp) uses `if/else` statements to express to time units with the correct number.

Listing 5.12: `timeconvcond1.cpp`

```
// File timeconvcond1.cpp

#include <iostream>

int main() {
    // Some useful conversion constants
    const int SECONDS_PER_MINUTE = 60,
              SECONDS_PER_HOUR   = 60*SECONDS_PER_MINUTE;  // 3600
    int hours, minutes, seconds;
    std::cout << "Please enter the number of seconds:";
    std::cin >> seconds;
    // First, compute the number of hours in the given number
    // of seconds
    hours = seconds / SECONDS_PER_HOUR;  // 3600 seconds = 1 hours
    // Compute the remaining seconds after the hours are
    // accounted for
    seconds = seconds % SECONDS_PER_HOUR;
    // Next, compute the number of minutes in the remaining
    // number of seconds
    minutes = seconds / SECONDS_PER_MINUTE;  // 60 seconds = 1 minute
    // Compute the remaining seconds after the minutes are
    // accounted for
    seconds = seconds % SECONDS_PER_MINUTE;
    // Report the results
    std::cout << hours;
    // Decide between singular and plural form of hours
    if (hours == 1)
        std::cout << " hour ";
    else
       std::cout << " hours ";
    std::cout << minutes;
    // Decide between singular and plural form of minutes
    if (minutes == 1)
        std::cout << " minute ";
    else
        std::cout << " minutes ";
    std::cout << seconds;
    // Decide between singular and plural form of seconds
    if (seconds == 1)
        std::cout << " second";
    else
        std::cout << " seconds";
    std::cout << '\n';
}
```

The `if/else` statements within Listing 5.12 (timeconvcond1.cpp) are responsible for printing the correct version—singular or plural—for each time unit. One run of Listing 5.12 (timeconvcond1.cpp) produces

```
Please enter the number of seconds:10000
2 hours 46 minutes 40 seconds
```

All the words are plural since all the value are greater than one. Another run produces

```
Please enter the number of seconds:9961
2 hours 46 minutes 1 second
```

Note the word *second* is singular as it should be.

```
Please enter the number of seconds:3601
1 hour 0 minutes 1 second
```

Here again the printed words agree with the number of the value they represent.

An improvement to Listing 5.12 (timeconvcond1.cpp) would not print a value and its associated time unit if the value is zero. Listing 5.13 (timeconvcond2.cpp) adds this feature.

Listing 5.13: `timeconvcond2.cpp`

```
// File timeconvcond1.cpp

#include <iostream>

int main() {
    // Some useful conversion constants
    const int SECONDS_PER_MINUTE = 60,
              SECONDS_PER_HOUR   = 60*SECONDS_PER_MINUTE;  // 3600
    int hours, minutes, seconds;
    std::cout << "Please enter the number of seconds:";
    std::cin >> seconds;
    // First, compute the number of hours in the given number
    // of seconds
    hours = seconds / SECONDS_PER_HOUR;  // 3600 seconds = 1 hours
    // Compute the remaining seconds after the hours are
    // accounted for
    seconds = seconds % SECONDS_PER_HOUR;
    // Next, compute the number of minutes in the remaining
    // number of seconds
    minutes = seconds / SECONDS_PER_MINUTE;  // 60 seconds = 1 minute
    // Compute the remaining seconds after the minutes are
    // accounted for
    seconds = seconds % SECONDS_PER_MINUTE;
    // Report the results
    if (hours > 0) {    // Print hours at all?
        std::cout << hours;
        // Decide between singular and plural form of hours
        if (hours == 1)
           std::cout << " hour ";
        else
           std::cout << " hours ";
    }
    if (minutes > 0) {   // Print minutes at all?
        std::cout << minutes;
        // Decide between singular and plural form of minutes
```

```
        if (minutes == 1)
            std::cout << " minute ";
        else
            std::cout << " minutes ";
    }
    // Print seconds at all?
    if (seconds > 0 || (hours == 0 && minutes == 0 && seconds == 0)) {
        std::cout << seconds;
        // Decide between singular and plural form of seconds
        if (seconds == 1)
            std::cout << " second";
        else
            std::cout << " seconds";
    }
    std::cout << '\n';
}
```

In Listing 5.13 (timeconvcond2.cpp) each code segment responsible for printing a time value and its English word unit is protected by an `if` statement that only allows the code to execute if the time value is greater than zero. The exception is in the processing of seconds: if all time values are zero, the program should print *0 seconds*. Note that each of the `if`/`else` statements responsible for determining the singular or plural form is nested within the `if` statement that determines whether or not the value will be printed at all.

One run of Listing 5.13 (timeconvcond2.cpp) produces

```
Please enter the number of seconds:10000
2 hours 46 minutes 40 seconds
```

All the words are plural since all the value are greater than one. Another run produces

```
Please enter the number of seconds:9961
2 hours 46 minutes 1 second
```

Note the word *second* is singular as it should be.

```
Please enter the number of seconds:3601
1 hour 1 second
```

Here again the printed words agree with the number of the value they represent.

```
Please enter the number of seconds:7200
2 hours
```

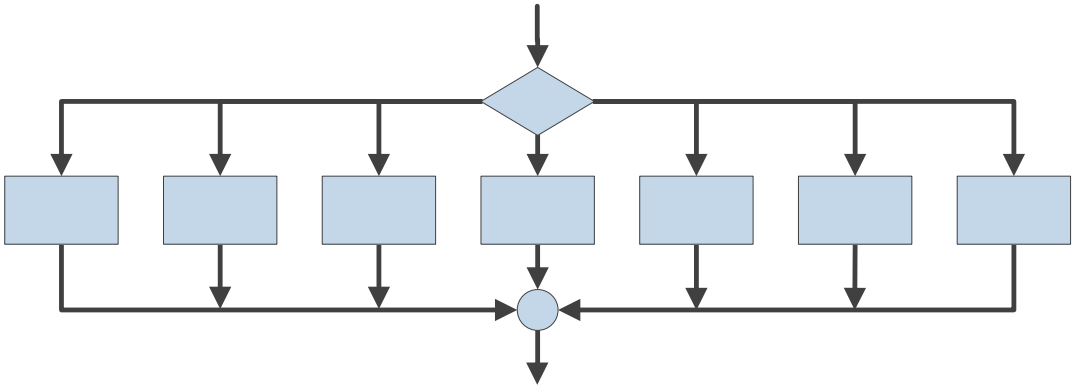
Another run produces:

```
Please enter the number of seconds:60
1 minute
```

Finally, the following run shows that the program handles zero seconds properly:

```
Please enter the number of seconds:0
0 seconds
```

Figure 5.5 Flowchart with multiple optional execution pathways



5.8 Multi-way if/else Statements

A simple `if/else` statement can select from between two execution paths. Suppose we wish to choose one execution path from among several possible paths, as shown in Figure 5.5?

Listing 5.8 (enhancedcheckrange.cpp) showed how to select from among three options. What if exactly one of many actions should be taken? Nested `if/else` statements are required, and the form of these nested `if/else` statements is shown in Listing 5.14 (digittoword.cpp).

Listing 5.14: `digittoword.cpp`

```
#include <iostream>

int main() {
    int value;
    std::cout << "Please enter an integer in the range 0...5: ";
    std::cin >> value;
    if (value < 0)
        std::cout << "Too small";
    else
        if (value == 0)
            std::cout << "zero";
        else
            if (value == 1)
                std::cout << "one";
             else
                if (value == 2)
                    std::cout << "two";
                else
                    if (value == 3)
                        std::cout << "three";
                    else
```

```
                        if (value == 4)
                            std::cout << "four";
                        else
                            if (value == 5)
                                std::cout << "five";
                            else
                                std::cout << "Too large";
    std::cout << '\n';
}
```

Observe the following about Listing 5.14 (digittoword.cpp):

- It prints exactly one of eight messages depending on the user's input.
- Notice that each `if` body contains a single printing statement and each `else` body, except the last one, contains an `if` statement. The control logic forces the program execution to check each condition in turn. The first condition that matches wins, and its corresponding `if` body will be executed. If none of the conditions are true, the last `else`'s *Too large* message will be printed.
- No curly braces are necessary to delimit the `if` or `else` bodies since each body contains only a single statement (although a single deeply nested `if`/`else` statement is a mighty big statement).

Listing 5.14 (digittoword.cpp) is formatted according to the conventions used in earlier examples. As a consequence, the mass of text drifts to the right as more conditions are checked. A commonly used alternative style, shown in Listing 5.15 (restyleddigittoword.cpp), avoids this rightward drift.

Listing 5.15: `restyleddigittoword.cpp`

```
#include <iostream>

int main() {
    int value;
    std::cout << "Please enter an integer in the range 0...5: ";
    std::cin >> value;
    if (value < 0)
        std::cout << "Too small";
    else if (value == 0)
        std::cout << "zero";
    else if (value == 1)
        std::cout << "one";
    else if (value == 2)
        std::cout << "two";
    else if (value == 3)
        std::cout << "three";
    else if (value == 4)
        std::cout << "four";
    else if (value == 5)
        std::cout << "five";
    else
        std::cout << "Too large";
    std::cout << '\n';
}
```

Based on our experience so far, the formatting of Listing 5.15 (restyleddigittoword.cpp) somewhat hides the true structure of the program's logic, but this style of formatting multi-way `if`/`else` statements is

so common that it is regarded as acceptable by most programmers. The sequence of `else if` lines all indented to the same level identifies this construct as a multi-way `if/else` statement.

Listing 5.16 (datetransformer.cpp) uses a multi-way `if/else` to transform a numeric date in month/-day format to an expanded US English form and an international Spanish form; for example, `2/14` would be converted to `February 14` and `14 febrero`.

Listing 5.16: `datetransformer.cpp`

```
#include <iostream>

int main() {
    std::cout << "Please enter the month and day as numbers: ";
    int month, day;
    std::cin >> month >> day;
    // Translate month into English
    if (month == 1)
        std::cout << "January";
    else if (month == 2)
        std::cout << "February";
    else if (month == 3)
        std::cout << "March";
    else if (month == 4)
        std::cout << "April";
    else if (month == 5)
        std::cout << "May";
    else if (month == 6)
        std::cout << "June";
    else if (month == 7)
        std::cout << "July";
    else if (month == 8)
        std::cout << "August";
    else if (month == 9)
        std::cout << "September";
    else if (month == 10)
        std::cout << "October";
    else if (month == 11)
        std::cout << "November";
    else
        std::cout << "December";
    // Add the day
    std::cout << " " << day << " or " << day << " de ";
    // Translate month into Spanish
    if (month == 1)
        std::cout << "enero";
    else if (month == 2)
        std::cout << "febrero";
    else if (month == 3)
        std::cout << "marzo";
    else if (month == 4)
        std::cout << "abril";
    else if (month == 5)
        std::cout << "mayo";
    else if (month == 6)
        std::cout << "junio";
    else if (month == 7)
```

```
        std::cout << "julio";
    else if (month == 8)
        std::cout << "agosto";
    else if (month == 9)
        std::cout << "septiembre";
    else if (month == 10)
        std::cout << "octubre";
    else if (month == 11)
        std::cout << "noviembre";
    else
        std::cout << "diciembre";
    std::cout << '\n';
}
```

A sample run of Listing 5.16 (datetransformer.cpp) is shown here:

```
Please enter the month and day as numbers: 5 20
May 20 or 20 de mayo
```

Figure 5.6 compares the structure of the `if/else` statements in a program such as Listing 5.15 (restyleddigittoword.cpp) to those in a program like Listing 5.9 (binaryconversion.cpp).

In a program like Listing 5.15 (restyleddigittoword.cpp), the `if/else` statements are nested, while in a program like Listing 5.9 (binaryconversion.cpp) the `if/else` statements are sequential.

C++ provides the tools to construct some very complicated conditional statements. It is important to resist the urge to make things overly complex. Consider the problem of computing the maximum of five integer values provided by the user. The complete solution is left as an exercise in Section 5.10, but here we will outline an appropriate strategy.

Suppose you allow the user to enter all the values at once; for example, for integer variables `n1`, `n2`, `n3`, `n4`, and `n5`:

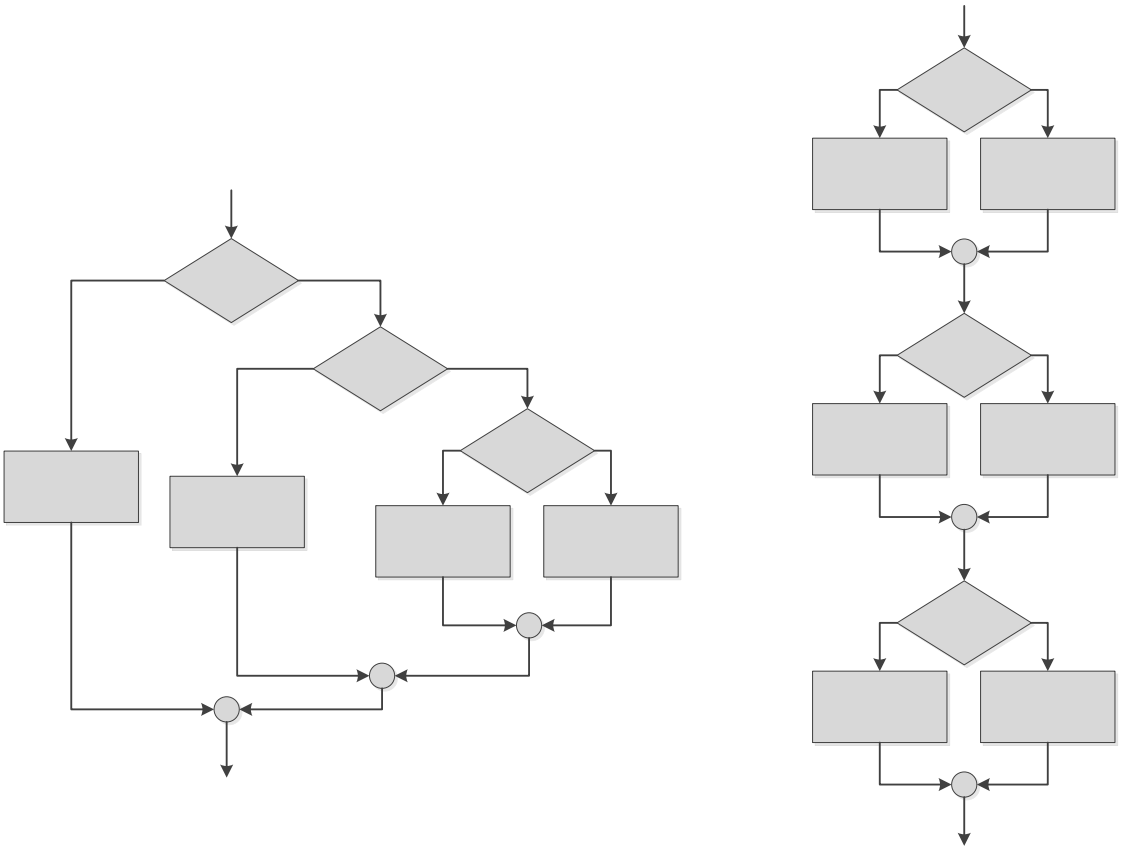
```
std::cout << "Please enter five integer values: ";
std::cin >> n1 >> n2 >> n3 >> n4 >> n5;
```

Now, allow yourself one extra variable called `max`. All variables have a meaning, and their names should reflect their meaning in some way. We'll let our additional `max` variable mean "maximum I have determined so far." The following is one approach to the solution:

1. Set `max` equal to `n1`. This means as far as we know at the moment, `n1` is the biggest number because `max` and `n1` have the same value.
2. Compare `max` to `n2`. If `n2` is larger than `max`, change `max` to have `n2`'s value to reflect the fact that we determined `n2` is larger; if `n2` is not larger than `max`, we have no reason to change `max`, so do not change it.
3. Compare `max` to `n3`. If `n3` is larger than `max`, change `max` to have `n3`'s value to reflect the fact that we determined `n3` is larger; if `n3` is not larger than `max`, we have no reason to change `max`, so do not change it.
4. Follow the same process for `n4` and `n5`.

In the end the meaning of the `max` variable remains the same–"maximum I have determined so far," but, after comparing `max` to all the input variables, we now know that it is *the* maximum value of all five input

Figure 5.6 The structure of the `if` statements in a program such as Listing 5.15 (restyleddigittoword.cpp) (left) vs. those in a program like Listing 5.9 (binaryconversion.cpp) (right)



numbers. The extra variable `max` is not strictly necessary, but it makes thinking about the problem and its solution easier.

Something to think about: Do you want a series of `if` statements or one large multiway `if/else` construct?

Also, you may be tempted to write logic such as

```
if (n1 >= n2 && n1 >= n3 && n1 >= n4 && n1 >=n5)
    std::cout << "The maximum is " << n1 << '\n';
else if ( n2 >= n1 && n2 >= n3 &&   // the rest omitted . . .
```

This will work, but this logic is much more complicated and less efficient (every >= and && operation requires a few machine cycles to execute). Since it is more complicated, it is more difficult to write correctly, in addition to being more code to type in. It is easy to use > by mistake instead of >=, which will not produce the correct results. Also, if you use this more complicated logic and decide later to add more variables, you will need to change *all* of the `if` conditions in your code and, of course, make sure to modify each one of the conditions correctly. If you implement the simpler strategy outlined before, you need only add one simple `if` statement for each additional variable.

Chapter 6 introduces loops, the ability to execute statements repeatedly. You easily can adapt the first approach to allow the user to type in as many numbers as they like and then have the program report the maximum number the user entered. The second approach with the more complex logic cannot be adapted in this manner. With the first approach you end up with cleaner, simpler logic, a more efficient program, and code that is easier to extend.

5.9 Errors in Conditional Statements

Consider Listing 5.17 (badequality.cpp).

Listing 5.17: badequality.cpp

```
#include <iostream>

int main() {
    int input;
    std::cout << "Please enter an integer:";
    std::cin >> input;
    if (input = 2)
        std::cout << "two\n";
   std::cout << "You entered " << input << '\n';
}
```

Listing 5.17 (badequality.cpp) demonstrates a common mistake—using the assignment operator where the equality operator is intended. This program, when run, always prints the message "two" and insists the user entered 2 regardless of the actual input. Recall from Section 4.3 that the assignment expression has a value. The value of an assignment expression is same as the value that is assigned; thus, the expression

```
input = 2
```

has the value 2. When you consider also that every integer can be treated as a Boolean value (see Section 5.1) and any non-zero value is interpreted as `true`, you can see that the condition of `if` statement

```
if (input = 2)
    std::cout << "two\n";
```

is always true. Additionally, the variable `input` is always assigned the value 2.

Since it is such a common coding error, most C++ compilers can check for such misuse of assignment. At warning Level 4, for example, Visual C++ will issue a warning when assignment appears where a conditional expression is expected:

warning C4706: assignment within conditional expression

Occasionally the use of assignment within a conditional expression is warranted, so the compiler does not perform this check by default. For our purposes it is good idea to direct the compiler to perform this extra check.

Carefully consider each compound conditional used, such as

```
value > 0  &&  value <= 10
```

found in Listing 5.7 (newcheckrange.cpp). Confusing logical *and* and logical *or* is a common programming error. If you substitute `||` for &&, the expression

```
x > 0  ||  x <= 10
```

is always true, no matter what value is assigned to the variable `x`. A Boolean expression that is always true is known as a *tautology*. Think about it. If `x` is an `int`, what value could the variable `x` assume that would make

```
x > 0  ||  x <= 10
```

false? Regardless of its value, one or both of the subexpressions will be true, so this compound logical *or* expression is always true. This particular *or* expression is just a complicated way of expressing the value true.

Another common error is contriving compound Boolean expressions that are always false, known as *contradictions*. Suppose you wish to *exclude* values from a given range; for example, reject values in the range 0...10 and accept all other numbers. Is the Boolean expression in the following code fragment up to the task?

```
// I want to use all but 0, 1, 2, ..., 10
if (value < 0  &&  value > 10)
    /*  Code to execute goes here . . . */
```

A closer look at the condition reveals it can *never* be true. What number can be both less than zero and greater than ten *at the same time*? None can, of course, so the expression is a contradiction and a complicated way of expressing *false*. To correct this code fragment, replace the && operator with `||`.

5.10 Exercises

1. What values can a variable of type `bool` assume?

2. Where does the term `bool` originate?

3. What is the integer equivalent to `true` in C++?
4. What is the integer equivalent to `false` in C++?
5. Is the value `-16` interpreted as true or false?
6. May an integer value be assigned to a `bool` variable?
7. Can `true` be assigned to an `int` variable?
8. Given the following declarations:

```
int x = 3, y = 5, z = 7;
bool b1 = true, b2 = false, b3 = x == 3, b4 = y < 3;
```

   evaluate the following Boolean expressions:

   (a) `x == 3`
   (b) `x < y`
   (c) `x >= y`
   (d) `x <= y`
   (e) `x != y - 2`
   (f) `x < 10`
   (g) `x >= 0 && x < 10`
   (h) `x < 0 && x < 10`
   (i) `x >= 0 && x < 2`
   (j) `x < 0 || x < 10`
   (k) `x > 0 || x < 10`
   (l) `x < 0 || x > 10`
   (m) `b1`
   (n) `!b1`
   (o) `!b2`
   (p) `b1 && b2`

9. Express the following Boolean expressions in simpler form; that is, use fewer operators. `x` is an `int`.

   (a) `!(x == 2)`
   (b) `x < 2 || x == 2`
   (c) `!(x < y)`
   (d) `!(x <= y)`
   (e) `x < 10 && x > 20`
   (f) `x > 10 || x < 20`
   (g) `x != 0`
   (h) `x == 0`

10. What is the simplest tautology?
11. What is the simplest contradiction?

12. Write a C++ program that requests an integer value from the user. If the value is between 1 and 100 inclusive, print "OK;" otherwise, do not print anything.

13. Write a C++ program that requests an integer value from the user. If the value is between 1 and 100 inclusive, print "OK;" otherwise, print "Out of range."

14. The following program attempts to print a message containing the English word corresponding to a given integer input. For example, if the user enters the value 3, the program should print `"You entered a three"`. In its current state, the program contains logic errors. Locate the problems and repair them so the program will work as expected.

```
#include <iostream>

int main() {
    std::cout << "Please in value in the range 1...5: ";
    int value;
    std::cin >> value;
    // Translate number into its English word
    if (month == 1)
        std::cout << "You entered a";
        std::cout << "one";
        std::cout << '\n';
    else if (month == 2)
        std::cout << "You entered a";
        std::cout << "two";
        std::cout << '\n';
    else if (month == 3)
        std::cout << "You entered a";
        std::cout << "three";
        std::cout << '\n';
    else if (month == 4)
        std::cout << "You entered a";
        std::cout << "four";
        std::cout << '\n';
    else if (month == 5)
        std::cout << "You entered a";
        std::cout << "five";
        std::cout << '\n';
    else   // Value out of range
        std::cout << "You entered a";
        std::cout << "value out of range";
        std::cout << '\n';
}
```

15. Consider the following section of C++ code:

```
// i, j, and k are ints
if (i < j) {
    if (j < k)
        i = j;
    else
        j = k;
```

```
}
else {
    if (j > k)
        j = i;
    else
        i = k;
}
std::cout << "i = " << i << " j = " << j << " k = " << k << '\n';
```

What will the code print if the variables `i`, `j`, and `k` have the following values?

(a) `i` is 3, `j` is 5, and `k` is 7

(b) `i` is 3, `j` is 7, and `k` is 5

(c) `i` is 5, `j` is 3, and `k` is 7

(d) `i` is 5, `j` is 7, and `k` is 3

(e) `i` is 7, `j` is 3, and `k` is 5

(f) `i` is 7, `j` is 5, and `k` is 3

16. Consider the following C++ program that prints one line of text:

```
#include <iostream>

int main() {
    int input;
    std::cin >> input;
    if (input < 10) {
        if (input != 5)
            std::cout << "wow ";
        else
            input++;
    }
    else {
        if (input == 17)
            input += 10;
        else
            std::cout << "whoa ";
    }
    std::cout << input << '\n';
}
```

What will the program print if the user provides the following input?

(a) 3

(b) 21

(c) 5

(d) 17

(e) -5

17. Why does the following section of code always print `"ByeHi"`?

```
int x;
std::cin >> x;
if (x < 0);
    std::cout << "Bye";
std::cout << "Hi\n";
```

18. Write a C++ program that requests five integer values from the user. It then prints the maximum and minimum values entered. If the user enters the values 3, 2, 5, 0, and 1, the program would indicate that 5 is the maximum and 0 is the minimum. Your program should handle ties properly; for example, if the user enters 2, 4, 2, 3, and 3, the program should report 2 as the minimum and 4 as maximum.

19. Write a C++ program that requests five integer values from the user. It then prints one of two things: if any of the values entered are duplicates, it prints `"DUPLICATES"`; otherwise, it prints `"ALL UNIQUE"`.

Chapter 6

Iteration

Iteration repeats the execution of a sequence of code. Iteration is useful for solving many programming problems. Iteration and conditional execution are key components of algorithm construction.

6.1 The while Statement

Listing 6.1 (counttofive.cpp) counts to five by printing a number on each output line.

Listing 6.1: `counttofive.cpp`

```
#include <iostream>

int main() {
    std::cout << 1 << '\n';
    std::cout << 2 << '\n';
    std::cout << 3 << '\n';
    std::cout << 4 << '\n';
    std::cout << 5 << '\n';
}
```

When compiled and run, this program displays

```
1
2
3
4
5
```

How would you write the code to count to 10,000? Would you copy, paste, and modify 10,000 printing statements? You could, but that would be impractical! Counting is such a common activity, and computers routinely count up to very large values, so there must be a better way. What we really would like to do is print the value of a variable (call it `count`), then increment the variable (`count++`), and repeat this process until the variable is large enough (`count == 5` or perhaps `count == 10000`). This process of executing the same section of code over and over is known as *iteration*, or *looping*, and in C++ we can implement loops in several different ways.

Listing 6.2 (iterativecounttofive.cpp) uses a `while` statement to count to five:

Listing 6.2: `iterativecounttofive.cpp`

```
#include <iostream>

int main() {
    int count = 1;              // Initialize counter
    while (count <= 5) {
        std::cout << count << '\n';  // Display counter, then
        count++;                // Increment counter
    }
}
```

Listing 6.2 (iterativecounttofive.cpp) uses a `while` statement to display a variable that is counting up to five. Unlike the approach taken in Listing 6.1 (counttofive.cpp), it is trivial to modify Listing 6.2 (iterativecounttofive.cpp) to count up to 10,000—just change the literal value 5 to 10000.

The line

```
while (count <= 5)
```

begins the `while` statement. The expression within the parentheses must be a Boolean expression. If the Boolean expression is true when the the program's execution reaches the `while` statement, the program executes the body of the `while` statement and then checks the condition again. The program repeatedly executes the statement(s) within the body of the `while` as long as the Boolean expression remains true.

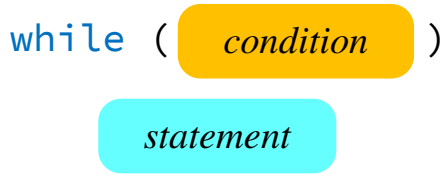
If the Boolean expression is true when the `while` statement is executed, the body of the `while` statement is executed, and the body is executed repeatedly as long as the Boolean expression remains true.

The statements

```
std::cout << count << '\n';
count++;
```

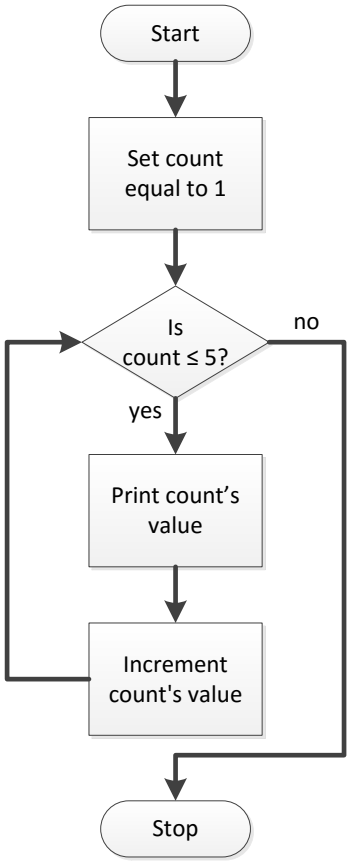
constitute the body of the `while` statement. The curly braces are necessary since more than one statement makes up the body.

The `while` statement has the general form:



- The reserved word `while` begins the `while` statement.
- The Boolean expression *condition* determines whether the body will be (or will continue to be) executed. The expression *must* be enclosed within parentheses as shown.
- The *statement* is the statement to be executed while the Boolean expression is true. The statement makes up the body of the `while` statement. The statement may be a compound statement (multiple statements enclosed within curly braces, see Section 5.4).

Figure 6.1 while flowchart for Listing 6.2 (iterativecounttofive.cpp)



Except for using the reserved word `while` instead of `if`, a `while` statement looks identical to an `if` statement. Sometimes beginning programmers confuse the two or accidentally type `if` when they mean `while` or vice-versa. Usually the very different behavior of the two statements reveals the problem immediately; however, sometimes, especially in nested complex logic, this mistake can be hard to detect.

Figure 6.1 shows how program execution flows through Listing 6.2 (iterativecounttofive.cpp).

The program checks the `while`'s condition before executing the body, and then re-checks the condition each time after it executes the body. If the condition is initially false the program's execution skips the body completely and continues executing the statements that follow the `while`'s body. If the condition is initially true, the program repeatedly executes the body until the condition becomes false, at which point the loop terminates. Program execution then continues with the statements that follow the loop's body, if any. Observe that the body may never be executed if the Boolean expression in the condition is initially false.

Listing 6.3 (countup.cpp) counts up from zero as long as the user wishes to do so.

Listing 6.3: `countup.cpp`

```
/*
 *  Counts up from zero.  The user continues the count by entering
 *  'Y'.  The user discontinues the count by entering 'N'.
 */

#include <iostream>


int main() {
    char input;         // The users choice
    int count = 0;      // The current count
    bool done = false;  // We are not done

    while (!done) {

        // Print the current value of count
        std::cout << count << '\n';
        std::cout << "Please enter \"Y\" to continue or \"N\" to quit: ";
        std::cin >> input;
        // Check for "bad" input
        if (input != 'Y' && input != 'y' && input != 'N' && input != 'n')
            std::cout << "\"" << input << "\""
                      << " is not a valid choice" << '\n';
        else if (input == 'Y' || input == 'y')
            count++;    // Keep counting
        else if (input == 'N' || input == 'n')
            done = true;  // Quit the loop
    }
}
```

A sample run of Listing 6.3 (countup.cpp) produces

```
0
Please enter "Y" to continue or "N" to quit: y
1
Please enter "Y" to continue or "N" to quit: y
2
Please enter "Y" to continue or "N" to quit: y
3
Please enter "Y" to continue or "N" to quit: q
"q" is not a valid choice
3
Please enter "Y" to continue or "N" to quit: r
"r" is not a valid choice
3
Please enter "Y" to continue or "N" to quit: W
"W" is not a valid choice
3
Please enter "Y" to continue or "N" to quit: Y
4
Please enter "Y" to continue or "N" to quit: y
5
Please enter "Y" to continue or "N" to quit: n
```

In Listing 6.3 (countup.cpp) the expression

```
input != 'Y' && input != 'y' && input != 'N' && input != 'n'
```

is true if the character variable `input` is not equal to one of the listed character literals. The Boolean variable `done` controls the loop's execution. It is important to note that the expression

```
!done
```

inside the `while`'s condition evaluates to the opposite truth value of the variable `done`; the expression does *not* affect the value of `done`. In other words, the `!` operator applied to a variable does not modify the variable's value. In order to actually change the variable `done`, you would need to reassign it, as in

```
done = !done;  // Invert the truth value
```

For Listing 6.3 (countup.cpp) we have no need to invert its value. We ensure that its value is `false` initially and then make it `true` when the user enters a capital or lower-case *N*.

Listing 6.4 (addnonnegatives.cpp) is a program that allows a user to enter any number of nonnegative integers. When the user enters a negative value, the program no longer accepts input, and it displays the sum of all the nonnegative values. If a negative number is the first entry, the sum is zero.

Listing 6.4: `addnonnegatives.cpp`

```
/*
 *  Allow the user to enter a sequence of nonnegative
 *  integers.  The user ends the list with a negative
 *  integer.  At the end the sum of the nonnegative
 *  integers entered is displayed.  The program prints
 *  zero if the user enters no nonnegative integers.
 */

#include <iostream>

int main() {
    int input = 0,   // Ensure the loop is entered
        sum = 0;     // Initialize sum

    // Request input from the user
    std::cout << "Enter numbers to sum, negative number ends list:";

    while (input >= 0) {    // A negative number exits the loop
        std::cin >> input;  // Get the value
        if (input >= 0)
            sum += input;   // Only add it if it is nonnegative
    }
    std::cout << "Sum = " << sum << '\n';  // Display the sum
}
```

The initialization of `input` to zero coupled with the condition `input >= 0` of the `while` guarantees that program will execute the body of the `while` loop at least once. The `if` statement ensures that a negative entry will not be added to `sum`. (Could the condition have used `>` instead of `>=` and achieved the same results?) When the user enters a negative integer the program will not update `sum`, and the condition of the while will no longer be true. The program's execution then leaves the loop and executes the print statement at the end.

Listing 6.4 (addnonnegatives.cpp) shows that a `while` loop can be used for more than simple counting. The program does not keep track of the number of values entered. The program simply accumulates the

entered values in the variable named `sum`.

It is a little awkward in Listing 6.4 (addnonnegatives.cpp) that the same condition appears twice, once in the `while` and again in the `if`. Furthermore, what if the user wishes to enter negative values along with nonnegative values? We can simplify the code with a common C++ idiom that uses `std::cin` and the extraction operator as a condition within a `while` statement.

If `x` is an integer, the expression

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

evaluates to `false` if the user does not enter a valid integer literal. Armed with this knowledge we can simplify and enhance Listing 6.4 (addnonnegatives.cpp) as shown in Listing 6.5 (addnumbers.cpp).

Listing 6.5: `addnumbers.cpp`

```
#include <iostream>

int main() {
    int input, sum = 0;
    std::cout << "Enter numbers to sum, type 'q' to end the list:";
    while (std::cin >> input)
        sum += input;
    std::cout << "Sum = " << sum << '\n';
}
```

The condition reads a value from the input stream and, if it is successful, it is interpreted as `true`. When the user enters `'q'`, the loop is terminated. If the user types `'q'` at the beginning, the loop is not entered. The `if` statement is no longer necessary, since the statement

```
sum += input;
```

can be executed only if `input` has been legitimately assigned. Also, the variable `input` no longer needs to initialized with a value simply so the loop is entered the first time; now it is assigned and then checked within the condition of the `while`.

In Listing 6.5 (addnumbers.cpp), the program's execution will terminate with any letter the user types; an entry of `'x'` or Ctrl-Z will terminate the sequence just as well as `'q'`.

If you use the

```
while (std::cin >> x) {
    // Do something . . .
}
```

idiom, be aware that when the loop is exited due to the input stream being unable to read a value of the type of variable `x`, the `std::cin` input stream object is left in an error state and cannot be used for the rest of the program's execution. It you wish to use this technique and reuse `std::cin` later, you must reset `std::cin` and extract and discard keystrokes entered since the last valid use of the extractor operator. This recovery process is covered in Section 13.2, but, for now, use this idiom to control a loop only if the program does not require additional user input later during its execution.

Listing 6.6 (powersof10.cpp) prints the powers of 10 from 1 to 1,000,000,000 (the next power of ten, 10,000,000,000, is outside the range of the `int` type).

Listing 6.6: powersof10.cpp

```
#include <iostream>

int main() {
    int power = 1;
    while (power <= 1000000000) {
        std::cout << power << '\n';
        power *= 10;
    }
}
```

Listing 6.6 (powersof10.cpp) produces

```
1
10
100
1000
10000
100000
1000000
10000000
100000000
1000000000
```

It is customary to right justify a column of numbers, but Listing 6.6 (powersof10.cpp) prints the powers of ten with their most-significant digit left aligned. We can right align the numbers using a stream object called a *steam manipulator*. The specific stream manipulator we need is named `std::setw`. `setw` means "set width." It can be used as

```
std::cout << std::setw(3) << x << '\n';
```

This statement prints the value of `x` right justified within a three character horizontal space on the screen. Listing 6.7 (powersof10justified.cpp) shows the affects of `setw`.

Listing 6.7: powersof10justified.cpp

```
#include <iostream>
#include <iomanip>

// Print the powers of 10 from 1 to 1,000,000,000
int main() {
    int power = 1;
    while (power <= 1000000000) {
        // Right justify each number in a field 10 wide
        std::cout << std::setw(10) << power << '\n';
        power *= 10;
    }
}
```

Listing 6.7 (powersof10justified.cpp) prints

```
          1
         10
        100
       1000
      10000
     100000
    1000000
   10000000
  100000000
 1000000000
```

Observe that in order to use `setw` the compiler needs to be made aware of it. The needed information about `std::setw` is not found in the `iostream` header file, so an additional preprocessor include directive is required:

```
#include <iomanip>
```

The `std::setw` manipulator "conditions" the output stream for the next item to be printed. The values passed to the "conditioned" stream are all right justified within the number of spaces specified by `std::setw`.

As an aside, this is good place to reveal another trick to improve the output of a C++ program. Listing 6.8 (powersof10withcommas.cpp) enhances Listing 6.7 (powersof10justified.cpp) by adding commas in the appropriate places in the displayed numbers.

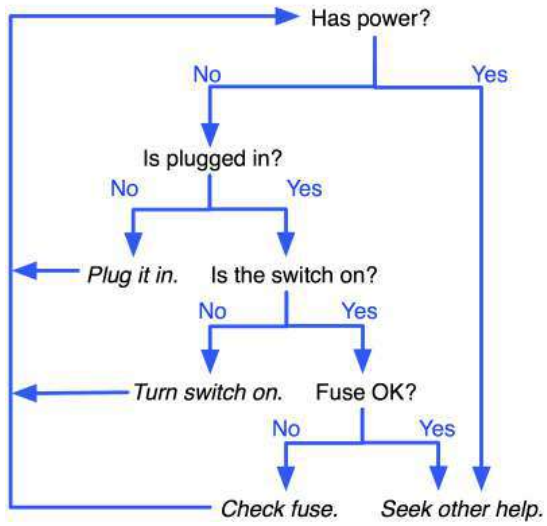
Listing 6.8: `powersof10withcommas.cpp`

```
#include <iostream>
#include <iomanip>
#include <locale>

// Print the powers of 10 from 1 to 1,000,000,000
int main() {
    int power = 1;
    std::cout.imbue(std::locale(""));
    while (power <= 1000000000) {
        // Right justify each number in a field 10 wide
        std::cout << std::setw(13) << power << '\n';
        power *= 10;
    }
}
```

Listing 6.8 (powersof10withcommas.cpp) prints

```
            1
           10
          100
        1,000
       10,000
      100,000
    1,000,000
   10,000,000
  100,000,000
1,000,000,000
```

Figure 6.2 Decision tree for troubleshooting a computer system Listing 6.9 (troubleshootloop.cpp)



The statement

```
std::cout.imbue(std::locale(""));
```

adjusts the `std::cout` object to print number with digit separators appropriate for the current country. If you replace the statement with

```
std::cout.imbue(std::locale("german"));
```

the output becomes

```
            1
           10
          100
        1.000
       10.000
      100.000
    1.000.000
   10.000.000
  100.000.000
1.000.000.000
```

because the German language uses periods in place of commas for digit separators. The `"german"` locale also would display a comma where the decimal point would appear in an English floating-point number.

We can use a `while` statement to make Listing 5.11 (troubleshoot.cpp) more convenient for the user. Recall that the computer troubleshooting program forces the user to rerun the program once a potential program has been detected (for example, turn on the power switch, then run the program again to see what else might be wrong). A more desirable decision logic is shown in Figure 6.2.

Listing 6.9 (troubleshootloop.cpp) incorporates a `while` statement so that the program's execution continues until the problem is resolved or its resolution is beyond the capabilities of the program.

Listing 6.9: `troubleshootloop.cpp`

```
#include <iostream>

int main() {
    std::cout << "Help!  My computer doesn't work!\n";
    char choice;
    bool done = false;  // Initially, we are not done

    while (!done) {  // Continue until we are done
        std::cout << "Does the computer make any sounds "
                  << "(fans, etc.) or show any lights? (y/n):";
        std::cin >> choice;
        // The troubleshooting control logic
        if (choice == 'n') { // The computer does not have power
            std::cout << "Is it plugged in? (y/n):";
            std::cin >> choice;
            if (choice == 'n') { // It is not plugged in, plug it in
                std::cout << "Plug it in.\n";
            }
            else { // It is plugged in
                std::cout << "Is the switch in the \"on\" position? (y/n):";
                std::cin >> choice;
                if (choice == 'n') { // The switch is off, turn it on!
                    std::cout << "Turn it on.\n";
                }
                else { // The switch is on
                    std::cout << "Does the computer have a fuse?  (y/n):";
                    std::cin >> choice;
                    if (choice == 'n') { // No fuse
                        std::cout << "Is the outlet OK? (y/n):";
                        std::cin >> choice;
                        if (choice == 'n') { // Fix outlet
                            std::cout << "Check the outlet's circuit "
                                      << "breaker or fuse.  Move to a "
                                      << "new outlet, if necessary.\n";
                        }
                        else { // Beats me!
                            std::cout << "Please consult a service "
                                      << "technician.\n";
                            done = true;  // Exhausted simple fixes
                        }
                    }
                    else { // Check fuse
                        std::cout << "Check the fuse. Replace if "
                                  << "necessary.\n";
                    }
                }
            }
        }
        else { // The computer has power
            std::cout << "Please consult a service technician.\n";
```

```
            done = true;  // Only troubleshoots power issues
        }
    }
}
```

The bulk of the body of the Listing 6.9 (troubleshootloop.cpp)is wrapped by a `while` statement. The Boolean variable `done` is often called a *flag*. You can think of the flag being down when the value is false and raised when it is true. In this case, when the flag is raised, it is a signal that the program should terminate.

Notice the last 11 lines of Listing 6.9 (troubleshootloop.cpp):

```
                    }
                }
            }
        }
        else { // The computer has power
            std::cout << "Please consult a service technician.\n";
            done = true;  // Only troubleshoots power issues
        }
    }
}
```

In the way this code is organized, the matching opening curly brace of a particular closing curly brace can be found by scanning upward in the source code until the closest opening curly brace at the same indentation level is found. Our programming logic is now getting complex enough that the proper placement of curly braces is crucial for human readers to more quickly decipher how the program should work. See Section 4.5 for guidelines on indentation and curly brace placement to improve code readability.

6.2 Nested Loops

Just like in `if` statements, `while` bodies can contain arbitrary C++ statements, including other `while` statements. A loop can therefore be nested within another loop. To see how nested loops work, consider a program that prints out a multiplication table. Elementary school students use multiplication tables, or times tables, as they learn the products of integers up to 10 or even 12. Figure 6.3 shows a 10×10 multiplication table. We want our multiplication table program to be flexible and allow the user to specify the table's size. We will begin our development work with a simple program and add features as we go. First, we will not worry about printing the table's row and column titles, nor will we print the lines separating the titles from the contents of the table. Initially we will print only the contents of the table. We will see we need a nested loop to print the table's contents, but that still is too much to manage in our first attempt. In our first attempt we will print the rows of the table in a very rudimentary manner. Once we are satisfied that our simple program works we can add more features. Listing 6.10 (timestable-1st-try.cpp) shows our first attempt at a muliplication table.

Listing 6.10: `timestable-1st-try.cpp`

```
#include <iostream>

int main() {
    int size;  // The number of rows and columns in the table
    std::cout << "Please enter the table size: ";
```

Figure 6.3 A 10 × 10 multiplication table

×	1	2	3	4	5	6	7	8	9	10
1	1	2	3	4	5	6	7	8	9	10
2	2	4	6	8	10	12	14	16	18	20
3	3	6	9	12	15	18	21	24	27	30
4	4	8	12	16	20	24	28	32	36	40
5	5	10	15	20	25	30	35	40	45	50
6	6	12	18	24	30	36	42	48	54	60
7	7	14	21	28	35	42	49	56	63	70
8	8	16	24	32	40	48	56	64	72	80
9	9	18	27	36	45	54	63	72	81	90
10	10	20	30	40	50	60	70	80	90	100

```
    std::cin >> size;
    // Print a size x size multiplication table
    int row = 1;
    while (row <= size) {           // Table has 10 rows.
        std::cout << "Row #" << row << '\n';
        row++;                        // Next row
    }
}
```

The output of Listing 6.10 (timestable-1st-try.cpp) is somewhat underwhelming:

```
Please enter the table size: 10
Row #1
Row #2
Row #3
Row #4
Row #5
Row #6
Row #7
Row #8
Row #9
Row #10
```

Listing 6.10 (timestable-1st-try.cpp) does indeed print each row in its proper place—it just does not supply the needed detail for each row. Our next step is to refine the way the program prints each row. Each row should contain `size` numbers. Each number within each row represents the product of the current row and current column; for example, the number in row 2, column 5 should be $2 \times 5 = 10$. In each row, therefore, we must vary the column number from from 1 to `size`. Listing 6.11 (timestable-2nd-try.cpp) contains the needed refinement.

Listing 6.11: `timestable-2nd-try.cpp`

```
#include <iostream>

int main() {
```

```
    int size;  // The number of rows and columns in the table
    std::cout << "Please enter the table size: ";
    std::cin >> size;
    // Print a size x size multiplication table
    int row = 1;
    while (row <= size) {           // Table has size rows.
        int column = 1;               // Reset column for each row.
        while (column <= size) {    // Table has size columns.
            int product = row*column; // Compute product
            std::cout << product << " ";   // Display product
            column++;                 // Next element
        }
        std::cout << '\n';                 // Move cursor to next row
        row++;                        // Next row
    }
}
```

We use a loop to print the contents of each row. The outer loop controls how many total rows the program prints, and the inner loop, executed in its entirity each time the program prints a row, prints the individual elements that make up a row.

The result of Listing 6.11 (timestable-2nd-try.cpp) is

```
Please enter the table size: 10
1 2 3 4 5 6 7 8 9 10
2 4 6 8 10 12 14 16 18 20
3 6 9 12 15 18 21 24 27 30
4 8 12 16 20 24 28 32 36 40
5 10 15 20 25 30 35 40 45 50
6 12 18 24 30 36 42 48 54 60
7 14 21 28 35 42 49 56 63 70
8 16 24 32 40 48 56 64 72 80
9 18 27 36 45 54 63 72 81 90
10 20 30 40 50 60 70 80 90 100
```

The numbers within each column are not lined up nicely, but the numbers are in their correct positions relative to each other. We can use the `std::setw` stream manipulator introduced in Listing 6.7 (powersof10justified.cpp) to right justify the numbers within a four-digit area. Listing 6.12 (timestable-3rd-try.cpp) contains this alignment adjustment.

Listing 6.12: `timestable-3rd-try.cpp`

```
#include <iostream>
#include <iomanip>

int main() {
    int size;  // The number of rows and columns in the table
    std::cout << "Please enter the table size: ";
    std::cin >> size;
    // Print a size x size multiplication table
    int row = 1;
    while (row <= size) {             // Table has size rows.
        int column = 1;                 // Reset column for each row.
        while (column <= size) {      // Table has size columns.
            int product = row*column;   // Compute product
```

```
            std::cout << std::setw(4) << product; // Display product
            column++;                             // Next element
        }
        std::cout << '\n';                            // Move cursor to next row
        row++;                                    // Next row
    }
}
```

Listing 6.12 (timestable-3rd-try.cpp) produces the table's contents in an attractive form:

```
Please enter the table size: 10
  1   2   3   4   5   6   7   8   9  10
  2   4   6   8  10  12  14  16  18  20
  3   6   9  12  15  18  21  24  27  30
  4   8  12  16  20  24  28  32  36  40
  5  10  15  20  25  30  35  40  45  50
  6  12  18  24  30  36  42  48  54  60
  7  14  21  28  35  42  49  56  63  70
  8  16  24  32  40  48  56  64  72  80
  9  18  27  36  45  54  63  72  81  90
 10  20  30  40  50  60  70  80  90 100
```

Input values of 5:

```
Please enter the table size: 5
  1   2   3   4   5
  2   4   6   8  10
  3   6   9  12  15
  4   8  12  16  20
  5  10  15  20  25
```

and 15:

```
Please enter the table size: 15
  1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
  2   4   6   8  10  12  14  16  18  20  22  24  26  28  30
  3   6   9  12  15  18  21  24  27  30  33  36  39  42  45
  4   8  12  16  20  24  28  32  36  40  44  48  52  56  60
  5  10  15  20  25  30  35  40  45  50  55  60  65  70  75
  6  12  18  24  30  36  42  48  54  60  66  72  78  84  90
  7  14  21  28  35  42  49  56  63  70  77  84  91  98 105
  8  16  24  32  40  48  56  64  72  80  88  96 104 112 120
  9  18  27  36  45  54  63  72  81  90  99 108 117 126 135
 10  20  30  40  50  60  70  80  90 100 110 120 130 140 150
 11  22  33  44  55  66  77  88  99 110 121 132 143 154 165
 12  24  36  48  60  72  84  96 108 120 132 144 156 168 180
 13  26  39  52  65  78  91 104 117 130 143 156 169 182 195
 14  28  42  56  70  84  98 112 126 140 154 168 182 196 210
 15  30  45  60  75  90 105 120 135 150 165 180 195 210 225
```

also give good results.

All that is left is to add the row and column titles and the lines that bound the edges of the table. Listing 6.13 (timestable.cpp) adds the necessary code.

Listing 6.13: timestable.cpp

```
#include <iostream>
#include <iomanip>

int main() {
    int size;  // The number of rows and columns in the table
    std::cout << "Please enter the table size: ";
    std::cin >> size;
    // Print a size x size multiplication table

    // First, print heading:  1  2  3  4   5   etc.
    std::cout << "    ";
    // Print column heading
    int column = 1;
    while (column <= size) {
        std::cout << std::setw(4) << column;  // Print heading for this column.
        column++;
    }
    std::cout << '\n';

    // Print line separator:   +------------------
    std::cout << "   +";
    column = 1;
    while (column <= size) {
        std::cout << "----";               // Print line for this column.
        column++;
    }
    std::cout << '\n';

    // Print table contents
    int row = 1;
    while (row <= size) {             // Table has size rows.
        std::cout << std::setw(2) << row << " |"; // Print heading for row.
        int column = 1;               // Reset column for each row.
        while (column <= size) {      // Table has size columns.
            int product = row*column; // Compute product
            std::cout << std::setw(4) << product; // Display product
            column++;                 // Next element
        }
        row++;                        // Next row
        std::cout << '\n';            // Move cursor to next row
    }
}
```

When the user supplies the value 10, Listing 6.13 (timestable.cpp) produces

```
Please enter the table size: 10
       1   2   3   4   5   6   7   8   9  10
   +----------------------------------------
 1 |   1   2   3   4   5   6   7   8   9  10
 2 |   2   4   6   8  10  12  14  16  18  20
 3 |   3   6   9  12  15  18  21  24  27  30
 4 |   4   8  12  16  20  24  28  32  36  40
 5 |   5  10  15  20  25  30  35  40  45  50
```

```
 6 |   6  12  18  24  30  36  42  48  54  60
 7 |   7  14  21  28  35  42  49  56  63  70
 8 |   8  16  24  32  40  48  56  64  72  80
 9 |   9  18  27  36  45  54  63  72  81  90
10 |  10  20  30  40  50  60  70  80  90 100
```

An input of 15 yields

```
Please enter the table size: 15
       1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
   +-------------------------------------------------------------
 1 |   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
 2 |   2   4   6   8  10  12  14  16  18  20  22  24  26  28  30
 3 |   3   6   9  12  15  18  21  24  27  30  33  36  39  42  45
 4 |   4   8  12  16  20  24  28  32  36  40  44  48  52  56  60
 5 |   5  10  15  20  25  30  35  40  45  50  55  60  65  70  75
 6 |   6  12  18  24  30  36  42  48  54  60  66  72  78  84  90
 7 |   7  14  21  28  35  42  49  56  63  70  77  84  91  98 105
 8 |   8  16  24  32  40  48  56  64  72  80  88  96 104 112 120
 9 |   9  18  27  36  45  54  63  72  81  90  99 108 117 126 135
10 |  10  20  30  40  50  60  70  80  90 100 110 120 130 140 150
11 |  11  22  33  44  55  66  77  88  99 110 121 132 143 154 165
12 |  12  24  36  48  60  72  84  96 108 120 132 144 156 168 180
13 |  13  26  39  52  65  78  91 104 117 130 143 156 169 182 195
14 |  14  28  42  56  70  84  98 112 126 140 154 168 182 196 210
15 |  15  30  45  60  75  90 105 120 135 150 165 180 195 210 225
```

If the user enters 7, the program prints

```
Please enter the table size: 7
       1   2   3   4   5   6   7
   +-----------------------------
 1 |   1   2   3   4   5   6   7
 2 |   2   4   6   8  10  12  14
 3 |   3   6   9  12  15  18  21
 4 |   4   8  12  16  20  24  28
 5 |   5  10  15  20  25  30  35
 6 |   6  12  18  24  30  36  42
 7 |   7  14  21  28  35  42  49
```

The user even can enter a 1:

```
Please enter the table size: 1
       1
   +----
 1 |   1
```

As we can see, the table automatically adjusts to the size and spacing required by the user's input.

This is how Listing 6.13 (timestable.cpp) works:

- It is important to distinguish what is done only once (outside all loops) from that which is done repeatedly. The column heading across the top of the table is outside of all the loops; therefore, it is printed all at once.

- The work to print the heading for the rows is distributed throughout the execution of the outer loop. This is because the heading for a given row cannot be printed until all the results for the previous row have been printed.

- A code fragment like

  ```
  if (x < 10)
      std::cout << " ";
  std::cout << x;
  ```

  prints x in one of two ways: if x is a one-digit number, it prints a space before it; otherwise, it does not print the extra space. The net effect is to right justify one and two digit numbers within a two character space printing area. This technique allows the columns within the times table to be properly right aligned.

- In the nested loop, row is the control variable for the outer loop; column controls the inner loop.

- The inner loop executes size times on every single iteration of the outer loop. How many times is the statement

  ```
  std::cout << product << " ";    // Display product
  ```

  executed? size × size times, one time for every product in the table.

- A newline is printed after the contents of each row is displayed; thus, all the values printed in the inner (column) loop appear on the same line.

Nested loops are used when an iterative process itself must be repeated. In our times table example, a while loop is used to print the contents of each row, but multiple rows must be printed. The inner loop prints the contents of each row, while the outer is responsible for printing all the rows.

Listing 6.14 (permuteabc.cpp) uses a triply-nested loop to print all the different arrangements of the letters A, B, and C. Each string printed is a *permutation* of ABC.

Listing 6.14: permuteabc.cpp

```
// File permuteabc.cpp

#include <iostream>

int main() {
    char first = 'A';        // The first letter varies from A to C
    while (first <= 'C') {
        char second = 'A';
        while (second <= 'C') { // The second varies from A to C
            if (second != first) {  // No duplicate letters
                char third = 'A';
                while (third <= 'C') { // The third varies from A to C
                    // Don't duplicate first or second letter
                    if (third != first && third != second)
                        std::cout << first << second << third << '\n';
                    third++;
                }
            }
            second++;
        }
```

```
        first++;
    }
}
```

Notice how the `if` statements are used to prevent duplicate letters within a given string. The output of Listing 6.14 (permuteabc.cpp) is all six permutations of ABC:

```
ABC
ACB
BAC
BCA
CAB
CBA
```

6.3 Abnormal Loop Termination

By default, a `while` statement executes until its condition becomes false. The executing program checks this condition only at the "top" of the loop. This means that even if the Boolean expression that makes up the condition becomes false before the program completes executing all the statements within the body of the loop, all the remaining statements in the loop's body must complete before the loop can once again check its condition. In other words, the `while` statement in and of itself cannot exit its loop somewhere in the middle of its body.

Ordinarily this behavior is not a problem. Usually the intention is to execute all the statements within the body as an indivisible unit. Sometimes, however, it is desirable to immediately exit the body or recheck the condition from the middle of the loop instead. C++ provides the `break` and `continue` statements to to give programmers more flexibility designing the control logic of loops.

6.3.1 The break statement

C++ provides the `break` statement to implement middle-exiting control logic. The `break` statement causes the immediate exit from the body of the loop. Listing 6.15 (addmiddleexit.cpp) is a variation of Listing 6.4 (addnonnegatives.cpp) that illustrates the use of `break`.

Listing 6.15: `addmiddleexit.cpp`

```
#include <iostream>

int main() {
    int input, sum = 0;
    std::cout << "Enter numbers to sum, negative number ends list:";
    while (true) {
        std::cin >> input;
        if (input < 0)
            break;       // Exit loop immediately
        sum += input;
    }
    std::cout << "Sum = " << sum << '\n';
}
```

The condition of the `while` in Listing 6.15 (addmiddleexit.cpp) is a tautology. This means the condition is true and can never be false. When the program's execution reaches the `while` statement it is guaranteed to enter the loop's body and the `while` loop itself does not provide a way of escape. The `if` statement in the loop's body:

```
if (input < 0)  // Is input negative
    break;        // If so, exit the loop immediately
```

provides the necessary exit. In this case the `break` statement, executed conditionally based on the value of the variable `input`, exits the loop. In Listing 6.15 (addmiddleexit.cpp) the `break` statement executes only when the user enters a negative number. When the program's execution encounters the `break` statement, it immediately jumps out of the loop. It skips any statements following the `break` within the loop's body. Since the statement

```
sum += input;  // Accumulate user input
```

appears after the `break`, it is not possible for the program to add a negative number to the `sum` variable.

Some software designers believe that programmers should use the `break` statement sparingly because it deviates from the normal loop control logic. Ideally, every loop should have a single entry point and single exit point. While Listing 6.15 (addmiddleexit.cpp) has a single exit point (the `break` statement), some programmers commonly use `break` statements within `while` statements in the which the condition for the while is not a tautology. Adding a `break` statement to such a loop adds an extra exit point (the top of the loop where the condition is checked is one point, and the `break` statement is another). Using multiple `break` statements within a single loop is particularly dubious and you should avoid that practice.

Why have the `break` statement at all if its use is questionable and it is dispensable? The logic in Listing 6.4 (addnonnegatives.cpp) is fairly simple, so the restructuring of Listing 6.15 (addmiddleexit.cpp) is straightforward; in general, the effort to restructure code to avoid a `break` statement may complicate the logic a bit and require the introduction of an additional Boolean variable. As shown in Figure 6.4, any program that uses a `break` statement can be rewritten so that the `break` statement is not used.

The no-`break` version introduces a Boolean variable, and the loop control logic is a little more complicated. The no-`break` version uses more memory (an extra variable) and more time to execute (requires an extra check in the loop condition during every iteration of the loop). This extra memory is insignificant, and except for rare, specialized applications, the extra execution time is imperceptible. In most cases, the more important issue is that the more complicated the control logic for a given section of code, the more difficult the code is to write correctly. In some situations, even though it violates the "single entry point, single exit point" principle, a simple `break` statement is an acceptable loop control option.

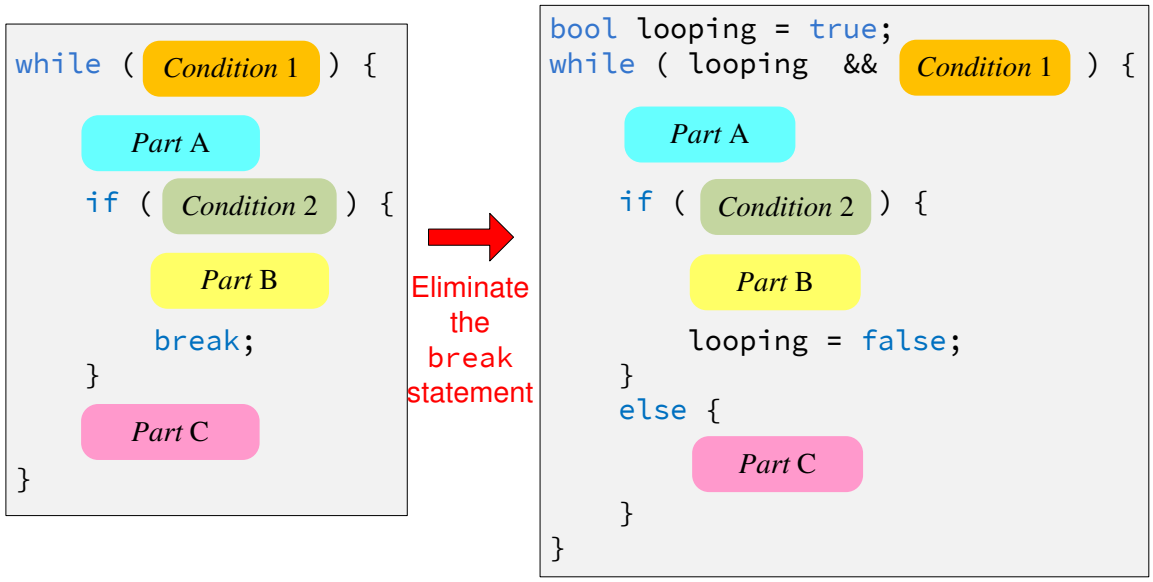
6.3.2 The goto Statement

The `break` statement exits the single loop in which it is located. A `break` statement is insufficient to jump completely out of the middle of a nested loop. The `goto` statement allows the program's execution flow to jump to a specified location within the function. Listing 6.16 (exitnested.cpp) uses a `goto` statement to jump out from the middle of a nested loop.

Listing 6.16: `exitnested.cpp`

```
#include <iostream>

int main() {
    // Compute some products
```

Figure 6.4 The code on the left generically represents any loop that uses a `break` statement. It is possible to transform the code on the left to eliminate the `break` statement, as the code on the right shows.




```
    int op1 = 2;
    while (op1 < 100) {
        int op2 = 2;
        while (op2 < 100) {
            if (op1 * op2 == 3731)
                goto end;
            std::cout << "Product is " << (op1 * op2) << '\n';
            op2++;
        }
        op1++;
    }
end:
    std::cout << "The end" << '\n';
}
```

When `op1 * op2` is 3731, program flow will jump to the specified label within the program. In this example, the label is named `end`, but this name is arbitrary. Like variable names, label names should be chosen to indicate their intended purpose. The label here named `end` comes after and outside the nested `while` loops.

A label's name is an identifier (see Section 3.3), and a label is distinguished by the colon that immediately follows its name. A label represents a target to which a `goto` can jump. A `goto` label must appear before a statement within a function.

With the `goto` statement, the `while` is superfluous; for example, Listing 6.2 (iterativecounttofive.cpp) could be rewritten without the `while` statement as shown in Listing 6.17 (gotoloop.cpp).

Listing 6.17: gotoloop.cpp

```
#include <iostream>

int main() {
    int count = 1;              // Initialize counter
top:
    if (count > 5)
        goto end;
    std::cout << count << '\n';  // Display counter, then
    count++;                     // Increment counter
    goto top;
end:
    ;   // Target is an empty statement
}
```

Early programming languages like FORTRAN and early versions of BASIC did not have structured statements like `while`, so programmers were forced to use `goto` statements to write loops. The problem with using `goto` statements is that it is easy to develop program logic that is very difficult to understand, even for the original author of the code. See the Wikipedia article about *spaghetti code* (`http://en.wikipedia.org/wiki/Spaghetti_code`). The structured programming revolution of the 1960s introduced constructs such as the `while` statement and resulted in the disappearance of the use of `goto` in most situations. All modern programming languages have a form of the `while` statement, so the `goto` statement in C++ is largely ignored except for the case of breaking out of a nested loop. You similarly should restrict your use of the `goto` statement to the abnormal exit of nested loops.

6.3.3 The continue Statement

When a program's execution encounters a `break` statement inside a loop, it skips the rest of the body of the loop and exits the loop. The `continue` statement is similar to the `break` statement, except the `continue` statement does not necessarily exit the loop. The `continue` statement skips the rest of the body of the loop and immediately checks the loop's condition. If the loop's condition remains true, the loop's execution resumes at the top of the loop. Listing 6.18 (continueexample.cpp) shows the `continue` statement in action.

Listing 6.18: continueexample.cpp

```
#include <iostream>

int main() {
    int input, sum = 0;
    bool done = false;
    while (!done) {
        std::cout << "Enter positive integer (999 quits): ";
        std::cin >> input;
        if (input < 0) {
            std::cout << "Negative value " << input << " ignored\n";
            continue;  // Skip rest of body for this iteration
        }
        if (input != 999) {
            std::cout << "Tallying " << input << '\n';
            sum += input;
        }
```

```
        else
            done = (input == 999);  // 999 entry exits loop
    }
    std::cout << "sum = " << sum << '\n';
}
```

Programmers do not use the `continue` statement as frequently as the `break` statement since it is easy to transform code using `continue` into an equivalent form that does not. Listing 6.19 (nocontinueexample.cpp) works exactly like Listing 6.18 (continueexample.cpp), but it avoids the `continue` statement.

Listing 6.19: `nocontinueexample.cpp`

```
#include <iostream>

int main() {
    int input, sum = 0;
    bool done = false;
    while (!done) {
        std::cout << "Enter positive integer (999 quits): ";
        std::cin >> input;
        if (input < 0)
            std::cout << "Negative value " << input << " ignored\n";
        else
            if (input != 999) {
                std::cout << "Tallying " << input << '\n';
                sum += input;
            }
            else
                done = (input == 999);  // 999 entry exits loop
    }
    std::cout << "sum = " << sum << '\n';
}
```

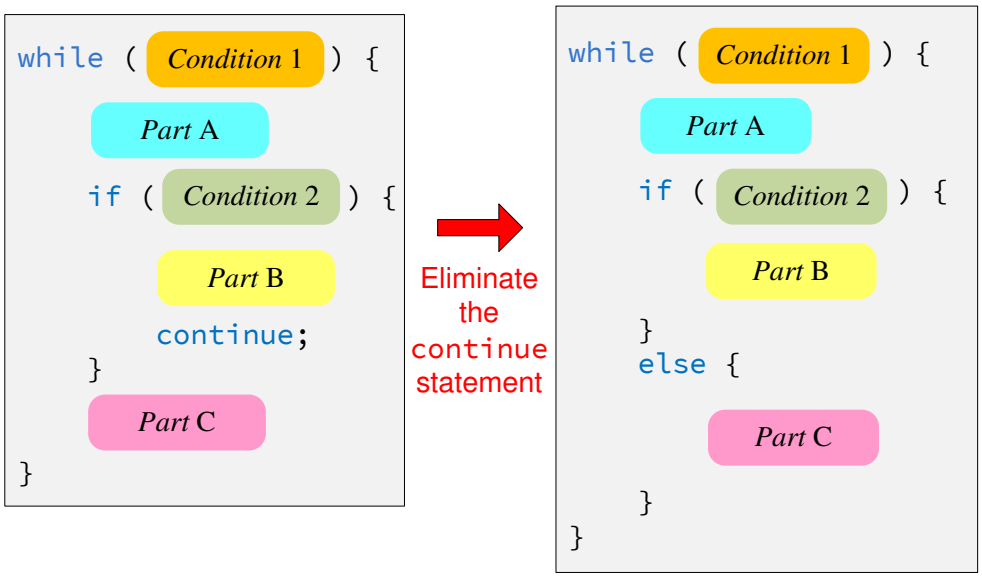
Figure 6.5 shows how we can rewrite any program that uses a `continue` statement into an equivalent form that does not use `continue`. The transformation is simpler than for `break` elimination (see Figure 6.4) since the loop's condition remains the same, and no additional variable is needed.

The version that uses `continue` is no more efficient than the version that uses `else`; in fact, the Visual C++ and GNU C++ compilers generate the same machine language code for Listing 6.18 (continueexample.cpp) and Listing 6.19 (nocontinueexample.cpp). Also, the logic of the `else` version is no more complex than the `continue` version. Therefore, unlike the `break` statement above, there is no compelling reason to use the `continue` statement. Sometimes a programmer may add a `continue` statement at the last minute to an existing loop body to handle an exceptional condition (like ignoring negative numbers in the example above) that initially went unnoticed. If the body of the loop is lengthy, the programmer can add a conditional statement with a `continue` near the top of the loop body without touching the logic of the rest of the loop. The `continue` statement thus merely provides a convenient alternative for the programmer. The `else` version is preferred.

6.4 Infinite Loops

An infinite loop is a loop without an exit. Once the program flow enters an infinite loop's body it cannot escape. Some infinite loops are by design; for example, a long-running server application, like a Web

Figure 6.5 The code on the left generically represents any loop that uses a `continue` statement. It is possible to transform the code on the left to eliminate the `continue` statement, as the code on the right shows.



server, may need to continuously check for incoming connections. This server application can perform this checking within a loop that runs indefinitely. All too often, however, beginning programmers create infinite loops by accident, and these infinite loops represent logic errors in their programs.

Intentional infinite loops should be made obvious. For example,

```
while (true) {
    /*  Do something forever . . . */
}
```

The Boolean literal `true` is always true, so it is impossible for the loop's condition to be false. The only ways to exit the loop is via a `break` statement, `return` statement (see Chapter 9), or an `exit` call (see Section 8.1) embedded somewhere within its body.

It is easy to write an intentional infinite loop. Accidental infinite loops are quite common, but can be puzzling for beginning programmers to diagnose and repair. Consider Listing 6.20 (findfactors.cpp) that attempts to print all the integers with their associated factors from 1 to 20.

Listing 6.20: `findfactors.cpp`

```
#include <iostream>

int main() {
    // List of the factors of the numbers up to 20
    int n = 1;
    const int MAX = 20;
    while (n <= MAX) {
        int factor = 1;
```

```
        std::cout << n << ": ";
        while (factor <= n)
            if (n % factor == 0) {
                std::cout << factor << " ";
                factor++;
            }
        std::cout << '\n';  // Go to next line for next n
        n++;
    }
}
```

It displays

```
1: 1
2: 1 2
3: 1
```

and then "freezes up" or "hangs," ignoring any user input (except the key sequence Ctrl C on most systems which interrupts and terminates the running program). This type of behavior is a frequent symptom of an unintentional infinite loop. The factors of 1 display properly, as do the factors of 2. The first factor of 3 is properly displayed and then the program hangs. Since the program is short, the problem may be easy to locate. In some programs, though, the error may be challenging to find. Even in Listing 6.20 (findfactors.cpp) the debugging task is nontrivial since it involves nested loops. (Can you find and fix the problem in Listing 6.20 (findfactors.cpp) before reading further?)

In order to avoid infinite loops, we must ensure that the loop exhibits certain properties:

- The loop's condition must not be a tautology (a Boolean expression that can never be false). For example,

  ```
  while (i >= 1 || i <= 10) {
      /*  Body omitted  */
  }
  ```

  is an infinite loop since any value chosen for `i` will satisfy one or both of the two subconditions. Perhaps the programmer intended to use a && instead of `||` to stay in the loop as long as `i` remains in the range 1...10.

  In Listing 6.20 (findfactors.cpp) the outer loop condition is

  ```
  n <= MAX
  ```

  If `n` is 21 and `MAX` is 20, then the condition is false, so this is not a tautology. Checking the inner loop condition:

  ```
  factor <= n
  ```

  we see that if `factor` is 3 and `n` is 2, then the expression is false; therefore, it also is not a tautology.

- The condition of a `while` must be true initially to gain access to its body. The code within the body must modify the state of the program in some way so as to influence the outcome of the condition that is checked at each iteration. This usually means code within the body of the loop modifies one of the variables used in the condition. Eventually the variable assumes a value that makes the condition false, and the loop terminates.

In Listing 6.20 (findfactors.cpp) the outer loop's condition involves the variable n and constant MAX. MAX cannot change, so to avoid an infinite loop it is essential that n be modified within the loop. Fortunately, the last statement in the body of the outer loop increments n. n is initially 1 and MAX is 20, so unless the circumstances arise to make the inner loop infinite, the outer loop should eventually terminate.

The inner loop's condition involves the variables n and factor. No statement in the inner loop modifies n, so it is imperative that factor be modified in the loop. The good news is factor is incremented in the body of the inner loop, but the bad news is the increment operation is protected within the body of the if statement. The inner loop contains one statement, the if statement. That if statement in turn has two statements in its body:

```
while (factor <= n)
    if (n % factor == 0) {
        std::cout << factor << " ";
        factor++;
    }
```

If the condition of the if is ever false, the state of the program will not change when the body of the inner loop is executed. This effectively creates an infinite loop. The statement that modifies factor must be moved outside of the if statement's body:

```
while (factor <= n) {
    if (n % factor == 0)
        std::cout << factor << " ";
    factor++;
}
```

Note that the curly braces are necessary for the statement incrementing factor to be part of the body of the while. This new version runs correctly.

Programmers can use a debugger to step through a program to see where and why an infinite loop arises. Another common technique is to put print statements in strategic places to examine the values of the variables involved in the loop's control. The original inner loop can be so augmented:

```
while (factor <= n) {
    std::cout << "factor = " << factor
              << "  n = " << n << '\n';
    if (n % factor == 0) {
        std::cout << factor << " ";
        factor++;
    }
}
```

It produces the following output:

```
1: factor = 1  n = 1
1
2: factor = 1  n = 2
1 factor = 2  n = 2
2
3: factor = 1  n = 3
1 factor = 2  n = 3
factor = 2  n = 3
```

```
factor = 2  n = 3
factor = 2  n = 3
factor = 2  n = 3
factor = 2  n = 3
   .
   .
   .
```

The program continues to print the same line until the user interrupts its execution. The output demonstrates that once `factor` becomes equal to 2 and `n` becomes equal to 3 the program's execution becomes trapped in the inner loop. Under these conditions:

1. `2 < 3` is true, so the loop continues and
2. `3 % 2` is equal to 1, so the `if` statement will not increment `factor`.

It is imperative that `factor` be incremented each time through the inner loop; therefore, the statement incrementing `factor` must be moved outside of the `if`'s guarded body.

6.5 Iteration Examples

We can implement some sophisticated algorithms in C++ now that we are armed with `if` and `while` statements. This section provides several examples that show off the power of conditional execution and iteration.

6.5.1 Drawing a Tree

Suppose we must write a program that draws a triangular tree, and the user provides the tree's height. A tree that is five levels tall would look like

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

whereas a three-level tree would look like

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

If the height of the tree is fixed, we can write the program as a simple variation of Listing 2.4 (arrow.cpp) which uses only printing statements and no loops. Our program, however, must vary its height and width based on input from the user.

Listing 6.21 (startree.cpp) provides the necessary functionality.

Listing 6.21: `startree.cpp`

```
#include <iostream>

int main() {
    int height;    // Height of tree
    std::cout << "Enter height of tree: ";
    std::cin >> height;  // Get height from user
    int row = 0;          // First row, from the top, to draw
    while (row < height) { // Draw one row for every unit of height
        // Print leading spaces
        int count = 0;
        while (count < height - row) {
            std::cout << " ";
            count++;
        }
        // Print out stars, twice the current row plus one:
        //  1. number of stars on left side of tree
        //     = current row value
        //  2. exactly one star in the center of tree
        //  3. number of stars on right side of tree
        //     = current row value
        count = 0;
        while (count < 2*row + 1) {
            std::cout << "*";
            count++;
        }
        // Move cursor down to next line
        std::cout << '\n';
        // Change to the next row
        row++;
    }
}
```

When a user runs Listing 6.21 (startree.cpp) and enters 7, the program displays

```
Enter height of tree: 7
       *
      ***
     *****
    *******
   *********
  ***********
 *************
```

Listing 6.21 (startree.cpp) uses two sequential `while` loops both nested within a `while` loop. The outer `while` loop is responsible for drawing one row of the tree each time its body is executed:

- The program will execute the outer `while` loop's body as long as the user enters a value greater than zero; if the user enters zero or less, the program terminates and does nothing. This is the expected behavior.
- The last statement in the body of the outer `while`:

  ```
  row++;
  ```

ensures that the variable `row` increases by one each time through the loop; therefore, it eventually will equal `height` (since it initially had to be less than `height` to enter the loop), and the loop will terminate. There is no possibility of an infinite loop here.

- The body of the outer loop consists of more than one statement; therefore, the body must be enclosed within curly braces. Whenever a group of statements is enclosed within curly braces a *block* is formed. Any variable declared within a block is local to that block. A variable's scope (the section of the source code in which the variable exists and can be used) is from its point of declaration to the end of the block in which it is declared. For example, the variables `height` and `row` are declared in the block that is `main`'s body; thus, they are local to `main`. The variable `count` is declared within the block that is the body of the outer `while` statement; therefore, `count` is local to the outer `while` statement. An attempt to use `count` *outside* the body of the outer `while` statement would be an error.

What does it mean for a variable `x` to be *local* to a particular section of code? It means `x` does not exist outside its scope. There may be other variables in the program named `x`, but they are different variables. If it seems odd that you can have two different variables in the same program with the same name, consider the fact that there can be two people in the same room with the same name. They are different people, but they have the same name. Similarly, the meaning of a variable depends on its context, and its name is not necessarily unique.

The two inner loops play distinct roles:

- The first inner loop prints spaces. The number of spaces printed is equal to the height of the tree the first time through the outer loop and decreases each iteration. This is the correct behavior since each succeeding row moving down contains fewer leading spaces but more asterisks.
- The second inner loop prints the row of asterisks that make up the tree. The first time through the outer loop, row is zero, so no left side asterisks are printed, one central asterisk is printed (the top of the tree), and no right side asterisks are printed. Each time through the loop the number of left-hand and right-hand stars to print both increase by one and the same central asterisk is printed; therefore, the tree grows one wider on each side each line moving down. Observe how the `2*row + 1` value expresses the needed number of asterisks perfectly.
- While it seems asymmetrical, note that no third inner loop is required to print trailing spaces on the line after the asterisks are printed. The spaces would be invisible, so there is no reason to print them!

6.5.2 Printing Prime Numbers

A *prime number* is an integer greater than one whose only factors (also called divisors) are one and itself. For example, 29 is a prime number (only 1 and 29 divide into it with no remainder), but 28 is not (2, 4, 7, and 14 are factors of 28). Prime numbers were once merely an intellectual curiosity of mathematicians, but now they play an important role in cryptography and computer security.

The task is to write a program that displays all the prime numbers up to a value entered by the user. Listing 6.22 (printprimes.cpp) provides one solution.

Listing 6.22: `printprimes.cpp`

```
#include <iostream>

int main() {
    int max_value;
```

```
    std::cout << "Display primes up to what value? ";
    std::cin >> max_value;
    int value = 2;  // Smallest prime number
    while (value <= max_value) {
        // See if value is prime
        bool is_prime = true;  // Provisionally, value is prime
        // Try all possible factors from 2 to value - 1
        int trial_factor = 2;
        while (trial_factor < value) {
            if (value % trial_factor == 0) {
                is_prime = false;  // Found a factor
                break; // No need to continue; it is NOT prime
            }
            trial_factor++;
        }
        if (is_prime)
            std::cout << value <<  " ";   // Display the prime number
        value++;   // Try the next potential prime number
    }
    std::cout << '\n';  // Move cursor down to next line
}
```

Listing 6.22 (printprimes.cpp), with an input of 90, produces:

```
Display primes up to what value? 90
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89
```

The logic of Listing 6.22 (printprimes.cpp) is a little more complex than that of Listing 6.21 (startree.cpp). The user provides a value for `max_value`. The main loop (outer `while` iterates over all the values from two to `max_value`:

- Two new variables, local to the body of the outer loop, are introduced: `trial_factor` and `is_prime`. `is_prime` is initialized to true, meaning `value` is assumed to be prime unless our tests prove otherwise. `trial_factor` takes on all the values from two to `value - 1` in the inner loop:

  ```
  int trial_factor = 2;
  while (trial_factor < value) {
      if (value % trial_factor == 0) {
          is_prime = false;  // Found a factor
          break;  // No need to continue; it is NOT prime
      }
      trial_factor++;
  }
  ```

  The expression `value % trial_factor` is zero when `trial_factor` divides into `value` with no remainder—exactly when `trial_factor` is a factor of `value`. If the executing program determines that any of the values of `trial_factor` is a factor of `value`, then it sets `is_prime` to false and exits the loop via the `break`. If the loop continues to completion, the program never sets `is_prime` to false, which means it found no factors and `value` is indeed prime.

- The `if` statement after the inner loop:

  ```
  if (is_prime)
      std::cout << value <<  " "; // Display the prime number
  ```

simply checks the status of `is_prime`. If `is_prime` is true, then `value` must be prime, so it prints `value` along with an extra space for separation from output it may produce during subsequent iterations.

Some important questions we can ask include:

1. **If the user enters a 2, will it be printed?**

   In this case `max_value` = `value` = 2, so the condition of the outer loop

   ```
   value <= max_value
   ```

   is true, since `2 <= 2`. `is_prime` is set to true, but the condition of the inner loop

   ```
   trial_factor < value
   ```

   is not true (2 is not less than 2). Thus, the inner loop is skipped, `is_prime` is not changed from true, and 2 is printed. This behavior is correct because 2 is the smallest prime number (and the only even prime).

2. **if the user enters a number less than 2, is anything printed?**

   The `while` condition ensures that values less than two are not considered. The body of the `while` will never be entered. Only the newline is printed, and no numbers are displayed. This behavior is correct.

3. **Is the inner loop guaranteed to always terminate?**

   In order to enter the body of the inner loop, `trial_factor` must be less than `value`. `value` does not change anywhere in the loop. `trial_factor` is not modified anywhere in the `if` statement within the loop, and it is incremented within the loop immediately after the `if` statement. `trial_factor` is, therefore, incremented during each iteration of the loop. Eventually, `trial_factor` will equal `value`, and the loop will terminate.

4. **Is the outer loop guaranteed to always terminate?**

   In order to enter the body of the outer loop, `value` must be less than or equal to `max_value`. `max_value` does not change anywhere in the loop. `value` is increased in the last statement within the body of the outer loop, and `value` is not modified anywhere else. Since the inner loop is guaranteed to terminate as shown in the previous answer, eventually `value` will exceed `max_value` and the loop will end.

We can rearrange the logic of the inner `while` to avoid the `break` statement. The current version is:

```
while (trial_factor < value) {
    if (value % trial_factor == 0) {
        is_prime = false;  // Found a factor
        break; // No need to continue; it is NOT prime
    }
    trial_factor++;
}
```

We can rewrite it as:

```
while (is_prime && trial_factor < value) {
    is_prime = (value % trial_factor != 0);
    trial_factor++;     // Try next factor
}
```

This version without the `break` introduces a slightly more complicated condition for the `while` but removes the `if` statement within its body. `is_prime` is initialized to true before the loop. Each time through the loop it is reassigned. `trial_factor` will become false if at any time `value % trial_factor` is zero. This is exactly when `trial_factor` is a factor of `value`. If `is_prime` becomes false, the loop cannot continue, and if `is_prime` never becomes false, the loop ends when `trial_factor` becomes equal to `value`. Due to operator precedence, the parentheses are not necessary. The parentheses do improve readability, since an expression including both `==` and `!=` is awkward for humans to parse. When parentheses are placed where they are not needed, as in

```
x = (y + 2);
```

the compiler simply ignores them, so there is no efficiency penalty in the compiled code.

We can shorten the loop even further:

```
while (is_prime && trial_factor < value)
    is_prime = (value % trial_factor++ != 0);
```

This version uses the post-increment operator within the test expression (see Section 4.9). Recall that with the post-increment operator the value of the variable is used in the surrounding expression (if any), and then the variable is incremented. Since the `while`'s body now contains only one statement, the curly braces are not needed.

6.6 Exercises

1. In Listing 6.4 (addnonnegatives.cpp) could the condition of the `if` statement have used > instead of >= and achieved the same results? Why?

2. In Listing 6.4 (addnonnegatives.cpp) could the condition of the `while` statement have used > instead of >= and achieved the same results? Why?

3. Use a loop to rewrite the following code fragment so that it uses just one `std::cout` and one `'\n'`.

```
std::cout <<  2 << '\n';
std::cout <<  4 << '\n';
std::cout <<  6 << '\n';
std::cout <<  8 << '\n';
std::cout << 10 << '\n';
std::cout << 12 << '\n';
std::cout << 14 << '\n';
std::cout << 16 << '\n';
```

4. In Listing 6.4 (addnonnegatives.cpp) what would happen if the statement containing `std::cin` is moved out of the loop? Is moving the assignment out of the loop a good or bad thing to do? Why?

5. How many asterisks does the following code fragment print?

```
int a = 0;
while (a < 100) {
    std::cout << "*";
    a++;
}
std::cout << '\n';
```

6. How many asterisks does the following code fragment print?

```
int a = 0;
while (a < 100)
    std::cout << "*";
std::cout << '\n';
```

7. How many asterisks does the following code fragment print?

```
int a = 0;
while (a > 100) {
    std::cout << "*";
    a++;
}
std::cout << '\n';
```

8. How many asterisks does the following code fragment print?

```
int a = 0;
while (a < 100) {
    int b = 0;
    while (b < 55) {
        std::cout << "*";
        b++;
    }
    std::cout << '\n';
}
```

9. How many asterisks does the following code fragment print?

```
int a = 0;
while (a < 100) {
    if (a  % 5 == 0)
        std::cout << "*";
    a++;
}
std::cout << '\n';
```

10. How many asterisks does the following code fragment print?

```
int a = 0;
while (a < 100) {
    int b = 0;
    while (b < 40) {
        if ((a + b) % 2 == 0)
            std::cout << "*";
        b++;
    }
    std::cout << '\n';
    a++;
}
```

11. How many asterisks does the following code fragment print?

```
int a = 0;
while (a < 100) {
    int b = 0;
    while (b < 100) {
        int c = 0;
        while (c < 100) {
            std::cout << "*";
            c++;
        }
        b++;
    }
    a++;
}
std::cout << '\n';
```

12. What is printed by the following code fragment?

```
int a = 0;
while (a < 100)
    std::cout << a++;
std::cout << '\n';
```

13. What is printed by the following code fragment?

```
int a = 0;
while (a > 100)
    std::cout << a++;
std::cout << '\n';
```

14. Rewrite the following code fragment using a `break` statement and eliminating the `done` variable. Your code should behave identically to this code fragment.

```
bool done = false;
int n = 0, m = 100;
while (!done && n != m) {
    std::cin >> n;
    if (n < 0)
        done = true;
    std::cout << "n = " << n << '\n';
}
```

15. Rewrite the following code fragment so it eliminates the `continue` statement. Your new code's logic should be simpler than the logic of this fragment.

```
int x = 100, y;
while (x > 0) {
    std::cin >> y;
    if (y == 25) {
        x--;
        continue;
    }
    std::cin >> x;
```

```
    std::cout << "x = " << x << '\n';
}
```

16. Suppose you were given some code from the 1960s in a language that did not support structured statements like `while`. Your task is to modernize it and adapt it to C++. The following code fragment has been adapted to C++ already, but you must now structure it with a `while` statement to replace the `goto`s. Your code should be `goto` free and still behave identically to this code fragment.

```
     int i = 0;
top: if (i >= 10)
         goto end;
     std::cout << i << '\n';
     i++;
     goto top;
end:
```

17. What is printed by the following code fragment?

```
    int a = 0;
    while (a < 100);
        std::cout << a++;
    std::cout << '\n';
```

18. Write a C++ program that accepts a single integer value entered by the user. If the value entered is less than one, the program prints nothing. If the user enters a positive integer, n, the program prints an $n \times n$ box drawn with `*` characters. If the users enters 1, for example, the program prints

```
*
```

If the user enters a 2, it prints

```
**
**
```

An entry of three yields

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

and so forth. If the user enters 7, it prints

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

that is, a 7×7 box of `*` symbols.

19. Write a C++ program that allows the user to enter exactly twenty double-precision floating-point values. The program then prints the sum, average (arithmetic mean), maximum, and minimum of the values entered.

20. Write a C++ program that allows the user to enter any number of nonnegative double-precision floating-point values. The user terminates the input list with any negative value. The program then prints the sum, average (arithmetic mean), maximum, and minimum of the values entered. The terminating negative value is **not** used in the computations. If the first number the user supplies is negative, the program simply prints the text *NO NUMBERS PROVIDED*.

21. Redesign Listing 6.21 (startree.cpp) so that it draws a sideways tree pointing right; for example, if the user enters 7, the program would print

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

22. Redesign Listing 6.21 (startree.cpp) so that it draws a sideways tree pointing left; for example, if the user enters 7, the program would print

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

Chapter 7

Other Conditional and Iterative Statements

The `if`/`else` and `while` statements are sufficient to implement any algorithms that involve conditional execution and looping. The `break` and `continue` statements are convenient but are not necessary. C++ provides some additional conditional and iterative statements that are more convenient to use in some circumstances. These additional statements include

- `switch`: an alternative to some multi-way `if`/`else` statements
- the conditional operator: an expression that exhibits the behavior of an `if`/`else` statement
- `do`/`while`: a loop that checks its condition after its body is executed
- `for`: a loop convenient for counting

These alternate constructs allow certain parts of algorithms to expressed more clearly and succinctly. This chapter explores these other forms of expressing conditional execution and iteration.

7.1 The switch Statement

The `switch` statement provides a convenient alternative for some multi-way `if`/`else` statements like the one in Listing 5.15 (restyleddigittoword.cpp). Listing 7.1 (switchdigittoword.cpp) is a new implementation of Listing 5.15 (restyleddigittoword.cpp) that uses a `switch` statement instead of a multi-way `if`/`else` statement.

Listing 7.1: `switchdigittoword.cpp`

```
#include <iostream>

int main() {
    int value;
    std::cout << "Please enter an integer in the range 0...5: ";
    std::cin >> value;
    switch (value) {
      case 0:
        std::cout << "zero";
        break;
```

```
      case 1:
        std::cout << "one";
        break;
      case 2:
        std::cout << "two";
        break;
      case 3:
        std::cout << "three";
        break;
      case 4:
        std::cout << "four";
        break;
      case 5:
        std::cout << "five";
        break;
      default:
        if (value < 0)
            std::cout << "Too small";
        else
            std::cout << "Too large";
        break;
    }
    std::cout << '\n';
}
```

The general form of a `switch` is:

```
switch ( integral expression ) {
    case integral constant 1 :
        statement sequence 1
        break;
    case integral constant 2 :
        statement sequence 2
        break;
    case integral constant 3 :
        statement sequence 3
        break;
            ⋮
    case integral constant n :
        statement sequence n
        break;
    default:
        default statement sequence
}
```

In a `switch` statement

- The reserved word `switch` identifies a `switch` statement.
- The required parenthesized expression that follows the word `switch` must evaluate to an integral value. Any integer type, characters, and Boolean expressions are acceptable. Floating point expressions and other non-integer types are forbidden.

- The body of the `switch` is enclosed by required curly braces.

- Each occurrence of the word `case` is followed by an integral *constant* and a colon (:). We call the integral constant a `case` *label*. This label can be either a literal value or a `const` symbolic value (see Section 3.6). In particular, non-`const` variables and other expressions are expressly forbidden.

  The case label defines a position within the code; it is not an executable statement. A case label represents a target to which the program's execution flow can jump.

  If the `case` label matches the `switch`'s expression, then the statements that follow that label are executed up until the `break` statement is encountered. The statements and `break` statement that follow each `case` label are optional. One way to execute one set of statements for more than one `case` label is to provide empty statements for one or more of the labels, as in:

  ```
  std::cin >> key;  // get key from user
  switch (key) {
    case 'p':
    case 'P':
      std::cout << "You choose \"P\"\n";
      break;
    case 'q':
    case 'Q':
      done = true;
      break;
  }
  ```

  Here either an upper- or lowercase *P* result in the same action— *You chose P* is printed. If the user enters either an upper- or lowercase *Q*, the `done` Boolean variable is set to true. If the user enters neither *P* nor *Q*, none of the statements in the `switch` is executed.

  The `break` statement is optional. When a `case` label is matched, the statements that follow are executed until a `break` statement is encountered. The control flow then transfers out of the body of the `switch`. In this way, the `break` within a `switch` works just like a `break` within a loop: the rest of the body of the statement is skipped and program execution resumes at the next statement following the body. A missing `break` statement, a common error, when its omission is not intentional, causes the statements of the succeeding `case` label to be executed. The process continues until a `break` is encountered or the end of the `switch` body is reached.

- The `default` label is matched if none of the `case` labels match. It serves as a catch all option like the final `else` in a multi-way `if/else` statement. The `default` label is optional. If it is missing and none of the `case` labels match the expression, then no statement within the `switch`'s body is executed.

The `switch` statement has two restrictions that make it less general than the multi-way `if/else`:

- The `switch` argument must be an integral expression.

- Case labels must be constant integral values. Integral literals and constants are acceptable. Variables or expressions are **not** allowed.

To illustrate these restrictions, consider the following `if/else` statement that translates easily to an equivalent `switch` statement:

```
if (x == 1) {
    // Do 1 stuff here . . .
}
else if (x == 2) {
    // Do 2 stuff here . . .
}
else if (x == 3) {
    // Do 3 stuff here . . .
}
```

The corresponding `switch` statement is:

```
switch (x) {
  case 1:
    // Do 1 stuff here . . .
    break;
  case 2:
    // Do 2 stuff here . . .
    break;
  case 3:
    // Do 3 stuff here . . .
    break;
}
```

Now consider the following `if/else`:

```
if (x == y) {
    // Do "y" stuff here . . .
}
else if (x > 2) {
    // Do "> 2" stuff here . . .
}
else if (z == 3) {
    // Do 3 stuff here . . .
}
```

This code cannot be easily translated into a `switch` statement. The variable `y` cannot be used as a `case` label. The second choice checks for an inequality instead of an exact match, so direct translation to a `case` label is impossible. In the last condition, a different variable is checked, `z` instead of `x`. The control flow of a `switch` statement is determined by a single value (for example, the value of `x`), but a multi-way `if/else` statement is not so constrained.

Where applicable, a `switch` statement allows programmers to compactly express multi-way selection logic. Most programmers find a `switch` statement easier to read than an equivalent multi-way `if/else` construct.

A positive consequence of the `switch` statement's restrictions is that it allows the compiler to produce more efficient code for a `switch` than for an equivalent `if/else`. If a choice must be made from one of several or more options, and the `switch` statement can be used, then the `switch` statement will likely be faster than the corresponding multi-way `if/else`.

©2018 Richard L. Halterman

7.2 The Conditional Operator

As purely a syntactical convenience, C++ provides an alternative to the `if/else` construct called the *conditional operator*. It has limited application but is convenient nonetheless. The following code fragment assigns one of two things to `x`:

- the result of `y/z`, if `z` is nonzero, or
- zero, if `z` is zero; we wish to avoid the run-time error of division by zero.

```
// Assign a value to x:
if (z != 0)
    x = y/z;  // Division is possible
else
    x = 0;    // Assign a default value instead
```

This code has two assignment statements, but only one is executed at any given time. The conditional operator makes for a more compact statement:

```
// Assign a value to x:
x = (z != 0) ? y/z : 0;
```

The general form of a conditional expression is:

(*condition*) ? *expression 1* : *expression 2*

- *condition* is a normal Boolean expression that might appear in an `if` statement. Parentheses around the condition are not required but should be used to improve the readability.
- *expression 1* is the overall value of the conditional expression if *condition* is true.
- *expression 2* is the overall value of the conditional expression if *condition* is false.

The conditional operator uses two symbols (`?` and `:`) and three operands. Since it has three operands it is classified as a *ternary* operator (C++'s only one). The overall type of a conditional expression is the more dominant of exp_1 and exp_2 The conditional expression can be used anywhere an expression can be used. It is not a statement itself; it is used within a statement.

As another example, the *absolute value* of a number is defined in mathematics by the following formula:

$$|n| = \begin{cases} n, & \text{when } n \geq 0 \\ -n, & \text{when } n < 0 \end{cases}$$

In other words, the absolute value of a positive number or zero is the same as that number; the absolute value of a negative number is the additive inverse (negative of) of that number. The following C++ expression represents the *absolute value* of the variable `n`:

```
(n < 0) ? -n : n
```

Some argue that the conditional operator is cryptic, and thus its use reduces a program's readability. To seasoned C++ programmers it is quite understandable, but it is used sparingly because of its very specific nature.

7.3 The do/while Statement

An executing program checks the condition of a `while` statement (Section 6.1) before executing any of the statements in its body; thus, we say a `while` loop is a *top-checking* loop. Sometimes this sequence of checking the condition first then executing the body is inconvenient; for example, consider Listing 7.2 (goodinputonly.cpp).

Listing 7.2: `goodinputonly.cpp`

```
#include <iostream>

int main() {
    int in_value = -1;
    std::cout << "Please enter an integer in the range 0-10: ";
    // Insist on values in the range 0...10
    while (in_value < 0 || in_value > 10)
        std::cin >> in_value;
    // in_value at this point is guaranteed to be within range
    std::cout << "Legal value entered was " << in_value << '\n';
}
```

The loop in Listing 7.2 (goodinputonly.cpp) traps the user in the `while` until the user provides a number in the desired range. Here's how it works:

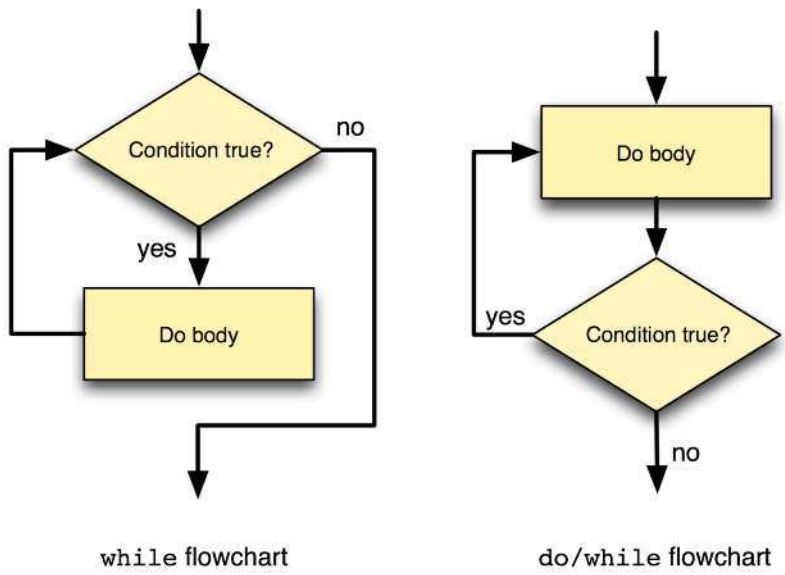
- The condition of the `while` specifies a set that includes all values that are *not* in the desired range. The initialization of `in_value` to -1 ensures the condition of the `while` will be true initially, and, thus, the program always will execute the loop's body at least one time.
- The user does not get a chance to enter a value until program's execution is inside the loop.
- The only way the user can escape the loop is to enter a value that violates the condition—precisely a value in the desired range.

The initialization of `in_value` before the loop check is somewhat artificial. It is there only to ensure entry into the loop's body. It seems unnatural to check for a valid value *before* the user gets a chance to enter it. A loop that checks its condition after its body is executed at least once would be more appropriate. The `do`/`while` statement is a *bottom-checking* loop that behaves exactly in this manner. Listing 7.3 (betterinputonly.cpp) uses a `do`/`while` statement to check for valid input.

Listing 7.3: `betterinputonly.cpp`

```
#include <iostream>

int main() {
    int in_value;
    std::cout << "Please enter an integer in the range 0-10: ";
```

Figure 7.1 The flowcharts for while and do/while loops




```
    // Insist on values in the range 0...10
    do
        std::cin >> in_value;
    while (in_value < 0 || in_value > 10);
    // in_value at this point is guaranteed to be within range
    std::cout << "Legal value entered was " << in_value << '\n';
}
```

Notice that there is no need to initialize `in_value` since its value is not used until after it is assigned through the input stream `std::cin`. Figure 7.1 compares the flowcharts of a `while` and `do/while` loop.

The `do/while` statement has the general form:

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

- The reserved words `do` and `while` identify a `do/while` statement. The `do` and `while` keywords

delimit the loop's body, but curly braces are still required if the body consists of more than one statement.

- The *condition* is associated with the `while` at the end of the loop. The *condition* is a Boolean expression and must be enclosed within parentheses.
- The *statement* is exactly like the *statement* in the general form of the `while` loop (see Section 6.1). It can be a compound statement enclosed within curly braces.

The body of a `do/while` statement, unlike the `while` statement, is guaranteed to execute at least once.

The `do/while` loop is a convenience to the programmer and is not an essential programming construct. It is easy to transform any code that uses a `do/while` statement into code that behaves identically that uses a `while` statement instead. In practice, programmers use `while` loops much more frequently than `do/while` loops because more algorithms require top-checking loops than bottom-checking loops. The `do/while` statement is included in C++ for a reason, however. Transforming an algorithm that can be expressed more naturally with a bottom-checking loop into one the uses a top-checking loop can lead to awkward code. Use `do/while` when appropriate.

7.4 The for Statement

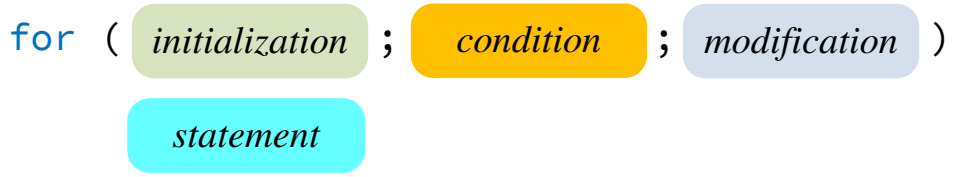
Recall Listing 6.2 (iterativecounttofive.cpp). It simply counts from one to five. Counting is a frequent activity performed by computer programs. Certain program elements are required in order for any program to count:

- A variable must be used to keep track of the count; in Listing 6.2 (iterativecounttofive.cpp), `count` is the aptly named counter variable.
- The counter variable must be given an initial value. In the case of Listing 6.2 (iterativecounttofive.cpp), the initial value is 1.
- The variable must be modified (usually incremented) as the program counts. The statement

  ```
  count++;
  ```

  increments `count` in Listing 6.2 (iterativecounttofive.cpp).
- A way must be provided to determine if the counting has completed. In Listing 6.2 (iterativecounttofive.cpp), the condition of the `while` statement determines if the counting is complete or must contine.

C++ provides a specialized loop that packages these four programming elements into one convenient statement. Called the `for` statement, its general form is

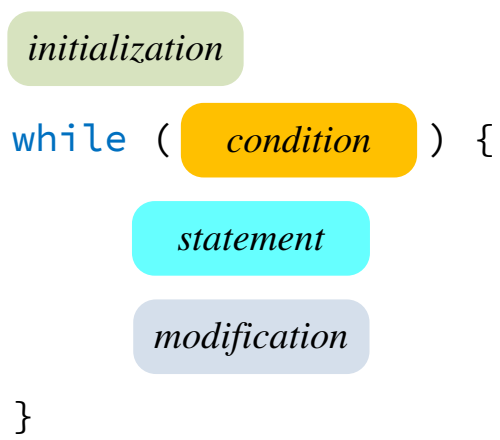


- The reserved word `for` identifies a `for` statement.
- The loop is controlled by a special variable called the *loop variable*.
- The header, contained in parentheses, contains three parts, each separated by semicolons:
  - **Initialization**. The *initialization* part assigns an initial value to the loop variable. The loop variable may be declared here as well; if it is declared here, then its scope is limited to the `for` statement. This means you may use that loop variable only within the loop. It also means you are free to reuse that variable's name outside the loop to declare a different variable with the same name as the loop variable.

    The initialization part is performed one time.
  - **Condition**. The *condition* part is a Boolean expression, just like the condition of a `while` statement. The condition is checked each time *before* the body is executed.
  - **Modification**. The *modification* part generally changes the loop variable. The change should be such that the condition will eventually become false so the loop will terminate. The modification is performed during each iteration *after* the body is executed.

    Notice that the last part (*modification*) is not following by a semicolon; semicolons are used strictly to separate the three parts.
- The *statement* is like the body of any other loop. It may be a compound statement within curly braces.

Any `for` loop can be rewritten as a `while` loop. The general form of the `for` loop given above can be written equivalently as



Listing 7.4 (forcounttofive.cpp) uses a `for` statement to count to five.

Listing 7.4: forcounttofive.cpp

```
#include <iostream>

int main() {
    for (int count = 1; count <= 5; count++)
        std::cout << count << '\n';  // Display counter
}
```

With a `while` loop, the four counting components (variable declaration, initialization, condition, and modification can be scattered throughout the code. With a `for` loop, a programmer should be able to determine all the important information about the loop's control by looking at one statement.

Recall Listing 6.13 (timestable.cpp) that prints a multiplication table on the screen. We can organize its code better by converting all the `while` statements to `for` statements. The result uses far less code, as shown in Listing 7.5 (bettertimestable.cpp).

Listing 7.5: `bettertimestable.cpp`

```
#include <iostream>
#include <iomanip>

int main() {
    int size;  // The number of rows and columns in the table
    std::cout << "Please enter the table size: ";
    std::cin >> size;
    // Print a size x size multiplication table

    // First, print heading
    std::cout << "      ";
    for (int column = 1; column <= size; column++)
        std::cout << std::setw(4) << column; // Print heading for this column.
    std::cout << '\n';
    // Print line separator
    std::cout << "     +";
    for (int column = 1; column <= size; column++)
        std::cout << "----";         // Print separator for this column.
    std::cout << '\n';
    // Print table contents
    for (int row = 1; row <= size; row++) {
        std::cout << std::setw(4) << row << " |";  // Print row label.
        for (int column = 1; column <= size; column++)
            std::cout << std::setw(4) << row*column;   // Display product
        std::cout << '\n';                 // Move cursor to next row
    }
}
```

A `for` loop is ideal for stepping through the rows and columns. The information about the control of both loops is now packaged in the respective `for` statements instead of being spread out in various places in `main`. In the `while` version, it is easy for the programmer to forget to update one or both of the counter variables (`row` and/or `column`). The `for` makes it harder for the programmer to forget the loop variable update, since it is done right up front in the `for` statement header.

It is considered bad programming practice to do either of the following in a `for` statement:

- **Modify the loop control variable within the body of the loop**—if the loop variable is modified within the body, then the logic of the loop's control is no longer completely isolated to the `for` statement's header. The programmer must look elsewhere within the statement to understand completely how the loop works.

- **Prematurely exit the loop with a `break`**—this action also violates the concept of keeping all the loop control logic in one place (the `for`'s header).

The language allows both of these practices, but experience shows that it is best to avoid them. If it seems necessary to violate this advice, consider using a different kind of loop. The `while` and `do`/`while` loops do not imply the same degree of control regularity expected in a `for` loop.

Listing 7.6 (permuteabcd.cpp) is a rewrite of Listing 6.14 (permuteabc.cpp) that replaces its `while` loops with `for` loops and adds an additional character.

Listing 7.6: `permuteabcd.cpp`

```
// File permuteabcd.cpp

#include <iostream>

int main() {
  for (char first = 'A'; first <= 'D'; first ++)
    for (char second = 'A'; second <= 'D'; second++)
      if (second != first)  // No duplicate letters
        for (char third = 'A'; third <= 'D'; third++)
          if (third != first && third != second)
             for (char fourth = 'A'; fourth <= 'D'; fourth++)
                if (fourth != first && fourth != second && fourth != third)
                    std::cout << first << second << third << fourth << '\n';
}
```

Notice that since all the variable initialization and incrementing is taken care of in the `for` statement headers, we no longer need compound statements in the loop bodies, so the curly braces are unnecessary. Listing 7.6 (permuteabcd.cpp) prints all 24 permutations of ABCD:

```
ABCD
ABDC
ACBD
ACDB
ADBC
ADCB
BACD
BADC
BCAD
BCDA
BDAC
BDCA
CABD
CADB
CBAD
CBDA
CDAB
CDBA
DABC
DACB
DBAC
DBCA
DCAB
DCBA
```

Listing 7.7 (forprintprimes.cpp) is a rewrite of Listing 6.22 (printprimes.cpp) that replaces its `while` loops with `for` loops.

Listing 7.7: forprintprimes.cpp

```
#include <iostream>

int main() {

    int max_value;
    std::cout << "Display primes up to what value? ";
    std::cin >> max_value;
    for (int value = 2; value <= max_value; value++) {
        // See if value is prime
        bool is_prime = true;  // Provisionally, value is prime
        // Try all possible factors from 2 to value - 1
        for (int trial_factor = 2;
              is_prime && trial_factor < value;
              trial_factor++)
            is_prime = (value % trial_factor != 0);
        if (is_prime)
            std::cout << value <<  " ";   // Display the prime number
    }
    std::cout << '\n';  // Move cursor down to next line
}
```

As shown in Listing 7.7 (forprintprimes.cpp), the conditional expression in the `for` loop is not limited to a simple test of the loop control variable; it can be any legal Boolean expression. Programmers can use the logical *and* (&&), *or* (||), and *not* (!) operators to create complex Boolean expressions, if necessary. The modification part of the `for` loop is not limited to simple arithmetic and can be quite elaborate. For example:

```
for (double d  = 1000; d >= 1; std::cin >> d) {
    /*  Body goes here  */
}
```

Here d is reassigned from the input stream. If necessary, multiple variables can be initialized in the initialization part:

```
for (int i = 0, j = 100; i < j; i++) {
    /*  Body goes here  */
}
```

While the `for` statement supports such complex headers, simpler is usually better. Ordinarily the `for` loop should manage just one control variable, and the initialization, condition, and modification parts should be straightforward. If a particular programming situation warrants an overly complicated `for` construction, consider using another kind of loop.

Any or all of the parts of the `for` statement (initialization, condition, modification, and body) may be omitted:

- **Initialization**. If the initialization is missing, as in

  ```
  for (; i < 10; i++)
      /*  Body goes here  */
  ```

  then no initialization is performed by the `for` loop, and it must be done elsewhere.

- **Condition**. If the condition is missing, as in

```
for (int i = 0; ; i++)
    /*  Body goes here  */
```

  then the condition is true by default. A `break` or `goto` must appear in the body unless an infinite loop is intended.

- **Modification**. If the modification is missing, as in

```
for (int i = 0; i < 10; )
    /*  Body goes here  */
```

  then the `for` performs no automatic modification; the modification must be done by a statement in the body to avoid an infinite loop.

- **Body**. An empty body, as in

```
for (int i = 0; i < 10; i++) {}
```

  or

```
for (int i = 0; i < 10; i++);
```

  results in an empty loop. Some programmers use an empty loop to produce a non-portable delay in the program's execution. A programmer may, for example, need to slow down a graphical animation. Such an attempt using an empty loop is non-portable for several reasons. If the program actually executes the loop, slower computers will delay longer than faster computers. The timing of the program's delay will differ from one computer to another. Worse yet, some compilers may detect that such code has no functional effect and optimize away the empty loop. This means the compiler will ignore the `for` statement altogether.

As mentioned in Section 5.3, be careful about accidentally putting a semicolon at the end of the `for` header, as in

```
for (int i = 0; i < 10; i++);
    /*  Intended body goes here  */
```

The semicolon terminates the `for` statement, and the intended body that follows is not the body, even though it may be properly indented.

One common C/C++ idiom to make an intentional infinite loop is to use a `for` statement with all control information missing:

```
for ( ;; )
    /*  Body goes here  */
```

Omitting all the parts of the `for` header is a statement from the programmer that says "I know what I am doing—I really want an infinite loop here." In reality the loop may not be infinite at all; its body could contain a `break` or `goto` statement.

While the `for` statement supports the omission of parts of its header, such constructs should be avoided. The intention of the `for` loop is to allow the programmer to see all the aspects of the loop's control in one place. If some of these control responsibilities are to be handled elsewhere (not in the `for`'s header) then consider using another kind of loop.

Programmers usually select a simple name for the control variable of a `for` statement. Recall that variable names should be well chosen to reflect the meaning of their use within the program. It may come as a surprise that `i` is probably the most common name used for an integer control variable in a `for` loop. This practice has its roots in mathematics where variables such as i, j, and k are commonly used to index vectors and matrices. Such mathematical structures have programming analogs in arrays and `vector`s, which we explore in Chapter 11. Computer programmers make considerable use of `for` loops in array and vector processing, so programmers have universally adopted this convention of short control variable names. Thus, it generally is acceptable to use simple identifiers like `i` as loop control variables.

C++ allows the `break`, `continue`, and `goto` statements to be used in the body of a `for` statement. Like with the `while` and `do`/`while` statements, `break` causes immediate loop termination, `continue` causes the condition to be immediately checked to determine if the iteration should continue, and `goto` jumps to a label somewhere in the function. As previously mentioned, however, `for` loop control should be restricted to its header, and the use of `break`, `continue`, and `goto` within `for` loops should be avoided.

Any `for` loop can be rewritten with a `while` loop and behave identically. For example, consider the `for` loop

```
for (int i = 1; i <= 10; i++)
    std::cout << i << '\n';
```

and next consider the `while` loop that behaves exactly the same way:

```
int i = 1;
while (i <= 10) {
    std::cout << i << '\n';
    i++;
}
```

Which is better? The `for` loop conveniently packages the loop control information in its header, but in the `while` loop this information is distributed throughout the small section of code. The `for` loop thus provides a better organization of the loop control code. Does one loop outperform the other? No, most compilers produce essentially the same code for both constructs. Thus, the `for` loop is preferred in this example.

7.5 Exercises

1. Consider the following code fragment.

```
int x;
std::cin >> x;
switch (x + 3) {
  case  5:
    std::cout << x << '\n';
    break;
  case 10:
    std::cout << x - 3 << '\n';
    break;
  case 20:
    std::cout << x + 3 << '\n';
    break;
}
```

(a) What is printed when the user enters 2?

(b) What is printed when the user enters 5?

(c) What is printed when the user enters 7?

(d) What is printed when the user enters 17?

(e) What is printed when the user enters 20?

2. Consider the following code fragment.

```
char ch;
std::cin >> ch;
switch (ch) {
  case 'a':
    std::cout << "*\n";
    break;
  case 'A':
    std::cout << "**\n";
    break;
  case 'B':
  case 'b':
    std::cout << "***\n";
  case 'C':
  case 'c':
    std::cout << "****\n";
    break;
  default:
    std::cout << "*****\n";
}
```

(a) What is printed when the user enters a?

(b) What is printed when the user enters A?

(c) What is printed when the user enters b?

(d) What is printed when the user enters B?

(e) What is printed when the user enters C?

(f) What is printed when the user enters c?

(g) What is printed when the user enters t?

3. What is printed by the following code fragment?

```
int x = 0;
do {
    std::cout << x << " ";
    x++;
} while (x < 10);
std::cout << '\n';
```

4. What is printed by the following code fragment?

```
int x = 20;
do {
    std::cout << x << " ";
    x++;
} while (x < 10);
std::cout << '\n';
```

5. What is printed by the following code fragment?

```
for (int x = 0; x < 10; x++)
    std::cout << "*";
std::cout << '\n';
```

6. Rewrite the following code fragment so that a `switch` is used instead of the `if`/`else` statements.

```
int value;
char ch;
std::cin >> ch;
if (ch == 'A')
    value = 10;
else if (ch == 'P')
    value = 20;
else if (ch == 'T')
    value = 30;
else if (ch == 'V')
    value = 40;
else
    value = 50;
std::cout << value << '\n';
```

7. Rewrite the following code fragment so that a multi-way `if`/`else` is used instead of the `switch` statement.

```
int value;
char ch;
std::cin >> ch;
switch( ch) {
case 'A':
    value = 10;
    break;
case 'P':
    std::cin >> value;
    break;
case 'T':
    value = ch;
    break;
case 'V':
    value = ch + 1000;
    break;
default:
    value = 50;
}
```

```
std::cout << value << '\n';
```

8. Rewrite the following code fragment so that a multi-way `if`/`else` is used instead of the `switch` statement.

```
int value;
char ch;
std::cin >> ch;
switch (ch) {
case 'A':
    std::cout << ch << '\n';
    value = 10;
    break;
case 'P':
case 'E':
    std::cin >> value;
    break;
case 'T':
    std::cin >> ch;
    value = ch;
case 'C':
    value = ch;
    std::cout << "value=" << value << ", ch=" << ch << '\n';
    break;
case 'V':
    value = ch + 1000;
    break;
}
std::cout << value << '\n';
```

9. Rewrite the following code fragment so a `while` loop is used instead of the `for` statement.

```
for (int i = 100; i > 0; i--)
    std::cout << i << '\n';
```

10. Rewrite the following code fragment so that it uses the conditional operator instead of an `if` statement:

```
if (value % 2 != 0)   // Is value even?
    value = value + 1;  // If not, make it even.
```

11. Rewrite the following code fragment so that it uses the conditional operator instead of an `if`/`else` statement:

```
if (value % 2 == 0)   // Is value even?
    value = 0;          // If so, make it zero.
else
    value = value + 1;  // Otherwise, make it even.
```

12. Would the following multi-way `if`/`else` be a good candidate to rewrite as a `switch` statement? If so, rewrite the code using a `switch`; otherwise, explain why it is impractical to do so.

```
int x, y;
std::cin >> x >> y;
if (x < 10)
    y = 10;
else if (x == 5)
    y = 5;
else if (x == y)
    y = 0;
else if (y > 10)
    x = 10;
else
    x = y;
```

Chapter 8

Using Functions

Suppose you must write a C++ program that computes the square root of a number supplied by the user. Listing 8.1 (computesquareroot.cpp) provides a simple implementation.

Listing 8.1: computesquareroot.cpp

```
// File squareroot.cpp

#include <iostream>

int main() {
    double input;

    // Get value from the user
    std::cout << "Enter number: ";
    std::cin >> input;
    double diff;
    // Compute a provisional square root
    double root = 1.0;

    do {  // Loop until the provisional root
          // is close enough to the actual root
        root = (root + input/root) / 2.0;
        std::cout << "root is " << root << '\n';
        // How bad is the approximation?
        diff = root * root - input;
    }
    while (diff > 0.0001 || diff < -0.0001);

    // Report approximate square root
    std::cout << "Square root of " << input << " = " << root << '\n';
}
```

The program is based on a simple algorithm, Newton's Method, that uses successive approximations to zero in on an answer that is within 0.0001 of the true answer.

One sample run is

```
Enter number: 2
```

```
root is 1.5
root is 1.41667
root is 1.41422
Square root of 2 = 1.41422
```

The actual square root is approximately 1.4142135623730951 and so the result is within our accepted tolerance (0.0001). Another run is

```
Enter number: 100
root is 50.5
root is 26.2401
root is 15.0255
root is 10.8404
root is 10.0326
root is 10.0001
root is 10
Square root of 100 = 10
```

which is, of course, the exact answer.

While this code may be acceptable for many applications, better algorithms exist that work faster and produce more precise answers. Another problem with the code is this: What if you are working on a significant scientific or engineering application and must use different formulas in various parts of the source code, and each of these formulas involve square roots in some way? In mathematics, for example, you use square root to compute the distance between two geometric points (x_1, y_1) and (x_2, y_2) as

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

and, using the quadratic formula, the solution to the equation $ax^2 + bx + c = 0$ is

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

In electrical engineering and physics, the root mean square of a set of values $\{a_1, a_2, a_3, \ldots, a_n\}$ is

$$\sqrt{\frac{a_1^2 + a_2^2 + a_3^2 + \ldots + a_n^2}{n}}$$

Suppose we are writing one big program that, among many other things, needs to compute distances and solve quadratic equations. Must we copy and paste the relevant portions of the square root code in Listing 8.1 (computesquareroot.cpp) to each location in our source code that requires a square root computation? Also, what if we develop another program that requires computing a root mean square? Will we need to copy the code from Listing 8.1 (computesquareroot.cpp) into every program that needs to compute square roots, or is there a better way to package the square root code and reuse it?

Code is made reusable by packaging it in *functions*. A function is a unit of reusable code. In Chapter 9 we will write our own reusable functions, but in this chapter we examine some of the functions available in the C++ standard library. C++ provides a collection of standard precompiled C and C++ code stored in libraries. Programmers can use parts of this library code within their own code to build sophisticated programs.

Figure 8.1 Conceptual view of the square root function

8.1 Introduction to Using Functions

In mathematics, a *function* computes a result from a given value; for example, from the function definition $f(x) = 2x + 3$, we can compute $f(5) = 13$ and $f(0) = 3$. A function in C++ works like a mathematical function. To introduce the function concept, we will look at the standard C++ function that implements mathematical square root.

In C++, a function is a named sequence of code that performs a specific task. A program itself consists of a collection of functions. One example of a function is the mathematical square root function. Such a function, named `sqrt`, is available to C and C++ programs (see Section 8.2). The square root function accepts one numeric value and produces a `double` value as a result; for example, the square root of 16 is 4, so when presented with 16.0, `sqrt` responds with 4.0. Figure 8.1 visualizes the square root function.

For the programmer using the `sqrt` function within a program, the function is a black box; the programmer is concerned more about *what* the function does, not *how* it does it.

This `sqrt` function is exactly what we need for our square root program, Listing 8.1 (computesquareroot.cpp). The new version, Listing 8.2 (standardsquareroot.cpp), uses the library function `sqrt` and eliminates the complex logic of the original code.

Listing 8.2: `standardsquareroot.cpp`

```
#include <iostream>
#include <cmath>

int main() {
    double input;

    // Get value from the user
    std::cout << "Enter number: ";
    std::cin >> input;

    // Compute the square root
    double root = sqrt(input);

    // Report result
    std::cout << "Square root of " << input << " = " << root << '\n';
}
```

The line

```
#include <cmath>
```

directs the preprocessor to augment our source code with the declarations of a collection of mathematical functions in the cmath library. The `sqrt` function is among them. Table 8.1 lists some of the other commonly used mathematical functions available in the cmath library. The compiler needs this augmented code so it can check to see if we are using the `sqrt` function properly.

The expression

```
sqrt(input)
```

is a *function invocation*, also known as a *function call*. A function provides a service to the code that uses it. Here, our `main` function is the *caller*[1] that uses the service provided by the `sqrt` function. We say `main` calls, or invokes, `sqrt` passing it the value of `input`. The expression `sqrt(input)` evaluates to the square root of the value of the variable `input`. Behind the scenes—inside the black box as it were—precompiled C code uses the value of the `input` variable to compute its square root. There is nothing special about this precompiled C code that constitutes the `sqrt` function; it was written by a programmer or team of programmers working for the library vendor using the same tools we have at our disposal. In In Chapter 9 we will write our own functions, but for now we will enjoy the functions that others have provided for us.

When calling a function, a pair of parentheses follow the function's name. Information that the function requires to perform its task must appear within these parentheses. In the expression

```
sqrt(input)
```

`input` is the information the function needs to do its work. We say `input` is the *argument*, or *parameter*, passed to the function. We also can say "we are passing `input` to the `sqrt` function."

While we might say "we are passing `input` to the `sqrt` function," the program really is not giving the function access to `main`'s `input` variable. The `sqrt` function itself cannot change the value of `main`'s `input` variable, it simply uses the variable's value to perform the computation.

The following simple analogy may help explain how the communication works between `main` and `sqrt`, The `main` function has work to do, but instead of doing all the work itself, it delegates some of the work (in this case the hard part) to `sqrt`. When `main` needs to compute the square root of `input`, it writes down the value of its `input` variable on a piece of paper and hands it to `sqrt`. `main` then sits idly until `sqrt` finishes its work. The `sqrt` function accepts `main`'s note and begins working on the task (computing the square root of the number on the note `main` gave it). When it is finished, `sqrt` does two things: `sqrt` hands back to `main` a different piece of paper with the answer, and `sqrt` throws away the piece of paper `main` originally passed to it. When `main` receives the note from `sqrt` it uses the information on the note and then discards the note. The `main` function then can continue with its other business.

The `sqrt` function thus has no access to `main`'s original `input` variable; it has only a copy of `input`, as if "written on a piece of paper." (Similarly, if the `sqrt` function uses any variables to do its work, `main` is oblivious to them and has no way to access them.) After `sqrt` is finished and returns to `main` its computed answer, `sqrt` discards its copy of `input` (by analogy, the function "throws away" the paper with the copy of `input` that `main` gave it). Thus, during a function call the parameter is a temporary, transitory value used only to communicate information to the function. The parameter lives only as long as the function is executing.

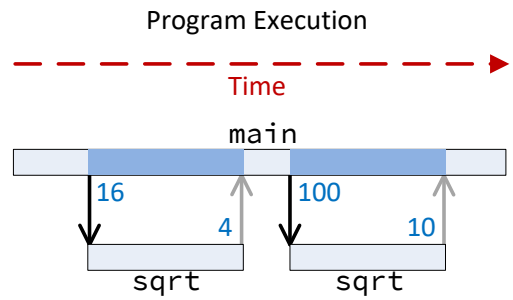
Figure 8.2 illustrates a program's execution involving simple function calls.

Figure 8.2 shows that a program's execution begins in its `main` function. Here `main` calls the `sqrt` function twice. A vertical bar represents the time that a function is *active*, or "alive." A function's variables

[1] The term *client* can be used as well, although we reserve the term *client* for code that interacts with objects (see Chapter 13).

Figure 8.2 The diagram on the right visualizes the execution of the program on the left. Time flows from left to right. A rectangular bar represents the time that a function is active. A C++ program's execution begins with its **`main`** function. Here, **`main`** calls the **`sqrt`** function twice. The shaded parts of **`main`**'s bar shows the times **`main`** has to wait for **`sqrt`** to complete.

```
int main() {
    double value;

    //  Assign variable
    value = 16;

    //  Compute s square root
    double root = sqrt(value);

    //  Compute another
    root = sqrt(100);
}
```



exist while a function is active. Observe that the `main` function is active for the duration of the program's execution. The `sqrt` becomes active twice, exactly the two times `main` calls it.

The `sqrt` function can be called in other ways, as Listing 8.3 (usingsqrt.cpp) illustrates:

Listing 8.3: `usingsqrt.cpp`

```
/*
 * This program shows the various ways the
 * sqrt function can be used.
 */

#include <iostream>
#include <cmath>

int main() {
    double x = 16.0;
    // Pass a literal value and display the result
    std::cout << sqrt(16.0) << '\n';
    // Pass a variable and display the result
    std::cout << sqrt(x) << '\n';
    // Pass an expression
    std::cout << sqrt(2 * x - 5) << '\n';
    // Assign result to variable
    double y = sqrt(x);
    // Use result in an expression
    y = 2 * sqrt(x + 16) - 4;
    // Use result as argument to a function call
    y = sqrt(sqrt(256.0));
    std::cout << y << '\n';
}
```

Figure 8.3 Conceptual view of the maximum function

The `sqrt` function accepts a single numeric argument. The parameter that a caller can pass to `sqrt` can be a literal number, a numeric variable, an arithmetic expression, or even a function invocation that produces an acceptable numeric result.

Some C++ functions, like `sqrt`, compute a value and return it to the caller. The caller can use this result in various ways, as shown in Listing 8.3 (usingsqrt.cpp). The next to the last statement passes the result of calling `sqrt` to `sqrt`, thereby computing $\sqrt{\sqrt{256}}$, which is 4.

If the caller code attempts to pass a parameter to the function that is incompatible with the type expected by the function, the compiler will issue an error.

```
std::cout << sqrt("16") << '\n'; // Illegal, a string is not a number
```

The compiler is able to determine that the above statement is illegal based on the additional information the preprocessor added to the code via the

```
#include<cmath>
```

directive.

Listing 8.3 (usingsqrt.cpp) shows that a program can call the `sqrt` function as many times and in as many places as needed. As noted in Figure 8.1, to the caller of the square root function, the function is a black box; the caller is concerned strictly about *what* the function does, not *how* the function accomplishes its task.

We safely can treat all functions as black boxes. We can use the service that a function provides without being concerned about its internal details. We are guaranteed that we can influence the function's behavior only via the parameters that we pass, and that nothing else we do can affect what the function does or how it does it. Furthermore, the function cannot affect any of our code, apart from what we do with the value it computes.

Some functions take more than one parameter; for example, the C++ `max` function requires two arguments in order to produce a result. The `max` function selects and returns the larger of the two parameters. The `max` function is visualized in Figure 8.3.

The `max` function could be used as

```
std::cout << "The larger of " << 4 << " and " << 7
          << " is " << max(4, 7) << '\n';
```

Notice that the parameters are contained in parentheses following the function's name, and the parameters are separated by commas.

From the caller's perspective a function has three important parts:

- **Name**. Every function has a name that identifies the location of the code to be executed. Function names follow the same rules as variable names; a function name is another example of an identifier (see Section 3.3).
- **Parameter type(s)**. A caller must provide the exact number and types of parameters that a function expects. If a caller attempts to call a function with too many or too few parameters, the compiler will issue an error message and not compile the code. Similarly, if the caller passes parameters that are not compatible with the types specified for the function, the compiler will report appropriate error messages.
- **Result type**. A function can compute a result and return this value to the caller. The caller's use of this result must be compatible with the function's specified result type. The result type returned to the caller and the parameter types passed in by the caller can be completely unrelated.

These three crucial pieces of information are formally described for each function in a specification known as a function *prototype*. The prototype for the `sqrt` function is

```
double sqrt(double)
```

and the `max` function's prototype can be expressed as [2]

```
int max(int, int)
```

In a function prototype, the return type is listed first, followed by the function's name, and then the parameter types appear in parentheses. Sometimes it is useful to list parameter names in the function's prototype, as in

```
double sqrt(double n)
```

or

```
int max(int a, int b)
```

The specific parameter names are irrelevant. The names make it easier to describe what the function does; for example, `sqrt`computes the square root of `n` and `max` determines the larger of `a` and `b`.

When using a library function the programmer must include the appropriate `#include` directive in the source code. The file specified in an `#include` directive contains prototypes for library functions. In order to use the `sqrt` function, a program must include the

```
#include <cmath>
```

preprocessor directive. For the `max` function, include

```
#include <algorithm>
```

although under Visual C++, includeing `<iostream>` is sufficient.

Armed with the function prototypes, the compiler can check to see if the calling code is using the library functions correctly. An attempt to use the `sqrt` function as

```
std::cout << sqrt(4.0, 7.0) << '\n';  // Error
```

[2] The prototype for the actual library function max uses *generic types*; generic types are beyond the scope of this introductory book, so the prototype provided here is strictly for illustrating a point.

will result in an error because the prototype for `sqrt` specifies only one numeric parameter, not two.

Some functions do not accept parameters; for example, the C++ function to generate a pseudorandom number, `rand`, is called with no arguments:

```
std::cout << rand() << '\n';
```

The `rand` function returns an `int` value, but the caller does not pass the function any information to do its task. The `rand` prototype is

```
int rand()
```

Notice the empty parentheses that indicate this function does not accept any parameters.

Unlike mathematical functions that must produce a result, C++ does not require a function to return a value to its caller. The C++ function `exit` expects an integer value from the caller, but it does not return a result back to the caller. A prototype for a function that returns nothing uses `void` as the return type, as in:

```
void exit(int);
```

The `exit` function immediately terminates the program's execution. The integer argument passed to `exit` is returned to the operating system which can use the value to determine if the program terminated normally or due to an error. C++ programs automatically return zero when `main` finishes executing—no `exit` call is necessary.

Note that since `exit` does not return a value to the caller, code such as

```
std::cout << exit(8) << '\n';  // Illegal!
```

will not compile since the expression `exit(8)` evaluates to nothing, and the `std::cout` stream object requires an actual value of some kind to print. A `void` function is useful for the *side effects* it produces instead a value it computes. Example side effects include printing something on the console, sending data over a network, or animating a graphical image.

8.2 Standard Math Functions

The `cmath` library provides much of the functionality of a scientific calculator. Table 8.1 lists only a few of the available functions.

mathfunctions Module
`double sqrt(double x)` Computes the square root of a number: `sqrt(x)` = $\sqrt{x}$
`double exp(double x)` Computes e raised a power: `exp(x)` = e^x
`double log(double x)` Computes the natural logarithm of a number: `log(x)` = $\log_e x = \ln x$
`double log10(double x)` Computes the common logarithm of a number: `log(x)` = $\log_{10} x$
`double cos(double)` Computes the cosine of a value specified in radians: `cos(x)` = $\cos x$; other trigonometric functions include sine, tangent, arc cosine, arc sine, arc tangent, hyperbolic cosine, hyperbolic sine, and hyperbolic tangent
`double pow(double x, double y)` Raises one number to a power of another: `pow(x,y)` = x^y
`double fabs(double x)` Computes the absolute value of a number: `fabs(x)` = $\lvert x \rvert$

Table 8.1: A few of the functions from the `cmath` library

The `cmath` library also defines a constant named `HUGE_VAL`. Programmers can use this constant to represent infinity or an undefined value such the slope of a vertical line or a fraction with a zero denominator. A complete list of the numeric functions available to C++ can be found at `http://www.cplusplus.com/reference/clibrary/cmath/`.

Be careful to put the function's arguments in the proper order when calling a function; for example, the call `pow(10,2)` computes $10^2 = 100$, but the call `pow(2,10)` computes $2^{10} = 1,024$.

A C++ program that uses any of the functions from the `cmath` library must use the following preprocessor `#include` directive:

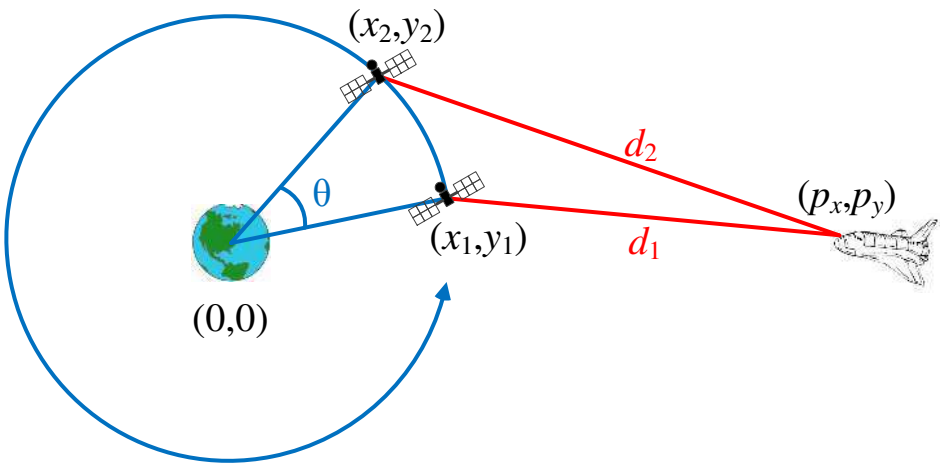
```
#include <cmath>
```

Functions in the `cmath` library are ideal for solving problems like the one shown in Figure 8.4. Suppose a spacecraft is at a fixed location in space relative to some planet. The spacecraft's distance to the planet, therefore, also is fixed. A satellite is orbiting the planet in a circular orbit. We wish to compute how much farther away the satellite will be from the spacecraft when it has progressed 10 degrees along its orbital path.

We will let the origin of our coordinate system (0,0) be located at the center of the planet which corresponds also to the center of the circular orbital path. The satellite is initially at point (x_1, y_1) and the spacecraft is stationary at point (p_x, p_y). The spacecraft is located in the same plane as the satellite's orbit. We need to compute the difference in the distances between the moving point (satellite) and the fixed point (spacecraft) at two different times during the satellite's orbit.

Facts from mathematics provide solutions to the following two problems:

Figure 8.4 Orbital distance problem. In this diagram, the satellite begins at point (x_1, y_1), a distance of d_1 from the spacecraft. The satellite's orbit takes it to point (x_2, y_2) after an angle of θ rotation. The distance to its new location is d_2.



1. **Problem**: We must recompute the location of the moving point as it moves along the circle.

   Solution: Given an initial position (x_1, y_1) of the moving point, a rotation of θ degrees around the origin will yield a new point at (x_2, y_2), where

$$\begin{aligned} x_2 &= x_1 \cos\theta - y_1 \sin\theta \\ y_2 &= x_1 \sin\theta + y_1 \cos\theta \end{aligned}$$

2. **Problem**: The distance between the moving point and the fixed point must be recalculated as the moving point moves to a new position.

   Solution: The distance d_1 in Figure 8.4 between two points (p_x, p_y) and (x_1, y_1) is given by the formula

$$d_1 = \sqrt{(x_1 - p_x)^2 + (y_1 - p_y)^2}$$

   Similarly, the distance d_2 in Figure 8.4 is

$$d_2 = \sqrt{(x_2 - p_x)^2 + (y_2 - p_y)^2}$$

Listing 8.4 (orbitdist.cpp) uses these mathematical results to compute the difference in the distances.

Listing 8.4: orbitdist.cpp

```
#include <iostream>
#include <cmath>

int main() {
    // Location of orbiting point is (x,y)
    double x;   // These values change as the
```

```
    double y;   // satellite moves
    const double PI = 3.14159;

    // Location of fixed point is always (100, 0),
    // AKA (p_x, p_y).  Change these as necessary.
    const double p_x = 100;
    const double p_y = 0;

    // Radians in 10 degrees
    const double radians = 10 * PI/180;

    // Precompute the cosine and sine of 10 degrees
    const double COS10 = cos(radians);
    const double SIN10 = sin(radians);

    // Get starting point from user
    std::cout << "Enter initial satellite coordinates (x,y):";
    std::cin >> x >> y;

    // Compute the initial distance
    double d1 = sqrt((p_x - x)*(p_x - x) + (p_y - y)*(p_y - y));

    // Let the satellite orbit 10 degrees
    double x_old = x;  // Remember x's original value
    x = x*COS10 - y*SIN10;  // Compute new x value
    // x's value has changed, but y's calculate depends on
    // x's original value, so use x_old instead of x.
    y = x_old*SIN10 + y*COS10;

    // Compute the new distance
    double d2 = sqrt((p_x - x)*(p_x - x) + (p_y - y)*(p_y - y));

    // Print the difference in the distances
    std::cout << "Difference in distances: " << d2 - d1 << '\n';
}
```

We can use the square root function to improve the efficiency of our primes program. Instead of trying all the factors of n up to $n-1$, we need only try potential factors up to the square root of n. Listing 8.5 (moreefficientprimes.cpp) uses the `sqrt` function to reduce the number of factors that need be considered.

Listing 8.5: `moreefficientprimes.cpp`

```
#include <iostream>
#include <cmath>

int main() {

    int max_value;
    std::cout << "Display primes up to what value? ";
    std::cin >> max_value;
    for (int value = 2; value <= max_value; value++) {
        // See if value is prime
        bool is_prime = true;  // Provisionally, value is prime
        double r = value, root = sqrt(r);
        // Try all possible factors from 2 to the square
```

```
        // root of value
        for (int trial_factor = 2;
              is_prime && trial_factor <= root; trial_factor++)
            is_prime = (value % trial_factor != 0);
        if (is_prime)
            std::cout << value <<  " ";   // Display the prime number
    }
    std::cout << '\n';  // Move cursor down to next line
}
```

The `sqrt` function comes in three forms:

```
double sqrt(double)
float sqrt(float)
long double sqrt(long double)
```

The function names are the same, but the parameter types differ. We say that the `sqrt` function is *overloaded*. (Overloaded functions are covered in more detail in Section 10.3.) When a caller invokes the `sqrt` function, the compiler matches the call to the closest matching prototype. If the caller passes a `double` parameter, the compiler generates code to call the `double` version. If the caller instead passes a `float` variable, the compiler selects the `float` version of `sqrt`. When an `int` is passed to `sqrt`, the compiler cannot decide which version to use, because an `int` can be converted automatically to either a `float`, `double`, or `long double`. The compiler thus needs some help to resolve the ambiguity, so we introduced an additional variable of type `double` so the compiler will use the `double` version of the `sqrt` function. Another option is to use a type cast to convert the integer value into one of the types acceptable to the `sqrt` function.

8.3 Maximum and Minimum

C++ provides standard functions for determining the maximum and minimum of two numbers. Listing 8.6 (maxmin.cpp) exercises the standard `min` and `max` functions.

Listing 8.6: `maxmin.cpp`

```
#include <iostream>
#include <algorithm>

int main() {
    int value1, value2;
    std::cout << "Please enter two integer values: ";
    std::cin >> value1 >> value2;
    std::cout << "max = " << std::max(value1, value2)
              << ", min = " << std::min(value1, value2) << '\n';
}
```

To use the standard `max` and `min` functions in a program you must include the `<algorithm>` header.

8.4 clock Function

The `clock` function from the `<ctime>` library requests from the operating system the amount of time an executing program has been running. The units returned by the call `clock()` is system dependent, but it can be converted into seconds with the constant `CLOCKS_PER_SEC`, also defined in the `ctime` library. Under Visual C++, the `CLOCKS_PER_SEC` constant is 1,000, which means the call `clock()` returns the number of milliseconds that the program has been running.

Using two calls to the `clock` function you can measure *elapsed time*. Listing 8.7 (timeit.cpp) measures how long it takes a user to enter a character from the keyboard.

Listing 8.7: `timeit.cpp`

```
#include <iostream>
#include <ctime>

int main() {
    char letter;
    std::cout << "Enter a character: ";
    clock_t seconds = clock();    // Record starting time
    std::cin >> letter;
    clock_t other = clock();      // Record ending time
    std::cout << static_cast<double>(other - seconds)/CLOCKS_PER_SEC
              << " seconds\n";
}
```

The type `clock_t` is a type defined in the `<ctime>` header file. `clock_t` is equivalent to an `unsigned long`, and you can perform arithmetic on `clock_t` values and variables just as if they are `unsigned long`s. In the expression

```
static_cast<double>(other - seconds)/CLOCKS_PER_SEC
```

the cast is required to force floating-point division; otherwise, the result is truncated to an integer value.

Listing 8.8 (measureprimespeed.cpp) measures how long it takes a program to display all the prime numbers up to half a million using the algorithm from Listing 7.7 (forprintprimes.cpp).

Listing 8.8: `measureprimespeed.cpp`

```
#include <iostream>
#include <ctime>
#include <cmath>

// Display the prime numbers between 2 and 500,000 and
// time how long it takes

int main() {
    clock_t start_time = clock(),   // Record start time
            end_time;
    for (int value = 2; value <= 500000; value++) {
        // See if value is prime
        bool is_prime = true;  // Provisionally, value is prime
        // Try all possible factors from 2 to n - 1
        for (int trial_factor = 2;
              is_prime && trial_factor < value;
```

```
                trial_factor++)
            is_prime = (value % trial_factor != 0);
        if (is_prime)
            std::cout << value <<  " ";   // Display the prime number
    }
    std::cout << '\n';  // Move cursor down to next line
    end_time = clock();
    // Print the elapsed time
    std::cout << "Elapsed time: "
              << static_cast<double>(end_time - start_time)/CLOCKS_PER_SEC
              << " sec." << '\n';
}
```

On one system, the program took 93 seconds, on average, to print all the prime numbers up to 500,000. By comparison, the newer, more efficient version, Listing 8.5 (moreefficientprimes.cpp), which uses the square root optimization takes only 15 seconds to display all the primes up to 500,000. Exact times will vary depending on the speed of the computer.

As it turns out, much of the program's execution time is taken up printing the output, not computing the prime numbers to print. We can compare the algorithms better by redirecting the program's output to a file. If the executable program is named primes.exe, you can redirect its output at the command line by issuing the command

```
primes > run1.out
```

This creates a text file named run1.out that can be viewed with any text editor. Its contents are exactly what would have been printed to the screen if the redirection is not used.

When run using redirection, the time difference is even more dramatic: The unoptimized version generates the prime numbers up to 500,000 in 77 seconds, while the optimized square root version requires only 2 seconds to generate the same number of primes! An even faster prime generator can be found in Listing 11.25 (fasterprimes.cpp); it uses a completely different algorithm to generate prime numbers.

You must `#include` the `<ctime>` header to use the standard `time` function in a program.

8.5 Character Functions

The C library provides a number of character functions that are useful to C++ programmers. Listing 8.9 (touppercase.cpp) converts lowercase letters to uppercase letters.

Listing 8.9: `touppercase.cpp`

```
#include <iostream>
#include <cctype>

int main() {
    for (char lower = 'a'; lower <= 'z'; lower++) {
       char upper = toupper(lower);
       std::cout << lower << " => " << upper << '\n';
    }
}
```

The first lines printed by Listing 8.9 (touppercase.cpp) are

```
a => A
b => B
c => C
d => D
```

Interestingly, the `toupper` function returns an `int`, not a `char`. At the enhanced warning level 4 for Visual C++ a cast is required to assign the result to the variable `upper`:

```
char upper = static_cast<char>(toupper(lower));
```

Some of the more useful character functions are described in Table 8.2.

`charfunctions` Module
`int toupper(int ch)` Returns the uppercase version of the given character; returns the original character if no uppercase version exists (such as for punctuation or digits)
`int tolower(int ch)` Returns the lowercase version of the given character; returns the original character if no lowercase version exists (such as for punctuation or digits)
`int isupper(int ch)` Returns a nonzero value (true) if `ch` is an uppercase letter (`'A'–'Z'`); otherwise, it returns 0 (false)
`int islower(int ch)` Returns a nonzero value (true) if `ch` is an lowercase letter (`'a'–'z'`); otherwise, it returns 0 (false)
`int isalpha(int ch)` Returns a nonzero value (true) if `ch` is a letter from the alphabet (`'A'–'Z'` or `'a'–'z'`); otherwise, it returns 0 (false)
`int isdigit(int ch)` Returns a nonzero value (true) if `ch` is a digit (`'0'–'9'`); otherwise, it returns 0 (false)

Table 8.2: A few of the functions from the `cctype` library

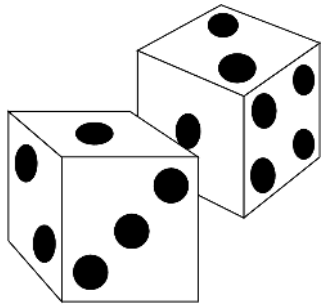
Other functions exist to determine if a character is a punctuation character like a comma or semicolon (`ispunct`), a space, tab, or newline character (`isspace`).

To use the standard C character functions in your C++ program, you must include the `<cctype>` header file.

8.6 Random Numbers

Some applications require behavior that appears random. Random numbers are useful particularly in games and simulations. For example, many board games use a die (one of a pair of dice) to determine how many places a player is to advance. (See Figure 8.5.) A die or pair of dice are used in other games of chance. A die is a cube containing spots on each of its six faces. The number of spots range from one to six. A player rolls a die or sometimes a pair of dice, and the side(s) that face up have meaning in the game being played. The value of a face after a roll is determined at random by the complex tumbling of the die. A software adaptation of a game that involves dice would need a way to simulate the random roll of a die.

Figure 8.5 A pair of dice



All algorithmic random number generators actually produce *pseudorandom* numbers, not true random numbers. A pseudorandom number generator has a particular period, based on the nature of the algorithm used. If the generator is used long enough, the pattern of numbers produced repeats itself exactly. A sequence of true random numbers would not contain such a repeating subsequence. The good news is that all practical algorithmic pseudorandom number generators have periods that are large enough for most applications.

C++ programmers can use two standard C functions for generating pseudorandom numbers: `srand` and `rand`:

```
void srand(unsigned)
int rand()
```

`srand` establishes the first value in the sequence of pseudorandom integer values. Each call to `rand` returns the next value in the sequence of pseudorandom values. Listing 8.10 (simplerandom.cpp) shows how a sequence of 100 pseudorandom numbers can be printed.

Listing 8.10: `simplerandom.cpp`

```
#include <iostream>
#include <cstdlib>

int main() {
    srand(23);
    for (int i = 0; i < 100; i++) {
        int r = rand();
        std::cout << r << " ";

    }
    std::cout << '\n';
}
```

The numbers printed by the program appear to be random. The algorithm is given a seed value to begin, and a formula is used to produce the next value. The seed value determines the sequence of numbers generated; identical seed values generate identical sequences. If you run the program again, the same sequence is displayed because the same seed value, 23, is used. In order to allow each program run to display different sequences, the seed value must be different for each run. How can we establish a different seed value for each run? The best way to make up a "random" seed at run time is to use the `time` function which is found in the `ctime` library. The call `time(0)` returns the number of seconds since midnight January 1, 1970.

This value obviously differs between program runs, so each execution will use a different seed value, and the generated pseudorandom number sequences will be different. Listing 8.11 (betterrandom.cpp) incorporates the `time` function to improve its randomness over multiple executions.

Listing 8.11: `betterrandom.cpp`

```
#include <iostream>
#include <cstdlib>
#include <ctime>

int main() {
    srand(static_cast<unsigned>(time(0)));
    for (int i = 0; i < 100; i++) {
        int r = rand();
        std::cout << r << " ";

    }
    std::cout << '\n';
}
```

Each execution of Listing 8.11 (betterrandom.cpp) produces a different pseudorandom number sequence. The actual type of value that `time` returns is `time_t`, so the result from a call to `time` must be cast to `unsigned int` before being used with `srand`.

Notice that the numbers returned by `rand` can be rather large. The pseudorandom values range from 0 to a maximum value that is implementation dependent. The maximum value for Visual C++ś `rand` function is 32,767, which corresponds to the largest 16-bit `signed int` value. The `cstdlib` header defines the constant `RAND_MAX` that represents the largest value in the range. The following statement

```
std::cout << RAND_MAX << '\n';
```

would print the value of `RAND_MAX` for a particular system.

Ordinarily we need values in a more limited range, like 1...100. Simple arithmetic with the modulus operator can produce the result we need. If n is any nonnegative integer and m is any positive integer, the expression

$$n \% m$$

produces a value in the range $0 \ldots m-1$.

This means the statement

```
int r = rand() % 100;
```

can assign only values in the range 0...99 to `r`. If we really want values in the range 1...100, what can we do? We simply need only add one to the result:

```
int r = rand() % 100 + 1;
```

This statement produces pseudorandom numbers in the range 1...100.

We now have all we need to write a program that simulates the rolling of a die.

Listing 8.12 (die.cpp) simulates rolling die.

Listing 8.12: `die.cpp`

```
#include <iostream>
#include <cstdlib>
#include <ctime>

int main() {
    // Set the random seed value
    srand(static_cast<unsigned>(time(0)));

    // Roll the die three times
    for (int i = 0; i < 3; i++) {
        // Generate random number in the range 1...6
        int value = rand() % 6 + 1;


        // Show the die
        std::cout <<  "+-------+\n";
        switch (value) {
          case 1:
            std::cout << "|       |\n";
            std::cout << "|   *   |\n";
            std::cout << "|       |\n";
            break;
          case 2:
            std::cout << "| *     |\n";
            std::cout << "|       |\n";
            std::cout << "|     * |\n";
            break;
          case 3:
            std::cout << "|     * |\n";
            std::cout << "|   *   |\n";
            std::cout << "| *     |\n";
            break;
          case 4:
            std::cout << "| *   * |\n";
            std::cout << "|       |\n";
            std::cout << "| *   * |\n";
            break;
          case 5:
            std::cout << "| *   * |\n";
            std::cout << "|   *   |\n";
            std::cout << "| *   * |\n";
            break;
          case 6:
            std::cout << "| * * * |\n";
            std::cout << "|       |\n";
            std::cout << "| * * * |\n";
            break;
          default:
            std::cout <<  " ***  Error: illegal die value ***\n";
            break;
        }
        std::cout << "+-------+\n";
    }
```

```
}
```

The output of one run of Listing 8.12 (die.cpp) is

```
+-------+
| *   * |
|       |
| *   * |
+-------+
+-------+
| * * * |
|       |
| * * * |
+-------+
+-------+
|       |
|   *   |
|       |
+-------+
```

Since the values are pseudorandomly generated, actual output will vary from one run to the next.

8.7 Exercises

1. Suppose you need to compute the square root of a number in a C++ program. Would it be a good idea to write the code to perform the square root calculation? Why or why not?

2. In C++ source code what is one way to help you distinguish a variable name from a function name?

3. Which one of the following values could be computed by the `rand` function?

   `4.5` `34` `-1` `RAND_MAX + 1`

4. What does `clock_t` represent?

5. What does `CLOCKS_PER_SEC` represent?

6. Ordinarily how often should a program call the `srand` function?

7. In Listing 8.2 (standardsquareroot.cpp), what does the `main` function do while the `sqrt` function is computing the square root of the argument that `main` provides?

8. Consider each of the following code fragments below that could be part of a C++ program. Each fragment contains a call to a standard C/C++ library function. Answer each question in one of the following three ways:

   - If the code fragment contains a compile-time error, write the word *error* for the answer.
   - If the code fragment contains no compile-time errors and you can determine its output at compile-time, provide the fragment's literal output.
   - If the code fragment contains no compile-time errors but you cannot determine its exact output at compile-time, provide one possible evaluation and write the word *example* for the answer and provide one possible literal output that the code fragment could produce.

(a) `std::cout << sqrt(4.5) << '\n';`

(b) `std::cout << sqrt(4.5, 3.1) << '\n';`

(c) `std::cout << rand(4) << '\n';`

(d)
```
double d = 16.0;
std::cout << sqrt(d) << '\n';
```

(e) `std::cout << srand() << '\n';`

(f) `std::cout << rand() << '\n';`

(g)
```
int i = 16;
std::cout << sqrt(i) << '\n';
```

(h) `std::cout << srand(55) << '\n';`

(i) `std::cout << tolower('A') << '\n';`

(j) `std::cout << exp() << '\n';`

(k) `std::cout << sqrt() << '\n';`

(l) `std::cout << toupper('E') << '\n';`

(m) `std::cout << toupper('e') << '\n';`

(n) `std::cout << toupper("e") << '\n';`

(o) `std::cout << exp(4.5) << '\n';`

(p) `std::cout << toupper('h', 5) << '\n';`

(q) `std::cout << ispunct('!') << '\n';`

(r) `std::cout << tolower("F") << '\n';`

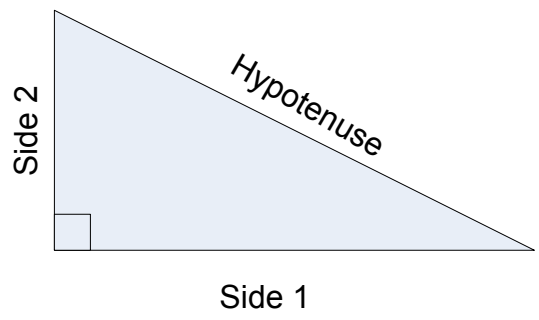
(s)
```
char ch = 'D';
std::cout << tolower(ch) << '\n';
```

(t) `std::cout << exp(4.5, 3) << '\n';`

(u) `std::cout << toupper('7') << '\n';`

(v)
```
double a = 5, b = 3;
std::cout << exp(a, b) << '\n';
```

Figure 8.6 Right triangle



(w)
```
std::cout << exp(3, 5, 2) << '\n';
```

(x)
```
std::cout << tolower(70) << '\n';
```

(y)
```
double a = 5;
std::cout << exp(a, 3) << '\n';
```

(z)
```
double a = 5;
std::cout << exp(3, a) << '\n';
```

9. From geometry: Write a computer program that given the lengths of the two sides of a right triangle adjacent to the right angle computes the length of the hypotenuse of the triangle. (See Figure 8.6.) If you are unsure how to solve the problem mathematically, do a web search for the *Pythagorean theorem*.

Chapter 9

Writing Functions

As programs become more complex, programmers must structure their programs in such a way as to effectively manage their complexity. Most humans have a difficult time keeping track of too many pieces of information at one time. It is easy to become bogged down in the details of a complex problem. The trick to managing complexity is to break down the problem into more manageable pieces. Each piece has its own details that must be addressed, but these details are hidden as much as possible within that piece. The problem is ultimately solved by putting these pieces together to form the complete solution.

So far all of our programs have been written within one function—`main`. As the number of statements within a function increases, the function can become unwieldy. The code within such a function that does all the work by itself is called *monolithic code*. Monolithic code that is long and complex is undesirable for several reasons:

- **It is difficult to write correctly**. All the details in the entire piece of code must be considered when writing any statement within that code.

- **It is difficult to debug**. If the sequence of code does not work correctly, it is often difficult to find the source of the error. The effects of an erroneous statement that appears earlier in the code may not become apparent until a correct statement later uses the erroneous statement's incorrect result.

- **It is difficult to extend**. All the details in the entire sequence of code must be well understood before it can be modified. If the code is complex, this may be a formidable task.

Using a divide and conquer strategy, a programmer can decompose a complicated function (like `main`) into several simpler functions. The original function can then do its job by delegating the work to these other functions. In this way the original function can be thought of as a "work coordinator."

Besides their code organization aspects, functions allow us to bundle functionality into reusable parts. In Chapter 8 we saw how library functions can dramatically increase the capabilities of our programs. While we should capitalize on library functions as much as possible, sometimes we need a function exhibiting custom behavior that is not provided by any standard function. Fortunately we can create our own functions, and the same function may be used (called) in numerous places within a program. If the function's purpose is general enough and we write the function properly, we may be able to reuse the function in other programs as well.

©2018 Richard L. Halterman

9.1 Function Basics

Recall the "handwritten" square root code we saw in Listing 8.1 (computesquareroot.cpp). We know that the better option is the standard library function `sqrt`; however, we will illustrate custom function development by writing our own square root function based on the code in Listing 8.1 (computesquareroot.cpp). In Listing 9.1 (customsquareroot.cpp) we see the definition for the `square_root` function.

Listing 9.1: customsquareroot.cpp

```
#include <iostream>
#include <iomanip>
#include <cmath>

// Compute an approximation of
// the square root of x
double square_root(double x) {
    double diff;
    // Compute a provisional square root
    double root = 1.0;

    do { // Loop until the provisional root
         // is close enough to the actual root
        root = (root + x/root) / 2.0;
        //std::cout << "root is " << root << '\n';
        // How bad is the approximation?
        diff = root * root - x;
    } while (diff > 0.0001 || diff < -0.0001);
    return root;
}

int main() {
    // Compare the two ways of computing the square root
    for (double d = 1.0; d <= 10.0; d += 0.5)
       std::cout << std::setw(7) << square_root(d) << " : " << sqrt(d) << '\n';
}
```

The `main` function in Listing 9.1 (customsquareroot.cpp) compares the behavior of our custom `square_root` function to the `sqrt` library function. Its output:

```
      1 : 1
1.22474 : 1.22474
1.41422 : 1.41421
1.58116 : 1.58114
1.73205 : 1.73205
1.87083 : 1.87083
      2 : 2
2.12132 : 2.12132
2.23607 : 2.23607
2.34521 : 2.34521
2.44949 : 2.44949
2.54952 : 2.54951
2.64577 : 2.64575
2.73861 : 2.73861
2.82843 : 2.82843
2.91548 : 2.91548
```

```
       3 : 3
3.08221 : 3.08221
3.16228 : 3.16228
```

shows a few small differences in the results. Clearly we should use the standard `sqrt` function instead of ours.

There are two aspects to every C++ function:

- **Function definition**. The definition of a function specifies the function's return type and parameter types, and it provides the code that determines the function's behavior. In Listing 9.1 (customsquareroot.cpp) the definition of the `square_root` function appears above the `main` function.
- **Function invocation**. A programmer uses a function via a function invocation. The `main` function invokes both our `square_root` function and the `sqrt` function. Every function has exactly one definition but may have many invocations.

A function definition consists of four parts:

- **Name**—every function in C++ has a name. The name is an identifier (see Section 3.3). As with variable names, the name chosen for a function should accurately portray its intended purpose or describe its functionality.
- **Type**—every function has a return type. If the function returns a value to its caller, its type corresponds to the type of the value it returns. The special type `void` signifies that the function does not return a value.
- **Parameters**—every function must specify the types of parameters that it accepts from callers. The parameters appear in a parenthesized comma-separated list like in a function prototype (see Section 8.1). Unlike function prototypes, however, parameters usually have names associated with each type.
- **Body**—every function definition has a body enclosed by curly braces. The body contains the code to be executed when the function is invoked.

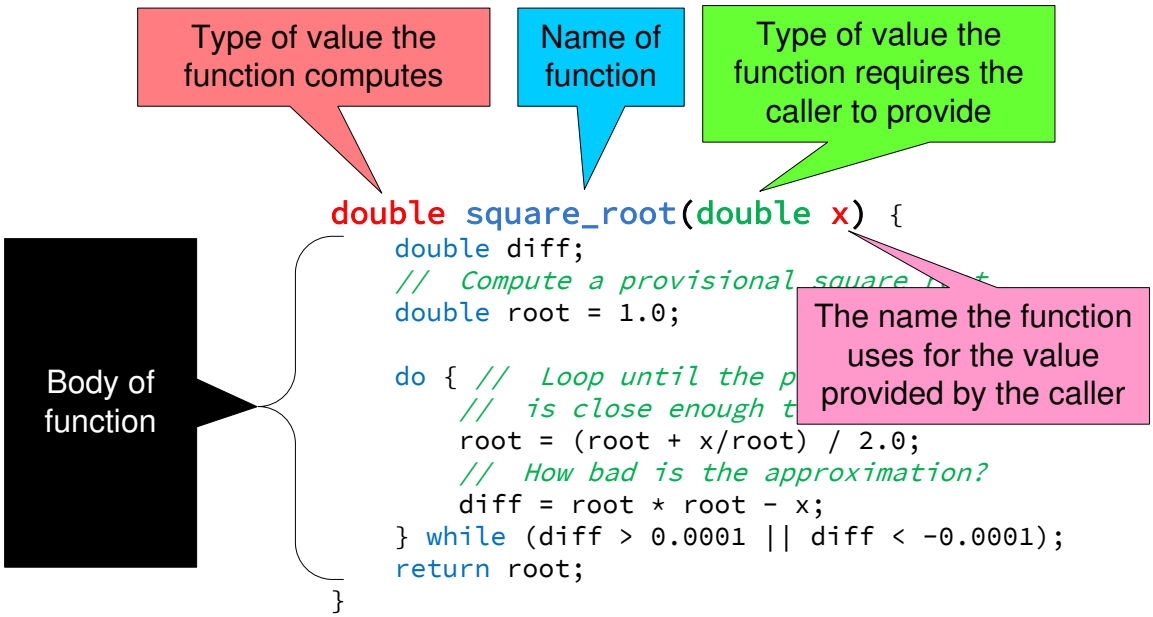
Figure 9.1 dissects a our `square_root` function definition.

The simplest function accepts no parameters and returns no value to the caller. Listing 9.2 (simplefunction.cpp) is a variation of Listing 4.1 (adder.cpp) that contains such a simple function definition.

Listing 9.2: `simplefunction.cpp`

```
#include <iostream>

// Definition of the prompt function
void prompt() {
    std::cout << "Please enter an integer value: ";
}

int main() {
    int value1, value2, sum;
    std::cout << "This program adds together two integers.\n";
    prompt();    // Call the function
    std::cin >> value1;
```

Figure 9.1 Function definition dissection



```
    prompt();    // Call the function again
    std::cin >> value2;
    sum = value1 + value2;
    std::cout << value1 << " + " << value2 << " = " << sum << '\n';
}
```

The `prompt` function simply prints a message. The program runs as follows:

1. The program's execution, like in all C++ programs, begins with the first executable statement in the function named `main`. The first line in the `main` function simply declares some variables needed for compiler housekeeping, so the next line actually begins the executable code.
2. The first executable statement prints the message of the program's intent.
3. The next statement is a call of the `prompt` function. At this point the program's execution transfers to the body of the `prompt` function. The code within `prompt` is executed until the end of its body or until a `return` statement is encountered. Since `prompt` contains no `return` statement, all of `prompt`'s body (the one print statement) will be executed.
4. When `prompt` is finished, control is passed back to the point in `main` immediately *after* the call of `prompt`.
5. The next action after `prompt` call reads the value of `value1` from the keyboard.
6. A second call to `prompt` transfers control back to the code within the `prompt` function. It again prints its message.
7. When the second call to `prompt` is finished, control passes back to `main` at the point of the second input statement that assigns `value2` from the keyboard.
8. The remaining two statements in `main` are executed, and then the program's execution terminates.

As another simple example, consider Listing 9.3 (countto10.cpp).

Listing 9.3: `countto10.cpp`

```
#include <iostream>

int main() {
    for (int i = 1; i <= 10; i++)
        std::cout << i << '\n';
}
```

which simply counts to ten:

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

If counting to ten in this way is something we want to do frequently within a program, we can write a function as shown in Listing 9.4 (countto10func.cpp) and call it as many times as necessary.

Listing 9.4: countto10func.cpp

```
#include <iostream>

// Count to ten and print each number on its own line
void count_to_10() {
    for (int i = 1; i <= 10; i++)
        std::cout << i << '\n';
}

int main() {
    std::cout << "Going to count to ten . . .";
    count_to_10();
    std::cout << "Going to count to ten again. . .";
    count_to_10();
}
```

Our `prompt` and `countto10` functions are a bit underwhelming. The `prompt` function could be eliminated, and each call to `prompt` could be replaced with the statement in its body. The same could be said for the `countto10` function, although it is convenient to have the simple one-line statement that hides the complexity of the loop. Using the `prompt` function does have one advantage, though. If `prompt` is removed and the two calls to `prompt` are replaced with the print statement within prompt, we have to make sure that the two messages printed are identical. If we simply call `prompt`, we know the two messages printed will be identical because only one possible message can be printed (the one in the body of `prompt`).

We can alter the behavior of a function through a mechanism called *parameter passing*. If a function is written to accept information from the caller, the caller must supply the information in order to use the function. The caller communicates the information via one or more parameters as required by the function. The `countto10` function does us little good if we sometimes want to count up to a different number. Listing 9.5 (countton.cpp) generalizes Listing 9.4 (countto10func.cpp) to count as high as the caller needs.

Listing 9.5: countton.cpp

```
#include <iostream>

// Count to n and print each number on its own line
void count_to_n(int n) {
    for (int i = 1; i <= n; i++)
        std::cout << i << '\n';
}

int main() {
    std::cout << "Going to count to ten . . .";
    count_to_n(10);
    std::cout << "Going to count to five . . .";
    count_to_n(5);
}
```

When the caller, in this case `main`, issues the call

```
count_to_n(10);
```

the argument 10 is assigned to n before the function's statements begin executing.

A caller must pass exactly one integer parameter (or other type that is assignment-compatible with integers) to `count_to_n` during a call. An attempt to do otherwise will result in a compiler error or warning:

```
count_to_n();       // Error, missing parameter during the call
count_to_n(3, 5);   // Error, too many parameters during the call
count_to_n(3.2);    // Warning, possible loss of data (double to int)
```

We can enhance the `prompt` function's capabilities as shown in Listing 9.6 (betterprompt.cpp)

Listing 9.6: `betterprompt.cpp`

```
#include <iostream>

// Definition of the prompt function
int prompt() {
    int result;
    std::cout << "Please enter an integer value: ";
    std::cin >> result;
    return result;
}

int main() {
    int value1, value2, sum;
    std::cout << "This program adds together two integers.\n";
    value1 = prompt();    // Call the function
    value2 = prompt();    // Call the function again
    sum = value1 + value2;
    std::cout << value1 << " + " << value2 << " = " << sum << '\n';
}
```

In this version, `prompt` takes care of the input, so `main` does not have to use any input statements. The assignment statement within `main`:

```
value1 = prompt();
```

implies `prompt` is no longer a `void` function; it must return a value that can be assigned to the variable `value1`. Furthermore, the value that `prompt` returns must be assignment compatible with an `int` because `value1`'s declared type is `int`. A quick look at the first line of `prompt`'s definition confirms our assumption:

```
int prompt()
```

This indicates that `prompt` returns an `int` value.

Because `prompt` is declared to return an `int` value, it must contain a `return` statement. A `return` statement specifies the exact value to return to the caller. When a `return` is encountered during a function's execution, control immediately passes back to the caller. The value of the function call is the value specified by the `return` statement, so the statement

```
value1 = prompt();
```

assigns to the variable `value1` the value indicated when the `return` statement executes.

Note that in Listing 9.6 (betterprompt.cpp), we declared a variable named `result` inside the `prompt` function. This variable is local to the function, meaning we cannot use this particular variable outside of `prompt`. It also means we are free to use that same name outside of the `prompt` function in a different context, and that use will not interfere with the `result` variable within `prompt`. We say that `result` is a *local variable*.

We can further enhance our `prompt` function. Currently `prompt` always prints the same message. Using parameters, we can customize the message that `prompt` prints. Listing 9.7 (evenbetterprompt.cpp) shows how parameters are used to provide a customized message within `prompt`.

Listing 9.7: evenbetterprompt.cpp

```
#include <iostream>

// Definition of the prompt function
int prompt(int n) {
    int result;
    std::cout << "Please enter integer #" << n << ": ";
    std::cin >> result;
    return result;
}

int main() {
    int value1, value2, sum;
    std::cout << "This program adds together two integers.\n";
    value1 = prompt(1);    // Call the function
    value2 = prompt(2);    // Call the function again
    sum = value1 + value2;
    std::cout << value1 << " + " << value2 << " = " << sum << '\n';
}
```

In Listing 9.7 (evenbetterprompt.cpp), the parameter influences the message that it printed. The user is now prompted to enter value #1 or value #2. The call

```
value1 = prompt(1);
```

passes the value 1 to the `prompt` function. Since `prompt`'s parameter is named `n`, the process works as if the assignment statement

```
n = 1;
```

were executed as the first action within `prompt`.

In the first line of the function definition:

```
int prompt(int n)
```

`n` is called the *formal parameter*. A formal parameter is used like a variable within the function's body, but it is declared in the function's parameter list; it is not declared in the function's body. A *formal* parameter is a parameter as used in the *formal* definition of the function.

At the point of the function call:

```
value1 = prompt(1);
```

the parameter (or argument) passed into the function, 1, is called the *actual parameter*. An *actual* parameter is the parameter *actually* used during a call of the function. When a function is called, any actual parameters

are assigned to their corresponding formal parameters, and the function begin executing. Another way to say it is that during a function call, the actual parameters are *bound* to their corresponding formal parameters.

The parameters used within a function definition are called *formal parameters*. Formal parameters behave as local variables within the function's body; as such, the name of a formal parameter will not conflict with any local variable or formal parameter names from other functions. This means as a function developer you may choose a parameter name that best represents the parameter's role in the function.

If you are writing a function, you cannot predict the caller's actual parameters. You must be able to handle any value the caller sends. The compiler will ensure that the types of the caller's parameters are compatible with the declared types of your formal parameters.

To remember the difference between formal and actual parameters, remember this:

- A *formal* parameter is a parameter declared and used in a function's *formal* definition.
- An *actual* parameter is a parameter supplied by the caller when the caller *actually* uses (invokes or calls) the function.

When the call

```
value1 = prompt(1);
```

is executed in `main`, and the statement

```
std::cout << "Please enter integer #" << n << ": ";
```

within the body of `prompt` is executed, `n` will have the value 1. Similarly, when the call

```
value2 = prompt(2);
```

is executed in `main`, and the statement

```
std::cout << "Please enter integer #" << n << ": ";
```

within the body of `prompt` is executed, `n` will have the value 2. In the case of

```
value1 = prompt(1);
```

`n` within `prompt` is bound to 1, and in the case of

```
value2 = prompt(2);
```

`n` within `prompt` is bound to 2.

A function's definition requires that all formal parameters be declared in the parentheses following the function's name. A caller does not provide actual parameter type declarations when calling the function. Given the `square_root` function defined in Listing 9.1 (customsquareroot.cpp), the following caller code fragment is illegal:

```
double number = 25.0;
// Legal, pass the variable's value to the function
std::cout << square_root(number) << '\n';
// Illegal, do not declare the parameter during the call
std::cout << square_root(double number) << '\n';
```

The function definition is responsible for declaring the types of its parameters, not the caller.

9.2 Using Functions

The general form of a function definition is

type name (*parameterlist*) {
    body
}

- The *type* of the function indicates the type of value the function returns. Often a function will perform a calculation and the result of the calculation must be communicated back to the place where the function was invoked. The special type `void` indicates that the function does not return a value.
- The *name* of the function is an identifier (see Section 3.3). The function's name should indicate the purpose of the function.
- The *parameterlist* is a comma separated list of pairs of the form

  type name

  where *type* is a C++ type and *name* is an identifier representing a parameter. The caller of the function communicates information into the function via parameters. The parameters specified in the parameter list of a function definition are called *formal parameters*. A parameter is also known as an *argument*. The parameter list may be empty; an empty parameter list indicates that no information may be passed into the function by the caller.
- The *body* is the sequence of statements, enclosed within curly braces, that define the actions that the function is to perform. The statements may include variable declarations, and any variables declared within the body are local to that function.

  The body may contain only one statement, many statements, or no statements at all; regardless, the curly braces always are required.

Observe that multiple pieces of information can be passed into a function via multiple parameters, but only one piece of information can be passed out of the function via the return value. Recall the greatest

Figure 9.2 Cutting plywood

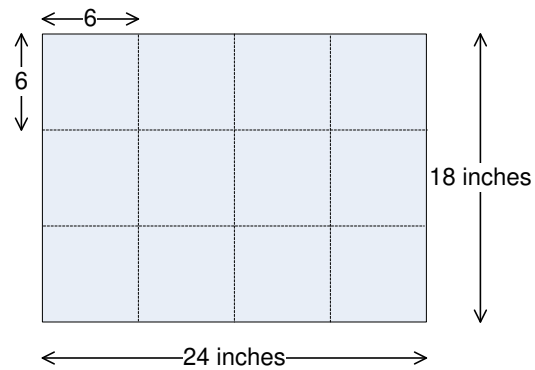
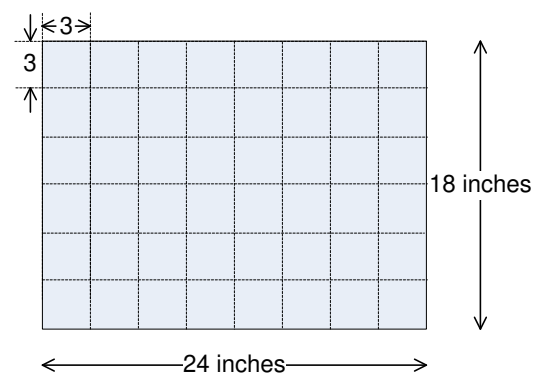


Figure 9.3 Squares too small



common divisor (also called greatest common factor) from elementary mathematics. To determine the GCD of 24 and 18 we list all of their common factors and select the largest one:

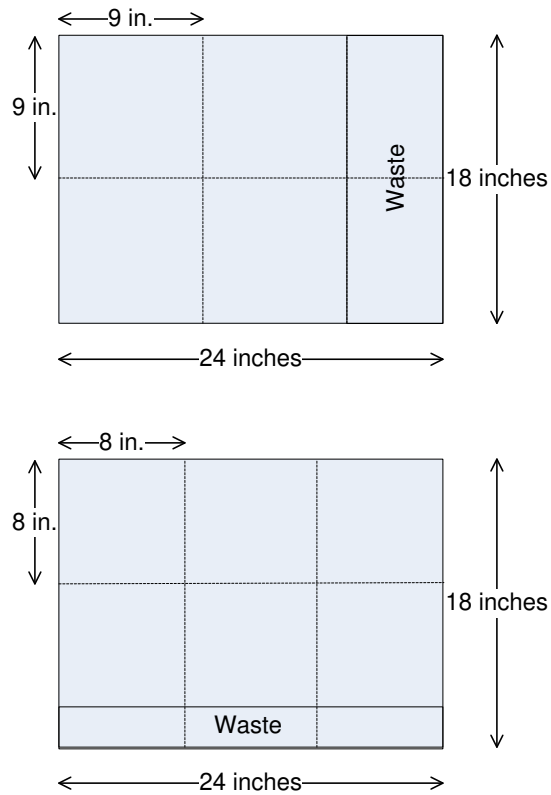
24: 1, 2, 3, 4, $\boxed{6}$, 8, 12, 24
18: 1, 2, 3, $\boxed{6}$, 9, 18

The greatest common divisor function is useful when reducing fractions to lowest terms; for example, consider the fraction $\frac{18}{24}$. The greatest common divisor of 18 and 24 is 6, and so we divide the numerator and the denominator of the fraction by 6: $\frac{18 \div 6}{24 \div 6} = \frac{3}{4}$. The GCD function has applications in other areas besides reducing fractions to lowest terms. Consider the problem of dividing a piece of plywood 24 inches long by 18 inches wide into square pieces of maximum size without wasting any material. Since the GCD(24, 18) = 6, we can cut the plywood into twelve 6 inch $\times$ 6 inch square pieces as shown in Figure 9.2.

If we cut the plywood into squares of any other size without wasting the any of the material, the squares would have to be smaller than 6 inches $\times$ 6 inches; for example, we could make forty-eight 3 inch $\times$ 3 inch squares as shown in pieces as shown in Figure 9.3.

©2018 Richard L. Halterman

Figure 9.4 Squares too large



If we cut squares larger than 6 inches × 6 inches, not all the plywood can be used to make the squares. Figure 9.4. shows how some larger squares would fare.

In addition to basic arithmetic and geometry, the GCD function plays a vital role in cryptography, enabling secure communication across an insecure network.

Listing 9.8: gcdprog.cpp

```
#include <iostream>

int main() {
    // Prompt user for input
    int num1, num2;
    std::cout << "Please enter two integers: ";
    std::cin >> num1 >> num2;

    // Determine the smaller of num1 and num2
    int min = (num1 < num2) ? num1 : num2;

    // 1 is definitely a common factor to all ints
    int largestFactor = 1;
```

```
    for (int i = 2;  i <= min;  i++)
        if (num1 % i == 0 && num2 % i == 0)
           largestFactor = i; // Found larger factor
    std::cout <<  largestFactor << '\n';
}
```

Listing 9.8 (gcdprog.cpp) implements a straight-forward but naive algorithm that seeks potential factors by considering every integer less than the smaller of the two values provided by the user. This algorithm is not very efficient, especially for larger numbers. Its logic is easy to follow, with no deep mathematical insight required. Soon we will see a better algorithm for computing GCD.

If we need to compute the GCD from several different places within our program, we should package the code in a function rather than copying it to multiple places. The following code fragment defines a C++ function that that computes the greatest common divisor of two integers. It determines the largest factor (divisor) common to its parameters:

```
int gcd(int num1, int num2) {
    // Determine the smaller of num1 and num2
    int min = (num1 < num2) ? num1 : num2;
    // 1 is definitely a common factor to all ints
    int largestFactor = 1;
    for (int i = 2;  i <= min;  i++)
        if (num1 % i == 0 && num2 % i == 0)
            largestFactor = i; // Found larger factor
    return largestFactor;
}
```

This function is named `gcd` and expects two integer arguments. Its formal parameters are named `num1` and `num2`. It returns an integer result. Its body declares three local variables: `min`, `largestFactor`, and `i` (`i` is local to the `for` statement). The last line in its body is a `return` statement. A return statement is required for functions that return a value. A `void` function is not required to have a `return` statement. If a `void` function does have a `return` statement, it must simply consist of `return` followed by a semicolon (in other words, it cannot return a value, like `gcd`'s `return` statement does). A `void` function that does not contain a `return` statement simply returns at the end of its body.

Recall from Section 6.5 that local variables have meaning only within their scope. This means that when you write a function you can name a local variable without fear that its name may be used already in another part of the program. Two different functions can use local variables named `x`, and these are two different variables that have no influence on each other. Anything local to a function definition is hidden to all code outside that function definition.

Since a formal parameter is a local variable, you can reuse the names of formal parameters in different functions without a problem.

It may seem strange that we can use the same name in two different functions within the same program to refer to two distinct variables. The block of statements that makes up a function definition constitutes a context for local variables. A simple analogy may help. In the United States, many cities have a street named *Main Street*; for example, there is a thoroughfare named Main Street in San Francisco, California. Dallas, Texas also has a street named Main Street. Each city and town provides its own context for the use of the term *Main Street*. A person in San Francisco asking "How do I get to Main Street?" will receive the directions to San Francisco's Main Street, while someone in Dallas asking the same question will receive Dallas-specific instructions. In a similar manner, assigning a variable within a function block localizes its identity to that function. We can think of a program's execution as a person traveling around the U.S. When

in San Francisco, all references to *Main Street* mean San Francisco's Main Street, but when the traveler arrives in Dallas, the term *Main Street* means Dallas' Main Street. A program's thread of execution cannot execute more than one statement at a time, which means the compiler can use its current context to interpret any names it encounters within a statement. Similarly, at the risk of overextending the analogy, a person cannot be physically located in more than one city at a time. Furthermore, Main Street may be a bustling, multi-lane boulevard in one large city, but a street by the same name in a remote, rural township may be a narrow dirt road! Similarly, two like-named variables may have two completely different types. A variable named `x` in one function may represent an integer, while a different function may use a string variable named `x`.

Another advantage of local variables is that they occupy space in the computer's memory only when the function is executing. Space is allocated for local variables and parameters when the function begins executing. When the function is finished and control returns to the caller, the variables and parameters go out of scope, and the memory they held is freed up for other purposes within the running program. This process of local variable allocation and deallocation happens each time a caller invokes the function. More information about how C++ handles memory management during a program's execution can be found in Section 18.1.

Once we have written a complete function definition we can use the function within our program. We invoke a programmer-defined function in exactly the same way as a standard library function like `sqrt` (Section 8.2) or `rand` (Section 8.6). If the function returns a value (that is, it is not declared `void`), then we can use its invocation anywhere an expression of that type is allowed. The parameters used for the function call are known as actual parameters. The function `gcd` can be called as part of an assignment statement:

```
int factor = gcd(val, 24);
```

This call uses the variable `val` as its first actual parameter and the literal value 24 as its second actual parameter. Variables, expressions, and literals can be freely used as actual parameters. The function then computes and returns its result. This result is assigned to the variable `factor`.

How does the function call and parameter mechanism work? It's actually quite simple. The actual parameters, in order, are assigned (bound) to each of the formal parameters in the function definition, then control is passed to the body of the function. When the function's body is finished executing, control passes back to the point in the program where the function was called. The value returned by the function, if any, replaces the function call expression. In the statement

```
int factor = gcd(val, 24);
```

an integer value is assigned to `factor`. The expression on the right is a function call, so the function is invoked to determine what to assign. The value of the variable `val` is assigned to the formal parameter `num1`, and the literal value 24 is assigned to the formal parameter `num2`. The body of the `gcd` function is then executed. When the `return` statement in the body is encountered, program execution returns back to where the function was called. The argument of the return statement becomes the value that is assigned to `factor`. This process of copying actual parameters to formal parameters works exactly like during assignment. This means the compiler, where possible, automatically will widen or narrow (see Section 4.2) the value of an actual parameter to make it compatible with its corresponding formal parameter; for example, if `val` is declared to a `char`, its value would automatically be copied to a temporary location and converted to an `int`. This temporary value would then be bound to the formal parameter `num1`. Note that `gcd` could be called from many different places within the same program, and, since different parameter values could be passed at each of these different invocations, `gcd` could compute a different result at each invocation.

Other invocation examples include:

- `std::cout << gcd(36, 24);`

  This example simply prints the result of the invocation. The value 36 is bound to `num1` and 24 is bound to `num2` for the purpose of the function call. The value 12 will be printed, since 12 is the greatest common divisor of 36 and 24..

- `x = gcd(x - 2, 24);`

  The execution of this statement would evaluate `x - 2` and bind its value to `num1`. `num2` would be assigned 24. The result of the call is then assigned to `x`. Since the right side of the assignment statement is evaluated *before* being assigned to the left side, the original value of `x` is used when calculating `x - 2`, and the function return value then updates `x`.

- `x = gcd(x - 2, gcd(10, 8));`

  This example shows two invocations in one statement. Since the function returns an integer value its result can itself be used as an actual parameter in a function call. Passing the result of one function call as an actual parameter to another function call is called *function composition*.

The compiler will report an error if a function call does not agree with the function's definition. Possible problems include:

- **Number of actual parameters do not agree with the number of formal parameters**. The number of parameters must agree exactly. For example, the statement

  ```
  int factor = gcd(24);   // Error: too few parameters
  ```

  is illegal given the above definition of `gcd`, since only one actual parameter is provided when two are required.

- **Passing an actual parameter that is not assignment compatible with the formal parameter**. For example, passing the `std::cout` object when an `int` has been defined, as in

  ```
  int factor = gcd(36, std::cout);  // Error: second parameter is wrong type
  ```

  The compiler will detect that `std::cout` is not a valid `int` and report an error.

- **Using the result in a context where an expression of that type is not allowed**. For example, a function that returns `void` cannot be used where an `int` is expected:

  ```
  std::cout << srand(2);  // Error: srand does not return anything
  ```

  The compiler will disallow this code.

9.3 Pass by Value

The default parameter passing mechanism in C++ is classified as *pass by value*, also known as *call by value*. This means the value of the actual parameter is copied to the formal parameter for the purpose of executing the function's code. Since it is working on a copy of the actual parameter, the function's execution cannot affect the value of the actual parameter owned by the caller.

Listing 9.9 (passbyvalue.cpp) illustrates the consequences of pass by value.

Listing 9.9: passbyvalue.cpp

```
#include <iostream>

/*
 *  increment(x)
 *     Illustrates pass by value protocol.
 */
void increment(int x) {
   std::cout << "Beginning execution of increment, x = "
             << x << '\n';
   x++;   // Increment x
   std::cout << "Ending execution of increment, x = "
             << x << '\n';
}

int main() {
    int x = 5;
    std::cout << "Before increment, x = " << x << '\n';
    increment(x);
    std::cout << "After increment, x = " << x << '\n';
}
```

For additional drama we chose to name the actual parameter the same as the formal parameter. Since the actual parameter and formal parameter are declared and used in different contexts and represent completely different memory locations, their names can be the same without any problems.

Listing 9.9 (passbyvalue.cpp) produces

```
Before increment, x = 5
Beginning execution of increment, x = 5
Ending execution of increment, x = 6
After increment, x = 5
```

The memory for the variable `x` in `main` is unaffected since `increment` works on a copy of the actual parameter.

C++ supports another way of passing parameters called *pass by reference*. Pass by reference is introduced in Section 10.9.

A function communicates its return value to the caller in the same way that the caller might pass a parameter by value. In the `prompt` function we saw earlier:

```
int prompt(int n) {
    int result;
    std::cout << "Please enter integer #" << n << ": ";
    std::cin >> result;
    return result;
}
```

the `return` statement is

```
return result;
```

The variable `result` is local to `prompt`. We informally may say we are returning the `result` variable, but, in fact, we really are returning only the *value* of the `result` variable. The caller has no access to

the local variables declared within any function it calls. In fact, the local variables for a function exist only when the function is active (that is, executing). When the function returns to its caller all of its local variables disappear from memory. During subsequent invocations, the function's local variables reappear when the function becomes active and disappear again when it finishes.

9.4 Function Examples

This section contains a number of examples of how we can use functions to organize a program's code.

9.4.1 Better Organized Prime Generator

Listing 9.10 (primefunc.cpp) is a simple enhancement of Listing 8.5 (moreefficientprimes.cpp). It uses the square root optimization and adds a separate `is_prime` function.

Listing 9.10: `primefunc.cpp`

```
#include <iostream>
#include <cmath>

/*
 *  is_prime(n)
 *     Determines the primality of a given value
 *     n an integer to test for primality
 *     Returns true if n is prime; otherwise, returns false
 */
bool is_prime(int n) {
    bool result = true;  // Provisionally, n is prime
    double r = n, root = sqrt(r);
    // Try all possible factors from 2 to the square
    // root of n
    for (int trial_factor = 2;
          result && trial_factor <= root; trial_factor++)
         result = (n % trial_factor != 0);
    return result;
}

/*
 *  main
 *     Tests for primality each integer from 2
 *     up to a value provided by the user.
 *     If an integer is prime, it prints it;
 *     otherwise, the number is not printed.
 */
int main() {
    int max_value;
    std::cout << "Display primes up to what value? ";
    std::cin >> max_value;
    for (int value = 2; value <= max_value; value++)
        if (is_prime(value)) // See if value is prime
            std::cout << value <<  " "; // Display the prime number
    std::cout << '\n';  // Move cursor down to next line
}
```

Listing 9.10 (primefunc.cpp) illustrates several important points about well-organized programs:

- The complete work of the program is no longer limited to the `main` function. The effort to test for primality is delegated to a separate function. `main` is focused on a simpler task: generating all the numbers to be considered and using another function (`is_prime`) to do the hard work of determining if a given number is prime. `main` is now simpler and more logically *coherent*. A function is coherent when it is focused on a single task. Coherence is a desirable property of functions. If a function becomes too complex by trying to do too many different things, it can be more difficult to write correctly and debug when problems are detected. A complex function should be decomposed into several, smaller, more coherent functions. The original function would then call these new simpler functions to accomplish its task. Here, `main` is not concerned about *how* to determine if a given number is prime; `main` simply delegates the work to `is_prime` and makes use of the `is_prime` function's findings.

- Each function is preceded by a thorough comment that describes the nature of the function. It explains the meaning of each parameter, and it indicates what the function should return. The comment for `main` may not be as thorough as for other functions; this is because `main` usually has no parameters, and it always returns a code to the operating system upon the program's termination.

- While the exterior comment indicates *what* the function is to do, comments within each function explain in more detail *how* the function accomplishes its task.

The call to `is_prime` returns true or false depending on the value passed to it. This means we can express a condition like

```
if (is_prime(value) == true)  . . .
```

more compactly as

```
if (is_prime(value))  . . .
```

because if `is_prime(value)` is true, `true == true` is true, and if `is_prime(value)` is false, `false == true` is false. The expression `is_prime(value)` suffices.

Just as it is better for a loop to have exactly one entry point and exactly one exit point, preferably a function will have a single `return` statement. Simple functions with a small number of `return`s are generally tolerable, however. Consider the following version of `is_prime`:

```
bool is_prime(int n) {
    for (int trialFactor = 2;
          trialFactor <= sqrt(static_cast<double>(n));
          trialFactor++)
        if (n % trialFactor == 0) // Is trialFactor a factor?
            return false;  // Yes, return right away
    return true;  // Tried them all, must be prime
}
```

This version uses two `return` statements, but eliminates the need for a local variable (`result`). Because a `return` statement exits the function immediately, no `break` statement is necessary. The two `return` statements are close enough textually in source code that the logic is fairly transparent.

9.4.2 Command Interpreter

Some functions are useful even if they accept no information from the caller and return no result. Listing 9.11 (calculator.cpp) uses such a function.

Listing 9.11: calculator.cpp

```
#include <iostream>
#include <cmath>

/*
 *  help_screen
 *     Displays information about how the program works
 *     Accepts no parameters
 *     Returns nothing
 */
void help_screen() {
    std::cout << "Add:  Adds two numbers\n";
    std::cout << "      Example: a 2.5 8.0\n";
    std::cout << "Subtract:  Subtracts two numbers\n";
    std::cout << "      Example: s 10.5 8.0\n";
    std::cout << "Print:  Displays the result of the latest operation\n";
    std::cout << "      Example: p\n";
    std::cout << "Help:  Displays this help screen\n";
    std::cout << "      Example: h\n";
    std::cout << "Quit:  Exits the program\n";
    std::cout << "      Example: q\n";
}

/*
 *  menu
 *     Display a menu
 *     Accepts no parameters
 *     Returns the character entered by the user.
 */
char menu() {
    // Display a menu
    std::cout << "=== A)dd S)ubtract P)rint H)elp Q)uit ===\n";
    // Return the char entered by user
    char ch;
    std::cin >> ch;
    return ch;
}

/*
 *  main
 *     Runs a command loop that allows users to
 *     perform simple arithmetic.
 */
int main() {
    double result = 0.0, arg1, arg2;
    bool done = false;  // Initially not done
    do {
        switch (menu()) {
          case 'A':     // Addition
```

```
            case 'a':
              std::cin >> arg1 >> arg2;
              result = arg1 + arg2;
              std::cout << result << '\n';
              break;
            case 'S':     // Subtraction
            case 's':
              std::cin >> arg1 >> arg2;
              result = arg1 - arg2;
              // Fall through, so it prints the result
            case 'P':     // Print result
            case 'p':
              std::cout << result << '\n';
              break;
            case 'H':     // Display help screen
            case 'h':
              help_screen();
              break;
            case 'Q':     // Quit the program
            case 'q':
              done = true;
              break;
          }
      }
      while (!done);
}
```

The `help_screen` function needs no information from `main`, nor does it return a result. It behaves exactly the same way each time it is called. The `menu` function returns the character entered by the user.

9.4.3 Restricted Input

Listing 7.3 (betterinputonly.cpp) forces the user to enter a value within a specified range. We now can easily adapt that concept to a function. Listing 9.12 (betterinputfunc.cpp) uses a function named `get_int_range` that does not return until the user supplies a proper value.

Listing 9.12: `betterinputfunc.cpp`

```
#include <iostream>

/*
 *  get_int_range(first, last)
 *     Forces the user to enter an integer within a
 *     specified range
 *     first is either a minimum or maximum acceptable value
 *     last is the corresponding other end of the range,
 *     either a maximum or minimum *     value
 *     Returns an acceptable value from the user
 */
int get_int_range(int first, int last) {
    // If the larger number is provided first,
    // switch the parameters
    if (first > last) {
        int temp = first;
```

```
        first = last;
        last = temp;
    }
    // Insist on values in the range first...last
    std::cout << "Please enter a value in the range "
              << first << "..." << last << ": ";
    int in_value;   // User input value
    bool bad_entry;
    do {
        std::cin >> in_value;
        bad_entry = (in_value < first || in_value > last);
        if (bad_entry) {
            std::cout << in_value << " is not in the range "
                      << first << "..." << last << '\n';
            std::cout << "Please try again: ";
        }
    }
    while (bad_entry);
    // in_value at this point is guaranteed to be within range
    return in_value;
}

/*
 *  main
 *     Tests the get_int_range function
 */
int main() {
    std::cout << get_int_range(10, 20) << '\n';
    std::cout << get_int_range(20, 10) << '\n';
    std::cout << get_int_range(5, 5) << '\n';
    std::cout << get_int_range(-100, 100) << '\n';
}
```

Listing 9.12 (betterinputfunc.cpp) forces the user to enter a value within a specified range. This functionality could be useful in many programs.

In Listing 9.12 (betterinputfunc.cpp)

- The high and low values are specified by parameters. This makes the function more flexible since it could be used elsewhere in the program with a completely different range specified and still work correctly.
- The function is supposed to be called with the lower number passed as the first parameter and the higher number passed as the second parameter. The function will also accept the parameters out of order and automatically swap them to work as expected; thus,

  ```
  num = get_int_range(20, 50);
  ```

  will work exactly like

  ```
  num = get_int_range(50, 20);
  ```

- The Boolean variable `bad_entry` is used to avoid evaluating the Boolean expression twice (once to see if the bad entry message should be printed and again to see if the loop should continue).

9.4.4 Better Die Rolling Simulator

Listing 9.13: betterdie.cpp

```
#include <iostream>
#include <cstdlib>
#include <ctime>

/*
 *  initialize_die
 *     Initializes the randomness of the die
 */
void initialize_die() {
    // Set the random seed value
    srand(static_cast<unsigned>(time(0)));
}

/*
 *  show_die(spots)
 *     Draws a picture of a die with number of spots
 *     indicated
 *     spots is the number of spots on the top face
 */
void show_die(int spots) {
    std::cout <<  "+-------+\n";
    switch (spots) {
      case 1:
        std::cout << "|       |\n";
        std::cout << "|   *   |\n";
        std::cout << "|       |\n";
        break;
      case 2:
        std::cout << "| *     |\n";
        std::cout << "|       |\n";
        std::cout << "|     * |\n";
        break;
      case 3:
        std::cout << "|     * |\n";
        std::cout << "|   *   |\n";
        std::cout << "| *     |\n";
        break;
      case 4:
        std::cout << "| *   * |\n";
        std::cout << "|       |\n";
        std::cout << "| *   * |\n";
        break;
      case 5:
        std::cout << "| *   * |\n";
        std::cout << "|   *   |\n";
        std::cout << "| *   * |\n";
        break;
      case 6:
        std::cout << "| * * * |\n";
        std::cout << "|       |\n";
        std::cout << "| * * * |\n";
```

```
        break;
      default:
        std::cout <<  " ***  Error: illegal die value ***\n";
        break;
    }
    std::cout << "+-------+\n";
}

/*
 *  roll
 *     Returns a pseudorandom number in the range 1...6
 */
int roll() {
    return rand() % 6 + 1;
}

/*
 *  main
 *      Simulates the roll of a die three times
 */
int main() {

    // Initialize the die
    initialize_die();

    // Roll the die three times
    for (int i = 0; i < 3; i++)
        show_die(roll());
}
```

In Listing 9.13 (betterdie.cpp), `main` is no longer concerned with the details of pseudorandom number generation, nor is it responsible for drawing the die. These important components of the program are now in functions, so their details can be perfected independently from `main`.

Note how the result of the call to `roll` is passed directly as an argument to `show_die`.

9.4.5 Tree Drawing Function

Listing 9.14: `treefunc.cpp`

```
#include <iostream>

/*
 *  tree(height)
 *     Draws a tree of a given height
 *     height is the height of the displayed tree
 */
void tree(int height) {
    int row = 0;        // First row, from the top, to draw
    while (row < height) { // Draw one row for every unit of height
        // Print leading spaces
        int count = 0;
        while (count < height - row) {
            std::cout << " ";
```

```
            count++;
        }
        // Print out stars, twice the current row plus one:
        //  1. number of stars on left side of tree
        //     = current row value
        //  2. exactly one star in the center of tree
        //  3. number of stars on right side of tree
        //     = current row value
        count = 0;
        while (count < 2*row + 1) {
            std::cout << "*";
            count++;
        }
        // Move cursor down to next line
        std::cout << '\n';
        // Change to the next row
        row++;
    }

}

/*
 * main
 *    Allows users to draw trees of various heights
 */
int main() {
    int height;    // Height of tree
    std::cout << "Enter height of tree: ";
    std::cin >> height;  // Get height from user
    tree(height);
}
```

Observe that the name `height` is being used as a local variable in `main` and as a formal parameter name in `tree`. There is no conflict here, and the two `height`s represent two different locations in memory. Furthermore, the fact that the statement

```
tree(height);
```

uses `main`'s `height` as an actual parameter and `height` happens to be the name as the formal parameter is simply a coincidence. During the call, the value of `main`'s `height` variable is copied into to the formal parameter in `tree` also named `height`. The compiler can keep track of which `height` is which based on where each is declared.

9.4.6 Floating-point Equality

Recall from Listing 4.17 (imprecise5th.cpp) that floating-point numbers are not mathematical real numbers; a floating-point number is finite, and is represented internally as a quantity with a binary mantissa and exponent. Just as 1/3 cannot be represented finitely in the decimal (base 10) number system, 1/10 cannot be represented exactly in the binary (base 2) number system with a fixed number of digits. Often, no problems arise from this imprecision, and in fact many software applications have been written using floating-point numbers that must perform precise calculations, such as directing a spacecraft to a distant planet. In such cases even small errors can result in complete failures. Floating-point numbers can and are used safely and effectively, but not without appropriate care.

To build our confidence with floating-point numbers, consider Listing 9.15 (simplefloataddition.cpp), which increments a double-precision floating-point number by a double-precision floating-point amount and then checks it against a given value.

Listing 9.15: simplefloataddition.cpp

```
#include <iostream>

int main() {
    double x = 0.9;
    x += 0.1;
    if (x == 1.0)
        std::cout << "OK\n";
    else
        std::cout << "NOT OK\n";
}
```

All seems well judging by the output of Listing 9.15 (simplefloataddition.cpp):

```
OK
```

Next, consider Listing 9.16 (badfloatcheck.cpp) which attempts to control a loop with a double-precision floating-point number.

Listing 9.16: badfloatcheck.cpp

```
#include <iostream>

int main() {
    // Count to ten by tenths
    for (double i = 0.0; i != 1.0; i += 0.1)
        std::cout << "i = " << i << '\n';
}
```

When compiled and executed, Listing 9.16 (badfloatcheck.cpp) begins as expected, but it does not end as expected:

```
i = 0
i = 0.1
i = 0.2
i = 0.3
i = 0.4
i = 0.5
i = 0.6
i = 0.7
i = 0.8
i = 0.9
i = 1
i = 1.1
i = 1.2
i = 1.3
i = 1.4
i = 1.5
i = 1.6
i = 1.7
```

```
i = 1.8
i = 1.9
i = 2
i = 2.1
```

We expect it stop when the loop variable `i` equals 1, but the program continues executing until the user types **Ctrl-C**. We are adding 0.1, just as in Listing 9.15 (simplefloataddition.cpp), but now there is a problem. Since 0.1 cannot be represented exactly within the constraints of the double-precision floating-point representation, the repeated addition of 0.1 leads to round off errors that accumulate over time. Whereas 0.1 + 0.9 rounded off may equal 1, 0.1 added to itself 10 times may be 1.000001 or 0.999999, neither of which is exactly 1.

Listing 9.16 (badfloatcheck.cpp) demonstrates that the `==` and `!=` operators are of questionable worth when comparing floating-point values. The better approach is to check to see if two floating-point values are *close enough*, which means they differ by only a very small amount. When comparing two floating-point numbers x and y, we essentially must determine if the absolute value of their difference is small; for example, $|x-y| < 0.00001$. The C `abs` function was introduced in Section 8.2, and we can incorporate it into an equals function, as shown in Listing 9.17 (floatequals.cpp).

Listing 9.17: `floatequals.cpp`

```
#include <iostream>
#include <cmath>

/*
 *  equals(a, b, tolerance)
 *     Returns true if a = b or |a - b| < tolerance.
 *     If a and b differ by only a small amount
 *     (specified by tolerance), a and b are considered
 *     "equal."  Useful to account for floating-point
 *     round-off error.
 *     The == operator is checked first since some special
 *     floating-point values such as HUGE_VAL require an
 *     exact equality check.
 */
bool equals(double a, double b, double tolerance) {
    return a == b || fabs(a - b) < tolerance;
}

int main() {
    for (double i = 0.0; !equals(i, 1.0, 0.0001); i += 0.1)
        std::cout << "i = " << i << '\n';
}
```

The third parameter, named `tolerance`, specifies how close the first two parameters must be in order to be considered equal. The `==` operator must be used for some special floating-point values such as `HUGE_VAL`, so the function checks for `==` equality as well. Since C++ uses short-circuit evaluation for Boolean expressions involving logical *OR* (see Section 5.2), if the `==` operator indicates equality, the more elaborate check is not performed.

You should use a function like `equals` when comparing two floating-point values for equality.

9.4.7 Multiplication Table with Functions

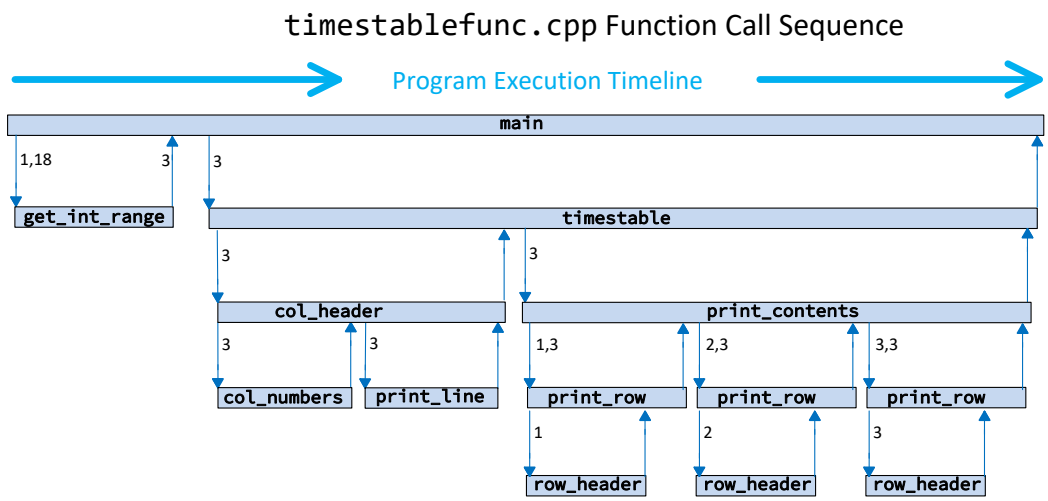
When we last visited our multiplication table program in Listing 7.5 (bettertimestable.cpp), we used nested `for` loops to display a user-specified size. We can decompose the code into functions as shown in Listing 9.18 (timestablefunction.cpp). Our goal is to have a collection of functions that each are very simple. We also want the program's overall structure to be logically organized.

Listing 9.18: `timestablefunction.cpp`

```
#include <iostream>
#include <iomanip>

// Print the column labels for an n x n multiplication table.
void col_numbers(int n) {
    std::cout << "      ";
    for (int column = 1; column <= n; column++)
        std::cout << std::setw(4) << column; // Print heading for this column.
    std::cout << '\n';
}

// Print the table's horizontal line at the top of the table
// beneath the column labels.
void col_line(int n) {
    std::cout << "     +";
    for (int column = 1; column <= n; column++)
        std::cout << "----";         // Print separator for this row.
    std::cout << '\n';
}

// Print the title of each column across the top of the table
// including the line separator.
void col_header(int n) {
    // Print column titles
    col_numbers(n);

    // Print line separator
    col_line(n);
}

// Print the title that appears before each row of the table's
// body.
void row_header(int n) {
    std::cout << std::setw(4) << n << " |";  // Print row label.
}

// Print the line of text for row n
// This includes the row number and the
// contents of each row.
void print_row(int row, int columns) {
    row_header(row);
    for (int col = 1; col <= columns; col++)
        std::cout << std::setw(4) << row*col;   // Display product
    std::cout << '\n';                  // Move cursor to next row
}
```

```
// Print the body of the n x n multiplication table
void print_contents(int n) {
    for (int current_row = 1; current_row <= n; current_row++)
        print_row(current_row, n);
}

// Print a multiplication table of size n x n.
void timestable(int n) {
    // First, print column heading
    col_header(n);

    // Print table contents
    print_contents(n);
}

// Forces the user to enter an integer within a
// specified range first is either a minimum or maximum
// acceptable value last is the corresponding other end
// of the range, either a maximum or minimum value
// Returns an acceptable value from the user
int get_int_range(int first, int last) {
    // If the larger number is provided first,
    // switch the parameters
    if (first > last) {
        int temp = first;
        first = last;
        last = temp;
    }
    // Insist on values in the range first...last
    std::cout << "Please enter a value in the range "
              << first << "..." << last << ": ";
    int in_value;   // User input value
    bool bad_entry;
    do {
        std::cin >> in_value;
        bad_entry = (in_value < first || in_value > last);
        if (bad_entry) {
            std::cout << in_value << " is not in the range "
                      << first << "..." << last << '\n';
            std::cout << "Please try again: ";
        }
    }
    while (bad_entry);
    // in_value at this point is guaranteed to be within range
    return in_value;
}

int main() {
    // Get table size from user; allow values in the
    // range 1...18.
    int size = get_int_range(1, 18);

    // Print a size x size multiplication table
    timestable(size);
}
```

Figure 9.5 A trace of the activation of the various functions in Listing 9.18 (timestablefunction.cpp) when the user enters 3 for a 3×3 multiplication table. Time flows from left to right. The horizontal bars show the lifetimes of the various functions involved.



In Listing 9.18 (timestablefunction.cpp), each function plays a very specific role; for example, `row_header` prints the label for a particular row and then prints the vertical bar that separates the row label from the body of the table. We reuse the `get_int_range` function from Listing 9.12 (betterinputfunc.cpp). Notice that there no longer appear to be any nested loops in the program. The nested loop is not really gone, it now is hidden by the program's functional composition. Observe that the `print_contents` function contains a loop. Each time through the loop it calls `print_row`, but `print_row` itself contains a loop. The nested iteration, therefore, is still present.

Realistically, the functional decomposition within Listing 9.18 (timestablefunction.cpp) is extreme. The relative simplicity of the program does not not really justify eight separate functions each with such a narrow focus; however, more complex software systems are decomposed in this very manner. Not only does Listing 9.18 (timestablefunction.cpp) give us insight into how we can take a complicated problem and break it down into simpler, more manageable pieces, we can use the program to better understand how the function invocation process works. To see how, consider the situation where a user wishes to print a 3×3 multiplication table using Listing 9.18 (timestablefunction.cpp).

In Figure 9.5 an arrow pointing down corresponds to a function call. The labels on the down arrows represent parameters passed during the call. The up arrows indicate the return from a function. The label on an up arrow is the function's return value. Functions of type `void` have no labels on their return arrows.

Each horizontal bar in Figure 9.5 represents the duration of the program's execution that the function is *active*. The `main` function always is the first function executed in a C++ program, and in Listing 9.18 (timestablefunction.cpp) `main` immediately calls the `get_int_range` function. During the lifetime of `get_int_range`'s execution notice that `main` is still active. This means any local variables it may maintain are still stored in memory and have not lost their values. This means that `main`'s `value` variable

exists for the effective life of the program. This is not true for the variables used in `get_int_range`: `temp`, `in_value`, `bad_entry`, and the parameters `first` and `last`. These variables maintained by `get_int_range` appear automatically when `get_int_range` begins executing (the left end of its bar) and their space is released when `get_int_range` is finished (the right end of its bar)[1]. During the printing of a 3×3 table the program calls `print_row` three times, and for each call the function's parameters `row` and `column` and local variable `col` come to life and then disappear when the function returns.

9.5 Organizing Functions

The code we have seen so far provides a function definition before its invocation. Listing 9.19 (organization1.cpp) organizes its functions this way.

Listing 9.19: `organization1.cpp`

```
// Function definition precedes main where the function is called
#include <iostream>

int twice(int n) {
    return 2 * n;
}

int main() {
    std::cout << twice(5) << '\n';
}
```

When the compiler attempts to parse the following statement:

```
std::cout << twice(5) << '\n';
```

it needs to be able to determine if the expression `twice(5)` is valid in the given context; that is, it must determine if the expression `twice(5)` evaluates to something that can be sent to the `std::cout` object using the `<<` operator. The following points are important in the parsing process:

- The earlier definition of the `twice` function declares the identifier `twice` to the compiler. Consequently the compiler expects a caller to `twice` to pass a single argument that is compatible with the `int` type. The compiler also knows that the call to `twice` will return an integer value.
- The parentheses after the name `twice` in the statement indicate the expression is a function call.
- Within the parentheses is the integer literal 5, which clearly is compatible with the `int` type.
- The expression `twice(5)` thus evaluates to an `int`, and the `<<` operator can send integer values to the `std::cout` output stream object.

The compiler thus can verify that the code within the `main` function is using the `twice` function correctly.

If we instead place `twice`'s definition after the `main` function's definition, the compiler will report an error where `twice`'s call appears within `main`. This is because the compiler parses C++ source code line by line from top to bottom within the .cpp file. When the compiler sees the identifier `twice` first at its

[1]Technically, the run-time environment does not allocate the space for a local variable until after its point of declaration. For variables declared within blocks, like `temp` with the `if` body, the variable is discarded at the end of the block's execution.

invocation within `main`, it is an undefined symbol. The compiler needs to know a function's return type and parameter types in order to verify that the caller is invoking the function correctly.

As it turns out, we can satisfy the compiler's need to know about the `twice` function as used in `main` without providing its full definition. Listing 9.20 (organization2.cpp) provides a alternate organization of the source code in Listing 9.19 (organization1.cpp).

Listing 9.20: `organization2.cpp`

```
// Function declaration precedes main; function definition follows main
#include <iostream>

int twice(int);   // Declare function named twice

int main() {
    std::cout << twice(5) << '\n';
}

int twice(int n) {  // Define function named twice
    return 2 * n;
}
```

The statement

```
int twice(int);
```

is a prototype that serves as a *function declaration*. It essentially is a function definition without the function's body. A semicolon terminates the declaration, and no curly braces appear in a function declaration. We can provide a name the parameter, but that is not necessary. A function declaration provides all the information the compiler needs to determine if a caller is invoking the function correctly. The function is available to any callers that appear after the declaration within the source file.

Listing 9.21 (organization3.cpp) provides a slight variation of Listing 9.20 (organization2.cpp).

Listing 9.21: `organization3.cpp`

```
// Function declaration local to main; function definition follows main
#include <iostream>

int main() {
    int twice(int);   // twice function available anywhere in main
    std::cout << twice(5) << '\n';
}

int twice(int n) {  // Define function twice
    return 2 * n;
}
```

In Listing 9.21 (organization3.cpp) we declare the `twice` function within the `main` function itself. The makes `twice`'s declaration local to `main`. This means that any code within `main` that appears after `twice`'s declaration can use `twice`, but code outside of `main` cannot (unless other functions themselves independently provide their own declaration of `twice`. Listing 9.22 (localfuncdef1.cpp) and Listing 9.23 (localfuncdef2.cpp) provide a concrete example of the difference between a local and a global declaration.

Listing 9.22: `localfuncdef1.cpp`

```
#include <iostream>

// Global declarations available to all functions that follow the declarations
int two_times(int);
int three_times(int);

int main() {
    std::cout << two_times(5) << '\n';
    std::cout << three_times(5) << '\n';
}

int three_times(int n) {
   return two_times(n) + n;    // 3n = 2n + n
}

int two_times(int n) {
    return 2 * n;
}
```

The declaration of `two_times` appears above and outside of both `main` and `three_times`, and so both `main` and `three_times` may legitimately call `two_times`.

Listing 9.23: `localfuncdef2.cpp`

```
// Local declaration in main
#include <iostream>


int main() {
    // These function declarations are local to main
    int two_times(int);
    int three_times(int);
    std::cout << two_times(5) << '\n';
    std::cout << three_times(5) << '\n';
}

// The two_times function is not declared for three_times function!
int three_times(int n) {
   return two_times(n) + n;    // Compiler error
}

int two_times(int n) {
    return 2 * n;
}
```

The declaration for the `two_times` function given in `main` in Listing 9.23 (localfuncdef2.cpp) is not visible in the `three_times` function; thus, the program will not compile.

Note that while C++ allows us to *declare* a function within the body of another function, it does not allow us to *define* a function inside the body of another function. C++ does not support nested function definitions.

Some programmers prefer to declare all their functions before their use within the program (as in Listing 9.20 (organization2.cpp) or Listing 9.21 (organization3.cpp)) and define the functions after their use

within `main`. They argue that this better represents the way the compiled code executes—the `main` method begins running first, and it calls the other functions. As we read the source code from top to bottom we see the details of the `main` function before seeing the implementation details of the functions that `main` calls.

A more frequent way to organize functions within C++ source code is to separate function declarations, function invocations, and function definitions into their own distinct files. The function declarations go into one or more .h header files, the function definitions appear in one or more .cpp files, and the calling code (including `main`) appear in a separate .cpp file. The compiler compiles all the .cpp files separately, producing multiple machine language object files (.obj files in Visual C++). The linker then combines these separate object files into a single executable file. Section 10.6 covers this technique is detail.

9.6 Commenting Functions

It is good practice to comment a function's definition with information that aids programmers who may need to use or extend the function. The essential information includes:

- **The purpose of the function**. The function's purpose is not always evident merely from its name. This is especially true for functions that perform complex tasks. A few sentences explaining what the function does can be helpful.
- **The role of each parameter**. The parameter names and types are obvious from the definition, but the purpose of a parameter may not be apparent merely from its name. It is helpful indicate the purpose of each parameter, if any.
- **The nature of the return value**. While the function may do a number of interesting things as indicated in the function's purpose, what exactly does it return to the caller? It is helpful to clarify exactly what value the function produces, if any.

Other information is often required in a commercial environment:

- **Author of the function**. Specify exactly who wrote the function. An email address can be included. If questions about the function arise, this contact information can be invaluable.
- **Date that the function's implementation was last modified**. An additional comment can be added each time the function is updated. Each update should specify the exact changes that were made and the person responsible for the update.
- **References**. If the code was adapted from another source, list the source. The reference may consist of a Web URL.

The following fragment shows the beginning of a well-commented function definition:

```
/*
 *  distance(x1, y1, x2, y2)
 *     Computes the distance between two geometric points
 *     x1 is the x coordinate of the first point
 *     y1 is the y coordinate of the first point
 *     x2 is the x coordinate of the second point
 *     y2 is the y coordinate of the second point
 *     Returns the distance between (x1,y1) and (x2,y2)
 *     Author: Joe Algori (joe@eng-sys.net)
```

```
 *     Last modified:  2010-01-06
 *     Adapted from a formula published at
 *     http://en.wikipedia.org/wiki/Distance
 */
double distance(double x1, double y1, double x2, double y2) {
    ...
```

From the information provided

- callers know what the function can do for them,
- callers know how to use the function,
- subsequent programmers can contact the original author if questions arise about its use or implementation,
- subsequent programmers can check the Wikipedia reference if questions arise about its implementation, and
- subsequent programmers can judge the quality of the algorithm based upon its source of inspiration (in this case, *Wikipedia*).

9.7 Custom Functions vs. Standard Functions

Armed with our knowledge of function definitions, we can rewrite Listing 8.1 (computesquareroot.cpp) so the program uses a custom square root function. Listing 9.24 (squarerootfunction.cpp) shows one possibility.

Listing 9.24: squarerootfunction.cpp

```
#include <iostream>

double square_root(double x) {
    double diff;
    // Compute a provisional square root
    double root = 1.0;

    do {  // Loop until the provisional root
          // is close enough to the actual root
        root = (root + x/root) / 2.0;
        std::cout << "root is " << root << '\n';
        // How bad is the approximation?
        diff = root * root - x;
    }
    while (diff > 0.0001 || diff < -0.0001);
    return root;
}

int main() {
    double input;

    // Get value from the user
    std::cout << "Enter number: ";
```

```
    std::cin >> input;
    // Report square root
    std::cout << "Square root of " << input << " = "
              << square_root(input) << '\n';
}
```

Is Listing 9.24 (squarerootfunction.cpp) better than Listing 8.2 (standardsquareroot.cpp) which uses the standard `sqrt` function from the `cmath` library? Generally speaking, if you have the choice of using a standard library function or writing your own custom function that provides the same functionality, choose to use the standard library routine. The advantages of using the standard library routine include:

- Your effort to produce the custom code is eliminated entirely; you can devote more effort to other parts of the application's development.
- If you write your own custom code, you must thoroughly test it to ensure its correctness; standard library code, while not immune to bugs, generally has been subjected to a complete test suite. Library code is used by many developers, and thus any lurking errors are usually exposed early; your code is exercised only by the programs you write, and errors may not become apparent immediately. If your programs are not used by a wide audience, bugs may lie dormant for a long time. Standard library routines are well known and trusted; custom code, due to its limited exposure, is suspect until it gains wider exposure and adoption.
- Standard routines are typically tuned to be very efficient; it takes a great deal of effort to make custom code efficient.
- Standard routines are well-documented; extra work is required to document custom code, and writing good documentation is hard work.

Listing 9.25 (squarerootcomparison.cpp) tests our custom square root function over a range of 1,000,000,000 floating-point values.

Listing 9.25: `squarerootcomparison.cpp`

```
#include <iostream>
#include <cmath>

// Consider two floating-point numbers equal when
// the difference between them is very small.
bool equals(double a, double b, double tolerance) {
    return a == b || fabs(a - b) < tolerance;
}

double square_root(double x) {
    // Compute a provisional square root
    double root = 1.0;

    do {  // Loop until the provisional root
          // is close enough to the actual root
        root = (root + x/root) / 2.0;
    }
    while (!equals(root*root, x, 0.0001));
    return root;
}
```

```
int main() {
    for (double d = 0.0; d < 100000.0; d += 0.0001) {
        if (!equals(square_root(d), sqrt(d), 0.001))
            std::cout << d << ": Expected " << sqrt(d) << ", but computed "
                      << square_root(d) << '\n';
    }
}
```

Listing 9.25 (squarerootcomparison.cpp) uses our `equals` function from Listing 9.17 (floatequals.cpp). The third parameter specifies a tolerance; if the difference between the first two parameters is less than the specified tolerance, the first two parameters are considered equal. Our new custom `square_root` function uses the `equals` function. The `main` function uses the `equals` function as well. Observe, however, that the tolerance used within the square root computation is smaller than the tolerance `main` uses to check the result. The `main` function, therefore, uses a less strict notion of equality.

The output of Listing 9.25 (squarerootcomparison.cpp):

```
0: Expected 0, but computed 0.0078125
0.0001: Expected 0.01, but computed 0.0116755
0.0008: Expected 0.0282843, but computed 0.0298389
0.0009: Expected 0.03, but computed 0.0313164
0.001: Expected 0.0316228, but computed 0.0327453
```

shows that our custom square root function produces results outside of `main`'s acceptable tolerance for five values. Five wrong answers out of one billion tests represents a 0.0000005% error rate. While this error rate is very small, indicates our `square_root` function is not perfect. One of values that causes the function to fail may be very important to a particular application, so our function is not trustworthy.

9.8 Exercises

1. Is the following a legal C++ program?

```
int proc(int x) {
    return x + 2;
}

int proc(int n) {
    return 2*n + 1;
}

int main() {
    int x = proc(5);
}
```

2. Is the following a legal C++ program?

```
int proc(int x) {
    return x + 2;
}

int main() {
```

```
    int x = proc(5),
        y = proc(4);
}
```

3. Is the following a legal C++ program?

```
#include <iostream>

void proc(int x) {
    std::cout << x + 2 << '\n';
}

int main() {
    int x = proc(5);
}
```

4. Is the following a legal C++ program?

```
#include <iostream>

void proc(int x) {
    std::cout << x + 2 << '\n';
}

int main() {
    proc(5);
}
```

5. Is the following a legal C++ program?

```
#include <iostream>

int proc(int x, int y) {
    return 2*x + y*y;
}

int main() {
    std::cout << proc(5, 4) << '\n';
}
```

6. Is the following a legal C++ program?

```
#include <iostream>

int proc(int x, int y) {
    return 2*x + y*y;
}

int main() {
    std::cout << proc(5) << '\n';
}
```

7. Is the following a legal C++ program?

```
#include <iostream>

int proc(int x) {
    return 2*x*x;
}

int main() {
    std::cout << proc(5, 4) << '\n';
}
```

8. Is the following a legal C++ program?

```
#include <iostream>

proc(int x) {
    std::cout << 2*x*x << '\n';
}

int main() {
    proc(5);
}
```

9. The programmer was expecting the following program to print 200. What does it print instead? Why does it print what it does?

```
#include <iostream>

void proc(int x) {
    x = 2*x*x;
}

int main() {
    int num = 10;
    proc(num);
    std::cout << num << '\n';
}
```

10. Is the following program legal since the variable `x` is used in two different places (`proc` and `main`)? Why or why not?

```
#include <iostream>

int proc(int x) {
    return 2*x*x;
}

int main() {
    int x = 10;
    std::cout << proc(x) << '\n';
}
```

11. Is the following program legal since the actual parameter has a different name from the formal parameter (y vs. x)? Why or why not?

```
#include <iostream>

int proc(int x) {
    return 2*x*x;
}

int main() {
    int y = 10;
    std::cout << proc(y) << '\n';
}
```

12. Complete the `distance` function started in Section 9.6. Test it with several point coordinates to convince yourself that your implementation is correct.

13. What happens if a caller passes too many parameters to a function?

14. What happens if a caller passes too few parameters to a function?

15. What are the rules for naming a function in C++?

16. Consider the following function definitions:

```
#include <iostream>

int fun1(int n) {
    int result = 0;
    while (n) {
        result += n;
        n--;
    }
    return result;
}

void fun2(int stars) {
    for (int i = 0; i < stars; i++)
        std::cout << "*";
    std::cout << '\n';
}

double fun3(double x, double y) {
    return 2*x*x + 3*y;
}

bool fun4(char ch) {
    return ch >='A' && ch <= 'Z';
}

bool fun5(int a, int b, int c) {
    return (a <= b) ? (b <= c) : false;
}
```

```
int fun6() {
    return rand() % 2;
}
```

Examine each of the following print statements. If the statement is illegal, explain why it is illegal; otherwise, indicate what the statement will print.

(a) `std::cout << fun1(5) << '\n';`

(b) `std::cout << fun1() << '\n';`

(c) `std::cout << fun1(5, 2) << '\n';`

(d) `std::cout << fun2(5) << '\n';`

(e) `fun2(5);`

(f) `std::cout << fun3(5, 2) << '\n';`

(g) `std::cout << fun3(5.0, 2.0) << '\n';`

(h) `std::cout << fun3('A', 'B') << '\n';`

(i) `std::cout << fun3(5.0) << '\n';`

(j) `std::cout << fun3(5.0, 0.5, 1.2) << '\n';`

(k) `std::cout << fun4('T') << '\n';`

(l) `std::cout << fun4('t') << '\n';`

(m) `std::cout << fun4(5000) << '\n';`

(n) `fun4(5000);`

(o) `std::cout << fun5(2, 4, 6) << '\n';`

(p) `std::cout << fun5(4, 2, 6) << '\n';`

(q) `std::cout << fun5(2, 2, 6) << '\n';`

(r) `std::cout << fun5(2, 6) << '\n';`

(s)
```
if (fun5(2, 2, 6))
   std::cout << "Yes\n";
else
   std::cout << "No\n";
```

(t) `std::cout << fun6() << '\n';`

(u) `std::cout << fun6(4) << '\n';`

(v) `std::cout << fun3(fun1(3), 3) << '\n';`

(w) `std::cout << fun3(3, fun1(3)) << '\n';`

(x) `std::cout << fun1(fun1(fun1(3))) << '\n';`

(y) `std::cout << fun6(fun6()) << '\n';`

Chapter 10

Managing Functions and Data

This chapter covers some additional aspects of functions in C++. Recursion, a key concept in computer science is introduced.

10.1 Global Variables

All variables to this point have been local to functions or local to blocks within the bodies of conditional or iterative statements. Local variables have some very desirable properties:

- A local variable occupies memory only when the variable is in scope. When the program execution leaves the scope of a local variable, it frees up the memory for that variable. This freed up memory is then available for use by the local variables in other functions during their invocations.
- We can use the same variable name in different functions without any conflict. The compiler derives all of its information about a local variable used within a function from the declaration of that variable in that function. The compiler will not look for the declaration of a local variable in the definition of another function. Thus, there is no way a local variable in one function can interfere with a local variable declared in another function.

A local variable is transitory, so its value is lost in between function invocations. Sometimes it is desirable to have a variable that lives as long as the program is running; that is, until the `main` function completes. In contrast to a local variable, a *global* variable is declared outside of all functions and is not local to any particular function. In fact, any function that appears in the text of the source code after the point of the global variable's declaration may legally access and/or modify that global variable.

Listing 10.1 (globalcalculator.cpp) is a modification of Listing 9.11 (calculator.cpp) that uses a global variable named `result` that is shared by several functions in the program.

Listing 10.1: `globalcalculator.cpp`

```
#include <iostream>
#include <cmath>

/*
 *  help_screen
```

```
 *     Displays information about how the program works
 *     Accepts no parameters
 *     Returns nothing
 */
void help_screen() {
    std::cout << "Add:  Adds two numbers\n";
    std::cout << "      Example: a 2.5 8.0\n";
    std::cout << "Subtract:  Subtracts two numbers\n";
    std::cout << "      Example: s 10.5 8.0\n";
    std::cout << "Print:  Displays the result of the latest operation\n";
    std::cout << "      Example: p\n";
    std::cout << "Help:  Displays this help screen\n";
    std::cout << "      Example: h\n";
    std::cout << "Quit:  Exits the program\n";
    std::cout << "      Example: q\n";
}

/*
 *  menu
 *     Display a menu
 *     Accepts no parameters
 *     Returns the character entered by the user.
 */
char menu() {
    // Display a menu
    std::cout << "=== A)dd S)ubtract P)rint H)elp Q)uit ===\n";
    // Return the char entered by user
    char ch;
    std::cin >> ch;
    return ch;
}

/*
 *  Global variables used by several functions
 */
double result = 0.0, arg1, arg2;

/*
 *  get_input
 *      Assigns the globals arg1 and arg2 from user keyboard
 *      input
 */
void get_input() {
    std::cin >> arg1 >> arg2;
}

/*
 *  report
 *      Reports the value of the global result
 */
void report() {
    std::cout << result << '\n';
}

/*
```

```
 *  add
 *     Assigns the sum of the globals arg1 and arg2
        to the global variable result
 */

void add() {
   result = arg1 + arg2;
}

/*
 *  subtract
 *     Assigns the difference of the globals arg1 and arg2
 *     to the global variable result
 */

void subtract() {
   result = arg1 - arg2;
}

/*
 *  main
 *     Runs a command loop that allows users to
 *     perform simple arithmetic.
 */
int main() {
    bool done = false;  // Initially not done
    do {
        switch (menu()) {
          case 'A':     // Addition
          case 'a':
            get_input();
            add();
            report();
            break;
          case 'S':     // Subtraction
          case 's':
            get_input();
            subtract();
            report();
            break;
          case 'P':     // Print result
          case 'p':
            report();
            break;
          case 'H':     // Display help screen
          case 'h':
            help_screen();
            break;
          case 'Q':     // Quit the program
          case 'q':
            done = true;
            break;
        }
    }
    while (!done);
```

```
}
```

Listing 10.1 (globalcalculator.cpp) uses global variables `result`, `arg1`, and `arg2`. These names no longer appear in the `main` function. These global variables are accessed and/or modified in four different functions: `get_input`, `report`, `add`, and `subtract`.

When in the course of translating the statements within a function to machine language, the compiler resolves a variable it encounters as follows:

- If the variable has a local declaration (that is, it is a local variable or parameter), the compiler will use the local variable or parameter, even if a global variable of the same name exists. Local variables, therefore, take precedence over global variables. We say the local declaration *hides* the global declaration in the scope of the local variable.
- If the variable has no local declaration but is declared as a global variable, the compiler will use the global variable.
- If the variable has neither a local declaration nor a global declaration, then the variable is undefined, and its use is an error.

In the situation where a local variable hides a global variable of the same name, there is a way to access both the local variable and like-named global variable within the local variable's scope. Suppose a program has a global variable named `x` and a function with a local variable named `x`. The statement

```
x = 10;
```

within the scope of the local variable will assign the local `x`. The following statement will assign the global variable `x` in the scope of the local variable of the same name:

```
::x = 10;
```

The `::` operator is called the *scope resolution* operator. This special syntax may be used whenever a global variable is accessed within a function, but usually it only used when necessary to access a hidden global variable.

If the value of a local variable is used by a statement before that variable has been given a value, either through initialization or assignment, the compiler will issue a warning. For example, the Visual C++ compiler will issue a warning about code in the following function:

```
void uninitialized() {
    int x;  // Declare the variable
    std::cout << x;  // Then use it
}
```

The warning is

warning C4700: uninitialized local variable 'x' used

(The only way to avoid this warning in Visual C++ is to turn off *all* warnings.) A local variable has an undefined value after it is declared without being initialized. Its value should not be used until it has been properly assigned. Global variables, however, do not need to be initialized before they are used. Numeric global variables are automatically assigned the value zero. This means the initialization of `result` in Listing 10.1 (globalcalculator.cpp) is superfluous, since `result` will be assigned zero automatically. Boolean global variables are automatically assigned zero as well, as zero represents `false` (see Section 5.1).

When it is acceptable to use global variables, and when is it better to use local variables? In general, local variables are preferred to global variables for several reasons:

- When a function uses local variables exclusively and performs no other input operations (like using the `std::cin` object), its behavior is influenced only by the parameters passed to it. If a non-local variable appears, the function's behavior is affected by every other function that can modify that non-local variable. As a simple example, consider the following trivial function that appears in a program:

  ```
  int increment(int n) {
      return n + 1;
  }
  ```

  Can you predict what the following statement within that program will print?

  ```
  std::cout << increment(12) << '\n';
  ```

  If your guess is 13, you are correct. The `increment` function simply returns the result of adding one to its argument. The `increment` function behaves the same way each time it is called with the same argument.

  Next, consider the following three functions that appear in some program:

  ```
  int process(int n) {
      return n + m;  // m is a global integer variable
  }

  void assign_m() {
      m = 5;
  }

  void inc_m() {
      m++;
  }
  ```

  Can you predict the what the following statement within the program will print?

  ```
  std::cout << process(12) << '\n';
  ```

  We cannot predict what this statement in isolation will print. The following scenarios all produce different results:

  ```
  assign_m();
  std::cout << process(12) << '\n';
  ```

  prints 17,

  ```
  m = 10;
  std::cout << process(12) << '\n';
  ```

  prints 22,

  ```
  m = 0;
  inc_m();
  inc_m();
  std::cout << process(12) << '\n';
  ```

prints 14, and

```
assign_m();
inc_m();
inc_m();
std::cout << process(12) << '\n';
```

prints 19. The identical printing statements print different values depending on the cumulative effects of the program's execution up to that point.

It may be difficult to locate an error if that function fails because it may be the fault of *another* function that assigned an incorrect value to the global variable. The situation may be more complicated than the simple examples above; consider:

```
assign_m();
.
.   /* 30 statements in between, some of which may change a,
.      b, and m  */
.
if (a < 2 && b <= 10)
    m = a + b - 100;
.
.   /* 20 statements in between, some of which may change m */
.
std::cout << process(12) << '\n';
```

- A nontrivial program that uses non-local variables will be more difficult for a human reader to understand than one that does not. When examining the contents of a function, a non-local variable requires the reader to look elsewhere (outside the function) for its meaning:

  ```
  // Linear function
  double f(double x) {
      return m*x + b;
  }
  ```

  What are `m` and `b`? How, where, and when are they assigned or re-assigned?

- A function that uses only local variables can be tested for correctness in isolation from other functions, since other functions do not affect the behavior of this function. This function's behavior is only influenced only by its parameters, if it has any.

The exclusion of global variables from a function leads to *functional independence*. A function that depends on information outside of its scope to correctly perform its task is a dependent function. When a function operates on a global variable it depends on that global variable being in the correct state for the function to complete its task correctly. Nontrivial programs that contain many dependent functions are more difficult to debug and extend. A truly independent function that uses no global variables and uses no programmer-defined functions to help it out can be tested for correctness in isolation. Additionally, an independent function can be copied from one program, pasted into another program, and work without modification. Functional independence is a desirable quality.

Unlike global variables, global constants are generally safe to use. Code within functions that use global constants are dependent on those constants, but since constants cannot be changed, developers need not worry that other functions that have access to the global constants might disturb their values.

The use of global constants within functions has drawbacks in terms of program maintenance. As a program evolves, code is added and removed. If a global constant is removed or its meaning changes during the course of the program's development, the change will affect any function using the global constant.

Listing 10.2 (digitaltimer.cpp) uses global constants to assist the display of a digital timer.

Listing 10.2: digitaltimer.cpp

```
#include <iostream>
#include <iomanip>
#include <ctime>

// Some conversions from seconds
const clock_t SEC_PER_MIN = 60,                  // 60 sec  =  1 min
              SEC_PER_HOUR = 60 * SEC_PER_MIN,   // 60 min  =  1 hr
              SEC_PER_DAY = 24 * SEC_PER_HOUR;   // 24 hr   = 24 hr

/*
 *  print_time
 *     Displays the time in hr:min:sec format
 *     seconds is the number of seconds to display
 */

void print_time(clock_t seconds) {
    clock_t hours = 0, minutes = 0;

    // Prepare to display time ============================
    std::cout << '\n';
    std::cout << "     ";

    // Compute and display hours ==========================
    hours = seconds/SEC_PER_HOUR;
    std::cout << std::setw(2) << std::setfill('0') << hours << ":";

    // Remove the hours from seconds
    seconds %= SEC_PER_HOUR;

    // Compute and display minutes ========================
    minutes = seconds/SEC_PER_MIN;
    std::cout << std::setw(2) << std::setfill('0') << minutes << ":";
    // Remove the minutes from seconds
    seconds %= SEC_PER_MIN;

    // Compute and display seconds ========================
    std::cout << std::setw(2) << std::setfill('0') << seconds << '\n';
}

int main() {
    clock_t start = clock();      // Record starting time
    clock_t elapsed = (clock() - start)/CLOCKS_PER_SEC, // Elapsed time in sec.
            previousElapsed = elapsed;
    // Counts up to 24 hours (1 day), then stops
    while (elapsed < SEC_PER_DAY) {
        // Update the display only every second
        if (elapsed - previousElapsed >= 1) {
            // Remember when we last updated the display
```

```
            previousElapsed = elapsed;
            print_time(elapsed);
        }
        // Update elapsed time
        elapsed = (clock() - start)/CLOCKS_PER_SEC;
    }
}
```

In Listing 10.2 (digitaltimer.cpp):

- The `main` function controls the time initialization and update and deals strictly in seconds. The logic in `main` is kept relatively simple.
- The code that extracts the hours, minutes, and seconds from a given number of seconds is isolated in `print_time`. The `print_time` function can now be used anytime a value in seconds needs to be expressed in the *hours* `:` *minutes* `:` *seconds* format.
- The second conversion constants (`SEC_PER_HOUR`, `SEC_PER_MIN`, and `SEC_PER_DAY`) are global constants so that both functions can access them if necessary. In this case the functions use different constants , but it makes sense to place all the conversion factors in one place.

Since the two functions divide the responsibilities in a way that each can be developed independently, the design is cleaner and the program is easier to develop and debug. The use of constants ensures that the shared values cannot be corrupted by either function.

The exclusion from a function's definition of global variables and global constants does not guarantee that it will always produce the same results given the same parameter values; consider

```
int compute(int n) {
    int favorite;
    std::cout << "Please enter your favorite number: ";
    std::cin >> favorite;
    return n + favorite;
}
```

The `compute` function avoid globals, yet we cannot predict the value of the expression `compute(12)`. Recall the `increment` function from above:

```
int increment(int n) {
    return n + 1;
}
```

Its behavior is totally predictable. Furthermore, `increment` does not modify any global variables, meaning it cannot in any way influence the overall program's behavior. We say that `increment` is a *pure function*. A pure function cannot perform any input or output (for example, use the `std::cout` and `std::cin` objects), nor may it use global variables. While `increment` is pure, the `compute` function is impure. The following function is impure also, since it performs output:

```
int increment_and_report(int n) {
    std::cout << "Incrementing " << n << '\n';
    return n + 1;
}
```

A pure function simply computes its return value and has no other observable side effects.

A function that uses only pure functions and otherwise would be considered pure is itself a pure function; for example:

```
int double_increment(int n) {
    return increment(n) + 1;
}
```

`double_increment` is a pure function since `increment` is pure; however, `double_increment_with_report`:

```
int double_increment_with_report(int n) {
    return increment_and_report(n) + 1;
}
```

is not a pure function since it calls `increment_and_report` which is impure.

10.2 Static Variables

Space in the computer's memory for local variables and function parameters is allocated at run time when the function begins executing. When the function is finished and returns, the memory used for the function's local variables and parameters is freed up for other purposes. If a function is never called, the variable's local variables and parameters will never occupy the computer's memory.

Because a function's locals are transitory, a function cannot ordinarily retain any information between calls. C++ provides a way in which a variable local to a function can be retained in between calls. Listing 10.3 (counter.cpp) shows how declaring a local variable `static` allows it to remain in the computer's memory for the duration of the program's execution.

Listing 10.3: `counter.cpp`

```
#include <iostream>
#include <iomanip>

/*
 *  count
 *     Keeps track of a count.
 *     Returns the current count.
 */

int count() {
    // cnt's value is retained between calls because it
    // is declared static
    static int cnt = 0;
    return ++cnt;   // Increment and return current count
}

int main() {
    // Count to ten
    for (int i = 0;  i < 10;  i++)
       std::cout << count() << ' ';
    std::cout << '\n';
}
```

In Listing 10.3 (counter.cpp), the `count` function is called 10 times. Each time it returns a tally of the number of times it has been called:

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

By contrast, if you remove the word `static` from Listing 10.3 (counter.cpp), recompile it, and rerun it, it prints:

```
1 1 1 1 1 1 1 1 1 1
```

because new memory is allocated for the `cnt` variable each time `count` is called.

The local declaration

```
static int cnt = 0;
```

allocates space for `cnt` and assigns zero to it once—at the beginning of the program's execution. The space set aside for `cnt` is not released until the program finishes executing.

Recall Listing 9.7 (evenbetterprompt.cpp) that included the following function:

```
int prompt(int n) {
    int value;
    std::cout << "Please enter integer #" << n << ": ";
    std::cin >> value;
    return value;
}
```

The caller, `main`, used `prompt` function as follows:

```
int value1, value2, sum;
std::cout << "This program adds together two integers.\n";
value1 = prompt(1);    // Call the function
value2 = prompt(2);    // Call the function again
sum = value1 + value2;
```

The first call to `prompt` prints the message

```
Please enter integer #1:
```

and awaits the user's input. The second call prints the message

```
Please enter integer #2:
```

Another caller might use `prompt` within a loop like this:

```
int sum = 0;
for (int i = 1; i < 10; i++)
    sum += prompt(i);
```

Notice that it is the caller's responsibility to keep track of the proper number to pass to `prompt`. The caller may make a mistake and pass the wrong number or may not want to manage such details. It would be better to move the responsibility of tracking input count to `prompt`. `static` variables make that possible, as Listing 10.4 (promptwithstatic.cpp)

Listing 10.4: promptwithstatic.cpp

```
#include <iostream>

/*
 *  prompt requests an integer from the user and
 *  keeps track of the cumulative number of entries.
 *  Returns the value entered by the user.
 */
int prompt() {
    static int count = 0;
    int value;
    std::cout << "Please enter integer #" << ++count << ": ";
    std::cin >> value;
    return value;
}

int main() {
    int value1, value2, sum;
    std::cout << "This program adds together two integers.\n";
    value1 = prompt();    // Call the function
    value2 = prompt();    // Call the function again
    sum = value1 + value2;
    std::cout << value1 << " + " << value2 << " = " << sum << '\n';
}
```

Listing 10.4 (promptwithstatic.cpp) behaves just like Listing 9.7 (evenbetterprompt.cpp) but in the new version `main` does not have to keep track of the number of user entries.

Local `static` variables were inherited from the C programming language, but their need has diminished with the introduction of objects in C++ (see Chapter 14). Functions with `static` variables provide a way to implement executable code with persistent state. Objects provide a more natural and more flexible way to achieve the same effect.

10.3 Overloaded Functions

In C++, a program can have multiple functions with the same name. When two or more functions within a program have the same name, the function is said to be *overloaded*. The functions must be different somehow, or else the compiler would not know how to associate a call with a particular function definition. The compiler identifies a function by more than its name; a function is uniquely identified by its *signature*. A function signature consists of the function's name and its parameter list. In the parameter list, only the types of the formal parameters are important, not their names. If the parameter types do not match exactly, both in number and position, then the function signatures are different. Consider the following overloaded functions:

1. `void f() { /* ... */ }`

   This version has no parameters, so its signature differs from all the others which each have at least one parameter.

2. `void f(int x) { /* ... */ }`

This version differs from Version 3, since its single parameter is an `int`, not a `double`.

3. `void f(double x) { /* ... */ }`

   This version differs from Version 2, since its single parameter is a `double`, not an `int`.

4. `void f(int x, double y) { /* ... */ }`

   This version differs from Version 5 because, even though Versions 4 and 5 have the same number of parameters with the same types, the order of the types is different.

5. `void f(double x, int y) { /* ... */ }`

   See the comments for Version 4.

Overloaded functions are a convenience for programmers. If overloaded functions were not allowed (many programming languages do not support function overloading), new function names must be created for different functions that perform basically the same task but accept different parameter types. It is better for the programmer to choose the same name for the similar functions and let the compiler properly resolve the differences. Overloading becomes a more important issue for constructors, special functions called during object creation (Chapter 14).

10.4 Default Arguments

We can define functions that accept a varying number of parameters. Consider Listing 10.5 (countdown.cpp) that specifies a function that counts down:

Listing 10.5: `countdown.cpp`

```
#include <iostream>

// Prints a count down from n to zero.  The default
// starting value is 10.
void countdown(int n=10) {
    while (n >= 0) // Count down from n to zero
        std::cout << n-- << '\n';
}

int main() {
    countdown(5);
    std::cout << "----------" << '\n';
    countdown();
}
```

The formal parameter to the `countdown` function expressed as `n=10` represents a default parameter or default argument. If the caller does not supply an actual parameter, the formal parameter `n` is assigned 10. The following call

```
countdown()
```

prints

```
10
9
8
7
6
5
4
3
2
1
0
```

but the invocation

```
countdown(5)
```

displays

```
5
4
3
2
1
0
```

As we can see, when the caller does not supply a parameter specified by a function, and that parameter has a default value, the default value is used during the caller's call.

We may mix non-default and default parameters in the parameter lists of a function declaration, but all default parameters within the parameter list must appear after all the non-default parameters. This means the following definitions are acceptable:

```
int sum_range(int n, int m=100) {   // OK, default follows non-default
    int sum = 0;
    for (int val = n; val <= m; val++)
        sum += val;
    return val;
}
```

and

```
int sum_range(int n=0, int m=100) {   // OK, both default
    int sum = 0;
    for (int val = n; val <= m; val++)
        sum += val;
    return val;
}
```

but the following definition is illegal, since a default parameter precedes a non-default parameter in the function's parameter list:

```
int sum_range(int n=0, int m) { // Illegal, non-default follows default
    int sum = 0;
    for (int val = n; val <= m; val++)
```

```
        sum += val;
    return val;
}
```

Default arguments allow programmers to provide a highly tunable function that offer a simpler interface for its typical uses.

Overloading (see Section 10.3) enables programmers to write different function definitions for two different functions that have the same name. Mixing overloading and default arguments can produce ambiguities that the compiler will not allow; for example, the following overloaded functions are acceptable:

```
void f() { /* .... */ }
void f(int n) { /* .... */ }
```

as this overloads function `f`. If a caller invokes `f` as `f()`, the compiler will generate machine language code to call the first version of `f`. If instead we attempt to overload `f` as

```
void f() { /* .... */ }
void f(int n=0) { /* .... */ }
```

the compiler cannot determine if the call `f()` means the first overloaded version or the second with the parameter defaulting to zero. Because of this ambiguity, the compiler will report an error for attempts to call function `f` with no arguments. The other possible combination is illegal:

```
void f(int m) { /* .... */ }
void f(int n=0) { /* .... */ }
```

because these functions have the same signature, `f(int)`. C++ does not allow a program to contain multiple function definitions with the same signature.

10.5 Recursion

The *factorial* function is widely used in combinatorial analysis (counting theory in mathematics), probability theory, and statistics. The factorial of n is often expressed as $n!$. Factorial is defined for nonnegative integers as

$$n! \;=\; n \cdot (n-1) \cdot (n-2) \cdot (n-3) \cdots 2 \cdot 1$$

and $0!$ is defined to be 1. Thus $6! = 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 720$. Mathematicians precisely define factorial in this way:

$$n! \;=\; \begin{cases} 1 & \text{if } n \;=\; 0 \\ \\ n \cdot (n-1)! & \text{otherwise.} \end{cases}$$

This definition is *recursive* since the ! function is being defined, but ! is also used in the definition. A C++ function can be defined recursively as well. Listing 10.6 (factorialtest.cpp) includes a factorial function that exactly models the mathematical definition.

Listing 10.6: `factorialtest.cpp`

```
#include <iostream>

/*
 *  factorial(n)
```

```
 *     Computes n!
 *     Returns the factorial of n.
 */
int factorial(int n) {
    if (n == 0)
        return 1;
    else
        return n * factorial(n - 1);
}

int main() {
    // Try out the factorial function
    std::cout << " 0! = " << factorial(0) << '\n';
    std::cout << " 1! = " << factorial(1) << '\n';
    std::cout << " 6! = " << factorial(6) << '\n';
    std::cout << "10! = " << factorial(10) << '\n';
}
```

Observe that the `factorial` function in Listing 10.6 (factorialtest.cpp) uses no loop to compute its result. The `factorial` function simply calls itself. The call `factorial(6)` is computed as follows:

```
factorial(6) = 6 * factorial(5)
             = 6 * 5 * factorial(4)
             = 6 * 5 * 4 * factorial(3)
             = 6 * 5 * 4 * 3 * factorial(2)
             = 6 * 5 * 4 * 3 * 2 * factorial(1)
             = 6 * 5 * 4 * 3 * 2 * 1 * factorial(0)
             = 6 * 5 * 4 * 3 * 2 * 1 * 1
             = 6 * 5 * 4 * 3 * 2 * 1
             = 6 * 5 * 4 * 3 * 2
             = 6 * 5 * 4 * 6
             = 6 * 5 * 24
             = 6 * 120
             = 720
```

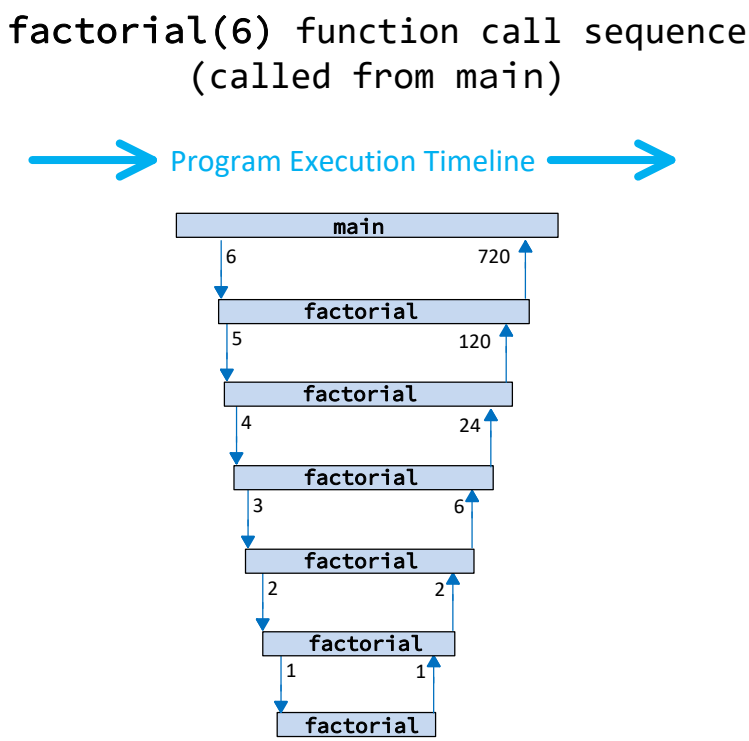
Note that the `factorial` function can be slightly optimized by changing the `if`'s condition from `(n == 0)` to `(n < 2)`. This change results in a function execution trace that eliminates two function calls at the end:

```
factorial(6) = 6 * factorial(5)
             = 6 * 5 * factorial(4)
             = 6 * 5 * 4 * factorial(3)
             = 6 * 5 * 4 * 3 * factorial(2)
             = 6 * 5 * 4 * 3 * 2 * 1
             = 6 * 5 * 4 * 3 * 2
             = 6 * 5 * 4 * 6
             = 6 * 5 * 24
             = 6 * 120
             = 720
```

Figure 10.1 shows the call sequence for `factorial(6)` invoked from within a `main` function.

A correct simple recursive function definition is based on four key concepts:

Figure 10.1 Traces the function activations of the recursive function `factorial` when called from `main` with an argument of 6. The arrows into an activation bar indicates the argument passed by the caller; the arrows out show the value passed back to the caller. The length of a bar represents the time during which that invocation of the function is active.



1. The function must optionally call itself within its definition; this is the *recursive case*.
2. The function must optionally *not* call itself within its definition; this is the *base case*.
3. Some sort of conditional execution (such as an `if/else` statement) selects between the recursive case and the base case based on one or more parameters passed to the function.
4. Each invocation that does not correspond to the base case must call itself with parameter(s) that move the execution closer to the base case. The function's recursive execution must converge to the base case.

Each recursive invocation must bring the function's execution closer to it base case. The `factorial` function calls itself in the `else` clause of the `if/else` statement. Its base case is executed if the condition of the `if` statement is true. Since the factorial is defined only for nonnegative integers, the initial invocation of `factorial` must be passed a value of zero or greater. A zero parameter (the base case) results in no recursive call. Any other positive parameter results in a recursive call with a parameter that is closer to zero than the one before. The nature of the recursive process progresses towards the base case, upon which the recursion terminates.

We can easily write a non-recursive factorial function, as Listing 10.7 (nonrecursfact.cpp) shows.

Listing 10.7: `nonrecursfact.cpp`

```
#include <iostream>

/*
 *  factorial(n)
 *     Computes n!
 *     Returns the factorial of n.
 */
int factorial(int n) {
    int product = 1;
    for (int i = n;  i > 0;  i--)
        product *= i;
    return product;
}

int main() {
    // Try out the factorial function
    std::cout << " 0! = " << factorial(0) << '\n';
    std::cout << " 1! = " << factorial(1) << '\n';
    std::cout << " 6! = " << factorial(6) << '\n';
    std::cout << "10! = " << factorial(10) << '\n';
}
```

Which `factorial` function is better, the recursive or non-recursive version? Generally, if the same basic algorithm is being used by both the recursive and non-recursive functions, then the non-recursive function will be more efficient. A function call is a relatively expensive operation compared to a variable assignment or comparison. The body of the non-recursive `factorial` function invokes no functions, but the recursive version calls a function—it calls itself—during all but the last recursive invocation. The iterative version of `factorial` is therefore more efficient than the recursive version.

Even though the iterative version of the factorial function is technically more efficient than the recursive version, on most systems you could not tell the difference. The execution time difference between the two versions is negligible. The reason is the factorial function "grows" fast, meaning it returns fairly large

results for relatively small arguments. In particular, `factorial(13)` is the largest value that fits within a 32-bit integer. Neither the iterative nor recursive version of `factorial` can compute 14! using the 32-bit `int` type.

Recall the `gcd` functions from Section 9.2. It computed he greatest common divisor (also known as greatest common factor) of two integer values. It works, but it is not very efficient. A better algorithm is used in Listing 10.8 (gcd.cpp). It is based on one of the oldest algorithms known, developed by the Greek mathematician Euclid around 300 B.C.

Listing 10.8: `gcd.cpp`

```
#include <iostream>

/*
 *  gcd(m, n)
 *     Uses Euclid's method to compute the greatest common divisor
 *     (also called greatest common factor) of  m and n.
 *     Returns the GCD of m and n.
 */
int gcd(int m, int n) {
    if (n == 0)
        return m;
    else
       return gcd(n, m % n);
}

int iterative_gcd(int num1, int num2) {
    // Determine the smaller of num1 and num2
    int min = (num1 < num2) ? num1 : num2;
    // 1 is definitely a common factor to all ints
    int largestFactor = 1;
    for (int i = 1;  i <= min;  i++)
        if (num1 % i == 0 && num2 % i == 0)
            largestFactor = i; // Found larger factor
    return largestFactor;
}

int main() {
    // Try out the gcd functions
    const int BEGIN = 1000000000,
              END   = 1000000003;
    for (int num1 = BEGIN; num1 <= END; num1++)
        for (int num2 = BEGIN; num2 <= END; num2++)
                std::cout << "iterative_gcd(" << num1 << "," << num2
                          << ") = " << iterative_gcd(num1, num2) << '\n';
    for (int num1 = BEGIN; num1 <= END; num1++)
        for (int num2 = BEGIN; num2 <= END; num2++)
                std::cout << "gcd(" << num1 << "," << num2
                          << ") = " << gcd(num1, num2) << '\n';
}
```

Running Listing 10.8 (gcd.cpp) you will see that the `gcd` and `iterative_gcd` functions compute the same results given the same arguments. Listing 10.8 (gcd.cpp) showcases the difference in performance between the two functions by computing the GCD of relatively large integers. The `gcd` function produces its result *much* faster than `iterative_gcd`. Note that this `gcd` function is recursive. The algorithm it

uses is much different from our original iterative version. Because of the difference in the algorithms, this recursive version is actually much more efficient than our original iterative version. A recursive function, therefore, cannot be dismissed as inefficient just because it is recursive.

While Listing 10.8 (gcd.cpp) expresses the `gcd` functions recursively, it is not hard to rewrite it so that it still follows Euclid's algorithm but uses a loop instead of recursion—this very task appears as an exercise at the end of the chapter. Often the concept of an algorithm solving certain kinds of problems is better understood when expressed recursively. Later, once the details of the recursive version are perfected, developers may rewrite the algorithm in an iterative fashion.

Listing 10.9 (histobar.cpp) provides another example of a recursive function. The `segments1` function uses iteration to draw segments that make up a bar that could be part of a histogram. The `segments2` function does that same thing, except it uses recursion.

Listing 10.9: `histobar.cpp`

```
#include <iostream>

// Draws a bar n segments long
// using iteration.
void segments1(int n) {
    while (n > 0) {
        std::cout << "*";
        n--;
    }
    std::cout << '\n';
}

// Draws a bar n segments long
// using recursion.
void segments2(int n) {
    if (n > 0) {
        std::cout << "*";
        segments2(n - 1);
    }
    else
        std::cout << '\n';
}

int main() {
    segments1(3);
    segments1(10);
    segments1(0);
    segments1(5);
    std::cout << "-------------\n";
    segments2(3);
    segments2(10);
    segments2(0);
    segments2(5);
}
```

The outout of Listing 10.9 (histobar.cpp) shows that the two functions produce the same results:

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

```
*****
-------------
***
**********

*****
```

A recursive function may call itself within its definition multiple times. Consider the following sequence of integer values:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots$$

This is beginning of the infinite *Fibonacci sequence* (see `https://en.wikipedia.org/wiki/Fibonacci_number`). It is a sequence of integers beginning with 0 followed by 1. Subsequent elements of the sequence are the sum of their two immediately preceding elements; thus, the third number is $0+1=1$, the fourth number is $1+1=2$, the fifth number is $1+2=3$, etc. The numbers that comprise the Fibonacci sequence are known as *Fibonacci numbers*. Note that 3 is a Fibonacci number but 4 is not.

The mathematical properties of Fibonacci numbers have bearing in such diverse fields as biology, economics, and art.

A common problem is computing the nth Fibonacci number. Zero is the 0th, 1 is the 1st, 1 is also the 2nd, 2 is the 3rd, 3 is the 4th, 5 is the 5th, etc.

A recursive C++ function to compute the nth Fibonacci number follows easily from the definition of the Fibonacci sequence:

```
// Returns the nth Fibonacci number
int fibonacci(int n) {
    if (n <= 0)
        return 0;
    else if (n == 1)
        return 1;
    else
        return fibonacci(n - 2) + fibonacci(n - 1);
}
```

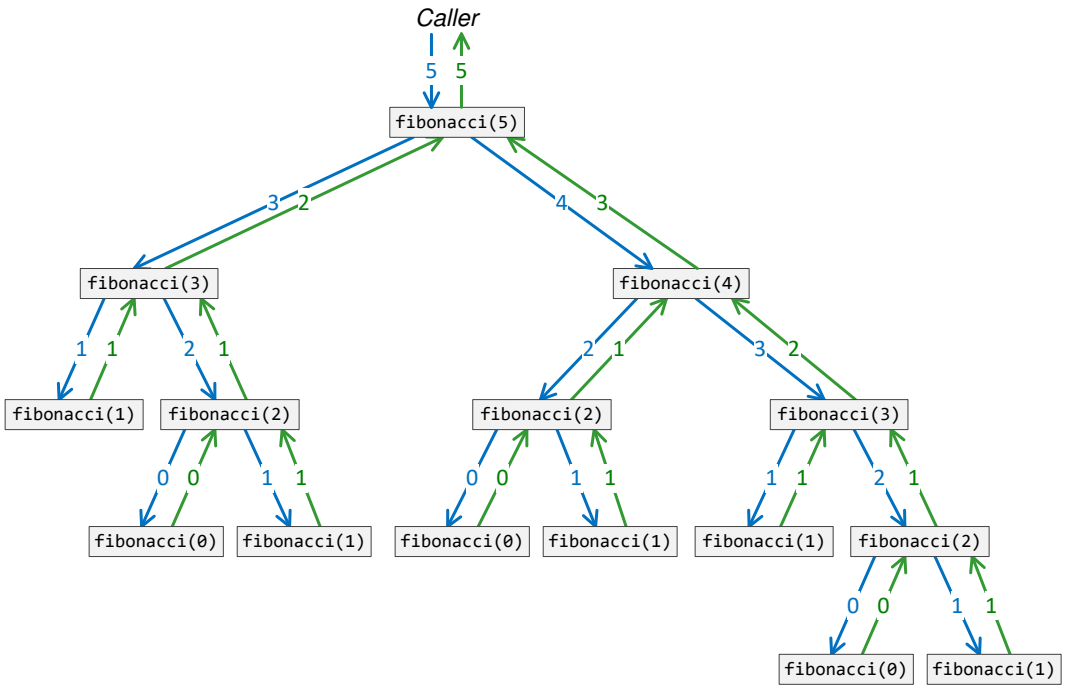
Figure 10.2 illstrates the recursive computation of `fibonacci(5)` using this `fibonacci` function. Note that the call `fibonacci(5)` invokes the `fibonacci` function a total of 15 times.

While this `factorial` function computes the correct result, this tendency to call itself so many times makes it impractical for larger values. Section 21.8 introduces an algorithm design technique that greatly improves the performance of this recursive function.

10.6 Making Functions Reusable

A function definition packages in one place functionality that we can exercise (call) from many different places within a program. Thus far, however, we have not seen how we can reuse easily function definitions in other *programs*. For example, our `is_prime` function in Listing 9.10 (primefunc.cpp) works well within Listing 9.10 (primefunc.cpp), and we could put it to good use in other programs that need to test primality (encryption software, for example, makes heavy use of prime numbers). We could use the copy-and-paste feature of our favorite text editor to copy the `is_prime` function definition from Listing 9.10 (primefunc.cpp) into the new encryption program we are developing.

Figure 10.2 The recursive computation of `fibonacci(5)`. Each rectangle represents an invocation of the `fibonacci` fuunction. The call at the top of the diagram represents the initial call of `fibonacci(5)`. An arrow pointing down indicates the argument being passed into an invocation of `fibonacci`, and an arrow pointing up represents the value returned by that invocation. An invocation of `fibonacci` with no arrow pointing down away from the invocation represents a base case; observe that any invocation receiving a 0 or 1 is a base case. We see that the recursive process for `fibonacci(5)` invokes the function a total of 15 times.



It is possible to reuse a function with this copy-and-paste technique only if the function definition does not use any programmer-defined global variables, programmer-defined global constants, nor other programmer-defined functions. If a function does use any of these programmer-defined external entities, we must copy and paste all of this extra code in order for the function to compile. If the names of the external variables or functions that we must copy into the new program conflict with existing names within the program, the program will not compile as is, and we must rename the offending variables or functions to resolve the conflicts. Changing code provides the opportunity to introduce bugs accidentally, so the process requires absolute care.

We can avoid the issues introduced by copying and pasting if the function we wish to reuse in other programs uses only local variables and parameters. Such a function is truly an independent function that can be reused easily in multiple programs.

The notion of copying source code from one program to another is not ideal, however. It is too easy for the copy to be incomplete or for some other error to be introduced during the copy. Furthermore, such code duplication is wasteful. If 100 programs on a particular system all need to use the `is_prime` function, under this scheme they must all include the `is_prime` code. This redundancy wastes space. Finally, in perhaps the most compelling demonstration of the weakness of this copy-and-paste approach, what if we discover a bug in the `is_prime` function that all 100 programs are built around? When we discover the error and fix it in one program, the other 99 programs still will contain the bug. Their source code must be updated, and they each then must be recompiled. The problem is even more complicated if the other 99 programs have 99 different authors. We then must contact multiple developers to tell them to fix their `is_prime` function. The situation would be similar if we updated a correct `is_prime` function to make it more efficient. The problem is this: all the programs using `is_prime` define their *own* `is_prime` function; while the function definitions are meant to be identical, there is no mechanism tying all these common definitions together. We really would like to reuse the function as is without copying it.

Fortunately, C++ provides a way to develop functions in separate files and combine the code from these independently developed functions into one program. We can compile the source files separately, and the linker can combine the compiled code into an executable. What we need is a way for the compiler to verify that the calling code in one source file is correctly invoking the function defined in another source file.

Listing 10.10 (prime.h) provides the first step in making a reusable `is_prime` function.

Listing 10.10: `prime.h`

```
bool is_prime(int);
```

The simple one-line code in Listing 10.10 (prime.h) can be stored in a file called prime.h. In Visual Studio, you simply add a new item to your project specified as a header file, name it prime.h, and you are ready to type the one line of code into the newly created file. This file contains the prototype for our `is_prime` function (see our earlier discussion of function prototypes in Section 8.1). Caller code that intends to use our `is_prime` function must `#include` this file so that compiler can check to see if the caller is using our `is_prime` function properly. An attempt, for example, to pass two arguments to `is_prime` would result in a compiler error since the prototype specifies a single integer argument.

A second file, which could be named prime.cpp, appears in Listing 10.11 (prime.cpp).

Listing 10.11: `prime.cpp`

```
// prime.cpp
#include <cmath>     // Needed for sqrt
#include "prime.h"   // is_prime prototype

/*
```

```
 *  is_prime(n)
 *     Determines the primality of a given value
 *     n an integer to test for primality
 *     Returns true if n is prime; otherwise, returns false
 */
bool is_prime(int n) {
    bool result = true;  // Provisionally, n is prime
    double r = n, root = sqrt(r);
    // Try all possible factors from 2 to the square
    // root of n
    for (int trial_factor = 2; result && trial_factor <= root; trial_factor++)
        result = (n % trial_factor != 0);
    return result;
}
```

The code in Listing 10.11 (prime.cpp) is placed in prime.cpp, a different file from prime.h. It provides an implementation of the `is_prime` function. Notice the `#include` preprocessor directive in Listing 10.11 (prime.cpp) that references the file prime.h. While not required, this serves as a good check to see if the implementation code in this file is faithful to the prototype specified in prime.h. This prime.cpp file is compiled separately, and the compiler will report an error if the implementation of `is_prime` disagrees with the information in the header file.

Note that the file prime.cpp does not contain a `main` function; `main` will appear in another file. Also observe that we do not need to `#include` the `iostream` header, since the `std::cout` and `std::cin` objects are not used anywhere in this file. The `cmath` header is `#include`d since `is_prime` uses the `sqrt` function.

The final piece of the program is the calling code. Suppose Listing 10.12 (primetester.cpp) is added to the project in a file named primetester.cpp.

Listing 10.12: `primetester.cpp`

```
#include <iostream>
#include "prime.h"

/*
 *  main
 *     Tests for primality each integer from 2
 *     up to a value provided by the user.
 *     If an integer is prime, it prints it;
 *     otherwise, the number is not printed.
 */
int main() {
    int max_value;
    std::cout << "Display primes up to what value? ";
    std::cin >> max_value;
    for (int value = 2; value <= max_value; value++)
        if (is_prime(value)) // See if value is prime
            std::cout << value <<  " "; // Display the prime number
    std::cout << '\n';  // Move cursor down to next line
}
```

Note that the file primetester.cpp uses a function named `is_prime`, but its definition is missing. The definition for `is_prime` is found, of course, in prime.cpp.

Visual Studio will automatically compile and link the `.cpp` files when it builds the project. Each `.cpp` is compiled independently on its own merits.

If you are using the Visual Studio Command Line tool, in order to build the program you would type

```
cl /EHsc /Za /W3 primetester.cpp prime.cpp
```

The executable file's name is determined by the name of the first source file listed, in this case primetester.

If you are using the GCC tools instead of Visual Studio, in order to make the executable program named primetester (or primetester.exe under the Microsoft Windows version of the GCC tools), you would issue the command

```
g++ -o primetester -Wall -O1 -std=c++11 prime.cpp primetester.cpp
```

The GNU C++ compiler will separately compile the two source files producing two machine language object files. The linker will then use those object files to create the executable. When the linker has created the executable, it automatically deletes the two object files.

The `is_prime` function is now more readily available to other programs. If it becomes an often used function in many programs, it can be compiled and placed into a special file called a *library*. In this form it need not be recompiled each time a new program is built that requires it. If our `is_prime` is placed in a dynamic library, its code can be loaded and linked at run time and shared by many executing programs. We do not cover library creation in this text.

In Listing 8.7 (timeit.cpp), Listing 8.8 (measureprimespeed.cpp), and Listing 10.2 (digitaltimer.cpp) we used the `clock` function from the `<ctime>` library to measure the elapsed time of sections of various executing programs. In each of these programs the programmer must be aware of the `clock_t` type and `CLOCKS_PER_SEC` constant, both defined in the `<ctime>` header file. Furthermore, the programmer must use the `clock` function properly and correctly perform some arithmetic and include a messy type cast operation. Armed with our knowledge of global variables (Section 10.1) and separate compilation of multiple source files, we can provide a better programming interface to the lower-level timing functions provided to the C library.

Listing 10.13 (timermodule.h) specifies some convenient timing functions.

Listing 10.13: `timermodule.h`

```
// Header file timermodule.h

// Reset the timer so it reads 0 seconds
void reset_timer();

// Start the timer.  The timer will begin measuring elapsed time.
void start_timer();

// Stop the timer.  The timer will retain the current elapsed
// time, but it will not measure any time while it is stopped.
void stop_timer();

// Return the cummulative time (in seconds) kept by the timer since
// it last was reset
double elapsed_time();
```

Listing 10.14 (timermodule.cpp) implements the functions declared in Listing 10.13 (timermodule.h).

Listing 10.14: timermodule.cpp

```
// File timermodule.h
// Implements the program timer module

#include <ctime>

// Global variable that keeps track of the elapsed time.
double elapsed;

// Global variable that counts the number of clock ticks since
// the most recent start time.
clock_t start_time;

// Global flag that indicates whether or not the
// timer is running.
bool running;

// Reset the timer so it reads 0 seconds
void reset_timer() {
    elapsed = 0.0;
    running = false;  // Ensure timer is not running
}

// Start the timer.  The timer will begin measuring elapsed time.
// Starting the timer if it already is running has no effect
void start_timer() {
    // Starting an already running timer has no effect
    if (!running) {
        running = true;        // Note that the timer is running
        start_time = clock();  // Record start time
    }
}

// Stop the timer.  The timer will retain the current elapsed
// time, but it will not measure any time while it is stopped.
// Stopping the timer if it is not currently running has no effect.
void stop_timer() {
    // Stopping a non-running timer has no effect
    if (running) {
        clock_t stop_time = clock();  // Record stop time
        running = false;              // Stop the clock
        // Add to the elapsed time how long it has been since we last
        // started the timer
        elapsed += static_cast<double>((stop_time - start_time))
                                       / CLOCKS_PER_SEC;
    }
}

// Return the cummulative running time (in seconds)
// kept by the timer since it last was reset
double elapsed_time() {
    if (running) { // Compute time since last reset
        clock_t current_time = clock();  // Record current time
        return elapsed + static_cast<double>((current_time - start_time))
```

```
                                                    / CLOCKS_PER_SEC;
    }
    else   // Timer stopped; elapsed already computed in stop_timer
        return elapsed;
}
```

Observe that the code in Listing 10.14 (timermodule.cpp) allows client code to stop the timer and restart it later without losing any previously accumulated time. The implementation uses three global variables—`elapsed`, `start_time`, and `running`—to maintain the state of the timer. One or more of these global variables is influenced by three functions—`start_timer`, `stop_timer`, `reset_time`. The fourth function returns the value of the `elapsed` variable.

Listing 10.15 (bettertimeit.cpp) simplifies Listing 8.7 (timeit.cpp) with our new timer module.

Listing 10.15: `bettertimeit.cpp`

```
#include <iostream>
#include "timermodule.h"  // Our timer module

int main() {
    char letter;
    std::cout << "Enter a character: ";
    start_timer();    // Start timing
    std::cin >> letter;
    stop_timer();     // Stop timing
    std::cout << elapsed_time() << " seconds" << '\n';
}
```

The code within Listing 10.15 (bettertimeit.cpp) is simpler and cleaner than the code in Listing 8.7 (timeit.cpp). All traces of the `clock_t` type and the messy arithmetic and casting are gone. The timer module provides a simple interface to callers that hides the details on how the timing actually happens.

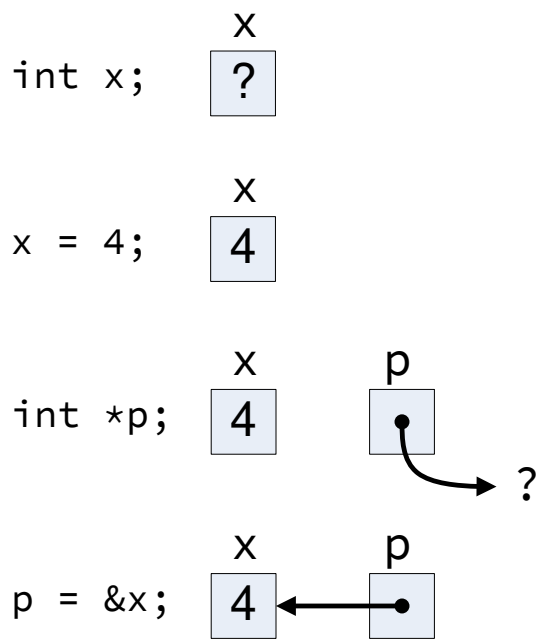
Despite the ease of use of our timer module, it has a servere limitation. Suppose you wish to measure how long it takes for a function to execute and also, during that function's execution, separately time a smaller section of code within that function. When the function is finished executing, you would like to know how long it took the function to do its job and how long a portion of its code took to execute. We essentially need two independent timers, but with our timer module it is not possible to conduct simultaneously more than one timing. We will see a far superior way to model a program execution timer in Section 16.2. We will use objects to enable us to maintain as many simultaneous stopwatches as we need.

10.7 Pointers

Ordinarily we need not be concerned about where variables live in the computer's memory during a program's execution. The compiler generates machine code that takes care of those details for us. Some systems software like operating systems and device drivers need to access specific memory locations in order to interoperate with hardware. Systems programmers, therefore, must be able to write code that can access such lower-level detail. Developers of higher-level applications sometimes need to access the address of variables to achieve specialized effects.

Each byte in a computer's memory is numbered with a unique address. The first address is 0, and the locations are numbered sequentially up to some maximum value allowed by the operating system and hardware. A C++ variable is stored in memory, so each variable lives at a particular address.

Figure 10.3 Pointer declaration and assignment



If x is a variable of any type, the expression

```
&x
```

represents the *address* of x. The & operator is called the *address of* operator. Regardless of the type of x, the expression

```
&x
```

is really just a number—the numeric address of the variable's memory location, but except for some situations in systems programming, programmers rarely need to treat this value as a number.

While an address is really just a nonnegative integer value, C++ uses a special notation when dealing with addresses. In the following declaration

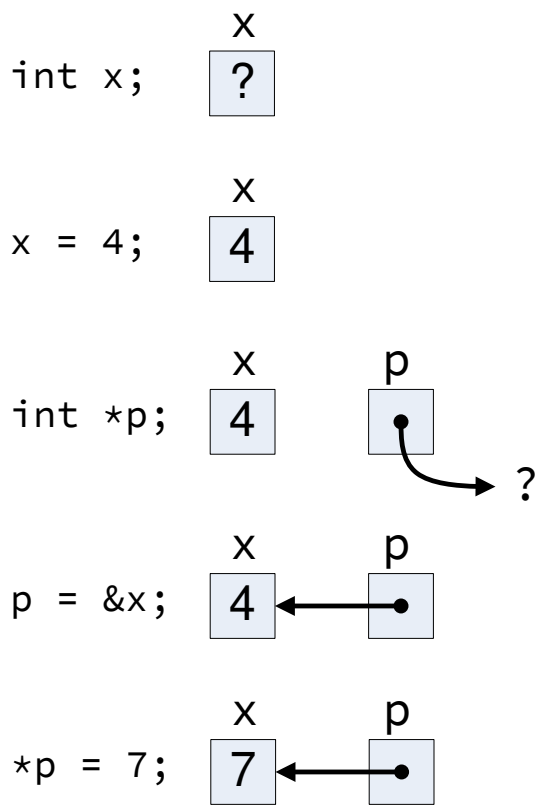
```
int *p;
```

the variable p is not an int itself; the * symbol in the declaration signifies that p is a *pointer* to an int. This means we can assign to p the address of an int. The following sequence of code

```
int x;
x = 4;
int *p;
p = &x;
```

assigns the address of x to p. We can visualize the execution of these four statements in Figure 10.3.

Figure 10.4 Assignment via a pointer



The * symbol used as shown above during a variable declaration indicates that the variable is a pointer. It will be used to refer to another variable or some other place in memory. In this case, the sequence of assignments allows pointer p to refer to variable x.

In order to access memory via a pointer, we use the unary * operator. When not used in the context of a declaration, the unary * operator is called the *pointer dereferencing* operator. Continuing the code sequence above,

```
int x;
x = 4;
int *p;
p = &x;
*p = 7;
```

the statement

```
*p = 7;
```

copies the value 7 into the address referenced by the pointer p. Figure 10.4 illustrates the full sequence.

Notice that the assignment to *p modifies variable x's value. The pointer p provides another way to

access the memory allocated for x.

It is important to note that the statement

```
*p = 5;
```

is the first assignment statement we have seen that uses more than just a single variable name on the left of the assignment operator. The statement is legal because the expression *p represents a memory location that can store a value. Here, *p stands in the place of the variable x, and we easily can assign a value to x.

The unary * operator has two distinct meanings depending on the context:

1. At the pointer's declaration, for example,

   ```
   double *p;
   ```

   the * indicates that p is a pointer to a double; it is not itself a double.

2. After the pointer's declaration, for example,

   ```
   *p = 0.08;
   ```

   the expression *p represents the contents of memory at the address assigned to p.

The * operator is the inverse of the & operator. & finds the address of a variable, and, when this address is assigned to a pointer variable like p, the * operator accesses the memory referenced by the pointer:

```
int v;        // v is a normal integer variable
int *p = &v;  // Pointer variable p points to v
*p = 5;       // Assigns 5 to variable v
```

We may declare a pointer and not assign it, as in

```
int *p;
```

We say p is an *uninitialized pointer*, sometimes called a *wild pointer*. If p is a local variable, its contents are undetermined bits. Because of p's declared type, we interpret these bits as an address, so the net effect is that the uninitialized pointer p points to a random location in the computer's memory. An attempt to dereference p, as in

```
*p = 500;
```

is certainly asking for trouble. This statement attempts to write the value 500 at some unknown-to-the-programmer memory location. Often the address is not part of the area of memory the operating system has set aside for the executing program, so the operating system steps in and issues a run-time error. This is the best possible result for misusing a wild pointer. It is possible, however, that the spurious address is within the executing program's domain. In this case the value 500 may overwrite another variable or the compiled machine language instructions of the program itself! Such errors are difficult to track down because the overwritten value of the variable cannot be detected until the program attempts to use the variable. The statement that misuses the uninitialized pointer may be far away in the source code (even in a different source file) from the code that attempts to use the clobbered variable. When the program fails, the programmer naturally looks around in the code where the failure occurred—the code in the vicinity where

the clobbered variable appears. Unfortunately, the misbehaving code is fine, and the error lies elsewhere. There often is no easy way to locate the true source of the error.

Suppose p is a variable that points in an int. The statement

```
*p = 5;
```

assigns the value 5 to the memory location to which p points. Compare this legal C++ statement to the following illegal C++statement:

```
p = 5;  // Illegal statement as is
```

This second statement attempts to make p refer to the memory address 5; it does not assign the value 5 to the memory to which p refers. C++ is very strict about disallowing the mixing of pointers and non-pointers across assignment.

Systems programmers sometimes need to assign a pointer to a particular address in memory, and C++ permits the assignment with a special kind of type cast, the reinterpret_cast:

```
p = reinterpret_cast<int *>(5);  // Legal
```

The familiar static_cast (see Section 4.2) will not work. Why is C++ so strict when it comes to assignments of pointers to non-pointers and vice versa? It is easy to make a mistake such as omitting the * operator when it is needed, so the special cast forces the programmer to pause and consider whether the mixed assignment truly is necessary or whether attempting to do so would be a mistake.

Modern C++ compiler supports the reserved word nullptr to represent a pointer to "nothing." It stands for "null pointer." The following statement

```
p = nullptr;
```

indicates that p is pointing nowhere. On most platforms, nullptr maps to address zero, which is out of bounds for any running program. Dereferencing p thus would result in a run-time error. Adept programmers can find the source of such a null pointer access problem quickly with a debugger.

C++ does not allow direct integer assignment to a pointer, as in

```
int *p = 5;
```

Here p must *point* to an integer; it is not an integer. If you wish to assign p to point to a particular memory address, you must use a special type cast:

```
int *p = reinterpret_cast<int *>(5);
```

C++ does allow the literal value 0 to be used in place of nullptr. The statement

```
p = 0;
```

achieves the same result as assigning nullptr to p. This is how C++ programmers assigned a pointer to point to nothing before the nullptr keyword was available. Since newer compilers support existing C++ source code, the literal zero assignment still works. You should use the nullptr keyword because it improves the source code readability. Since 0 can represent both an integer value and a pointer to any type, both of the following statements are legal if p has been declared to be a pointer to an integer:

```
p = 0;
*p = 0;
```

The first statement assigns null to p, while the second statement sets the data to which p points to zero. Said another way, the first statement changes where p points; the second statement changes the memory to which p p points. Superficially, the two statements look very similar and are easy to confuse. Next, consider the statements

```
p = nullptr;    // OK
*p = nullptr;   // Error, will not compile
```

The second statement contains an error because it is illegal to assign the nullptr literal to anything other than a pointer type.

The nullptr reserved word is part of the C++11 standard. The name nullptr is simply an identifier (for example, a variable or function name) for older compilers. Before the nullptr constant became available the literal 0 (zero) was considered the null pointer reference. For backwards compatibility with older code, C++11 allows you to use 0 in place of nullptr, but if possible you should avoid this practice when writing new code. The nullptr literal allows the complier to perform better type checking. To see why, suppose the programmer believes the variable p is a pointer to an integer, but p is instead a simple integer:

```
// Programmer believes p is a pointer to an integer
p = 0;
```

In this case the compiler is powerless to detect a problem because p is an integer, and zero is a valid integer value. If p really is an integer rather a pointer, the compiler will flag the following code:

```
// Programmer believes p is a pointer to an integer
p = nullptr;
```

because the nullptr may not be assigned to a non-pointer type.

The nullptr constant and the notion of a pointer pointing nowhere becomes more useful when building dynamic data structures (see, for example, Section 18.3).

10.8 Reference Variables

C++ supports another kind of variable that is many ways similar to a pointer. When the & symbol is used as part of the type name during a variable declaration, as in

```
int x;
int& r = x;
```

we say r is a *reference variable*. This declaration creates a variable r that refers to the same memory location as the variable x. We say that r *aliases* x. Unlike a pointer variable, we may treat r as if it were an int variable—no dereferencing with * is necessary. Listing 10.16 (referencevar.cpp) demonstrates how reference variables can alias other variables.

Listing 10.16: referencevar.cpp

```
#include <iostream>

int main() {
    int x = 5;
    int y = x;
    int& r = x;
    std::cout << "x = " << x << '\n';
    std::cout << "y = " << y << '\n';
    std::cout << "r = " << r << '\n';
    std::cout << "Assign 7 to x\n";
    x = 7;
    std::cout << "x = " << x << '\n';
    std::cout << "y = " << y << '\n';
    std::cout << "r = " << r << '\n';
    std::cout << "Assign 8 to y\n";
    y = 8;
    std::cout << "x = " << x << '\n';
    std::cout << "y = " << y << '\n';
    std::cout << "r = " << r << '\n';
    std::cout << "Assign 2 to r\n";
    r = 2;
    std::cout << "x = " << x << '\n';
    std::cout << "y = " << y << '\n';
    std::cout << "r = " << r << '\n';
}
```

The output Listing 10.16 (referencevar.cpp):

```
x = 5
y = 5
r = 5
Assign 7 to x
x = 7
y = 5
r = 7
Assign 8 to y
x = 7
y = 8
r = 7
Assign 2 to r
x = 2
y = 8
r = 2
```

clearly demonstrates that the variables `x` and `r` represent the same quantity. Reassigning `x` changes `r` in exactly the same way, and reassigning `r` changes `x` in exactly the same way. The variable `y`, on the other hand, is independent from both `x` and `r`. Reassigning either `x` or `r` does not affect `y`, and reassigning `y` affects neither `x` nor `r`.

The space around the & symbol used to declare a reference variable is not significant; specifically, the statement

```
int& r = x;
```

is equivalent to

```
int &r = x;
```

Reference variables are similar to pointer variables, but there are some important differences. Reference syntax is simpler than pointer syntax because it is not necessary to dereference a reference variable in order to assign the memory location to which it refers. If `num` is an `int`, `ptr` is a pointer to an `int`, and `ref` is a reference to an `int`, consider the following statements:

```
num = ref;   // Assign num from ref, no need to deference
num = *ptr;  // Assign num from ptr, must dereference with *
```

A reference variable has two big limitations over a pointer variable:

- A reference variable must be initialized with an actual variable when it is declared. A pointer variable may be declared without an initial value and assigned later. Consider the following statements:

  ```
  int *p;   // Legal, we will assign p later
  int& r;   // Illegal, we must initialize r when declaring it
  ```

  Attempting to compile this code under Visual C++ prompts the compiler to issue the error

  error C2530: 'r' : references must be initialized

- There is no way to bind a reference variable to a different variable during its lifetime. Consider the following code fragemnt:

  ```
  int x = 5, y = 7;
  int *p = &x;  // Binds p to point to x
  int& r = x;   // Binds r to x
  p = &y;       // Reassign p to point to y
  r = y;        // Assign y's value to x (via r)
  ```

  The statement

  ```
  r = y;
  ```

  does not bind `r` to the `y` variable. The declaration of `r` binds `r` to the `x` variable for the life of `r`. This statement simply assigns `y`'s value to `x` via the reference `r`. In contrast, we may freely bind pointer variables to any variables we choose at any time.

A reference variable, therefore, in the examples provided here works like a pointer that must be bound to a variable and may not be redirected to point anywhere else. Also, unlike with pointers, it is illegal to attempt to assign `nullptr` to a reference variable. There are some other differences between references and pointers that we will not explore here. Reference variables provide a simpler syntax than pointer variables since we do not use the pointer dereferencing operator (`*`) when working with references.

There is one other major difference between pointers and references—C++ adopted pointers as is from the C programming language, but C does not provide references. C++ programmers often use library functions written in C, so it is important to not mix references with C code.

Section 10.9 reveals a very important practical application of pointers and references in their role of enabling pass by reference to functions.

10.9 Pass by Reference

The default technique for passing parameters to functions is pass by value (see Section 9.3). C++ also supports *pass by reference*, also known as *call by reference*, which allows functions to alter the values of formal parameters passed by callers.

Consider Listing 10.17 (faultyswap.cpp), which uses a function that attempts to interchange the values of its two integer parameters.

Listing 10.17: faultyswap.cpp

```
#include <iostream>

/*
 *  swap(a, b)
 *     Attempts to interchange the values of
 *     its parameters a and b.  That it does, but
 *     unfortunately it only affects the local
 *     copies.
 */
void swap(int a, int b) {
   int temp = a;
   a = b;
   b = temp;
}

/*
 *  main
 *     Attempts to interchange the values of
 *     two variables using a faulty swap function.
 */
int main() {
    int var1 = 5, var2 = 19;
    std::cout << "var1 = " << var1 << ", var2 = " << var2 << '\n';
    swap(var1, var2);
    std::cout << "var1 = " << var1 << ", var2 = " << var2 << '\n';
}
```

The output of Listing 10.17 (faultyswap.cpp) is

```
var1 = 5, var2 = 19
var1 = 5, var2 = 19
```

Unfortunately, the `swap` function simply interchanges copies of the actual parameters, not the actual parameters themselves. We really would like to write a function that interchanges the caller's variables.

Pass by reference is necessary to achieve the desired effect. C++ can do pass by reference in two ways: pointer parameters and reference parameters.

10.9.1 Pass by Reference via Pointers

Pointers (see Section 10.7) allow us to access the memory locations of variables. We can use this capability to allow a function to modify the values of variables that are owned by its caller. Listing 10.18 (swapwithpointers.cpp) provides a correct version of our variable interchange program.

Listing 10.18: swapwithpointers.cpp

```
#include <iostream>

/*
 *  swap(a, b)
 *     Interchanges the values of memory
 *     referenced by its parameters a and b.
 *     It effectively interchanges the values
 *     of variables in the caller's context.
 */
void swap(int *a, int *b) {
   int temp = *a;
   *a = *b;
   *b = temp;
}

/*
 *  main
 *     Interchanges the values of two variables
 *     using the swap function.
 */
int main() {
    int var1 = 5, var2 = 19;
    std::cout << "var1 = " << var1 << ", var2 = " << var2 << '\n';
    swap(&var1, &var2);
    std::cout << "var1 = " << var1 << ", var2 = " << var2 << '\n';
}
```

The output of Listing 10.18 (swapwithpointers.cpp) is

```
var1 = 5, var2 = 19
var1 = 19, var2 = 5
```

which is the result we were trying to achieve. The `swap` function can manipulate `main`'s variables directly since we passed it pointers to those variables.

In Listing 10.18 (swapwithpointers.cpp):

- The formal parameters to `swap`, `a` and `b`, are pointers to integers; they are not integers themselves. In order to access the integer to which the pointer named `a` refers, it must be dereferenced. That is why any use of `a` in `swap`'s body is prefixed with the pointer dereferencing operator, `*`. The statement

  ```
  int temp = *a;
  ```

  assigns to the local variable `temp` the value of the variable to which `a` points. Since `main` passes the address of `var1` as the first parameter in its call to `swap`, in this case `a` points to `var1`, so `*a` is effectively another way to access the memory location of `var1` in `main`. The function thus assigns the value of `main`'s `var1` variable to `temp`.

- In `swap`'s statement

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

  since `*a` is effectively `main`'s `var1` variable and `*b` is effectively `main`'s `var2` variable, this statement assigns the value of `main`'s `var2` to its `var1` variable.

- The `swap` function's

  ```
  *b = temp;
  ```

  statement assigns the local `temp` value to `main`'s `var2` variable, since `swap`'s `b` parameter points to `main`'s `var2`.

- Within `main`, the call

  ```
  swap(&var1, &var2);
  ```

  passes the addresses of its local variables to `swap`.

In reality, pass by reference with pointers is still using pass by value. Instead of passing copies of values, we are passing copies of addresses. In Listing 10.18 (swapwithpointers.cpp), for example, the values of the addresses of `var1` and `var2` are copied to the formal parameters `a` and `b`. The difference is we are not attempting to reassign `a` or `b`; we are reassigning memory to which `a` and `b` point. Whether we use the original address or a copy of the address, it is still the same address—the same numeric location in memory.

10.9.2 Pass by Reference via References

Both C and C++ support pass by reference with pointers (see Section 10.9.1). Since C++ programs often use C libraries, C++ programmers must be familiar with the pointer technique for pass by reference. C++, however, provides a simpler way of implementing pass by reference using *reference parameters*. (See Section 10.8 for an introduction to reference variables.) Listing 10.19 (swapwithreferences.cpp) uses reference parameters in our variable interchange program.

Listing 10.19: `swapwithreferences.cpp`

```
#include <iostream>

/*
 *  swap(a, b)
 *     Interchanges the values of memory
 *     referenced by its parameters a and b.
 *     It effectively interchanges the values
 *     of variables in the caller's context.
 */
void swap(int& a, int& b) {
   int temp = a;
   a = b;
   b = temp;
}

/*
 *  main
 *     Interchanges the values of two variables
 *     using the swap function.
 */
int main() {
    int var1 = 5, var2 = 19;
    std::cout << "var1 = " << var1 << ", var2 = " << var2 << '\n';
    swap(var1, var2);
    std::cout << "var1 = " << var1 << ", var2 = " << var2 << '\n';
}
```

The syntax of Listing 10.19 (swapwithreferences.cpp) is a bit cleaner than that of Listing 10.18 (swapwithpointers.cpp). In Listing 10.19 (swapwithreferences.cpp):

- The formal parameters to `swap`, `a` and `b`, are *references* to integers; this is signified by the & symbol following `int` in their declarations. Because `a` is a reference we use it exactly like an integer; there is no need to dereference it with the `*` operator to change the value of the integer it aliases. Because `a` is a reference, however, it is an alias to another variable or memory location elsewhere. This means if we modify `a`, we also modify the variable it references, in this case `var1` in `main`. The statement

  ```
  int temp = a;
  ```

  assigns to the local variable `temp` the value of `a`, but since `a` is another way to get to `var1`, this statement ultimately assigns `var1`'s value to `temp`.

- Within `main`, the call

  ```
  swap(var1, var2);
  ```

  does not require any special decoration. It looks like a normal pass by value function invocation. A programmer must be aware that `swap` uses reference parameters and that any function that uses reference parameters can change the actual values passed to it.

Reference parameters were introduced into C++ so that some of the more advanced object-oriented features could be implemented more cleanly. Some argue that for simpler situations like the `swap` function, pointer pass by reference is more desirable than reference parameter pass by reference because with pointer pass by reference, the caller is forced to pass addresses of the actual parameters that may be modified. In this way there can be no doubt that pass by reference is going on. With reference parameters, pass by value and pass by reference cannot be distinguished at the call site, since a call with reference parameters looks exactly like a pass by value invocation.

In general, pure pass by value functions are preferred to pass by reference functions. Functions using pass by reference cause *side effects*. This means they can change the state of the program in ways that can be determined only by looking inside of the function and seeing how it works. Functions that access global variables (see Section 10.1) can also cause side effects. Program development is much easier when functions can be treated as black boxes that perform computations in isolation without the possibility of affecting anything outside of their local context. The result of a function's work can be assigned to a variable thus changing the state of the program, but that change is the responsibility of the caller, not the responsibility of the function itself. With pass by value, the function's parameters and local variables come into existence when the function executes, the parameters and local variables disappear when the function is finished, and nothing else is affected by the function's execution.

Side-effect-free functions can be developed and tested in isolation from the rest of the program. Once programmers are satisfied with their correctness, they need not be touched again as the remainder of the system is developed. Functions with side effects, however, have dependencies to other parts of the program, and changes elsewhere in the system may require programmers to modify and re-evaluate existing functions.

10.10 Higher-order Functions

The functions we have seen so far have accepted only data as parameters. Since functions can contain conditional statements and loops, they can do different things based on the data they receive. The code within each function, however, is fixed at compile time. Consider the following function:

```
int evaluate(int x, int y) {
    return x + y;
}
```

The call

```
evaluate(10, 2)
```

evaluates to 12. Since the `evaluate` function returns the sum of its two parameters. The call

```
evaluate(-4, 6);
```

returns 2. The function returns a different result this time because we passed different parameters to it. The `evaluate` function in a sense behaves differently depending on the arguments passed by its caller; however, it always *adds* its two parameters. There is no way we can call `evaluate` and expect it to multiply its two parameters instead. The `evaluate` function is hard-coded to perform addition.

What if we wanted the `evaluate` function to be able to perform different arithmetic operations at different times during a program's execution? Unfortunately, `evaluate` as it currently is written cannot adapt to perform a different arithmetic operation. The good news is that we can rewrite `evaluate` so that it can flexibly adapt to a caller's changing arithmetic needs.

C++ allows programmers to pass functions as parameters to other functions. A function even may return a function as a result. A function that accepts one or more functions as parameters or returns a function as a result is known as a *higher-order function*. As we will see, higher-order functions open up new programming possibilities enabling us to customize the behavior of a function by plugging into it different functions to achieve different effects.

C++ achieves higher-order functions via *function pointers*. During a program's execution, the compiled machine language code for a function must reside in the computer's memory for the program to be able to invoke the function. That means every function has a memory address just as each variable has its own specific memory address. A pointer to a function holds the starting address for the compiled code of a particular function.

Listing 10.20 (arithmeticeval.cpp) provides an example of a higher-order function in action.

Listing 10.20: `arithmeticeval.cpp`

```
#include <iostream>

int add(int x, int y) {
    return x + y;
}

int multiply(int x, int y) {
    return x * y;
}

int evaluate(int (*f)(int, int), int x, int y) {
    return f(x, y);
}

int main() {
    std::cout << add(2, 3) << '\n';
    std::cout << multiply(2, 3) << '\n';
    std::cout << evaluate(&add, 2, 3) << '\n';
```

```
    std::cout << evaluate(&multiply, 2, 3) << '\n';
}
```

The first parameter of the `evaluate` function,

```
int (*f)(int, int)
```

represents a function pointer. The parameter's name is `f`, and `f` is a pointer to a function that accepts two integer parameters and returns an integer result. In Listing 10.20 (arithmeticeval.cpp), the first parameter a caller must pass to `evaluate` is the address of a function with a prototype that matches the formal parameter specified in `evaluate`'s definition. Both the `add` and `multiply` functions qualify because they accept two integer parameters and return an integer result. The expression

```
evaluate(&add, 2, 3)
```

passes the address of the `add` function to `evaluate`. In the body of the `evaluate` function we see that `evaluate` invokes the function specified by its first parameter (`f`), passing to this function its second (`x`) and third (`y`) parameters as the two parameters that function expects. The net effect, therefore, of the call

```
evaluate(&add, 2, 3)
```

is the same as

```
add(2, 3)
```

C++ has a somewhat relaxed syntax for function pointers that cannot be applied to pointers to data. When invoking `evaluate` in source code we may omit the ampersand in front of the function argument, as is

```
std::cout << evaluate(add, 2, 3) << '\n';
```

Since the compiler knows that `add` is a function and since parentheses do not follow the word `add`, the compiler deduces that this is not a call to `add` but rather the address of the `add` function.

When the name of a previously declared function appears by itself within C++ source code it represents a pointer to that function.

Function pointers are not restricted to function parameters. Given the definition of `add` above, the following code fragment is legal C++:

```
// Declare func to be a pointer to a function that accepts
// two integer parameters and returns an integer result
int (*func)(int, int);

// Assign add to func
func = add;

// Call add through func
std::cout << func(7, 2) << '\n';
```

This code fragment as part of a complete C++ program would print 9.

Higher-order functions via function pointers provide a powerful tool for developing flexible programs. With functions as parameters we can dynamically customize the behavior of a function, essentially "plugging in" new functionality by passing in different functions. We will put higher-order functions to good use in Chapter 12.

10.11 Exercises

1. Consider the following C++ code:

```
#include <iostream>

int sum1(int n) {
    int s = 0;
    while (n > 0) {
        s++;
        n--;
    }
    return s;
}

int input;

int sum2() {
    int s = 0;
    while (input > 0) {
        s++;
        input--;
    }
    return s;
}

int sum3() {
    int s = 0;
    for (int i = input; i > 0; i--)
        s++;
    return s;
}

int main() {
    // See each question below for details
}
```

   (a) What is printed if `main` is written as follows?

```
int main() {
    input = 5;
    std::cout << sum1(input) << '\n';
    std::cout << sum2() << '\n';
```

```
    std::cout << sum3() << '\n';
}
```

(b) What is printed if `main` is written as follows?

```
int main() {
    input = 5;
    std::cout << sum1(input) << '\n';
    std::cout << sum3() << '\n';
    std::cout << sum2() << '\n';
}
```

(c) What is printed if `main` is written as follows?

```
int main() {
    input = 5;
    std::cout << sum2() << '\n';
    std::cout << sum1(input) << '\n';
    std::cout << sum3() << '\n';
}
```

(d) Which of the functions `sum1`, `sum2`, and `sum3` produce a side effect? What is the side effect?

(e) Which function may not use the `input` variable?

(f) What is the scope of the variable `input`? What is its lifetime?

(g) What is the scope of the variable `i`? What is its lifetime?

(h) Which of the functions `sum1`, `sum2`, and `sum3` manifest good functional independence? Why?

2. Consider the following C++ code:

```
#include <iostream>

int next_int1() {
    static int cnt = 0;
    cnt++;
    return cnt;
}

int next_int2() {
    int cnt = 0;
    cnt++;
    return cnt;
}

int global_count = 0;

int next_int3() {
    global_count++;
    return global_count;
}

int main() {
    for (int i = 0; i < 5; i++)
```

```
        std::cout << next_int1() << " "
                  << next_int2() << " "
                  << next_int3() << '\n';
}
```

(a) What does the program print?

(b) Which of the functions `next_int1`, `next_int2`, and `next_int3` is the best function for the intended purpose? Why?

(c) What is a better name for the function named `next_int2`?

(d) The `next_int3` function works in this context, but why is it not a good implementation of a function that always returns the next largest integer?

3. The following C++ program is split up over three source files. The first file, counter.h, consists of

```
int read();
int increment();
int decrement();
```

The second file, counter.cpp, contains

```
static int count;

int read() {
    return count;
}

int increment() {
    if (count < 5)
        count++;
}

int decrement() {
    if (count > 0)
        count--;
}
```

The third file, main.cpp, is incomplete:

```
#include <iostream>
#include "counter.h"

int main() {
    // Add code here

}
```

(a) Add statements to `main` that enable it to produce the following output:

```
3
2
4
```

The restriction is that the only ouput statement you are allowed to use (three times) is

```
std::cout << read() << '\n';
```

(b) Under the restriction of using the same output statement above, what code could you add to `main` so that it would produce the following output?

```
6
```

4. Consider the following C++ code:

```
#include <iostream>

int max(int n) {
    return n;
}

int max(int m, int n) {
    return (m >= n)? m : n;
}

int max(int m, int n, int r) {
    int x = m;
    if (n > x)
        x = n;
    if (r > x)
        x = r;
    return x;
}

int main() {
    std::cout << max(4) << '\n';
    std::cout << max(4, 5) << '\n';
    std::cout << max(5, 4) << '\n';
    std::cout << max(1, 2, 3) << '\n';
    std::cout << max(2, 1, 3) << '\n';
    std::cout << max(2, 1, 2) << '\n';
}
```

(a) Is the program legal since there are three different functions named `max`?

(b) What does the program print?

5. Consider the following function:

```
int proc(int n) {
    if (n < 1)
        return 1;
    else
        return proc(n/2) + proc(n - 1);
}
```

Evaluate each of the following expressions:

(a) `proc(0)`
(b) `proc(1)`
(c) `proc(2)`
(d) `proc(3)`
(e) `proc(5)`
(f) `proc(10)`
(g) `proc(-10)`

6. Rewrite the `gcd` function so that it implements Euclid's method but uses iteration instead of recursion.

7. If `x` is a variable, how would you determine its address in the computer's memory?

8. What is printed by the following code fragment?

```
int x = 5, y = 3, *p = &x, *q = &y;
std::cout << "x = " << x << ", y = " << y << '\n';
x = y;
std::cout << "x = " << x << ", y = " << y << '\n';
x = 7;
std::cout << "x = " << x << ", y = " << y << '\n';
*p = 10;
std::cout << "x = " << x << ", y = " << y << '\n';
p = q;
*p = 20;
std::cout << "x = " << x << ", y = " << y << '\n';
```

9. Given the declarations:

```
int x, y, *p, *q;
```

indicate what each of the following code fragments will print.

(a)
```
p = &x;
x = 5;
std::cout << *p << '\n';
```

(b)
```
x = 5;
p = &x;
std::cout << *p << '\n';
```

(c)
```
p = &x;
*p = 8;
std::cout << *p << '\n';
```

(d)
```
p = &x;
q = &y;
x = 100;
y = 200;
*q = *p;
std::cout << x << ' ' << y << '\n';
std::cout << *p << ' ' << *q  << '\n';
```

(e)
```
p = &x;
q = &y;
x = 100;
y = 200;
q = p;
std::cout << x << ' ' << y << '\n';
std::cout << *p << ' ' << *q  << '\n';
```

(f)
```
x = 5;
y = 10;
p = q = &y;
std::cout << *p << ' ' << *q << '\n';
*p = 100;
*q = 1;
std::cout << x << ' ' << y << '\n';
```

(g)
```
x = 5;
y = 10;
p = q = &x;
*p = *q = y;
std::cout << x << ' ' << y << '\n';
```

10. The following function does not behave as expected:

```
/*
 *  (Faulty function)
 *
 *  get_range
 *     Establishes a range of integers.  The lower value must
 *     be greater than or equal to min, and the upper value
 *     must be less than or equal to max.
 *     min is the lowest acceptable lower value.
 *     max is the highest acceptable upper value.
 *     lower is assigned the lower limit of the range
 *     upper is assigned the upper limit of the range
 */
void get_range(int min, int max, int lower, int upper) {
    std::cout << "Please enter a data range within the bounds "
              << min << "..." << max << ": ";
    do { // Loop until acceptable values are provided
        std::cin >> lower >> upper;
        if (lower < min)
            std::cout << lower << " is too low, please try again.\n";
        if (upper > max)
            std::cout << upper << " is too high, please try again.\n";
    }
    while (lower < min || upper > max);
}
```

(a) Modify the function so that it works using pass by reference with pointers.

(b) Modify the function so that it works using pass by reference with reference parameters.

11. Classify the following functions as pure or impure. `x` is a global variable and LIMIT is a global constant.

(a)
```
int f1(int m, int n) {
    return 2*m + 3*n;
}
```

(b)
```
int f2(int n) {
    return n - LIMIT;
}
```

(c)
```
int f3(int n) {
    return n - x;
}
```

(d)
```
void f4(int n) {
    std::cout << 2*n << '\n';
}
```

(e)
```
int f5(int n) {
    int m;
    std::cin >> m;
    return m * n;
}
```

(f)
```
int f6(int n) {
    int m = 2*n, p;
    p = 2*m - 5;
    return p - n;
}
```

12. Complete the following function that assigns to its `mx` and `my` reference parameters the components of the midpoint of the points (x_1, y_1) and (x_2, y_2), represented by the parameters `x1`, `y1`, `x2`, and `y2`.

```
// Computes the midpoint of the points (x1, y1) and (x2, y2).
// The point (mx, my) represents the midoint.
void midpoint(double x1, double y1, double x2, double y2,
              double& mx, double& my) {
    // Add your code . . .
}
```

13. Complete the following function that assigns to its `ix` and `iy` reference parameters the components of the point of intersection of two lines. The first line passes through the points (x_1, y_1) and (x_2, y_2); the second line passes through the points (x_3, y_3) and (x_4, y_4). If the two lines do not intersect in a single point (that is, they are parallel to each other), the function should assign `INFINITY` to both `ix` and `iy`. `INFINITY` is a double-precision floating-point constant defined in the cmath header file. It represents a very large number that you effectively can treat as infinity.

```
// Computes the point of intersection of two lines.
// The first line passes through the points
// (x1, y1) and (x2, y2).  The second line passes through the
// points (x3, y3) and (x4, y4).  The function assigns
// (ix, iy) as the intersection point.
// If the two lines do not intersect in a singal point, the function
// computes (INFINITY, INFINITY).
void intersection(double x1, double y1, double x2, double y2,
                  double x3, double y3, double x4, double y4,
                  double& ix, double& iy) {
    // Add your code . . .
}
```

14. Rewrite the recursive `gcd` function found in Listing 10.8 (gcd.cpp) so that it uses the same basic algorithm of Euclid but uses iteration instead of recursion.

Chapter 11

Sequences

The variables we have used to this point can assume only one value at a time. As we have seen, we can use individual variables to create some interesting and useful programs; however, variables that can represent only one value at a time do have their limitations. Consider Listing 11.1 (averagenumbers.cpp) which averages five numbers entered by the user.

Listing 11.1: averagenumbers.cpp

```
#include <iostream>

int main() {
    double n1, n2, n3, n4, n5;
    std::cout << "Please enter five numbers: ";
    // Allow the user to enter in the five values.
    std::cin >> n1 >> n2 >> n3 >> n4 >> n5;
    std::cout << "The average of " << n1 << ", " << n2 << ", "
              << n3 << ", " << n4 << ", " << n5 << " is "
              << (n1 + n2 + n3 + n4 + n5)/5 << '\n';
}
```

A sample run of Listing 11.1 (averagenumbers.cpp) looks like:

```
Please enter five numbers: 9 3.5 0.2 100 15.3
The average of 9.0, 3.5, 0.2, 100.0, 15.3 is 25.6
```

The program conveniently displays the values the user entered and then computes and displays their average.

Suppose the number of values to average must increase from five to 25. If we use Listing 11.1 (averagenumbers.cpp) as a guide, we must introduce twenty additional variables, and the overall length of the program will necessarily grow. Averaging 1,000 numbers using this approach is impractical.

Listing 11.2 (averagenumbers2.cpp) provides an alternative approach for averaging numbers.

Listing 11.2: averagenumbers2.cpp

```
#include <iostream>

int main() {
```

```
    double sum = 0.0, num;
    const int NUMBER_OF_ENTRIES = 5;
    std::cout << "Please enter " << NUMBER_OF_ENTRIES << " numbers: ";
    for (int i = 0; i < NUMBER_OF_ENTRIES; i++) {
        std::cin >> num;
        sum += num;
    }
    std::cout << "The average of " << NUMBER_OF_ENTRIES << " values is "
              << sum/NUMBER_OF_ENTRIES << '\n';
}
```

Listing 11.2 (averagenumbers2.cpp) behaves slightly differently from Listing 11.1 (averagenumbers.cpp), as the following sample run using the same data shows:

```
Please enter 5 numbers: 9 3.5 0.2 100 15.3
The average of the 5 values is 25.6
```

Listing 11.2 (averagenumbers2.cpp) can be modified to average 25 values much more easily than Listing 11.1 (averagenumbers.cpp) that must use 25 separate variables—just change the constant `NUMBER_OF_ENTRIES`. In fact, the coding change to average 1,000 numbers is no more difficult. However, unlike the original average program, this new version does not display the numbers entered. This is a significant difference; it may be necessary to retain all the values entered for various reasons:

- All the values can be redisplayed after entry so the user can visually verify their correct entry.
- The programmer may want to display the values in some more persistent way; for example, the user may instead type the values in a graphical user interface component, like a visual grid (spreadsheet).
- A more sophisticated program may need to process the values in a different way; for example, we may wish to display just the values entered above a certain value (like greater than zero), but the limit is not determined until after the user finishes entering all the numbers.

In all of these situations we must retain the values of all the variables for future recall.

We need to combine the advantages of both of the above programs; specifically we want to be able to

- retain every individual value, and
- avoid defining separate variables to store all the individual values

These may seem like contradictory requirements, but C++ provides several standard data structures that simultaneously provide both of these advantages. In this chapter we will examine the common sequence types available in C++: *vectors* and *arrays*.

Vectors and arrays are *sequence* types because a sequence implies its elements are ordered. A nonempty sequence has the following properties:

- Every nonempty sequence has a unique *first* element.
- Every nonempty sequence has a unique *last* element.
- Every element in a nonempty sequence except for the first element has a unique *predecessor* element.
- Every element in a nonempty sequence except for the last element has a unique *successor* element.

We call this a *linear ordering*. In a linear ordering you can begin at the first element and repeatedly visit successor elements until you reach the last element. There never is any ambiguity about which element comes next in a sequence.

The data structures we examine in this chapter, `std::vector`s, primitive arrays, and `std::array`s, are all sequence types.

11.1 Vectors

A vector in C++ is an object that manages a block of memory that can hold multiple values simultaneously; a vector, therefore, represents a collection of values. A vector has a name, and we may access the values it contains via their position within the block of memory managed by the vector. A vector stores a sequence of values, and the values must all be of the same type. A collection of values all of the same type is said to be *homogeneous*.

11.1.1 Declaring and Using Vectors

In order to use a vector object within a C++ program, you must add the preprocessor directive

```
#include <vector>
```

The `vector` type is part of the standard (`std`) namespace, so its full name is `std::vector`, just like the full name of `cout` is `std::cout` (see Section 2.1). If you include the directive

```
using std::vector;
```

in your source file, you can use the shorter name `vector` within your code.

We may declare a vector object that can hold integers as simply as

```
std::vector<int> vec_a;
```

The type within the angle brackets may be any valid C++ data type. When declared this way, the vector `vec_a` initially is empty.

We can declare a vector with a particular initial size as follows:

```
std::vector<int> vec_b(10);
```

Here `vec_b` initially holds 10 integers. All 10 elements are zero by default. Note that the vector's size appears within parentheses following the vector's name.

We may declare a vector object of a given size and specify the initial value of all of its elements:

```
std::vector<int> vec_c(10, 8);
```

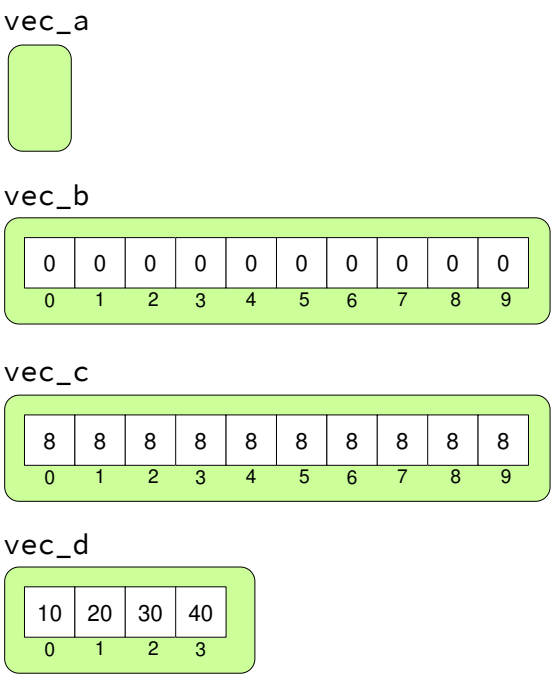
In this example `vec_c` initially holds 10 integers, all having the value 8. Note that the first number within the parentheses following the vector's name indicates the number of elements, and the second argument specifies the initial value of all the elements.

We may declare a vector and specify each and every element separately:

```
std::vector<int> vec_d{10, 20, 30, 40, 50};
```

Figure 11.1 Four different vector declaration statements and conceptual illustrations of the resulting vectors

```
vector<int> vec_a;
vector<int> vec_b(10);
vector<int> vec_c(10, 8);
vector<int> vec_d{ 10, 20, 30, 40 };
```



Note that the elements appear within curly braces, not parentheses. The list of elements within the curly braces constitutes a *vector initializer list*. This kind of declaration is practical only for relatively small vectors. Figure 11.1 provides a conceptual illustration of the vectors `vec_a`, `vec_b`, `vec_c`, and `vec_d`.

Like any other variable, a vector can be local or global, and it must be declared before it is used.

The vector initializer list syntax:

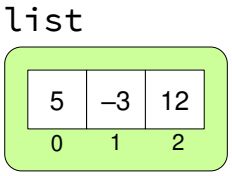
```
std::vector<int> vec_d{10, 20, 30, 40, 50};
```

is a C++11 language feature and, therefore, is not be supported by older C++ compilers; in particular, the C++ compilers available for Visual Studio editions prior to Visual Studio 2013 do not support this initializer list syntax.

Once we have a non-empty vector object, we can access its elements using the vector's name, the square bracket operator, and a integer index:

Figure 11.2 The numbers below the boxes represent indices, or positions, within the vector. Note that the first element in a vector is found at index 0, not 1.

```
vector<int> list(3);
list[0] = 5;
list[1] = -3;
list[2] = 12;
```




```
std::vector<int> list(3);  // Declare list to be a vector of three ints
list[0] = 5 ;              // Make the first element 5
list[1] = -3 ;             // Make the second element -3
list[2] = 12 ;             // Make the last element 12
std::cout << list[1] << '\n'; // Print the element at index 1
```

This code fragment shows how the square brackets allow us to access an individual element based on that element's position within the vector. The number within the square brackets indicates the distance from the beginning of the vector. The expression `list[0]` therefore indicates the element at the very beginning (a distance of zero from the beginning), and `list[1]` is the second element (a distance of one away from the beginning). After executing these assignment statements, the `list` vector conceptually looks like Figure 11.2.

C++ classifies the square brackets, `[]`, as a binary operator, since it requires two operands: a vector's name and an index.

Vectors may hold any valid C++ data type. The following code fragment declares three vectors of differing types:

```
std::vector<int> list;
std::vector<double> collection{ 1.0, 3.5, 0.5, 7.2 };
std::vector<char> letters{ 'a', 'b', 'c' };
```

Here `list` is empty but can contain integer values, `collection` is a vector of double-precision floating-point numbers initially containing the values contained in the initializer list, and `letters` holds the lowercase versions of the first three letters of the English alphabet. Figure 11.3 illustrates these three vector objects.

We can observe two key points from Figures 11.1–11.3:

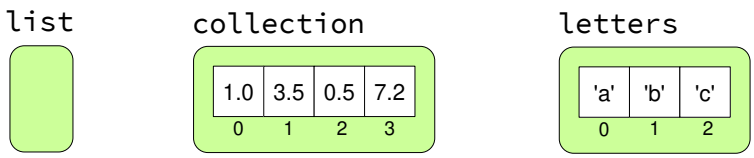
- Vectors store their elements in a contiguous block of memory. This means, for example, the memory occupied by the element at index 2 follows immediately after the memory occupied by the element at index 1 and immediately before the element at index 3.
- Elements in a vector are located by a numeric index. The first element is at index *zero*, not one.

In an expression such as

```
list[3]
```

Figure 11.3 Vectors containing different types of elements. Note that while two different vector objects may contain different types of elements, all the elements within a particular vector object must all have the same type.

```
vector<int> list;
vector<double> collection{ 1.0, 3.5, 0.5, 7.2 };
vector<char> letters{ 'a', 'b', 'c' };
```



the expression within the square brackets, in this case 3, is called an *index* or *subscript*. The subscript terminology comes from mathematicians who use subscripts to reference elements in a mathematical sequence (for example, V_2 represents the second element in the mathematical sequence V). Unlike the convention often used in mathematics, however, the first element in a vector is at position *zero*, not one. The expression `list[2]` can be read aloud as "`list` sub two." As a consequence of a zero beginning index, if vector `a` holds n elements, the last element in `a` is `a[`$n-1$`]`, not `a[`n`]`.

An element of a vector accessed via its index behaves just like a variable of that type; for example, suppose we declare `nums` as

```
std::vector<double> nums(10);
```

This declaration specifies a collection of 10 double-precision floating-point elements. The following code fragment shows how we can manipulate some of those elements:

```
// Print the fourth element
std::cout << nums[3] << '\n';
// The third element is the average of the first and last elements
nums[2] = (nums[0] + nums[9])/2;
// Assign elements at indices 1 and 4 from user input
std::cin >> nums[1] >> nums[4];
```

Note that the `<<` operator associated with the `std::cout` output stream object is not designed to work with vector objects as a whole:

```
std::cout << nums << '\n';   // <--- Will not compile!
```

The expression within the `[]` operator must be compatible with an integer. Suppose `a` is a vector, `x` is a numeric variable, `max` is a function that returns a numeric value, and `b` is a vector that holds numeric values. The following examples illustrate the variety of expressions that qualify as legal vector indices:

- an numeric literal: `a[34]`
- an numeric variable: `a[x]`

- an arithmetic expression: `a[x + 3]`
- an integer result of a function call that returns an numeric value: `a[max(x, y)]`
- an element of a vector that contains numeric values: `a[b[3]]`

A floating-point index is permissible but discouraged. The compiler will issue a warning about using a floating-point index with good reason. A vector may have an element at index 2 or index 3, but it is not possible to have an element located between indices 2 and 3; therefore, the executing program will truncate a floating-point index to an integer in order to select the proper element within the vector.

11.1.2 Traversing a Vector

The action of moving through a vector visiting each element is known as *traversal*. `for` loops are ideal for vector traversals. If `a` is an integer vector containing 10 elements, the following loop prints each element in `a`:

```
for (int i = 0;  i < 10;  i++)
    std::cout << a[i] << '\n';
```

The loop control variable, `i`, steps through each valid index of vector `a`. Variable `i`'s value starts at 0 and ends at 9, the last valid position in vector `a`.

The following loop prints contents of vector `a` in reverse order:

```
for (int i = 9;  i >= 0;  i--)
    std::cout << a[i] << '\n';
```

The following code produces a vector named `set` containing the integer sequence 0, 1, 2, 3, . . . , 999:

```
std::vector<int> set(1000);
for (int i = 0; i < 1000; i++)
    set[i] = i;
```

We now have all the tools we need to build a program that flexibly averages numbers while retaining all the values entered. Listing 11.3 (vectoraverage.cpp) uses a vector and a loop to achieve the generality of Listing 11.2 (averagenumbers2.cpp) with the ability to retain all input for later redisplay.

Listing 11.3: `vectoraverage.cpp`

```
#include <iostream>
#include <vector>

int main() {
    double sum = 0.0;
    const int NUMBER_OF_ENTRIES = 5;
    std::vector<double> numbers(NUMBER_OF_ENTRIES);

    std::cout << "Please enter " << NUMBER_OF_ENTRIES << " numbers: ";
    // Allow the user to enter in the values.
    for (int i = 0;  i < NUMBER_OF_ENTRIES;  i++) {
        std::cin >> numbers[i];
        sum += numbers[i];
    }
```

```
    std::cout << "The average of ";
    for (int i = 0;  i < NUMBER_OF_ENTRIES - 1;  i++)
        std::cout << numbers[i] << ", ";
    // No comma following last element
    std::cout << numbers[NUMBER_OF_ENTRIES - 1] << " is "
              << sum/NUMBER_OF_ENTRIES << '\n';
}
```

The output of Listing 11.3 (vectoraverage.cpp) is similar to the original Listing 11.1 (averagenumbers.cpp) program:

```
Please enter 5 numbers:  9 3.5 0.2 100 15.3
The average of 9.0, 3.5, 0.2, 100.0, 15.3 is 25.6
```

Unlike the original program, however, we now conveniently can extend this program to handle as many values as we wish. We need only change the definition of the `NUMBER_OF_ENTRIES` constant to allow the program to handle any number of values. This centralization of the definition of the vector's size eliminates duplicating a hard-coded value and leads to a program that is more maintainable. Suppose every occurrence of `NUMBER_OF_ENTRIES` were replaced with the literal value 5. The program would work exactly the same way, but changing the size would require touching many places within the program. When duplicate information is scattered throughout a program, it is a common error to update some but not all of the information when a change is to be made. If all of the duplicate information is not updated to agree, the inconsistencies result in errors within the program. By faithfully using the `NUMBER_OF_ENTRIES` constant throughout the program instead of the literal numeric value, we eliminate the possibility of such inconsistency.

The first loop in Listing 11.3 (vectoraverage.cpp) collects all five input values from the user. The second loop only prints the first four because it also prints a trailing comma after each element. Since no comma should be displayed after the last element, the program prints the last element after the loop is finished.

The compiler will insist that the programmer use a numeric value for an index, but the programmer must ensure that the index used is within the bounds of the vector. Since the index may consist of an arbitrary integer expression whose value cannot be determined until run time, the compiler cannot check for out-of-bound vector accesses; for example, in the code

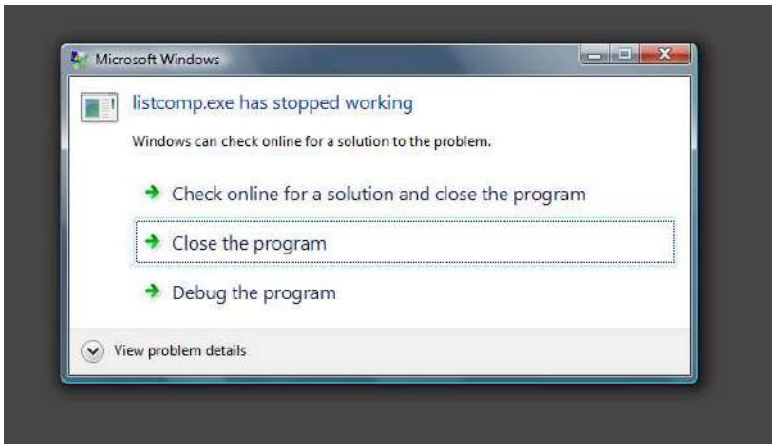
```
int x;
std::vector<int> v(10);  // Make a vector with 10 spaces available
std::cin >> x;           // User enters x at run time
v[x] = 1;                // Is this okay?  What is x?
```

the compiler cannot predict what number the user will enter. This means that misuse of a vector index can lead to run-time errors. To illustrate the problem, consider Listing 11.4 (vectoroutofbounds.cpp).

Listing 11.4: `vectoroutofbounds.cpp`

```
#include <iostream>
#include <vector>

int main() {
    const int SIZE = 3;
    std::vector<int> a{5, 5, 5};
    // Print the contents of the vector
    std::cout << "a contains ";
    for (int i = 0; i < SIZE; i++)
```

Figure 11.4 Memory access error under Visual C++




```
        std::cout << a[i] << " ";
    std::cout << '\n';
    // Change all the 5s in vector a to 8s
    for (int i = 0; i <= SIZE; i++)   // Bug: <= should be <
        a[i] = 8;
    // Reprint the contents of the vector
    std::cout << "a contains ";
    for (int i = 0; i < SIZE; i++)
        std::cout << a[i] << " ";
    std::cout << '\n';
}
```

Listing 11.4 (vectoroutofbounds.cpp) contains a logic error; the reassignment loop goes one past the end of vector a. Attempting to access elements outside the bounds of a vector produces what is known as *undefined behavior*. The C++ language standard uses this term to indicate a program's behavior is not specified, and complier writers are free to do whatever they want. Often, running programs that venture into undefined behavior will crash, but sometimes they may continue executing with no indication of a problem and appear to behave correctly most of the time. In other words, the actual program behavior is system dependent and compiler dependent. Consider code that represents undefined behavior to be a logic error, since its action is inconsistent across platforms and compilers. Simply said, the program's behavior is unpredictable. Unpredictable behavior is incorrect behavior.

In most cases, an out-of-bounds access simply accesses memory outside the vector. If this includes memory that does not belong to the executing program, modern operating systems will terminate the program and produce an error message. Under Visual C++ when the program is built as a debug version (the default when using the IDE), the program prints the contents of the vector the first time but crashes before it can print it out a second time. Microsoft Windows then displays the dialog box shown in Figure 11.4.

The program running under Linux or macOS may simply print in the console:

```
Segmentation fault
```

If your program is using a vector, and it terminates with such a message, you should check its source code carefully for places where out-of-bounds vector accesses are possible.

The following code fragment shows some proper and improper vector accesses:

```
std::vector<int> numbers(10);  // Declare the vector
numbers[0] = 5;            // Put value 5 first
numbers[9] = 2;            // Put value 2 last
numbers[-1] = 5;           // Out of bounds; first valid index is 0
numbers[10] = 5;           // Out of bounds; last valid index is 9
numbers[1.3] = 5;          // Compiler warning, 1.3 is not an int
```

In a vector traversal with a `for` loop such as

```
for (int i = 0;  i < SIZE;  i++)
    std::cout << a[i] << '\n';
```

where you know `SIZE` is the number of elements in the vector, it is easy to ensure that `i` cannot fall outside the bounds of vector `a`, but you should check an arbitrary index value before using it. In the following code:

```
int x;
std::vector<int> a(10);  // Make a vector with 10 spaces available
std::cin >> x;           // User enters x at run time
// Ensure x is within the bounds of the vector
if (0 <= x && x < 10)
    a[x] = 1;   // This is safe now
else
    std::cout << "Index provided is out of range\n";
```

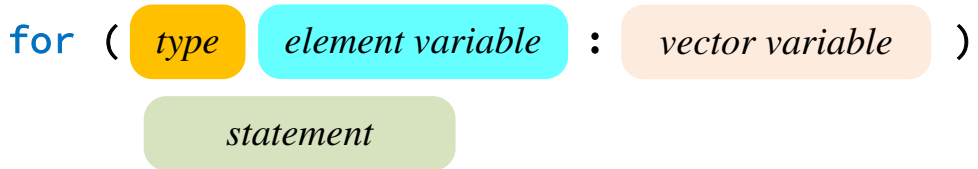
the `if` statement ensures the vector access is within the vector's bounds.

C++11 supports a variation of the `for` statement that uses special syntax for objects like vectors that support traversal. Commonly known as the *range-based for* or "foreach" statement, this version of the `for` statement permits vector traversal without an index variable keeping track of the position. The following code fragment uses a range-based `for` statement to print the contents of an integer vector named `vec`:

```
for (int n : vec)
    std::cout << n << ' ';
```

You can read this statement as "for each `int` `n` in `vec`, `std::cout << n << ''`," The colon (`:`) therefore is pronounced "in." In the first interation of this range-based `for` loop the variable `n` represents the first element in the vector `vec`. In the second iteration `n` represents the second element. The third time through `n` is the third element, and so forth. The declared variable assumes the role of a different vector element during each iteration of the loop. Note that the range-based `for` loop requires no control variable to keep track of the current index within the vector; the loop itself takes care of that detail, freeing the programmer from that task.

The general form of this range-based `for` statement is



If the element variable within the range-based `for` loop is declared to be a reference, the code within the body of the loop may modify the vector's elements. Listing 11.5 (foreachexample.cpp) allows a user to populate a vector with 10 numbers and then prints the vector's contents.

Listing 11.5: `foreachexample.cpp`

```
#include <iostream>
#include <vector>

int main() {
    // Declare a vector of ten numbers
    std::vector<double> vec(10);

    // Allow the user to populate the vector
    std::cout << "Please enter 10 numbers: ";
    for (double& elem : vec)
        std::cin >> elem;

    // Print the vector's contents
    for (double elem : vec)
        std::cout << elem << '\n';
}
```

Note that the first range-based `for` statement in Listing 11.5 (foreachexample.cpp) uses a reference variable to assign each element in the vector. The second range-based `for` statement does not need to use a reference variable because it does not change any of the contents of vector `vec`.

It is not always possible to use the range-based `for` statement when traversing vectors. The range-based `for` statement iterates forward through the vector elements and cannot move backwards. Also, the range-based `for` statement is not convenient if you want to consider only a portion of the elements in the vector. Examples include visiting every other element in the vector or considering only the first third of the elements. In these specialized cases you can use a standard `for` loop with an integer control variable. Section 20.5 explores some other options for traversing vectors in creative ways.

11.1.3 Vector Methods

A vector is an object, and objects differ from the simple types like `int`, `double`, and `bool`, in a number of ways. Most objects have access to special functions called *methods*. C++ programmers often refer to methods as *member functions*. A method is a function associated with a class of objects. A method invocation involves a slightly different syntax than a function invocation; for example, if `obj` is an object that supports a method named `f` that accepts no parameters, we can invoke `f` on behalf of `obj` with the statement:

```
obj.f();
```

The dot operator connects an object with a method to invoke. Other than this special invocation syntax, methods work very much like the global functions introduced in Chapter 8. A method may accept parameters and may return a value.

Vectors support a number of methods, but we will focus on seven of them:

- `push_back`—inserts a new element onto the back of a vector
- `pop_back`—removes the last element from a vector
- `operator[]`—provides access to the value stored at a given index within the vector
- `at`—provides bounds-checking access to the value stored at a given position within the vector
- `size`—returns the number of values currently stored in the vector
- `empty`—returns true if the vector contains no elements; returns false if the vector contains one or more elements
- `clear`—makes the vector empty.

We have seen how to declare a vector of a particular size and use the space provided; however, we are not limited by a vector's initial size. In order to add an element to a vector, we use the `push_back` method as follows:

```
std::vector<int> list;  // Make an empty vector that can hold integers
list.push_back(5);      // Add 5 to the end of list
list.push_back(-3);     // Add -3 to the end of the list
list.push_back(12);     // Add 12 to the end of list
```

After executing the three `push_back` calls above, the `list` vector conceptually looks just like Figure 11.2. The size of the vector adjusts automatically as new elements are inserted onto the back. Each `push_back` method call increases the number of elements in a vector by one.

The `pop_back` method performs the opposite action of `push_back`. A call to `pop_back` removes the last element from a vector, effectively reducing the number of elements in the vector by one. The following code fragment produces a vector named `list` that contains only the element 5.

```
std::vector<int> list;  // Declare list to be a vector
list.push_back(5);      // Add 5 to the end of list
list.push_back(-3);     // Add -3 to the end of the list
list.push_back(12);     // Add 12 to the end of list
list.pop_back();        // Removes 12 from the list
list.pop_back();        // Removes -3 from the list
```

We have been using the `operator[]` method to access an element in the vector. The word `operator` is reserved in C++, and that makes this method even more interesting. The expression

```
vec.operator[](2)
```

is the long way to write

```
vec[2]
```

Programmers use the shorter syntax exclusively, but the longer expression better illustrates the fact that the square bracket ([]) operator really is a method in the `std::vector` class of objects.

As we have seen, a programmer must be vigilant to avoid using an out-of-bounds index with the `operator[]` method. The vector class provides an additional method, `at`, that provides index bounds checking. The expression `vec[x]` in and of itself provides no bounds checking and so may represent undefined behavior. The functionally equivalent expression `vec.at(x)` will check to ensure that the index `x` is within the bounds of the vector. If `x` is outside the acceptable range of indices, the method is guaranteed to produce a run-time error. Listing 11.6 (vectoroutofbounds2.cpp) is a variation of Listing 11.4 (vectoroutofbounds.cpp) that uses the `at` method instead of the `operator[]` method.

Listing 11.6: `vectoroutofbounds2.cpp`

```
#include <iostream>
#include <vector>

int main() {
    const int SIZE = 3;
    std::vector<int> a{5, 5, 5};
    // Print the contents of the vector
    std::cout << "a contains ";
    for (int i = 0; i < SIZE; i++)
        std::cout << a.at(i) << " ";
    std::cout << '\n';
    // Change all the 5s in vector a to 8s
    for (int i = 0; i <= SIZE; i++)   // Bug: <= should be <
        a.at(i) = 8;
    // Reprint the contents of the vector
    std::cout << "a contains ";
    for (int i = 0; i < SIZE; i++)
        std::cout << a.at(i) << " ";
    std::cout << '\n';
}
```

When compiled and executed Listing 11.6 (vectoroutofbounds2.cpp) is guaranteed to produce a run-time error. Run-time errors are bad, but undefined behavior is worse because it can manifest itself as unpredictable run-time errors and programs that behave differently across multiple platforms.

The `size` method returns the number of elements in a vector. The following code fragment prints the contents of vector `list`:

```
int n = list.size();
for (int i = 0; i < n; i++)
    std::cout << list[i] << " ";
std::cout << '\n';
```

The exact type that the `size` method returns here is a `std::vector<int>::size_type`. This type is defined within the `std::vector<int>` class. It is compatible with the `unsigned` type and may be assigned to an `int` variable as shown above. We can avoid the additional local variable `n` as follows:

```
for (unsigned i = 0; i < list.size(); i++)
    std::cout << list[i] << " ";
std::cout << '\n';
```

Notice that `i`'s declared type is `unsigned`, not `int`. This prevents a warning when comparing `i` to `list.size()`. A comparison between signed and unsigned integer types potentially is dangerous, and

the compiler will alert us to that fact. To see why you should not take this warning lightly, consider the following code that we would expect to print nothing:

```
unsigned u = 0;
int i = 0;
while (i < u - 1) {
    std::cout << i << '\n';
    i++;
}
```

but instead it prints as many `unsigned` values as the system supports! This is because even though 0 is not less than -1, -1 is a signed value, not an unsigned value. The unsigned data type cannot represent signed numbers. An attempt to compute `unsigned` 0 minus 1 on a 32-bit system produces 4,294,967,295, which definitely is not less than zero. To be safe, assign the value of the `size` method to an `int` variable or use `unsigned` variables to control loop iterations. Better yet, use the range-based `for` statement whenever possible.

The `empty` method is a convenience method; if `vec` is a vector, the expression `vec.empty()` is equivalent to the Boolean expression `vec.size() != 0`. Note that the `empty` method does not *make* the vector empty; it simply returns true *if* the vector is empty and false if it is not.

The `clear` method removes all the elements from a vector leaving it empty. Invoking `clear` on an initially empty vector has no effect. Immediately after clearing a vector, the vector's `size` method will return zero, its `empty` method will return true, and a call to its `operator[]` method with an index of any value (including zero) will exhibit undefined behavior.

11.1.4 Vectors and Functions

A function can accept a vector as a parameter as shown in Listing 11.7 (vectortofunc.cpp)

Listing 11.7: `vectortofunc.cpp`

```
#include <iostream>
#include <vector>

/*
 *  print(v)
 *     Prints the contents of an int vector
 *     v is the vector to print
 */
void print(std::vector<int> v) {
    for (int elem : v)
        std::cout << elem << " ";
    std::cout << '\n';
}

/*
 *  sum(v)
 *     Adds up the contents of an int vector
 *     v is the vector to sum
 *     Returns the sum of all the elements
 *     or zero if the vector is empty.
 */
int sum(std::vector<int> v) {
```

```
    int result = 0;
    for (int elem : v)
        result += elem;
    return result;
}

int main() {
    std::vector<int> list{ 2, 4, 6, 8, };
    // Print the contents of the vector
    print(list);
    // Compute and display sum
    std::cout << sum(list) << '\n';
    // Zero out all the elements of list
    int n = list.size();
    for (int i = 0; i < n; i++)
        list[i] = 0;
    // Reprint the contents of the vector
    print(list);
    // Compute and display sum
    std::cout << sum(list) << '\n';
}
```

Listing 11.7 (vectortofunc.cpp) produces

```
2 4 6 8
20
0 0 0 0
0
```

The `print` function's definition:

```
void print(std::vector<int> v) {
```

shows that a vector formal parameter is declared just like a non-vector formal parameter. In this case, the print function uses pass by value, so during the program's execution an invocation of `print` will copy the data in the actual parameter (`list`) to the formal parameter (`v`). Since a vector potentially can be quite large, it generally is inefficient to pass a vector by value as shown above. Pass by value requires a function invocation to create a new vector object for the formal parameter and copy all the elements of the actual parameter into the new vector which is local to the function. A better approach uses pass by reference, with a twist:

```
void print(const std::vector<int>& v) {
    for (int elem : v)
        std::cout << elem << " ";
    std::cout << '\n';
}
```

The & symbol indicates that a caller invoking `print` will pass `v` by reference (see Section 10.9). This copies the address of the actual parameter (owned by the caller) to the formal parameter `v` instead of making a copy of all the data in the caller's vector. Passing the address is much more efficient because on most systems an address is the same size as a single `int`, whereas a vector could, for example, contain 1,000,000 `int`s. With pass by value a function invocation would have to copy all those 1,000,000 integers from the caller's actual parameter into the function's formal parameter.

Section 10.9 indicated that call-by-value parameter passing is preferred to call-by-reference parameter passing. This is because a function using pass by value cannot modify the actual variable the caller passed to it. Observe closely that our new `print` function declares its formal parameter `v` to be a `const` reference. This means the function cannot modify the actual parameter passed by the caller. Passing a vector object as a constant reference allows us to achieve the efficiency of pass by reference with the safety of pass by value.

Passing by `const` reference is not the same as pass by value, though. The function receiving a parameter passed by value can modify the parameter and return a modified copy. A function using pass by `const` reference cannot modify the parameter passed to it.

Like the `print` function, the `sum` function in Listing 11.7 (vectortofunc.cpp) does not intend to modify the contents of its vector parameter. Since the `sum` function needs only look at the vector's contents, its vector parameter should be declared as a `const` reference. In general, if a function's purpose *is* to modify a vector, the reference should not be `const`. Listing 11.8 (makerandomvector.cpp) uses a function named `make_random` that fills a vector with pseudorandom integer values.

Listing 11.8: `makerandomvector.cpp`

```
#include <iostream>
#include <vector>
#include <cstdlib>   // For rand

/*
 *  print(v)
 *     Prints the contents of an int vector
 *     v is the vector to print; v is not modified
 */
void print(const std::vector<int>& v) {
    for (int elem : v)
        std::cout << elem << " ";
    std::cout << '\n';
}

/*
 *  make_random(v)
 *     Fills an int vector with pseudorandom numbers
 *     v is the vector to fill; v is modified
 *     size is the maximum size of the vector
 */
void make_random(std::vector<int>& v, int size) {
    v.clear();                  // Empties the contents of vector
    int n = rand() % size + 1;  // Random size for v
    for (int i = 0; i < n; i++)
        v.push_back(rand());    // Populate with random values
}

int main() {
    srand(2);  // Set pseudorandom number generator seed
    std::vector<int> list;
    // Print the contents of the vector
    std::cout << "Vector initially: ";
    print(list);
    make_random(list, 20);
    std::cout << "1st random vector: ";
```

```
    print(list);
    make_random(list, 5);
    std::cout << "2nd random vector: ";
    print(list);
    make_random(list, 10);
    std::cout << "3rd random vector: ";
    print(list);
}
```

The `make_random` function in Listing 11.8 (makerandomvector.cpp) calls the vector method `clear` which makes a vector empty. We call `clear` first because we want to ensure the vector is empty before we add more elements. The function then proceeds to add a random number of random integers to the empty vector.

A function may return a vector object. Listing 11.9 (primelist.cpp) is a practical example of a function that returns a vector.

Listing 11.9: `primelist.cpp`

```
#include <iostream>
#include <vector>
#include <cmath>

/*
 *  print(v)
 *     Prints the contents of an int vector
 *     v is the vector to print; v is not modified
 */
void print(const std::vector<int>& v) {
    for (int elem : v)
        std::cout << elem << " ";
    std::cout << '\n';
}

/*
 *  is_prime(n)
 *     Determines the primality of a given value
 *     n an integer to test for primality
 *     Returns true if n is prime; otherwise, returns false
 */
bool is_prime(int n) {
    if (n < 2)
        return false;
    else {
        bool result = true;  // Provisionally, n is prime
        double r = n, root = sqrt(r);
        // Try all possible factors from 2 to the square
        // root of n
        for (int trial_factor = 2; result && trial_factor <= root;
              trial_factor++)
            result = (n % trial_factor != 0);
        return result;
    }
}

/*
```

```
 *  primes(begin, end)
 *     Returns a vector containing the prime
 *     numbers in the range begin...end.
 *     begin is the first number in the range
 *     end is the last number in the range
 */
std::vector<int> primes(int begin, int end) {
   std::vector<int> result;
   for (int i = begin; i <= end;  i++)
       if (is_prime(i))
           result.push_back(i);
   return result;
}

int main() {
    int low, high;
    std::cout << "Please enter lowest and highest values in "
              << "the range: ";
    std::cin >> low >> high;
    std::vector<int> prime_list = primes(low, high);
    print(prime_list);
}
```

The `primes` function declares a local vector variable named `result`. The function examines every value in the range `begin`...`end`, inclusive. The `primes` function uses `is_prime` as a helper function. If the `is_prime` function classifies a value as a prime, code within the `primes` function adds the value to the `result` vector. After `primes` has considered all the values in the provided range, `result` will contain all the prime numbers in that range. In the end, `primes` returns the vector containing all the prime numbers in the specified range.

When returning a local variable that is a built-in scalar type like `int`, `double`, `char`, etc., a C++ function normally makes a copy of the local variable to return to the caller. Making a copy is necessary because local variables exist only while the function in which they are declared is actively executing. When the function is finished executing and returns back its caller, the run-time environment reclaims the memory held by the function's local variables and parameters so their space can be used by other functions.

The return value in `primes` is not a simple scalar type—it is an object that can be quite large, especially if the caller passes in a large range. Modern C++ compilers generate machine code that eliminates the need to copy the local vector `result`. The technique is known as *return value optimization*, and it comes into play when a function returns an object declared within the function. With return value optimization, the compiler "knows" that the variable will disappear and that the variable is to be returned to the caller; therefore, it generates machine language code that makes the space for the result in the caller's memory space, not the called function. Since the caller is maintaining the space for the object, it persists after the function returns. Because of return value optimization you can return vectors by value in situations like this one without fear of a time-consuming copy operation.

Due to the fact that vectors may contain a large number of elements, you usually should pass vectors to functions as reference parameters rather than value parameters:

- If the function *is* meant to modify the contents of the vector, declare the vector as a non-`const` reference (that is, omit the `const` keyword when declaring the parameter).
- If the function is *not* meant to modify the contents of the vector, declare the vector as a `const` reference.

It generally is safe to return a vector by value from a function if that vector is declared local to the function. Modern C++ compilers generate optimized code that avoid the overhead of copying the result back to the caller.

11.1.5 Multidimensional Vectors

The vectors we have seen thus far have been one dimensional—simple sequences of values. C++ supports higher-dimensional vectors. A *two-dimensional vector* is best visualized as a table with rows and columns. The statement

```
std::vector<std::vector<int>> a(2, std::vector<int>(3));
```

effectively declares `a` to be a two-dimensional (2D) vector of integers. It literally creates a vector with two elements, and each element is itself a vector containing three integers. Note that the type of `a` is a vector of vector of integers. A 2D vector is sometimes called a *matrix*. In this case, the declaration specifies that 2D vector `a` contains two rows and three columns. Figure 11.5 shows the logical structure of the vector created by the following sequence of code:

```
std::vector<std::vector<int>> a(2, std::vector<int>(3));
a[0][0] =  5;
a[0][1] = 19;
a[0][2] =  3;
a[1][0] = 22;
a[1][1] = -8;
a[1][2] = 10;
```

The two-dimensional vector `a` is said to be a 2×3 vector, meaning it has two rows and three columns (as shown in Figure 11.5). Rows are arranged horizontally, and the values in columns are arranged vertically. In each of the assignment statements above, for example

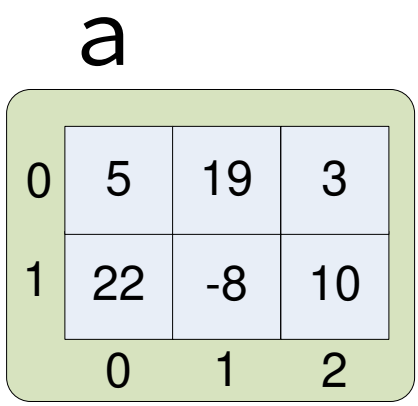
```
a[1][0] = 22;
```

the first index (here 1) signifies the row and the second index (here 0) denotes the column of the element within the vector. Literally, the expression `a[1][0]` means `(a[1])[0]`; that is, the element at index 0 of the vector at index 1 within `a`.

Using a syntax similar to the initialization lists of one-dimensional vectors, we can declare and initialize the 2D vector `a` from above as:

```
std::vector<std::vector<int>> a{{ 5, 19,  3},
                                {22, -8, 10}};
```

Figure 11.5 A 2×3 two-dimensional vector



Note that each row appears within its own set of curly braces, and each row looks like 1D vector initialization list.

To access an element of a 2D vector, use two subscripts:

```
a[r][c] = 4;                    // Assign element at row r, column c
std::cout << a[m][n] << '\n';   // Display element at row m, column n
```

The following function prints the contents of a 2D vector of `double`s:

```
void print(const std::vector<std::vector<double>>&  m) {
    for (unsigned row = 0; row < m.size(); row++) {
        for (unsigned col = 0; col < m[row].size(); col++)
            std::cout << std::setw(5) << m[row][col];
        std::cout << '\n';
    }
}
```

We can use range-based `for` statements to simplify the code:

```
void print(const std::vector<std::vector<double>>&  m) {
    for (const std::vector<double>& row : m) {  // For each row
        for (int elem : row)   // For each element in a row
            std::cout << std::setw(5) << elem;
        std::cout << '\n';
    }
    return sum;
}
```

The declaration of the parameter `m` is somewhat complicated. We can simplify the syntax by using a C++ type alias declaration. The statement

```
using Matrix = std::vector<std::vector<double>>;
```

creates a new name `Matrix` for the existing type of a 2D vector containing `double`s. Such type alias statements usually appear near the top of a source file and most often have global scope. It is less common to see a local type aliasing `using` statement within a function body.

You may encounter a type aliasing statement that uses the `typedef` keyword as shown here:

```
typedef std::vector<std::vector<double>> Matrix;
```

In this case this `typedef` directive works identically to the `using` type alias. C++ inherits the `typedef` keyword from the C programming language. The C++ `using` type alias is newer and supports type aliasing capabilities beyond those provided by the more primitive `typedef`. For this reason, you should prefer the `using` type aliasing to `typedef` when writing pure C++ code.

Given the `using` statement defining the new type name `Matrix`, we may express the parameter for the `print` function more simply:

```
void print(const Matrix& m) {
    for (const std::vector<double>& row : m) {  // For each row
        for (int elem : row)   // For each element in a row
            std::cout << std::setw(5) << elem;
        std::cout << '\n';
    }
    return sum;
}
```

We can take advantage of the type inference capability of C++11 to simplify the `print` function even further (see Section 3.10):

```
void print(const Matrix& m) {
    for (auto row : m) {  // For each row
        for (int elem : row)   // For each element in a row
            std::cout << std::setw(5) << elem;
        std::cout << '\n';
    }
    return sum;
}
```

Here we replaced the explicit type `const std::vector<double>&` with the word `auto`. The compiler is able to infer the type of the variable `row` from the context: `m` is a `Matrix` (that is, a `std::vector<std::vector<double>>`), so `row` must be an element of the 2D vector (which itself is a 1D vector).

Listing 11.10 (twodimvector.cpp) experiments with 2D vectors and takes advantage of the `using` type alias statement to simplify the code.

Listing 11.10: `twodimvector.cpp`

```
#include <iostream>
#include <iomanip>
#include <vector>
```

```
using Matrix = std::vector<std::vector<double>>;

// Allow the user to enter the elements of a matrix
void populate_matrix(Matrix& m) {
    std::cout << "Enter the " << m.size() << " rows of the matrix.\n";
    for (unsigned row = 0; row < m.size(); row++) {
        std::cout << "Row #" << row << " (enter " << m[row].size()
                  << " values):";
        for (double& elem : m[row])
            std::cin >> elem;
    }
}

void print_matrix(const Matrix m) {
    for (auto row : m) {
        for (double elem : row)
            std::cout << std::setw(5) << elem;
        std::cout << '\n';
    }
}

int main() {
    int rows, columns;
    std::cout << "How many rows? ";
    std::cin >> rows;
    std::cout << "How many columns? ";
    std::cin >> columns;
    // Declare the 2D vector
    Matrix mat(rows, std::vector<double>(columns));
    // Populate the vector
    populate_matrix(mat);
    // Print the vector
    print_matrix(mat);
}
```

An expression that uses just one index with a 2D vector represents a single row within the 2D vector. This row is itself a 1D vector. Thus, if `a` is a 2D vector and `i` is an integer, then the expression

```
a[i]
```

is a 1D vector representing row `i`.

We can build vectors with dimensions higher than two. Each "slice" of a 3D vector is simply a 2D vector, a 4D vector is a vector of 3D vectors, etc. For example, the statement

```
matrix[x][y][z][t] = 1.0034;
```

assigns 1.0034 to an element in a 4D vector of `double`s. In practice, vectors with more than two dimensions are rare, but advanced scientific and engineering applications sometimes require higher-dimensional vectors.

11.2 Arrays

C++ is an object-oriented programming language, and a vector is an example of a software object. C++ began as an extension of the C programming language, but C does not directly support object-oriented programming. Consequently, C does not have vectors available to represent sequence types. The C language uses a more primitive construct called an *array*. True to its C roots, C++ supports arrays as well as vectors. Some C++ libraries use arrays in instead of vectors. In addition, C++ programs can use any of the large number of C libraries that have been built up over the past 40+ years, and many of these libraries process arrays. While a more modern construct like `std::vector` may be better suited for may of the roles an array has played in the past, it nonetheless is important for C++ programmers to be familiar with arrays.

An array is a variable that refers to a block of memory that, like a vector, can hold multiple values simultaneously. An array has a name, and the values it contains are accessed via their position within the block of memory designated for the array. Also like a vector, the elements within an array must all be of the same type. Arrays may be local or global variables. Arrays are built into the core language of both C and C++. This means you do not need to add any `#include` directives to use an array within a program.

11.2.1 Static Arrays

Arrays come in two varieties, static and dynamic. A programmer must supply the size of a static array when declaring it; for example, the following statement:

```
// list is an array of 25 integers
int list[25];
```

declares `list` to be an array of 25 integers. The value within the square brackets specifies the number of elements in the array, and the size is fixed for the life of the array. The value within the square brackets must be a constant value determined at compile time. It can be a literal value or a symbolic constant, but it cannot be a variable. This is in contrast to vector declarations in which the initial vector size may be specified by a variable with a value determined at run time:

```
int x;
std::cin >> x;                // Get x's value from the user at run time
int list_1[x];                // Illegal for a static array
std::vector<int> list_2(x);   // OK for a vector
```

It is possible to declare an array and initialize it with a sequence of elements, with a syntax similar to that of vectors:

```
double collection[] = { 1.0, 3.5, 0.5, 7.2 };
```

The compiler can count the elements in the initializer list, so you need not supply a number within the square brackets. If you provide a number within the square brackets, it should be at least as large as the number of elements in the initialization list. The equals symbol is required for array initialization. You optionally can use the equals symbol as shown here when initializing vectors, but it is not required for vectors.

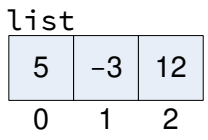
If the declaration omits an initializer list, as in:

```
int arr[100];
```

the initial values of the elements of the array depend on the context of the declaration:

Figure 11.6 A simple array with three elements

```
int list[3];
list[0] = 5;
list[1] = -3;
list[2] = 12;
```



- An executing program does not initialize the contents of a local arrays.

- The run-time environment initializes the values of global numeric arrays to all zeros.

As with vectors, once we declare an array, we can access its elements using the square bracket operator:

```
int list[3];      // Declare list to be an array of three ints
list[0] = 5 ;     // Make the first element 5
list[1] = -3 ;    // Make the second element -3
list[2] = 12 ;    // Make the last element 12
```

This code fragment shows that the square brackets used *after* the point of declaration allow us to access an individual element based on that element's position within the block of memory assigned to the array. This syntax is identical to that used with vector objects. As with vectors, the very first element in an array appears at index 0.

After executing these assignment statements, the `list` array conceptually looks like Figure 11.6.

The array square brackets operator has two distinct meanings depending on the context:

1. At the array's declaration, for example,

   ```
   double nums[10];
   ```

   the number within the square brackets specifies the number of elements that the array can hold. The compiler uses this number along with the type of the array to determine how much memory to allocate for the array.

2. After the array's declaration, for example,

   ```
   nums[3] = 10.45;
   ```

   the number within the square brackets represents an index locating a specific element within the memory allocated to the array.

Unlike with vectors, the compiler will insist that the programmer use an integral value for an index; for example, the following statement:

```
numbers[1.3] = 5;
```

is illegal if `numbers` is an array. Identically to vectors, the programmer must ensure that the array index is within the bounds of the array. Accessing an element via an index outside the bounds of an array results in undefined behavior.

We can pass an array to a function, as shown in Listing 11.11 (arraytofunc.cpp), the array-based version of Listing 11.7 (vectortofunc.cpp).

Listing 11.11: `arraytofunc.cpp`

```
#include <iostream>

/*
 *  print(a, n)
 *     Prints the contents of an int array
 *     a is the array to print
 *     n is the size of the array
 */
void print(int a[], int n) {
    for (int i = 0; i < n; i++)
        std::cout << a[i] << " ";
    std::cout << '\n';
}

/*
 *  sum(a, n)
 *     Adds up the contents of an int array
 *     a is the array to sum
 *     n is the size of the array
 */
int sum(int a[], int n) {
```

```
    int result = 0;
    for (int i = 0; i < n; i++)
        result += a[i];
    return result;
}

int main() {
    int list[] = { 2, 4, 6, 8 };
    // Print the contents of the array
    print(list, 4);
    // Compute and display sum
    std::cout << sum(list, 4) << '\n';
    // Zero out all the elements of list
    for (int i = 0; i < 4; i++)
        list[i] = 0;
    // Reprint the contents of the array
    print(list, 4);
    // Compute and display sum
    std::cout << sum(list, 4) << '\n';
}
```

As shown in the `print` function's definition:

```
void print(int a[], int n)
```

empty square brackets follow the name of an array formal parameter. The programmer can supply a number within the square brackets, but the compiler will ignore it. When calling a function that accepts an array parameter, as in

```
print(list, 4);
```

the programmer must pass the array's size along with the array name. This is because an array simply references a block of memory and has has no notion of its size.

An undecorated array name by itself in C++ source code behaves like a constant pointer to the beginning element of the array. Consequently, when an array is passed as an actual parameter during a function call, as in Listing 11.11 (arraytofunc.cpp):

```
print(list, 4);
```

the function is passed the address of the array—the array's address is bound to the formal parameter. So, while the function cannot affect the address of the array itself, the function has total access to the array's contents. In Listing 11.12 (cleararray.cpp), the function `clear` modifies the contents of any array sent to it, making all the elements zero.

Listing 11.12: `cleararray.cpp`

```
#include <iostream>

/*
 *  print(a, n)
 *     Prints the contents of an int array
 *     a is the array to print
 *     n is the size of the array
 */
void print(int a[], int n) {
```

```
    for (int i = 0; i < n; i++)
        std::cout << a[i] << " ";
    std::cout << '\n';
}

/*
 *  clear(a, n)
 *     Makes all the elements of array a zero
 *     a is the array to zero out
 *     n is the size of the array
 */
void clear(int a[], int n) {
    for (int i = 0; i < n; i++)
        a[i] = 0;
}

int main() {
    int list[] = { 2, 4, 6, 8 };
    // Print the contents of the array
    print(list, 4);
    // Zero out the array
    clear(list, 4);
    // Reprint the contents of the array
    print(list, 4);
}
```

The output of Listing 11.12 (cleararray.cpp) is

```
2 4 6 8
0 0 0 0
```

The `clear` function actually modifies the contents of `main`'s `list` array. The function works on the array's actual elements, not copies of its elements.

Arrays, therefore, by default are passed by reference with respect to the elements they contain. By default, an array's contents are open to corruption by errant functions. In order to protect an array so that a function may read its contents but not modify its contents, the parameter must be declared `const`, as in

```
int sum(const int a[], int n)
```

In Listing 11.11 (arraytofunc.cpp) there is no reason why the `print` and `sum` functions should be able to to modify the contents of their array parameters, so the `const` specifier should be added to their definitions.

As a general rule, all functions that accept array parameters should declare the array contents as `const` unless they need to modify the elements of the array.

As a general rule, all functions that accept array parameters should declare the array contents as `const` unless they need to modify the elements of the array. For example, use

```
int sum(const int a[], int n)
```

rather than

```
int sum(int a[], int n)
```

if the `sum` function must be able to examine the contents of the array but is not intended to modify its contents.

C++ does not directly support empty arrays. A physical array must contain at least one element. Sometimes it is appropriate to consider an array that is conceptually empty; for example, what if we wish to pass an "empty" array to the `sum` function? We can pass 0 as the second parameter since an empty array contains no elements, but what should we pass as the first parameter? Any array will do, but there is no need create an real array when none truly is needed. Instead, we may use `nullptr`:

```
quanitity = sum(nullptr, 0);
```

If you scrutinize the body of the `sum` function, you will see that this call will assign to `quantity` the correct answer, namely zero.

11.2.2 Pointers and Arrays

An array name used in C++ source code references a location in memory, the address of the first element (element at index 0) in the array. In this way an array name is similar to a constant pointer (see Section 10.7 for more information about C++ pointers). Because of this, we can treat in some ways an array identifier like a pointer. Similarly, we can direct a pointer to point to an array, and then treat the pointer itself as if it were an array.

Listing 11.13 (pointerprint.cpp) uses a pointer to traverse an array.

Listing 11.13: `pointerprint.cpp`

```
#include <iostream>

int main() {
    int a[] = { 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 },
        *p;

    p = &a[0];  // p points to first element of array a

    // Print out the contents of the array
    for (int i = 0; i < 10; i++) {
        std::cout << *p << ' ';  // Print the element p points to
        p++;  // Increment p so it points to the next element
    }
    std::cout << '\n';
}
```

The statement

```
p = &a[0];
```

sets p to point to the first element of array a. A shorter way to accomplish the same thing is

```
p = a;
```

since a is itself a reference to the array's location in memory. This assignment statement clearly demonstrates the association between array variables and pointer variables. Note that the opposite assignment (a = p) is impossible, because array a declared as above may not appear by itself on the left side of the assignment operator.

Pointer variables can participate in addition and subtraction expressions. The statement

```
p++;
```

changes the address stored in p so subsequently the pointer will point to the next integer position in memory. If p is assigned to array a, incrementing p redirects it to point to a's next element. [1] In Figure 11.7, pointer p is assigned to array a and then incremented to refer to various elements within a.

The expression

```
p[0]
```

is another way to write

```
*p
```

so the array indexing operator ([]) can be used with pointers. The expression

```
p[5]
```

represents the element at index 5 within the array pointed to by p.

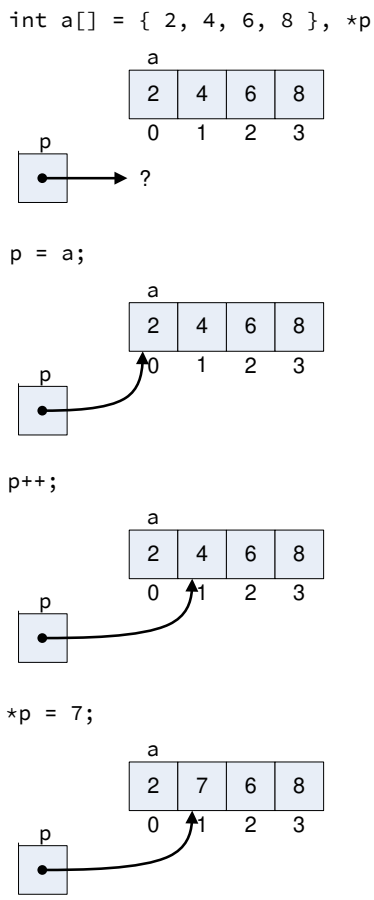
Listing 11.14 (pointerprint2.cpp) uses pointers in a different way to traverse an array.

Listing 11.14: pointerprint2.cpp

```
#include <iostream>

int main() {
    int a[] = { 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 },
        *begin, *end, *cursor;

    begin = a;      // begin points to the first element of array a
    end = a + 10;   // end points to just after the last element

    // Print out the contents of the array
    cursor = begin;
    while (cursor != end) {
        std::cout << *cursor << ' ';  // Print the element
        cursor++;  // Increment cursor so it points to the next element
```

[1] Each byte in memory has a unique numeric address. Since most C++ types require more than one byte of storage, incrementing a pointer by 1 does not simply add 1 to the address it holds. The amount added depends on the type of the pointer; for example, on systems using 32-bit (4 byte) integers, adding 1 to an integer pointer variable increments its address by 4, not 1. The compiler knows the type of the pointer variable because programmers must declare all variables. The compiler, therefore, can automatically adjust the arithmetic to work properly. If a pointer points to memory within an array and is of the same type as the array, incrementing the pointer correctly repositions the pointer to point to the next element in the array.

Figure 11.7 Incrementing a pointer to an array



```
    }
    std::cout << '\n';
}
```

In Listing 11.14 (pointerprint2.cpp), the code

```
cursor = begin;
while (cursor != end) {
    std::cout << *cursor << ' ';  // Print the element
    cursor++;  // Increment cursor so it points to the next element
}
```

can be expressed more succinctly as a `for` loop:

```
for (cursor = begin; cursor != end; cursor++)
    std::cout << *cursor << ' ';  // Print the element
```

If pointer `p` points to an array, the element at index `i` can be accessed as either `p[i]` or `*(p + i)`. The expression `*(p + i)` dereferences the address that is `i` positions away from the address referenced by `p`.

Sometimes pointer notation is used to represent an array parameter to a function. The array print function that begins

```
void print(const int a[], int n)
```

could instead be written

```
void print(const int *a, int n)
```

where `a` is a pointer to an array. The compiler treats the two forms identically in the machine language it produces.

Listing 11.15 (recursivearrayprint.cpp) uses two array printing functions, `iterative_print` and `recursive_print`. Both use the pointer notation when declaring their array parameters.

Listing 11.15: `recursivearrayprint.cpp`

```
#include <iostream>

/*
 *  iterative_print(a,n)
 *     Prints the contents of array a
 *     a is the array to print; a is not modified
 *     n is number of elements in a
 */
void iterative_print(const int *a, int n) {
    for (int i = 0; i < n; i++)
        std::cout << a[i] << ' ';
}

/*
 *  recursive_print(a,n)
 *     Prints the contents of array a
 *     a is the array to print; a is not modified
 *     n is number of elements in a
```

```
 */
void recursive_print(const int *a, int n) {
    if (n > 0) {
        std::cout << *a << ' ';   // Print the first element of the array
        recursive_print(a + 1, n - 1);  // Print rest of the array
    }
}

int main() {
   int list[] = { 23, -3, 4, 215, 0, -3, 2, 23, 100, 88, -10 };
   iterative_print(list, 11);
   recursive_print(list, 11);
}
```

The function `iterative_print` uses a loop, while `recursive_print` uses recursion instead. Inside the `recursive_print` function, the expression `a + 1` points to the second element in array `a`. `a + 1` essentially represents the rest of the array—everything except for the first element. The statement

```
recursive_print(a + 1, n - 1);
```

calls the function recursively, passing the remainder of the array (`a + 1`) with a length that is one less (`n - 1`) than before. The recursion terminates when the array's length is zero.

Instead of passing a pointer to an array and the array's size, we can pass two pointers. The first pointer points to the beginning of the array and another pointer points just past the end of the array. The following `print` function illustrates:

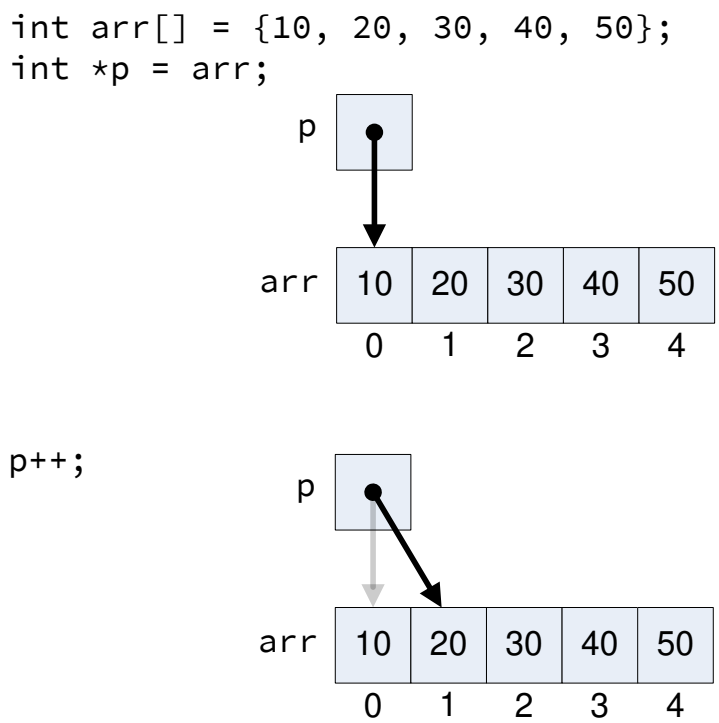
```
void print(int *begin, int *end) {
    for (int *elem = begin; elem != end; elem++)
        std::cout << *elem << ' ';
    std::cout << '\n';
}
```

Notice that this code does not appear to be using arrays at all! This code takes advantage of pointer arithmetic to move the `elem` pointer from the first element to each successive element in the array. Since the memory devoted to an array is a continuous, contiguous block of memory addresses, pointer arithmetic simply redirects a pointer from one element in an array to another element in that array. Listing 11.16 (pointerarithmetic.cpp) provides some more examples of pointer arithmetic in action.

Listing 11.16: `pointerarithmetic.cpp`

```
#include <iostream>

int main() {
    // Make an array
    int arr[] = {10, 20, 30, 40, 50};
    int *p = arr;               // p points to the first element
    std::cout << *p << '\n';    // Prints 10, does not change p
    std::cout << p[0] << '\n';  // Prints 10, does not change p
    std::cout << p[1] << '\n';  // Prints 20, does not change p
    std::cout << *p << '\n';    // Prints 10, does not change p
    p++;                        // Advances p to the next element
    std::cout << *p << '\n';    // Prints 20, does not change p
    p += 2;                     // Advance p two places
    std::cout << *p << '\n';    // Prints 40, does not change p
```

Figure 11.8 Pointer arithmetic. Incrementing pointer **p** moves it to the next element in the array.




```
    std::cout << p[0] << '\n';  // Prints 40, does not change p
    std::cout << p[1] << '\n';  // Prints 50, does not change p
    p--;                        // Moves p back one place
    std::cout << *p << '\n';    // Prints 30, does not change p
}
```

Listing 11.16 (pointerarithmetic.cpp) prints

```
10
10
20
10
20
40
40
50
30
```

Figure 11.8 illustrates how Listing 11.16 (pointerarithmetic.cpp) works.

Listing 11.16 (pointerarithmetic.cpp) also shows how we can use the square bracket array access operator with a pointer. The expression p[i] refers to the element in memory i positions from the location pointed to by p.

Going back to the `print` function that uses pointers:

```
void print(int *begin, int *end) {
    for (int *elem = begin; elem != end; elem++)
        std::cout << *elem << ' ';
    std::cout << '\n';
}
```

the pointer `elem` first points to the same element to which `begin` points (that is, the first element in the array), and within the loop it moves to the next element repeatedly until it passes the last element. A caller could invoke the function as

```
int list[25];                  // Allocate a static array
for (int i = 0; i < 25; i++)
    list[i] = rand() % 1000;   // Fill with pseudorandom values
print(list, list + 25);        // Print the elements of list
```

The advantage of the begin/end pointer approach is that it allows a programmer to pass a *slice* of the array to the function; for example, if we wish to print the elements of the array from index 3 to index 7, we would call the function as

```
print(list + 3, list + 8);  // Print elements at indices 3, 4, 5, 6, 7
```

11.2.3 Dynamic Arrays

Programmers need not worry about managing the memory used by static arrays. The compiler and run-time environment automatically ensure the array has enough space to hold all of its elements. The space held by local arrays is freed up automatically when the local array goes out of the scope of its declaration. Global arrays live for the lifetime of the executing program. Memory management for static arrays, therefore, works just like scalar variables.

Static arrays have a significant limitation: The size of a static array is determined at compile time. The programmer may change the size of the array in the source code and recompile the program, but once the program is compiled into an executable program any static array's size is fixed. For many applications the circumstances of the executing program must determine the size of an array. Static arrays lack such flexibility.

One approach to implement a flexibly-sized array defines the array to hold as many items as it conceivably will ever need. Listing 11.17 (largearrayaverage.cpp) uses this approach.

Listing 11.17: `largearrayaverage.cpp`

```
#include <iostream>

// Maximum number of expected values is 1,000,000
const int MAX_NUMBER_OF_ENTRIES = 1000000;
double numbers[MAX_NUMBER_OF_ENTRIES];

int main() {
    double sum = 0.0;
    int size;  // Actual number of entries

    // Get effective size of the array
    std::cout << "Please enter number of values to process: ";
```

```
    std::cin >> size;

    if (size > 0) {  // Nothing to do with no entries
        std::cout << "Please enter " << size << " numbers: ";
        // Allow the user to enter in the values.
        for (int i = 0;  i < size;  i++) {
            std::cin >> numbers[i];
            sum += numbers[i];
        }
        std::cout << "The average of ";
        for (int i = 0;  i < size - 1;  i++)
            std::cout << numbers[i] << ", ";
        // No comma following last element
        std::cout << numbers[size - 1] << " is "
                  << sum/size << '\n';
    }
}
```

Listing 11.17 (largearrayaverage.cpp) creates an array that can hold one million entries. The variable `size` keeps track of the actual size needed by the user. While the array's maximum size is one million, the part of the array actually used during a given execution may be much smaller.

Notice that the array `numbers` is a global variable and is not local to `main`. This is because most systems limit the amount of storage available to local variables within functions. Local variables reside in an area of memory known as the *stack*. Local variables exist in memory only when the function that uses them is invoked; therefore, the stack grows and shrinks as functions execute and return. Global variables, on the other hand, exist in the computer's memory for the life of the program's execution. Global variables are stored in what is known as *static memory*. There is a limit to the amount of static memory available for global variables (the amount of memory in RAM or virtual memory on disk, if nothing else), but the global limit usually is much higher then the local limit. One million double-precision floating-point numbers consumes 8,000,000 bytes (8 megabytes) in Visual C++ and on many other systems. The default stack size for local variables under Visual C++ is only one megabyte, although the stack size can be increased by adjusting the compiler and linker build options.

While the approach taken in Listing 11.17 (largearrayaverage.cpp) works, it wastes memory resources. In modern computing, a user may have multiple programs open at the same time. If each program is tying up the maximum amount of memory it may ever need, there may not be enough real memory (RAM) to go around, and the computer will be forced to use more virtual memory (shuttling portions of the running program's memory to and from the disk drive). Virtual memory access greatly degrades the speed of a program, and so the user's experience suffers. It is important that each program uses its resources wisely. Statically allocating the largest array that might ever be needed is not a good approach.

Fortunately, there is a way for a programmer to create an array of the exact size needed, even if that size is not known until run time. Listing 11.18 (flexiblearrayaverage.cpp) shows how.

Listing 11.18: `flexiblearrayaverage.cpp`

```
#include <iostream>

int main() {
    double sum = 0.0;
    double *numbers;  // Note: numbers is a pointer, not an array
    int size;  // Actual number of entries

    // Get effective size of the array
```

```
    std::cout << "Please enter number of values to process: ";
    std::cin >> size;

    if (size > 0) {  // Nothing to do with no entries
        std::cout << "Please enter " << size << " numbers: ";
        // Allocate the exact size needed
        numbers = new double[size];  // Dynamically allocated array
        // Allow the user to enter in the values.
        for (int i = 0;  i < size;  i++) {
            std::cin >> numbers[i];
            sum += numbers[i];
        }
        std::cout << "The average of ";
        for (int i = 0;  i < size - 1;  i++)
            std::cout << numbers[i] << ", ";
        // No comma following last element
        std::cout << numbers[size - 1] << " is "
                  << sum/size << '\n';
        delete [] numbers;  // Free up the space held by numbers
    }
}
```

Notice that `numbers` is not declared to be an array; it is a pointer:

```
double *numbers;  // Note: numbers is a pointer, not an array
```

This statement makes space for a single pointer, merely four bytes on a 32-bit system or eight bytes on a 64-bit system. The expression

```
new double[size]
```

allocates at run time a block of memory for an array that can hold exactly `size` double-precision floating-point elements. The value of this expression is the starting address of the newly allocated memory block.

The statement

```
numbers = new double[size];  // Dynamically allocated array
```

therefore assigns to `numbers` the address of the allocated memory. A dynamic array is simply a pointer (see Section 10.7) that points to a dynamically allocated block of memory. As Listing 11.18 (flexiblearrayaverage.cpp) shows, the value within the square brackets in a dynamic array allocation may be a variable determined at run time. Figure 11.9 compares the conceptual differences among of static and and dynamic arrays and vectors.

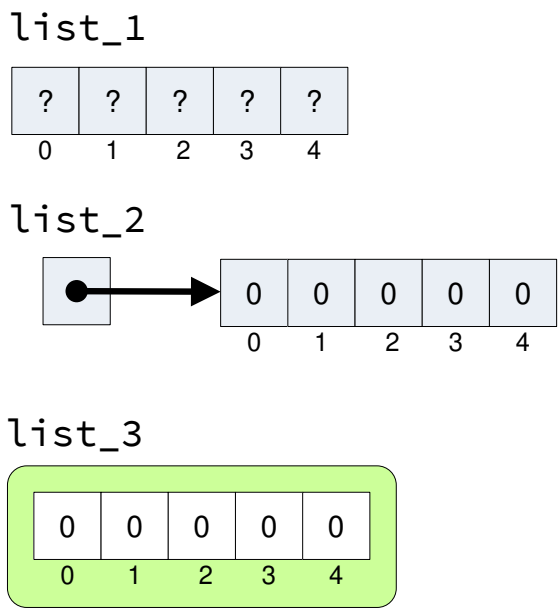
While we access a dynamic array via a pointer, a static array array behaves like a constant pointer; that is, a pointer that we cannot reassign to point elsewhere.

The reserved word `new` allocates memory for the array's elements. `new` technically is classified as an operator. The area of memory used for dynamic allocation via `new` is different from both the area used for local variables (the stack) and global variables (static memory). Dynamic memory comes from the *heap*. Section 18.1 provides more details about the memory available to an executing C++ program, but here it is sufficient to note that the heap has a capacity much larger than the stack.

The variable `numbers` is local to `main`, so it lives on the stack. It is a pointer, so it stores a memory address. The `new` expression returns an address to memory in the heap. So, even though the `numbers`

Figure 11.9 Comparing static arrays, dynamic arrays, and vectors

```
int list_1[5];
int *list_2 = new int[5];
vector<int> list_3(5);
```



variable itself is stored on the stack, the memory it references is located in the heap. We do not need to worry about our array being too big to be local to `main`, since its contents do not consume stack space.

An executing program automatically allocates on the stack the local variables of a function when a caller invokes the function. The executing program also automatically deallocates local variables when the function returns. The programmer does not need to explicitly manage local variables. Dynamically allocated memory, however, requires more attention on the part of the programmer. The statement

```
delete [] numbers;  // Free up the space held by numbers
```

uses the `delete` operator to free up the memory held by the `numbers` array. The programmer is responsible for deallocating memory that was allocated with `new`. Every use of the `new` operator should have a corresponding use of `delete` somewhere later in the program's execution. Notice that the square brackets (`[]`) are empty. The run-time environment keeps track of the amount of memory to free up.

You should develop the habit of ensuring that every use of the `new` operator has an associated call of `delete`. If the `delete` statement is omitted from Listing 11.18 (flexiblearrayaverage.cpp), the program in this case likely will behave no differently since most operating systems reclaim all the dynamic memory a program holds when the program finishes executing. Serious problems arise in longer running programs that must allocate and deallocate dynamic memory over time. Programs that run for extended periods of time, like web servers and operating system kernels, can crash due to failures to properly deallocate memory. The condition is known as a *memory leak*—the program overtime allocates more and more memory via `new` but never releases the memory back with `delete`. Eventually the program uses up all of the available heap space and crashes.

11.2.4 Copying an Array

It is important to remember that within C++ source code a static array variable behaves similarly to a constant pointer. At first it may seem plausible to make a copy of an array as follows:

```
int a[10], b[10];            // Declare two arrays
for (int i = 0; i < 10; i++) // Populate one of them
    a[i] = i;                // a is filled with increasing values
b = a;                       // Make a copy of array a?
```

Since `b` behaves like a constant pointer, we cannot reassign it; that is, the name `b` cannot appear on the left side of the assignment operator all by itself. Wherever `b` points, it must continue to point there during its lifetime. The code above will not compile.

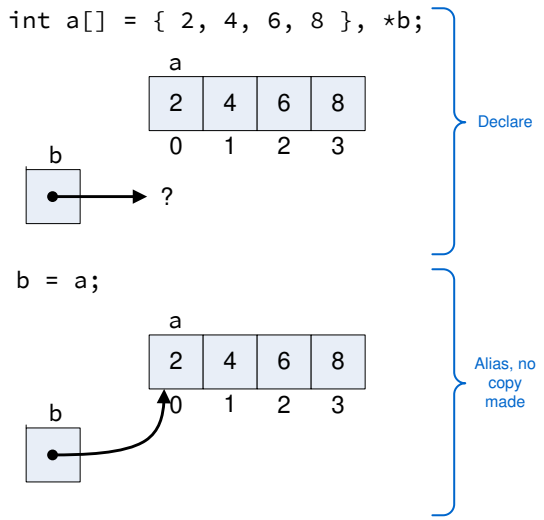
Perhaps the solution is to use a dynamic array (see Section 11.2.3 for information about dynamic arrays). If `b` is not `const`, can we copy the array through simple assignment? The following code is legal:

```
int a[10], *b;               // Declare two arrays, one dynamic
for (int i = 0; i < 10; i++) // Populate one of them
    a[i] = i;                // a is filled with increasing values
b = a;                       // Make a copy of array a?
```

but `b` is *not* a copy of `a`. `b` aliases `a`, so changing the contents of `a` will also change the contents of `b` identically, since `b` refers to the exact memory to which `a` refers. Figure 11.10 illustrates the aliasing.

This illustrates another key difference between vectors and arrays: It is not possible to assign one array variable to another through a simple assignment statement that copies all the elements from one array into another. If the arrays are static arrays, the simple assignment is illegal. If the arrays are dynamic arrays, the

Figure 11.10 Faulty array copy



assignment simply makes the two pointers point to the same block of memory, which is not the same effect as vector assignment.

The following code shows the proper way to make a copy of array a:

```
int a[10], *b;   // Declare two arrays, one dynamic
for (int i = 0; i < 10; i++) // Populate one of them
    a[i] = i;       // a is filled with increasing values
// Really make a copy of array a
b = new int[10];   // Allocate b
for (int i = 0;  i < 10;  i++)
    b[i] = a[i];
```

Separate space for the dynamic array must be allocated, and then each element from the original array must be copied into the new array. Figure 11.11 shows how this process works.

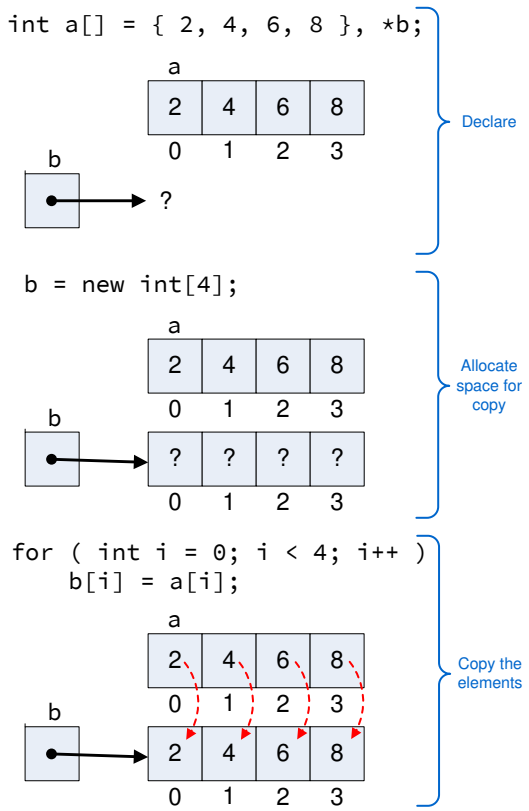
It is important to note that since the code above allocates b dynamically with new, code elsewhere within the program should eventually use delete to free up b's space when it is no longer used. Failure to properly deallocate b constitutes a memory leak.

Consider the following scenario:

```
int *a = new int[10],
    *b = new int[10];
// Do somethings with arrays a and b, and then
b = a;
```

Here dynamic array b already was in use before the assignment. Not only does this simple assignment create an alias, but it also creates a memory leak. If the reassigned array originally was pointing to a block of memory allocated on its behalf, and no other pointers reference that block of memory, that block is unreachable by the executing program.

Figure 11.11 Correct array copy



Listing 11.18 (flexiblearrayaverage.cpp) requires the user to enter up front the number of values to average. This is inconvenient, and people are notoriously poor counters. One solution is to allocate a minimal size array, and then resize it as necessary when it fills up. Listing 11.19 (resizearray.cpp) uses this approach.

Listing 11.19: `resizearray.cpp`

```
#include <iostream>

int main() {
    double sum = 0.0,  // Sum of the elements in the list
           *numbers,   // Dynamic array of numbers
           input;      // Current user entry

    // Initial size of array and amount to expand full array
    const int CHUNK = 3;
    int size = 0,          // Current number of entries
        capacity = CHUNK;  // Initial size of array


    // Allocate a modest-sized array to begin with
    numbers = new double[capacity];

    std::cout << "Please enter any number of nonnegative values "
              << "(negative value ends the list): ";
    std::cin >> input;

    while (input >= 0) {  // Continue until negative number entered
        if (size >= capacity) {  // Room left to add an element?
            capacity += CHUNK;   // Expand array
            double *temp = new double[capacity];  // Allocate space
            for (int i = 0; i < size; i++)
                temp[i] = numbers[i];  // Copy existing values
            delete [] numbers;   // Free up old space
            numbers = temp;      // Update numbers to new location
            std::cout << "Expanding by " << CHUNK << '\n';
        }
        numbers[size] = input;   // Add number to array at last position
        size++;                  // Update last position
        sum += input;            // Add to running sum
        std::cin >> input;       // Get next number
    }
    if (size > 0) {              // Can't average less than one number
        std::cout << "The average of ";
        for (int i = 0;  i < size - 1;  i++)
            std::cout << numbers[i] << ", ";
        // No comma following last element
        std::cout << numbers[size - 1] << " is "
                  << sum/size << '\n';
    }
    else
        std::cout << "No numbers to average\n";
    delete [] numbers;  // Free up the space held by numbers
}
```

Notice that the programmer of Listing 11.19 (resizearray.cpp) is expending a lot of effort to implement the functionality provided by `std::vector`. Unlike vectors, an array has a fixed size. It is impossible to change the size of a static array (short of editing the source code and recompiling). In order to change at run time the size of a dynamic array, you must

1. allocate a different block of memory of the desired size,
2. copy all of the existing elements into the new memory,
3. use `delete` to deallocate the previous memory block to avoid a memory leak, and
4. reassign the original dynamic array pointer to point to new memory block.

Listing 11.19 (resizearray.cpp) contains the code to perform this array resizing. A vector object manages a dynamic array, and its `push_back` method takes care these details for us.

11.2.5 Multidimensional Arrays

Just as C++ supports higher-dimensional vectors, it also supports multidimensional arrays. The following statement:

```
int a[2][3];
```

declares `a` to be a two-dimensional (2D) array of integers. In this case, the declaration specifies that array `a` contains two rows and three columns. Figure 11.12 shows the logical structure of the array created by the following sequence of code:

```
int a[2][3];  // a is a 2D array
a[0][0] =  5;
a[0][1] = 19;
a[0][2] =  3;
a[1][0] = 22;
a[1][1] = -8;
a[1][2] = 10;
```

The two-dimensional array `a` is said to be a 2×3 array, meaning it has two rows and three columns (as shown in Figure 11.12). Rows are arranged horizontally, and the values in columns are arranged vertically. In each of the assignment statements above, for example

```
a[1][0] = 22;
```

the first index (here 1) signifies the row and the second index (here 0) denotes the column of the element within the array.

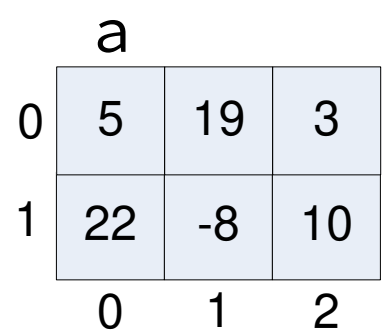
Using a syntax similar to 2D vectors, we could have declared and created the 2D array above as:

```
int a[2][3] = { {  5, 19,  3 },
                { 22, -8, 10 } };
```

You may omit the first index, as shown here:

```
int a[][3] = { {  5, 19,  3 },
               { 22, -8, 10 } };
```

Figure 11.12 A 2×3 two-dimensional array



For 2D arrays initialized in this manner the first subscript is optional, but the second subscript (that is, the size of each column) is required.

To access an element of a 2D array, use two subscripts:

```
a[r][c] = 4;                // Assign element at row r, column c
std::cout << a[m][n] << '\n';  // Display element at row m, column n
```

The following function prints the contents of a `ROWS` $\times$ `COLUMNS` 2D array of `double`s, where both `ROWS` and `COLUMNS` are constants:

```
void print(const double m[ROWS][COLUMNS]) {
    for (int row = 0; row < ROWS; row++) {
        for (int col = 0; col < COLUMNS; col++)
            std::cout << std::setw(5) << m[row][col];
        std::cout << '\n';
    }
}
```

We can omit the `ROW` size in the parameter declaration, but second set of square brackets must constant a constant integral expression. The declaration of the parameter `m` is quite complicated, and, as we did for 2D vectors, we can simplify the syntax by using a C++ type aliasing statement. statement

```
using Matrix = double[ROWS][COLUMNS];
```

defines a new type named `Matrix`.

Given the type alias `using` statement defining the new type name `Matrix`, we can express the parameter for the `print` function more simply:

```
void print(const Matrix m) {
    for (int row = 0; row < ROWS; row++) {
        for (int col = 0; col < COLUMNS; col++)
            std::cout << std::setw(5) << m[row][col];
        std::cout << '\n';
```

```
    }
}
```

Listing 11.20 (twodimarray.cpp) experiments with 2D arrays and takes advantage of the type alias statement to simplify the code.

Listing 11.20: twodimarray.cpp

```
#include <iostream>
#include <iomanip>

const int ROWS = 3,
          COLUMNS = 5;

// The name Matrix represents a new type
// that means a ROWS x COLUMNS
// two-dimensional array of double-precision
// floating-point numbers.
using Matrix = double[ROWS][COLUMNS];

// Allow the user to enter the elements of a matrix
void populate_matrix(Matrix m) {
    std::cout << "Enter the " << ROWS << " rows of the matrix.\n";
    for (int row = 0; row < ROWS; row++) {
        std::cout << "Row #" << row << " (enter " << COLUMNS << " values):";
        for (int col = 0; col < COLUMNS; col++)
            std::cin >> m[row][col];
    }
}

// We declare m constant because printing a matrix should not
// change it.
void print_matrix(const Matrix m) {
    for (int row = 0; row < ROWS; row++) {
        for (int col = 0; col < COLUMNS; col++)
            std::cout << std::setw(5) << m[row][col];
        std::cout << '\n';
    }
}

int main() {
    // Declare the 2D array
    Matrix mat;
    // Populate the array
    populate_matrix(mat);
    // Print the array
    print_matrix(mat);
}
```

An expression that uses just one index with a 2D array represents a single row within the 2D array. This row is itself a 1D array. Thus, if `a` is a 2D array and `i` is an integer, then the expression

```
a[i]
```

is a 1D array representing row `i`.

Figure 11.13 Physical layout of a C string

'H'	'o'	'w'	'd'	'y'	'!'	'\0'
0	1	2	3	4	5	6

As with vectors, C++ allows arrays with dimensions higher than two. Each "slice" of a 3D array is simply a 2D array, a 4D array is an array of 3D arrays, etc. For example, the statement

```
matrix[x][y][z][t] = 1.0034;
```

assigns 1.0034 to an element in a 4D array of `double`s. In practice, arrays with more than two dimensions are rare, but advanced scientific and engineering applications sometimes require higher-dimensional arrays.

11.2.6 C Strings

A string is a sequence of characters. C and C++ implement strings as arrays of `char`. The C++ language additionally supports string objects (see Section 13.1). In the C language, the only option is a `char` array. We use the term *C string* to refer to an array of characters as used in the C language. In this section, any mention of the term *string* refers to a C string.

A string is an array of characters. A string literal is a sequence of characters enclosed within quotation marks, as in

```
std::cout << "Howdy!\n";
```

All proper C strings are *null terminated*. This means the last character in the array is ASCII zero, which C++ represents by the character literal `'\0'`. Figure 11.13 shows the physical layout of the string `"Howdy!"` in memory.

Since strings are actually arrays, care must be taken when using string variables:

- Enough space must be reserved for number of characters in the string, including the null terminating character.
- The array of characters must be properly null terminated.

The following code fragment is safe and acceptable:

```
char *word = "Howdy!";
std::cout << word << '\n';
```

The variable `word` is declared to be a pointer to a character, and it is initialized to point to a string literal.

The following code fragment is less safe:

```
char word[256];
std::cin >> word;
```

The string `word` can hold 255 viable characters plus the null terminator. If the user types in relatively short words (length less than 255 characters), there is no problem. If at any time the user types in more characters than will fit in the `word` array, the executing program will have a problem. The problem is known as a *buffer overrun*. In the best case, buffer overruns lead to buggy programs. In the worst case, clever users can exploit buffer overruns to compromise software systems. Buffer overruns are always logic errors and you should take great care to avoid them.

The following code provides a safe way to get user input:

```
char word[10];
fgets(word, 10, stdin);
std::cout << word << '\n';
```

The `fgets` function is a standard C function. The second parameter specifies the maximum length of the string, including the terminating null character, that will be placed in the string `word`. The last argument, `stdin` is a C construct related to the C++ object `std::cin`. In order to use `fgets` within a program you must include the `<cstdio>` header.

The following code begs for disaster:

```
char *word;
std::cin >> word;
```

In this case `word` points to a random location in memory (it is uninitialized), and the code allocates no buffer to receive the input characters from `std::cin`. The program's behavior executing this code is undefined, but it likely will lead to the program crashing. Insidiously, depending on how the operating system manages memory, the program may run fine much of the time and crash only rarely. Regardless, the program contains a serious bug.

When passing an array to a function a caller must provide the size of the array so that the function may process the array properly. Since C strings are null terminated, such size information is not necessary. The `find_char` function in Listing 11.21 (findchar.cpp) determines if a particular character is present in a string.

Listing 11.21: `findchar.cpp`

```
#include <iostream>

bool find_char(const char *s, char ch) {
    while (*s != '\0')  { // Scan until we see the null character
        if (*s == ch)
            return true;   // Found the matching character
        s++;  // Advance to the next position within the string
    }
    return false;   // Not found
}

int main() {
    const char *phrase = "this is a phrase";
    // Try all the characters a through z
    for (char ch = 'a'; ch <= 'z'; ch++) {
        std::cout << '\'' << ch << '\''<< " is ";
        if (!find_char(phrase, ch))
            std::cout << "NOT ";
        std::cout << "in " << '\"' << phrase << '\"' << '\n';
    }
```

```
}
```

The output of Listing 11.21 (findchar.cpp) is

```
'a' is in "this is a phrase"
'b' is NOT in "this is a phrase"
'c' is NOT in "this is a phrase"
'd' is NOT in "this is a phrase"
'e' is in "this is a phrase"
'f' is NOT in "this is a phrase"
'g' is NOT in "this is a phrase"
'h' is in "this is a phrase"
'i' is in "this is a phrase"
'j' is NOT in "this is a phrase"
'k' is NOT in "this is a phrase"
'l' is NOT in "this is a phrase"
'm' is NOT in "this is a phrase"
'n' is NOT in "this is a phrase"
'o' is NOT in "this is a phrase"
'p' is in "this is a phrase"
'q' is NOT in "this is a phrase"
'r' is in "this is a phrase"
's' is in "this is a phrase"
't' is in "this is a phrase"
'u' is NOT in "this is a phrase"
'v' is NOT in "this is a phrase"
'w' is NOT in "this is a phrase"
'x' is NOT in "this is a phrase"
'y' is NOT in "this is a phrase"
'z' is NOT in "this is a phrase"
```

The `find_char` function in Listing 11.21 (findchar.cpp) uses pointer notation to traverse the string. It does not need to know in advance the number of characters in the string because it starts at the beginning and keeps scanning each character in turn until it finds the character it is looking for or encounters the null terminating character.

Recall from Section 5.1 that for Boolean conditions C++ treats a zero value as `false` and any non-zero value as `true`. Because of such a loose interpretation of Boolean expressions, we can write the `find_char` function from Listing 11.21 (findchar.cpp) more compactly as the following:

```
char find_char(const char *s, char ch) {
    // Scan until we see the null character or the character
    // we seek
    while (*s != '\0' && *s != ch)
        s++;  // Advance to the next position within the string
    return *s;   // Null character = false, any other is true
}
```

The only way out of the loop is to scan the null terminating character or the character sought. Here, if the loop encounters the null terminating character, it exits and returns that null character. The null character is simply ASCII zero—Boolean `false`. If the loop locates the sought character, it exits and returns that character which will not be ASCII zero. Any character except the null character has an ASCII value greater than zero; therefore, the calling code interprets the returned character as Boolean `true`.

Most routines that process strings depend on the strings to be null terminated in order to work properly. Some standard C string functions include

- `int strlen(const char *s)` returns the number of characters in string `s`, not including the null terminator.
- `char *strcpy(char *s, const char *t)` copies the contents of string `t` into string `s` up to and including the null terminator; `s` must point to a buffer large enough to hold all the characters of C string `t`.
- `char *strncpy(char *s, const char *t, unsigned n)` works like `strcpy` but copies a maximum of `n` characters; `s` must point to a buffer that can hold at least `n` characters.
- `int strcmp(const char *s, const char *t)` compares two strings for lexicographic (dictionary) ordering. The function returns an integer less than zero if `s` appears lexicographically before `t`. The function returns an integer greater than zero if `s` appears lexicographically after `t`. The function returns zero if the two strings are identical.

  The following code fragment

  ```
  std::cout << strcmp("ABC", "XYZ") << '\n';
  std::cout << strcmp("XYZ", "ABC") << '\n';
  std::cout << strcmp("ABC", "ABC") << '\n';
  ```

  prints

  ```
  -1
  1
  0
  ```

- `int strncmp(const char *s, const char *t, int n)` compares the first `n` characters of two strings for lexicographic (dictionary) ordering. The function returns an integer less than zero if the first `n` characters of `s` appear lexicographically before the first `n` characters of `t`; that is, `s` would appear before `t` is a dictionary. The function returns an integer greater than zero if the first `n` characters of `s` appear lexicographically after the first `n` characters of `t`. The function returns zero if the first `n` characters of the two strings are identical.

You should be familiar with C strings because C++ code can use C libraries, and C libraries often use C strings. Since C strings tend to be problematic, however, in most cases you should use the newer `string` objects (see Section 13.1) whenever possible.

11.2.7 Command-line Arguments

Some programs executed from the command line accept additional arguments. For example, in a Microsoft Windows command prompt the command

```
copy count.cpp count2.cpp
```

makes a copy of a file. The macOS and Linux equivalent would be

```
cp count.cpp count2.cpp
```

These filenames are called *command-line arguments*. They are provided in addition to the program's name. Command-line arguments allow the user to customize in some way the behavior of the program when launching it. In the example above, the user specifies which file to copy and the name of its copy. C++ programs that process command-line arguments do so via an array.

Listing 11.22 (cmdlineargs.cpp) is a program meant to be executed from the command line with extra arguments. It simply reports the extra information the user supplied.

Listing 11.22: cmdlineargs.cpp

```
#include <iostream>

int main(int argc, char *argv[]) {
    for (int i = 0; i < argc; i++)
        std::cout << '[' << argv[i] << "]\n";
}
```

The following shows one run of Listing 11.22 (cmdlineargs.cpp):

```
C:\Code>cmdlineargs -h 45 extra
[cmdlineargs]
[-h]
[45]
[extra]
```

The program did not print the first line shown in this program run. The command shell printed C:\Code> awaiting the user's command, and the user typed the remainder of the first line. In response, the program printed the remaining four lines. The parameter `argc` contains the number (count) of arguments on the command line typed by the user. The declaration `char *argv[]` indicates `argv` is an array of C strings. Notice that `argv[0]` is simply the name of the file containing the program. `argv[1]` is the string `"-h"`, `argv[2]` is the string `"45"`, and `argv[3]` is the string `"extra"`.

In Listing 11.23 (sqrtcmdline.cpp), the user supplies a range of integers on the command line. The program then prints all the integers in that range along with their square roots.

Listing 11.23: sqrtcmdline.cpp

```
#include <iostream>
#include <sstream>
#include <cmath>

int main(int argc, char *argv[]) {
    if (argc < 3)
        std::cout << "Supply range of values\n";
    else {
        int start, stop;
        std::stringstream st(argv[1]),
                          sp(argv[2]);
        st >> start;
        sp >> stop;
        for (int n = start; n <= stop; n++)
            std::cout << n << "  " << sqrt(n) << '\n';
    }
}
```

Feature	Vectors	Arrays
First element at index 0	Yes	Yes
Keeps track of its own size	Yes	No
Capacity expands automatically as needed	Yes	No
May be empty	Yes	No

Table 11.1: Vectors compared to arrays

Since the command-line arguments are strings, not integers, Listing 11.23 (sqrtcmdline.cpp) must convert the string parameters into their integer equivalents. The following shows some sample runs of Listing 11.23 (sqrtcmdline.cpp):

```
C:\Code>sqrtcmdline
Supply range of values

C:\Code>sqrtcmdline 2
Supply range of values

C:\Code>sqrtcmdline 2 10
2  1.41421
3  1.73205
4  2
5  2.23607
6  2.44949
7  2.64575
8  2.82843
9  3
10  3.16228
```

11.3 Vectors vs. Arrays

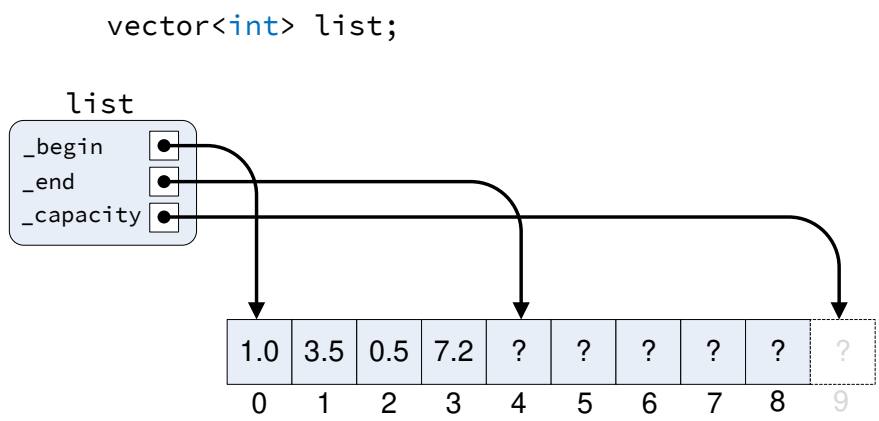
It is important to note is that arrays are not objects and, therefore, have no associated methods. The square bracket notation when used with arrays does not represent a special operator method. The square bracket array access notation is part of the core C++ language inherited from C. The creators of the C++ `vector` library designed vectors to behave as much as possible like primitive C arrays. Arrays have been part of the C language since its beginning, but vectors (added by C++) have adopted the syntactical features of arrays. Both arrays and vectors use square brackets for element access, and both locate their first element at index zero. Both provide access to a block of memory that can hold multiple elements. A vector object adds some additional capabilities and conveniences that make vectors the better choice for C++ developers. Table 11.1 summarizes many of the differences between vectors and arrays.

Vectors have a close association with arrays—a vector object is a thin wrapper around a dynamic array. A vector object simply manages several pointers that point to its dynamic array. Figure 11.14 provides a more accurate picture of the vector-array relationship.

The figure shows that a vector essentially maintains three pointers:

- The pointer labeled `_begin` points to the beginning of the block of memory allocated for the array that the vector manages. This corresponds to the location of the first element in the array.

Figure 11.14 A vector and its associated array.



- The pointer labeled `_end` points to the position in the array just past the last viable element in the vector.
- The `_capacity` pointer points to the memory location just past the block of memory allocated for the array.

The `_capacity` pointer is present because the array managed by a vector often will have more space allocated than the vector currently needs. An algorithm determines the amount of extra space needed to balance the demands of storage economy and fast `push_back` calls. When an executing program attempts to `push_back` an element onto a vector that has not reached its capacity, the operation is very fast; however, an attempt to `push_back` an element onto a vector that is at its capacity forces the run-time memory manager to do the following:

- allocate a large enough space elsewhere in memory to store the contents of the larger array, plus some extra room for future expansion,
- copy all the elements from the original array to the newly allocated array,
- add the new element onto the end of the newly allocated array,
- `delete` the memory held by the original array, and
- redirect the three pointers in the vector object to point to the appropriate places in the newly allocated array.

Resizing and copying the array is a relatively time-consuming process, especially as the size of the vector grows. The vector's capacity is tuned so that the average time to perform `push_back` is fast without the need to consume too much extra memory.

A vector adds value to a raw array by providing convenient methods for adding elements and resizing the array it manages. A vector keeps track of its own size. Arrays provide none of the convenient methods that vectors do. The overhead that a vector imposes over a raw dynamic array is negligible. You should prefer vectors over arrays when writing C++ code.

©2018 Richard L. Halterman

What if you are using vectors, but you need to use a C library that accepts only arrays? Fortunately it is easy to "unwrap" the array that the vector manages. If `vec` is a vector, the expression `&vec[0]` is the address of the first element within the vector. Given a function such as

```
void print(int a[], int n) {
    for (int i = 0; i < n; i++)
        std::cout << a[i] << ' ';
    std::cout << '\n';
}
```

that expects an array and its size, you can invoke it with your `vec` vector object as

```
print(&vec[0], vec.size());
```

Note that the first parameter is the starting address of the wrapped array, and the second parameter is the number of elements. Suppose instead you have a function that uses a pointer range, as in the following example:

```
void print(int *begin, int *end) {
    for (int *elem = begin; elem != end; elem++)
        std::cout << *elem << ' ';
    std::cout << '\n';
}
```

Here `begin` points to the beginning of an array, and `end` points to one position past the end of the array. You can invoke this function with your `vec` vector object as

```
print(&vec[0], &vec[0] + vec.size());
```

Note that the first parameter is the starting address of the wrapped array, and the second parameter uses pointer arithmetic to compute the memory address just past the end of the wrapped array.

Since the C++11 standard, the `std::vector` class has a method named `data` that returns a pointer to the first element of the array the vector manages. We can rewrite the above statement as

```
print(vec.data(), vec.data() + vec.size());
```

or better yet as

```
auto p = vec.data();
print(p, p + vec.size());
```

which avoids calling the `data` method twice.

As these examples show, you may use vector objects instead of arrays without any danger of not being able to use libraries that deal only with arrays.

If you happen to be using an array and need to use a function that expects a vector instead, it is easy to make a vector out of an existing array. If `arr` is a raw integer array containing `n` elements, the statement

```
std::vector<int> vec(arr, arr + n);
```

creates a new vector object `vec` with elements that are identical to `arr`. One disadvantage of this technique is that it copies all the elements in `arr` to a new dynamic array managed by the vector. This is because a vector object demands full control over the array it manages. A vector automatically frees up the memory of its array when it goes out of scope. Forcing a vector to manage a preexisting array could lead to problems in two ways:

- The code that created the array could later `delete` the contents of the array out from under the vector. Any attempted use of the vector after that point would be a problem.
- When the vector goes out of scope it automatically deallocates the array it manages. The code that originally created the array could afterward attempt to access the deallocated array with similar predictably disastrous results.

For these reasons the developers of the standard library designed the `std::vector` class to be in full control of its managed array. We thus cannot create a vector object from array created elsewhere without making a copy of the array.

We will see in Chapter 19 that it is relatively easy to write a function in a generic way so that it can accept and process either a vector or an array with equal efficiency.

Vectors are convenient data structures for working with dynamically-allocated arrays. C++ provides an object-oriented type equivalent to static C arrays. The `std::array` is the preferred way to represent statically-allocated arrays in C++.

In order to use the `std::array` class you must use the following preprocessor directive:

```
#include <array>
```

Instead of declaring a static array as

```
int arr[100];
```

you can write

```
std::array<int, 100> arr;
```

The first expression in the angle brackets specifies the type of elements held by the array, and the second expression specifies the array's statically-allocated size. The compiler must be able to compute the size; this means the size must be an integral literal or defined integral constant. Like a static C array, this size cannot be specified at run time. Also unlike `std::vector`s, the size of a `std::array` object remains fixed for the duration of the program's execution. Like a `std::vector`, however, a `std::array` object keeps track of its own size and supports reassignment. Listing 11.24 (stdarray.cpp) shows that we can determine the number of elements in a `std::array` object via its `size` method, just as we would with a `std::vector` object.

Listing 11.24: `stdarray.cpp`

```
#include <array>
#include <iostream>

int main() {
    std::array<int, 10> arr;
    std::cout << arr.size() << '\n';
}
```

Listing 11.24 (stdarray.cpp) prints

```
10
```

Because of the limitations of `std::array`s compared to `std::vector`s, the `std::array` class does not see as much use in C++ programs compared to the `std::vector` class.

Programmers must manage primitive arrays and associated pointers very carefully because array and pointer misuse is a common source of difficult to find and repair bugs within programs. Vectors effectively take care of much of the memory management problems that plague raw arrays. Additionally, vectors provide functionality and convenience that arrays cannot. As we have seen, it is easy to adapt a vector to a context that requires an array.

11.4 Prime Generation with a Vector

Listing 11.25 (fasterprimes.cpp) uses an algorithm developed by the Greek mathematician Eratosthenes who lived from 274 B.C. to 195 B.C. The principle behind the algorithm is simple: Make a list of all the integers two and larger. Two is a prime number, but any multiple of two cannot be a prime number (since a multiple of two has two as a factor). Go through the rest of the list and mark out all multiples of two (4, 6, 8, ...). Move to the next number in the list (in this case, three). If it is not marked out, it must be prime, so go through the rest of the list and mark out all multiples of that number (6, 9, 12, ...). Continue this process until you have listed all the primes you want.

Listing 11.25: `fasterprimes.cpp`

```
// File primesieve.cpp

#include <iostream>
#include <vector>
#include <ctime>

// Display the prime numbers between 2 and 500,000 and
// time how long it takes

// Largest potential prime considered
//const int MAX = 500000;
const int MAX = 500;

// Each position in the Boolean vector indicates
// if the number of that position is not prime:
// false means "prime," and true means "composite."
// Initially all numbers are prime until proven otherwise
std::vector<bool> nonprimes(MAX);  // Global vector initialized to all false

int main() {
    clock_t start_time = clock();   // Record start time

    // First prime number is 2; 0 and 1 are not prime
    nonprimes[0] = nonprimes[1] = true;

    // Start at the first prime number, 2.
    for (int i = 2; i < MAX; i++) {
        // See if i is prime
        if (!nonprimes[i])  {
            std::cout << i << " ";
            // It is prime, so eliminate all of its
            // multiples that cannot be prime
            for (int j = 2*i; j < MAX; j += i)
                nonprimes[j] = true;
        }
```

```
    }
    std::cout << '\n';  // Move cursor down to next line
    // Print the elapsed time
    std::cout << "Elapsed time: "
              << static_cast<double>(clock() - start_time)/CLOCKS_PER_SEC
              << " seconds\n";
}
```

Listing 11.26: fasterprimes2.cpp

```
#include <iostream>
#include <vector>
#include <ctime>

// Display the prime numbers between 2 and 500,000 and
// time how long it takes

// Largest potential prime considered
const int MAX = 2000000;

// Each position in the Boolean vector indicates
// if the number of that position is not prime:
// false means "prime," and true means "composite."
// Initially all numbers are prime until proven otherwise
std::vector<bool> nonprimes(MAX);  // Global vector initialized to all false

void count_primes1() {
    int count = 0;
    clock_t start_time = clock();   // Record start time
    for (int value = 2; value <= MAX; value++) {
        // See if value is prime
        bool is_prime = true;  // Provisionally, value is prime
        // Try all possible factors from 2 to the value - 1
        for (int trial_factor = 2;
              is_prime && trial_factor < value; trial_factor++)
            is_prime = (value % trial_factor != 0);
        if (is_prime)
            count++;  // Count the prime number
    }
    // Print the elapsed time
    std::cout << "Count = " << count << "  ";
    std::cout << "Elapsed time: "
              << static_cast<double>(clock() - start_time)/CLOCKS_PER_SEC
              << " seconds\n";
}

void count_primes2() {
    int count = 0;
    clock_t start_time = clock();   // Record start time
    for (int value = 2; value <= MAX; value++) {
        // See if value is prime
        bool is_prime = true;  // Provisionally, value is prime
        double r = value, root = sqrt(r);
        // Try all possible factors from 2 to the square
        // root of value
```

```
        for (int trial_factor = 2;
              is_prime && trial_factor <= root; trial_factor++)
            is_prime = (value % trial_factor != 0);
        if (is_prime)
            count++;  // Count the prime number
    }
    // Print the elapsed time
    std::cout << "Count = " << count << "  ";
    std::cout << "Elapsed time: "
              << static_cast<double>(clock() - start_time)/CLOCKS_PER_SEC
              << " seconds\n";
}

void count_primes3() {
    int count = 0;
    clock_t start_time = clock();   // Record start time

    // First prime number is 2; 0 and 1 are not prime
    nonprimes[0] = nonprimes[1] = true;

    // Start at the first prime number, 2.
    for (int i = 2; i < MAX; i++) {
        // See if i is prime
        if (!nonprimes[i])  {
            count++;  // It's prime, so count it
            // It is prime, so eliminate all of its
            // multiples that cannot be prime
            for (int j = 2*i; j < MAX; j += i)
                nonprimes[j] = true;
        }
    }
    // Print the elapsed time
    std::cout << "Count = " << count << "  ";
    std::cout << "Elapsed time: "
              << static_cast<double>(clock() - start_time)/CLOCKS_PER_SEC
              << " seconds\n";
}

int main() {
    count_primes1();
    count_primes2();
    count_primes3();
}
```

Recall Listing 8.8 (measureprimespeed.cpp), which also prints all the prime numbers up to 500,000. Using redirection (see Section 8.4), Listing 8.8 (measureprimespeed.cpp) takes 77 seconds. In comparison, Listing 11.25 (fasterprimes.cpp) takes about one second to perform the same task on the system. This is comparable to the square root version, Listing 8.5 (moreefficientprimes.cpp), which takes two seconds to run. If the goal is to print the prime numbers up to 1,000,000, the original version averages 271 seconds, or about four and one-half minutes. The square root version averages 4.5 seconds. The new, vector-based version averages 2 seconds. For 5,000,000 values the unoptimized original version takes a little over an hour and 33 minutes, the square root version takes a respectable 39 seconds, but vector version averages only 9 seconds to run.

Both the vector and array types represent linear sequences of elements. Vectors and arrays are convenient for storing collections of data, but they have some limitations. In Section 21.1 we will consider another kind of aggregate data structure called an *associative container*. Associative containers permit element access via a *key* rather than an index. Unlike an index, a key is not restricted to an integer expression. Associative containers are a better choice for some kinds of problems.

11.5 Exercises

1. Can you declare a vector to hold a mixture of `int`s and `double`s?

2. What happens if you attempt to access an element of a vector using a negative index?

3. Given the declaration

```
std::vector<int> list(100);
```

   (a) What expression represents the very first element of `list`?
   (b) What expression represents the very last element of `list`?
   (c) Write the code fragment that prints out the contents of `list`.
   (d) Is the expression `list[3.0]` legal or illegal?

4. Given the declarations

```
std::vector<int> list{2, 3, 1, 14, 4};
int x = 2;
```

   evaluate each of the following expressions:

   (a) `list[1]`
   (b) `list[x]`
   (c) `list.size()`
   (d) `list.empty()`
   (e) `list.at(3)`
   (f) `list[x] + 1`
   (g) `list[x + 1]`
   (h) `list[list[x]]`
   (i) `list[list.size() - 1]`

5. Is the following code fragment legal or illegal?

```
std::vector<int> list1(5), list2{ 3, 3, 3, 3, 3 };
list1 = list2;
```

6. Provide a single declaration statement that declares an integer vector named `list` that contains the values 45, −3, 16 and 8?

7. Does a vector keep track of the number of elements it contains?

8. Does an array keep track of the number of elements it contains?

9. Does the `std::array` class have more in common with a static array or a dynamic array?

10. Complete the following function that adds up all the *positive* values in an integer vector. For example, if vector `vec` contains the elements 3, -3, 5, 2, -1, and 2, the call `sum_positive(vec)` would evaluate to 12, since $3 + 5 + 2 + 2 = 12$. The function returns zero if the vector is empty. The function does not affect the contents of the vector.

```
int sum_positive(const std::vector<int>& v) {
    // Add your code...
}
```

11. Complete the following function that counts the even numbers in an integer vector. For example, if vector `vec` contains the elements 3, 5, 4, -1, and 0, the call `count_evens(vec)` would evaluate to 2, since the vector contains two even numbers: 4 and 0. The function returns zero if the vector is empty. The function does not affect the contents of the vector.

```
int count_evens(const std::vector<int>& v) {
    // Add your code...
}
```

12. Complete the following function that counts the even numbers in a 2D vector of integers.

```
int count_evens(const std::vector<std::vector<int>>& v) {
    // Add your code...
}
```

13. Complete the following function that compares two integer vectors to see if they contain exactly the same elements in exactly the same positions. The function returns true if the vectors are equal; otherwise, it returns false. For example, if vector `vec1` contains the elements 3, 5, 2, -1, and 2, and vector `vec2` contains the elements 3, 5, 2, -1, and 2, the call `equals(vec1, vec2)` would evaluate to true. If instead vector `vec2` contains the elements 3, 2, 5, -1, and 2, the call `equals(vec1, vec2)` would evaluate to false (the second and third elements are not in the same positions). Two vectors of unequal sizes cannot be equal. The function does not affect the contents of the vectors.

```
bool equals(const std::vector<int>& v1, const std::vector<int>& v2) {
    // Add your code...
}
```

14. Complete the following function that determines if all the elements in one vector also appear in another. The function returns true if all the elements in the second vector also appear in the first; otherwise, it returns false. For example, if vector `vec1` contains the elements 3, 5, 2, -1, 7, and 2, and vector `vec2` contains the elements 5, 7, and 2, the call `contains(vec1, vec2)` would evaluate to true. If instead vector `vec2` contains the elements 3, 8, -1, and 2, the call `contains(vec1, vec2)` would evaluate to false (8 does not appear in the first vector). Also If vector `vec2` contains the elements 5, 7, 2, and 5, the call `contains(vec1, vec2)` would evaluate to false (5 appears twice in `vec2` but only once in `vec1`, so `vec1` does not contain all the elements that appear in `vec2`). The function does not affect the contents of the vectors.

```
bool contains(const std::vector<int>& v1,
              const std::vector<int>& v2) {
    // Add your code...
}
```

15. Suppose your task is to implement the function with the prototype

```
void proc(std::vector<int> v);
```

When you implement the body of `proc`, how can you determine the size of vector `v`?

16. Consider the declaration

```
std::vector<std::vector<int>> collection(100, std::vector<int>(200));
```

(a) What does the expression `collection[15][29]` represent?

(b) How many elements does `collection` hold?

(c) Write the C++ code that prints all the elements in `collection`. All the elements in the same row should appear on the same line, and but each successive row should appear on its own line.

(d) What does the expression `collection[15]` represent?

17. Consider the declaration

```
std::vector<std::vector<std::vector<std::vector<int>>>>
    mesh(100, std::vector<int>(200, std::vector<int>(100,
                                        std::vector<int>(50))));
```

How many elements does `mesh` hold?

18. How is a C++ array different from a vector?

19. What advantages does a C++ vector provide over an array?

20. Provide the statement(s) that declare and create a static array named `a` that can hold 20 integers.

21. Provide the statement(s) that declare and ceate a dynamic array named `a` that can hold 20 integers.

22. What extra attention does a programmer need to give to a static array when its use within a program is finished?

23. What extra attention does a programmer need to give to a dynamic array when its use within a program is finished?

24. What extra attention does a programmer need to give to a vector array when its use within a program is finished?

25. Consider the following function that processes the elements of an array using a range:

```
bool proc(const int *begin, const int *end) {
    // Details omitted . . .
}
```

and an array and vector declared as shown here:

```
int a[10];
std::vector<int> v;
```

(a) Provide the statement that correctly calls `proc` with array `a`.

(b) Provide the statement that correctly calls `proc` with vector `v`.

26. Can you declare an array to hold a mixture of `int`s and `double`s?

27. What happens if you attempt to access an element of an array using a negative index?

28. Given the declaration

```
int list[100];
```

    (a) What expression represents the very first element of `list`?
    (b) What expression represents the very last element of `list`?
    (c) Write the code fragment that prints out the contents of `list`.
    (d) Is the expression `list[3.0]` legal or illegal?

29. Is the following code fragment legal or illegal?

```
int list1[5], list2[5] = { 3, 3, 3, 3, 3 };
list1 = list2;
```

30. Provide a single declaration statement that declares an integer array named `list` that contains the values 45, −3, 16 and 8?

31. Does an array keep track of the number of elements it contains?

32. Complete the following function that adds up all the *positive* values in an array of integers. For example, if array `arr` contains the elements 3, -3, 5, 2, −1, and 2, the call `sum_positive(arr)` would evaluate to 12, since 3 + 5 + 2 + 2 = 12. The function returns zero if the array is empty (that is, n < 1).

```
// Array a with length n
int sum_positive(const int *a, int n) {
    // Add your code...
}
```

33. Complete the following function that sums the even numbers in an array of integers. For example, if array `arr` contains the elements 3, 5, 2, −1, and 2, the call `sum_evens(arr)` would evaluate to 4, since 2 + 2 = 4. The function returns zero if the array is empty (that is, n < 1). The function does not affect the contents of the array.

```
// Array a with length n
int sum_evens(const int *a, int n) {
    // Add your code...
}
```

34. Suppose your task is to implement the function with the prototype

```
void proc(int a[]);
```

    When you implement the body of `proc`, how can you determine the size of array `a`?

35. Consider the declaration

```
int collection[100][200];
```

    What does the expression `collection[15][29]` represent?

36. Consider the declaration

```
int collection[100][200];
```

How many elements does `collection` hold?

37. Consider the declaration

```
int collection[100][200];
```

Write the C++ code that prints all the elements in `collection`. All the elements in the same row should appear on the same line, and but each successive row should appear on its own line.

38. Consider the declaration

```
int collection[100][200];
```

What does the expression `collection[15]` represent?

39. Consider the declaration

```
int mesh[100][200][100][50];
```

How many elements does `mesh` hold?

40. Rewrite the following expressions using pointer notation instead of array notation.

    (a) `a[4]`
    (b) `a[1]`
    (c) `a[0]`

41. Rewrite the following expressions using array notation instead of pointer notation.

    (a) `*(a + 3)`
    (b) `*a`
    (c) `*(a + 0)`

42. Rewrite the following code fragment using array notation instead of pointer notation:

```
void display(int *a, int n) {
    while (n) {
        std::cout << *a << " ";
        a++;
        n--;
    }
    std::cout << '\n';
}
```

43. Rewrite the following code fragment using pointer notation instead of array notation:

```
int sum(int *a, int n) {
    int s = 0;
    for (int i = 0; i < n; i++)
        s += a[i];
    return s;
}
```

44. Consider the following declaration:

```
char *word = "abcde";
```

Evaluate each of the following expressions. If an expression indicates undefined behavior, write *UB*.

(a) `word[1]`

(b) `*word`

(c) `word[5]`

45. Suppose your task is to implement the function with the prototype

```
void proc(char *s);
```

where `s` is a C string. When you implement the body of `proc`, how can you determine the length of string `s`?

46. Given the following declarations which appear in a function body:

```
double nums[100], *grid = new double[100];
```

(a) Where will the elements of the `nums` array live—static memory, the stack, or the heap?

(b) Where will the elements of the `grid` array live—static memory, the stack, or the heap?

47. What operator should eventually be used when the `new` operator is used to allocate memory? What is the consequence of its omission?

48. List some common errors programmers make when dealing with dynamic memory.

49. Complete the following function that counts the number of negative values in a 10×10 integer 2D array.

```
int count_negatives(int a[10][10]) {
    // Add your code...
}
```

Chapter 12

Sorting and Searching

Chapters 11 introduced the fundamentals of making and using vectors. In this chapter we explore some common algorithms for ordering elements within a vector and for locating elements by their value rather than by their index.

12.1 Sorting

We will use the generic term *sequence* to refer to either a vector or an array. Sorting—arranging the elements within a sequence into a particular order—is a common activity. For example, a sequence of integers may be arranged in ascending order (that is, from smallest to largest). A sequence of words (strings) may be arranged in lexicographical (commonly called alphabetic) order. Many sorting algorithms exist, and some perform much better than others. We will consider one sorting algorithm that is relatively easy to implement.

The selection sort algorithm is relatively easy to implement, and it performs acceptably for smaller sequences. If A is a sequence, and i and j represent indices within the sequence, selection sort works as follows:

1. Set $i = 0$.
2. Examine all the elements $A[j]$, where $j > i$. If any of these elements is less than $A[i]$, then exchange $A[i]$ with the smallest of these elements. (This ensures that all elements after position i are greater than or equal to $A[i]$.)
3. If i is less than the length of A, increase i by 1 and goto Step 2.

If the condition in Step 3 is not met, the algorithm terminates with a sorted sequence. The command to "goto Step 2" in Step 3 represents a loop. We can begin to translate the above description into C++ as follows:

```
// n is A's length
for (int i = 0; i < n - 1; i++) {
    // Examine all the elements
    // A[j], where j > i.
    // If any of these A[j] is less than A[i],
```

©2018 Richard L. Halterman

```
    // then exchange A[i] with the smallest of these elements.
}
```

The directive at Step 2 beginning with "Examine all the elements A[j], where $j > i$" also requires a loop. We continue refining our implementation with:

```
// n is A's length
for (int i = 0; i < n - 1; i++) {
    for (int j = i + 1; j < n; j++) {
        // Examine all the elements
        // A[j], where j > i.
    }
    // If any A[j] is less than A[i],
    // then exchange A[i] with the smallest of these elements.
}
```

In order to determine if any of the elements is less than `A[i]`, we introduce a new variable named `small`. The purpose of `small` is to keep track of the position of the smallest element found so far. We will set `small` equal to `i` initially because we wish to locate any element less than the element found at position `i`.

```
// n is A's length
for (int i = 0; i < n - 1; i++) {
    // small is the position of the smallest value we've seen
    // so far; we use it to find the smallest value less than A[i]
    int small = i;
    for (int j = i + 1; j < n; j++) {
        if (A[j] < A[small])
            small = j;  // Found a smaller element, update small
    }
    // If small changed, we found an element  smaller than A[i]
    if (small != i)
        // exchange A[small] and A[i]
}
```

Listing 12.1 (sortintegers.cpp) provides the complete C++ implementation of the `selection_sort` function within a program that tests it out.

Listing 12.1: `sortintegers.cpp`

```
#include <iostream>
#include <vector>

/*
 *  swap(a, b)
 *     Interchanges the values of memory
 *     referenced by its parameters a and b.
 *     It effectively interchanges the values
 *     of variables in the caller's context.
 */
void swap(int& a, int& b) {
   int temp = a;
   a = b;
   b = temp;
```

```
}

/*
 *   selection_sort
 *       Arranges the elements of vector a into ascending order.
 *       a is a vector that contains integers.
 */
void selection_sort(std::vector<int>& a) {
    int n = a.size();
    for (int i = 0;  i < n - 1;  i++) {
        // Note: i,small, and j represent positions within a
        // a[i], a[small], and a[j] represents the elements at
        // those positions.
        // small is the position of the smallest value we've seen
        // so far; we use it to find the smallest value less
        // than a[i]
        int small = i;
        // See if a smaller value can be found later in the vector
        for (int j = i + 1;  j < n;  j++)
            if (a[j] < a[small])
                small = j;  // Found a smaller value
        // Swap a[i] and a[small], if a smaller value was found
        if (i != small)
            swap(a[i], a[small]);
    }
}

/*
 *  print
 *     Prints the contents of a vector of integers.
 *     a is the vector to print.
 *     a is not modified.
 */
void print(const std::vector<int>& a) {
    int n = a.size();
    std::cout << '{';
    if (n > 0) {
        std::cout << a[0];  // Print the first element
        for (int i = 1; i < n; i++)
            std::cout << ',' << a[i];  // Print the rest
    }
    std::cout << '}';
}


int main() {
   std::vector<int> list{23, -3, 4, 215, 0, -3, 2, 23, 100, 88, -10};
   std::cout << "Before: ";
   print(list);
   std::cout << '\n';
   selection_sort(list);
   std::cout << "After: ";
   print(list);
   std::cout << '\n';
}
```

Listing 12.1 (sortintegers.cpp) uses a fancier `print` routine, separating the vector's elements with commas. The program's output is

```
Before: {23,-3,4,215,0,-3,2,23,100,88,-10}
After: {-10,-3,-3,0,2,4,23,23,88,100,215}
```

We really do not need to write our own function to interchange the values of two integers as shown in Listing 12.1 (sortintegers.cpp). The C++ standard library includes `std::swap` that works just like the `swap` function in Listing 12.1 (sortintegers.cpp); therefore, if you remove our custom `swap` definition in Listing 12.1 (sortintegers.cpp) and replace the call to `swap` with a call to `std::swap`, the program will work just as well.

12.2 Flexible Sorting

What if we want to change the behavior of the sorting function in Listing 12.1 (sortintegers.cpp) so that it arranges the elements in descending order instead of ascending order? It is actually an easy modification; simply change the line

```
if (a[j] < a[small])
```

to be

```
if (a[j] > a[small])
```

Suppose we want to change the sort so that it sorts the elements in ascending order except that all the even numbers in the vector appear before all the odd numbers? This would take a little more effort, but it still is possible to do.

The next question is more intriguing: How can we rewrite the `selection_sort` function so that, by passing an additional parameter, it can sort the vector in any way we want?

We can make our sort function more flexible by making it higher-order function (see Section 10.10) that accepts an ordering function as a parameter. Listing 12.2 (flexibleintsort.cpp) arranges the elements in a vector two different ways using the same `selection_sort` function.

Listing 12.2: flexibleintsort.cpp

```
#include <iostream>
#include <vector>

/*
 *  less_than(a, b)
 *     Returns true if a < b; otherwise, returns
 *     false.
 */
bool less_than(int a, int b) {
    return a < b;
}

/*
 *  greater_than(a, b)
```

```
 *     Returns true if a > b; otherwise, returns
 *     false.
 */
bool greater_than(int a, int b) {
    return a > b;
}

/*
 *   selection_sort(a, compare)
 *       Arranges the elements of a in an order determined
 *       by the compare function.
 *       a is a vector of integers.
 *       compare is a function that compares the ordering of
 *            two integers.
 */
void selection_sort(std::vector<int>& a, bool (*compare)(int, int)) {
    int n = a.size();
    for (int i = 0;  i < n - 1;  i++) {
        // Note: i,small, and j represent positions within a
        // a[i], a[small], and a[j] represents the elements at
        // those positions.
        // small is the position of the smallest value we've seen
        // so far; we use it to find the smallest value less
        // than a[i]
        int small = i;
        // See if a smaller value can be found later in the vector
        for (int j = i + 1;  j < n;  j++)
            if (compare(a[j], a[small]))
                small = j;  // Found a smaller value
        // Swap a[i] and a[small], if a smaller value was found
        if (i != small)
            std::swap(a[i], a[small]);  // Uses std::swap
    }
}

/*
 *  print
 *     Prints the contents of an integer vector
 *     a is the vector to print.
 *     a is not modified.
 */
void print(const std::vector<int>& a) {
    int n = a.size();
    std::cout << '{';
    if (n > 0) {
        std::cout << a[0];  // Print the first element
        for (int i = 1; i < n; i++)
            std::cout << ',' << a[i];  // Print the rest
    }
    std::cout << '}';
}


int main() {
   std::vector<int> list{ 23, -3, 4, 215, 0, -3, 2, 23, 100, 88, -10 };
```

```
    std::cout << "Original:   ";
    print(list);
    std::cout << '\n';
    selection_sort(list, less_than);
    std::cout << "Ascending:  ";
    print(list);
    std::cout << '\n';
    selection_sort(list, greater_than);
    std::cout << "Descending: ";
    print(list);
    std::cout << '\n';
}
```

Listing 12.2 (flexibleintsort.cpp) takes advantage of the standard `swap` function (see Section 12.1).

The output of Listing 12.2 (flexibleintsort.cpp) is

```
Original:   {23,-3,4,215,0,-3,2,23,100,88,-10}
Ascending:  {-10,-3,-3,0,2,4,23,23,88,100,215}
Descending: {215,100,88,23,23,4,2,0,-3,-3,-10}
```

The comparison function passed to the sort routine customizes the sort's behavior. The basic structure of the sorting algorithm does not change, but its notion of ordering is adjustable. If the second parameter to `selection_sort` is `less_than`, the sort routine arranges the elements into ascending order. If the caller passes `greater_than` instead, `selection_sort` rearranges vector's elements into descending order. More creative orderings are possible with more elaborate comparison functions.

Selection sort is a relatively efficient simple sort, but more advanced sorts are, on average, much faster than selection sort, especially for large data sets. One such general purpose sort is *Quicksort*, devised by C. A. R. Hoare in 1962. Quicksort is the fastest known general purpose sort. Since sorting is a common data processing activity, the standard C library provides a function named `qsort` that implements Quicksort. More information about Quicksort and `qsort` is available at `http://en.wikipedia.org/wiki/Quicksort`.

12.3 Search

Searching a vector for a particular element is a common activity. We will consider the two most common search strategies: linear search and binary search.

12.3.1 Linear Search

Listing 12.3 (linearsearch.cpp) uses a function named `locate` that returns the position of the first occurrence of a given element in a vector of integers; if the element is not present, the function returns -1.

Listing 12.3: `linearsearch.cpp`

```
#include <iostream>
#include <vector>
#include <iomanip>

/*
```

```
 *   locate(a, seek)
 *       Returns the index of element seek in vector a.
 *       Returns -1 if seek is not an element of a.
 *       a is the vector to search.
 *       seek is the element to find.
 */
int locate(const std::vector<int>& a, int seek) {
    int n = a.size();
    for (int i = 0;  i < n;  i++)
        if (a[i] == seek)
            return i;   // Return position immediately
    return -1;  // Element not found
}

/*
 *   format(i)
 *       Prints integer i right justified in a 4-space
 *       field.  Prints "****" if i > 9,999.
 */
void format(int i) {
    if (i > 9999)
        std::cout << "****" << '\n';  // Too big!
    else
        std::cout << std::setw(4) << i;
}


/*
 *  print(v)
 *     Prints the contents of an int vector.
 *     v is the vector to print.
 */
void print(const std::vector<int>& v) {
    for (int  i : v)
        format(i);
}


/*
 *    display(a, value)
 *      Draws an ASCII art arrow showing where
 *      the given value is within the vector.
 *      a is the vector.
 *      value is the element to locate.
 */
void display(const std::vector<int>& a, int value) {
    int position = locate(a, value);
    if (position >= 0) {
        print(a);                   // Print contents of the vector
        std::cout << '\n';
        position = 4*position + 7;  // Compute spacing for arrow
        std::cout << std::setw(position);
        std::cout << "   ^   " << '\n';
        std::cout << std::setw(position);
        std::cout << "   |   " << '\n';
```

```
        std::cout << std::setw(position);
        std::cout << "   +-- " << value << '\n';
    }
    else {
        std::cout << value << " not in ";
        print(a);
        std::cout << '\n';
    }
    std::cout << "======" << '\n';
}

int main() {
   std::vector<int> list{ 100, 44, 2, 80, 5, 13, 11, 2, 110 };
   display(list, 13);
   display(list, 2);
   display(list, 7);
   display(list, 100);
   display(list, 110);
}
```

The output of Listing 12.3 (linearsearch.cpp) is

```
 100  44   2  80   5  13  11   2 110
                      ^
                      |
                      +-- 13
======
 100  44   2  80   5  13  11   2 110
          ^
          |
          +-- 2
======
7 not in  100  44   2  80   5  13  11   2 110
======
 100  44   2  80   5  13  11   2 110
   ^
   |
   +-- 100
======
 100  44   2  80   5  13  11   2 110
                                  ^
                                  |
                                  +-- 110
======
```

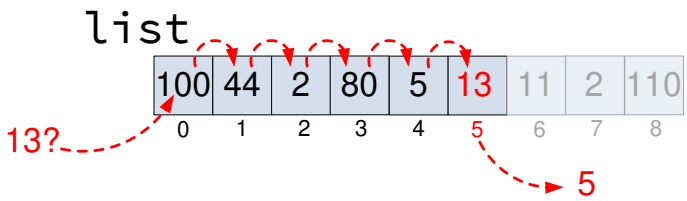
The key function in Listing 12.3 (linearsearch.cpp) is `locate`; all the other functions simply lead to a more interesting display of `locate`'s results. If `locate` finds a match, it immediately returns the position of the matching element; otherwise, if `locate` considers all the elements of the vector and finds no match, it returns -1. -1 is a good indication of failure, since -1 is not a valid index in a C++ vector.

The other functions are

- `format` prints an integer right justified within a four-space field. Extra spaces pad the beginning of the number if necessary.

Figure 12.1 Linear search. The grayed out elements are not considered during the search process.

```
int list[] = { 100, 44, 2, 80, 5, 13, 11, 2, 110 };
int x = locate(list, 9, 13);
```



- `print` prints out the elements in any vector using the `format` function to properly format the values. This alignment simplifies the `display` function.
- `display` uses `locate`, `print`, and `setw` to provide a graphical display of the operation of the `locate` function.

The kind of search performed by `locate` is known as *linear search*, since the process takes a straight line path from the beginning to the end of the vector, and it considers each element in order. Figure 12.1 illustrates linear search.

12.3.2 Binary Search

Linear search is acceptable for relatively small vectors, but the process of examining each element in a large vector is time consuming. An alternative to linear search is *binary search*. In order to perform binary search a vector's elements must be in sorted order. Binary search exploits this sorted structure of the vector using a clever but simple strategy that quickly zeros in on the element to find:

1. If the vector is empty, return -1.
2. Check the element in the middle of the vector. If the element is what you are seeking, return its position. If the middle element is larger than the element you are seeking, perform a binary search on the first half of the vector. If the middle element is smaller than the element you are seeking, perform a binary search on the second half of the vector.

This approach is analogous to looking for a telephone number in the phone book in this manner:

1. Open the book at its center. If the name of the person is on one of the two visible pages, look at the phone number.
2. If not, and the person's last name is alphabetically less the names on the visible pages, apply the search to the left half of the open book; otherwise, apply the search to the right half of the open book.
3. Discontinue the search with failure if the person's name should be on one of the two visible pages but is not present.

Listing 12.4 (binarysearch.cpp) contains a C++ function that implements the binary search algorithm.

Listing 12.4: binarysearch.cpp

```
#include <iostream>
#include <vector>
#include <iomanip>

/*
 *   binary_search(a, seek)
 *       Returns the index of element seek in vector a;
 *       returns -1 if seek is not an element of a
 *       a is the vector to search; a's contents must be
 *       sorted in ascending order.
 *       seek is the element to find.
 */
int binary_search(const std::vector<int>& a, int seek) {
    int first = 0,            // Initially the first position
        last = a.size() - 1,  // Initially the last position
        mid;                  // The middle of the vector
    while (first <= last) {
        mid = first + (last - first + 1)/2;
        if (a[mid] == seek)
            return mid;       // Found it
        else if (a[mid] > seek)
            last = mid - 1;   // continue with 1st half
        else  // a[mid] < seek
            first = mid + 1;  // continue with 2nd half
    }
    return -1;    // Not there
}

/*
 *   format(i)
 *       Prints integer i right justified in a 4-space
 *       field.  Prints "****" if i > 9,999.
 */
void format(int i) {
    if (i > 9999)
        std::cout << "****\n";  // Too big!
    else
        std::cout << std::setw(4) << i;
}


/*
 *  print(v)
 *     Prints the contents of an int vector.
 *     v is the vector to print.
 */
void print(const std::vector<int>& v) {
    for (int  i : v)
        format(i);
}
```

```
/*
 *    display(a, value)
 *      Draws an ASCII art arrow showing where
 *      the given value is within the vector.
 *      a is the vector.
 *      value is the element to locate.
 */
void display(const std::vector<int>& a, int value) {
    int position = binary_search(a, value);
    if (position >= 0) {
        print(a);                   // Print contents of the vector
        std::cout << '\n';
        position = 4*position + 7;  // Compute spacing for arrow
        std::cout << std::setw(position);
        std::cout << "   ^   " << '\n';
        std::cout << std::setw(position);
        std::cout << "   |   " << '\n';
        std::cout << std::setw(position);
        std::cout << "   +-- " << value << '\n';
    }
    else {
        std::cout << value << " not in ";
        print(a);
        std::cout << '\n';
    }
    std::cout << "======" << '\n';
}

int main() {
   // Check binary search on even- and odd-length vectors and
   // an empty vector
   std::vector<int> even_list{ 1, 2, 3, 4, 5, 6, 7, 8 },
               odd_list{ 1, 2, 3, 4, 5, 6, 7, 8, 9 },
               empty_list;

   for (int i = -1;  i <= 10; i++)
       display(even_list, i);
   for (int i = -1;  i <= 10; i++)
       display(odd_list, i);
   for (int i = -1;  i <= 10; i++)
       display(empty_list, i);
}
```

In the `binary_search` function:

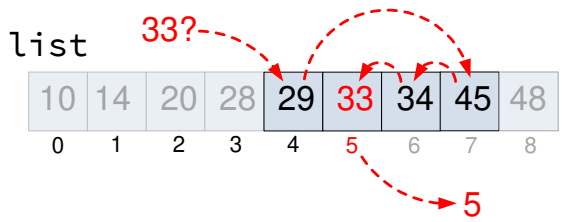
- The initializations of `first` and `last`:

  ```
  int first = 0,            // Initially the first position
      last = a.size() - 1,  // Initially the last position
  ```

  ensure that `first` is less than or equal to `last` for a nonempty vector. If the vector is empty, `first` is zero, and `last` is equal to `n` $-$ 1 which equals -1. So in the case of an empty vector `binary_search` will skip the loop body and immediately return -1. This is correct behavior because an empty vector cannot possibly contain any item we seek.

Figure 12.2 Binary search. The grayed out elements are not considered during the search.

```
std::vector<int> list = { 10, 14, 20, 28, 29, 33,
                          34, 45, 48 };
int x = binary_search(list, 33);
```



- The variable `mid` represents the midpoint value between `first` and `last`, and it is computed in the statement

  ```
  mid = first + (last - first + 1)/2;
  ```

  This arithmetic may look a bit complex. The more straightforward way to compute the average of two values would be

  ```
  mid = (first + last)/2;  // Alternate calculation
  ```

  The problem with this method of computing the midpoint is it fails to produce the correct results more times than the version found in Listing 12.4 (binarysearch.cpp). The problem arises if the vector is large, and `first` and `last` both have relatively large values. The expression `first + last` can overflow the range of integers producing a meaningless result. The subsequent division by two is too late to help. The result of the expression `first + last` would overflow the range of integers more often than the expression `first + (last - first + 1)/2` because `(last - first + 1)/2` would be a much smaller value to add to `first`. If you are thinking "I would never have thought of that problem," don't worry too much. A large number of competent, professional software engineers have fallen prey to this oversight (see `http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html`).

  The calculation of `mid` ensures that `first` $\leq$ `mid` $\leq$ `last`.

- If `mid` is the location of the sought element (checked in the first `if` statement), the loop terminates, and returns the correct position.

- The second `if` statement ensures that either `last` decreases or `first` increases each time through the loop. Thus, if the loop does not terminate for other reasons, eventually `first` will be larger than `last`, and the loop will terminate. If the loop terminates for this reason, the function returns -1. This is the correct behavior.

- The second `if` statement excludes the irrelevant elements from further search. The number of elements remaining to consider is effectively cut in half.

Figure 12.2 illustrates how binary search works.

The implementation of the binary search algorithm is more complicated than the simpler linear search algorithm. Ordinarily simpler is better, but for algorithms that process data structures that potentially hold large amounts of data, more complex algorithms employing clever tricks that exploit the structure of the data (as binary search does) often dramatically outperform simpler, easier-to-code algorithms.

For a fair comparison of linear vs. binary search, suppose we want to locate an element in a sorted vector. That the vector is ordered is essential for binary search, but it can be helpful for linear search as well. The revised linear search algorithm is

```
// This version requires vector a to be sorted in
// ascending order.
int linear_search(const std::vector<int>& a, int seek) {
    int n = a.size();
    for (int i = 0;  i < n && a[i] <= seek;  i++)
        if (a[i] == seek)
            return i;   // Return position immediately
    return -1;  // Element not found
}
```

Notice that, as in the original version of linear search, the loop will terminate when all the elements have been examined, but it also will terminate early when it encounters an element larger than the sought element. Since the vector is sorted, there is no need to continue the search once you have begun seeing elements larger than your sought value; `seek` cannot appear after a larger element in a sorted vector.

Suppose a vector to search contains n elements. In the worst case—looking for an element larger than any currently in the vector—the loop in linear search takes n iterations. In the best case—looking for an element smaller than any currently in the vector—the function immediately returns without considering any other elements. The number of loop iterations thus ranges from 1 to n, and so on average linear search requires $\frac{n}{2}$ comparisons before the loop finishes and the function returns.

Now consider binary search of a vector that contains n elements. After each comparison the size of the vector left to consider is one-half the original size. If the sought item is not found on the first probe, the number of remaining elements to search is $\frac{n}{2}$. After the next time through the loop, the number of elements left to consider is one-half of $\frac{n}{2}$, or $\frac{n}{4}$. After the third iteration, search space in the vector drops to one-half of $\frac{n}{4}$, which is $\frac{n}{8}$. This process of cutting the search space in half continues each time through the loop until the process locates the sought element or runs out of elements to consider. The problem of determining how many times a set of things can be divided in half until only one element remains can be solved with a base-2 logarithm. For binary search, the worst case scenario of not finding the sought element requires the loop to make $\log_2 n$ iterations.

How does this analysis help us determine which search is better? The quality of an algorithm is judged by two key characteristics:

- How much time (processor cycles) does it take to run?
- How much space (memory) does it take to run?

In our situation, both search algorithms process the sequence with only a few extra local variables, so for large sequences they both require essentially the same space. The big difference here is speed. Binary search performs more elaborate computations each time through the loop, and each operation takes time, so perhaps binary search is slower. Linear search is simpler (fewer operations through the loop), but perhaps its loop executes many more times than the loop in binary search, so overall it is slower.

We can deduce the faster algorithm in two ways: empirically and analytically. An empirical test is an experiment; we carefully implement both algorithms and then measure their execution times. The analyt-

ical approach analyzes the source code to determine how many operations the computer's processor must perform to run the program on a problem of a particular size.

Listing 12.5 (searchcompare.cpp) measures the running times of the two kinds of searches to compare the two algorithms empirically.

Listing 12.5: searchcompare.cpp

```
#include <iostream>
#include <iomanip>
#include <vector>
#include <ctime>

/*
 *   binary_search(a, seek)
 *       Returns the index of element seek in vector a;
 *       returns -1 if seek is not an element of a
 *       a is the vector to search; a's contents must be
 *       sorted in ascending order.
 *       seek is the element to find.
 */
int binary_search(const std::vector<int>& a, int seek) {
    int n = a.size();
    int first = 0,      // Initially the first element in vector
        last = n - 1,   // Initially the last element in vector
        mid;            // The middle of the vector
    while (first <= last) {
        mid = first + (last - first + 1)/2;
        if (a[mid] == seek)
            return mid;       // Found it
        else if (a[mid] > seek)
            last = mid - 1;   // continue with 1st half
        else  // a[mid] < seek
            first = mid + 1;  // continue with 2nd half
    }
    return -1;    // Not there
}


/*
 *   linear_search(a, seek)
 *       Returns the index of element seek in vector a.
 *       Returns -1 if seek is not an element of a.
 *       a is the vector to search.
 *       seek is the element to find.
 *       This version requires vector a to be sorted in
 *       ascending order.
 */
int linear_search(const std::vector<int>& a, int seek) {
    int n = a.size();
    for (int i = 0;  i < n && a[i] <= seek;  i++)
        if (a[i] == seek)
            return i;   // Return position immediately
    return -1;  // Element not found
}
```

```
/*
 *   Tests the execution speed of a given search function on a
 *   vector.
 *   search - the search function to test
 *   v      - the vector to search
 *   trials - the number of trial runs to average
 *   Returns the elapsed time in seconds
 *   The C++ chrono library defines the types
 *   system_clock::time_point and microseconds.
 */
double time_execution(int (*search)(const std::vector<int>&, int),
                      std::vector<int>& v, int trials) {
    int n = v.size();
    // Average the time over a specified number of trials
    double elapsed = 0.0;
    for (int iters = 0; iters < trials; iters++ ) {
        clock_t start_time = clock();  // Start the timer
        for (int i = 0; i < n; i++)    // Search for all elements
            search(v, i);
        clock_t end_time = clock();    // Stop the timer
        elapsed += static_cast<double>(end_time - start_time)/CLOCKS_PER_SEC;
    }
    return elapsed/trials;  // Mean elapsed time per run
}

int main() {
    std::cout << "---------------------------------------\n";
    std::cout << "  Vector        Linear         Binary\n";
    std::cout << "   Size         Search         Search\n";
    std::cout << "---------------------------------------\n";

    // Test the sorts on vectors with 1,000 elements up to
    // 10,000 elements.
    for (int size = 0; size <= 50000; size += 5000) {
        std::vector<int> list(size);  // Make a vector of the appropiate size

        // Ensure the elements are ordered low to high
        for (int i = 0; i < size; i++)
            list[i] = i;

        std::cout << std::setw(7) << size;
        // Search for all the elements in list using linear search
        // Compute running time averaged over five runs
        std::cout << std::fixed << std::setprecision(3) << std::setw(12)
                  << time_execution(linear_search, list, 5)
                  << " sec";
        // Search for all the elements in list binary search
        // Compute running time averaged over 25 runs
        std::cout << std::fixed << std::setprecision(3) << std::setw(12)
                  << time_execution(binary_search, list, 25)
                  << " sec\n";
    }
}
```

Listing 12.5 (searchcompare.cpp) applies linear search and binary search to vectors of various sizes and displays the results. The vector sizes range from 0 to 50,000. The program uses the function `time_execution` to compute the average running times of linear search and binary search. The `main` function directs which search `time_execution` should perform by passing as the first parameter a pointer to the appropriate function. The second parameter to `time_execution` specifies the number of runs the function should use to compute the average. Notice that in this program we average linear search over five runs and execute binary search over 25 runs. We subject the binary search function to more runs since it executes so quickly, and five runs is not an adequate sample size to evaluate its performance, especially for smaller vectors where the binary search's execution time is close to the resolution of the timer. Besides, since binary search does execute so quickly, we easily can afford to let `time_execution` run more tests and compute a more accurate average.

In Listing 12.5 (searchcompare.cpp) we use stream manipulators `std::fixed` and `std::setprecision` to dress up the output. The `std::fixed` manipulator adjusts `std::cout` to use a fixed number of decimal places, and the `std::setprecision` manipulator specifies the number of digits to display after the decimal point. The two manipulators in tandem allow us to align the columns of numbers by their decimal points.

A sample run of Listing 12.5 (searchcompare.cpp) displays

```
--------------------------------------
  Vector        Linear         Binary
   Size         Search         Search
--------------------------------------
       0       0.000 sec       0.000 sec
    5000       0.112 sec       0.001 sec
   10000       0.444 sec       0.002 sec
   15000       1.003 sec       0.003 sec
   20000       2.172 sec       0.007 sec
   25000       3.444 sec       0.005 sec
   30000       5.254 sec       0.006 sec
   35000       7.216 sec       0.008 sec
   40000       9.701 sec       0.009 sec
   45000      11.709 sec       0.018 sec
   50000      12.287 sec       0.012 sec
```

With a vector of size 50,000 linear search takes on average about 12 seconds on one system, while binary search requires a small fraction of a one second. The times for binary search are so small that the progression of times wanders a bit from perfectly ascending order. This is because the operating system is performing other tasks while our program is running. It gives each active program its own slice of processor time to run. The operating system can interrupt executing programs to give other tasks time to run. For most programs this processor time sharing is imperceptible, but it can make a significant difference in short-running programs. We can see in the results that the operating system must have attending to some other matters during the binary search of vectors of size 20,000 and 45,000, as these times are slightly higher than we would expect from the pattern of values. Ideally we would perform the tests multiple times and average the results to get a more accurate picture.

If we increase the vector's size to 500,000, linear search runs in 830 seconds (13 minutes, 50 seconds), while binary search still takes less than one second! Empirically, binary search performs dramatically better than linear search.

Action	Operations	Operation Cost	Iterations	Cost
`n = a.size()`	`=,a.size`	2	1	2
`i = 0`	`=`	1	1	1
`i < size && a[i] <= seek`	`<=,&&,[],<=`	4	$n/2$	$2n$
`a[i] == seek`	`[],==`	2	$n/2$	n
`return i` **or** `return -1`	`return`	1	1	1
			Total Cost	$3n+4$

Table 12.1: Analysis of Linear Search

One might wonder why a binary search on vector with 45,000 items is slightly slower than one on a vector containing 50,000 elements. This particular execution was performed on a computer running Microsoft Windows. Microsoft Windows' timer resolution via the `clock` function is milliseconds, so the values for binary search are near that timer's resolution. Windows is a multitasking operating system, meaning it manages a number of tasks (programs) simultaneously. The measurements above were performed on a "lightly loaded system," which means during the program's execution the user was not downloading files, browsing the web, or doing anything else in particular. Windows, as all modern multitasking OSs, runs a lot of services (other programs) in the background that steal processor time slices. It continually checks for network connections, tracks the users mouse movement, etc. The binary search algorithm's speed coupled with a multitasking OS with millisecond timer resolution can lead to such minor timing anomalies.

In addition to using the empirical approach, we can judge which algorithm is better by analyzing the source code for each function. Each arithmetic operation, assignment, logical comparison, and vector access requires time to execute. We will assume each of these activities requires one unit of processor "time." This assumption is not strictly true, but it will give acceptable results for relative comparisons. Since we will follow the same rules when analyzing both search algorithms, the relative results for comparison purposes will be fairly accurate.

We first consider linear search:

```
int linear_search(const std::vector<int>& a, int seek) {
    int n = a.size();
    for (int i = 0;  i < n && a[i] <= seek;  i++)
        if (a[i] == seek)
            return i;   // Return position immediately
    return -1;  // Element not found
}
```

We determined that, on average, the loop makes $\frac{n}{2}$ iterations for a vector of length n. The initialization of `i` happens only one time during each call to `linear_search`. All other activity involved with the loop except the `return` statements happens $\frac{n}{2}$ times. The function returns either `i` or -1, and only one return is executed during each call. Table 12.1 shows the breakdown for linear search.

The running time of the `linear_search` function thus can be expressed as a simple linear function: $L(n) = 3n+4$.

Action	Operations	Operation Cost	Iterations	Cost
`n = a.size()`	`=,a.size`	2	1	2
`first = 0`	`=`	1	1	1
`last = n - 1`	`=,-`	2	1	2
`first <= last`	`<=`	1	$\log_2 n$	$\log_2 n$
`mid = first + (last - first + 1)/2`	`=,+,-,+,/`	5	$\log_2 n$	$5\log_2 n$
`v[mid] == seek`	`[],==`	2	$\log_2 n$	$2\log_2 n$
`v[mid] > seek`	`[],>`	2	$\log_2 n$	$2\log_2 n$
`last = mid - 1` or `first = mid + 1`	`=,±`	2	$\log_2 n$	$2\log_2 n$
`return mid` **or** `return -1`	`return`	1	1	1
			Total Cost	$12\log_2 n + 6$

Table 12.2: Analysis of Binary Search

Next, we consider binary search:

```
int binary_search(const std::vector<int>& a, int seek) {
    int n = a.size(),    // Number of elements
        first = 0,       // Initially the first element in vector
        last = n - 1,    // Initially the last element in vector
        mid;             // The middle of the vector
    while (first <= last) {
        mid = first + (last - first + 1)/2;
        if (a[mid] == seek)
            return mid;       // Found it
        else if (a[mid] > seek)
            last = mid - 1;   // continue with 1st half
        else  // a[mid] < seek
            first = mid + 1;  // continue with 2nd half
    }
    return -1;    // Not there
}
```

We determined that in the worst case the loop in `binary_search` iterates $\log_2 n$ times if the vector contains n elements. The two initializations before the loop are performed once per call. Most of the actions within the loop occur $\log_2 n$ times, except that only one `return` statement can be executed per call, and in the `if`/`else` statement only one path can be chosen per loop iteration. Table 12.2 shows the complete analysis of binary search.

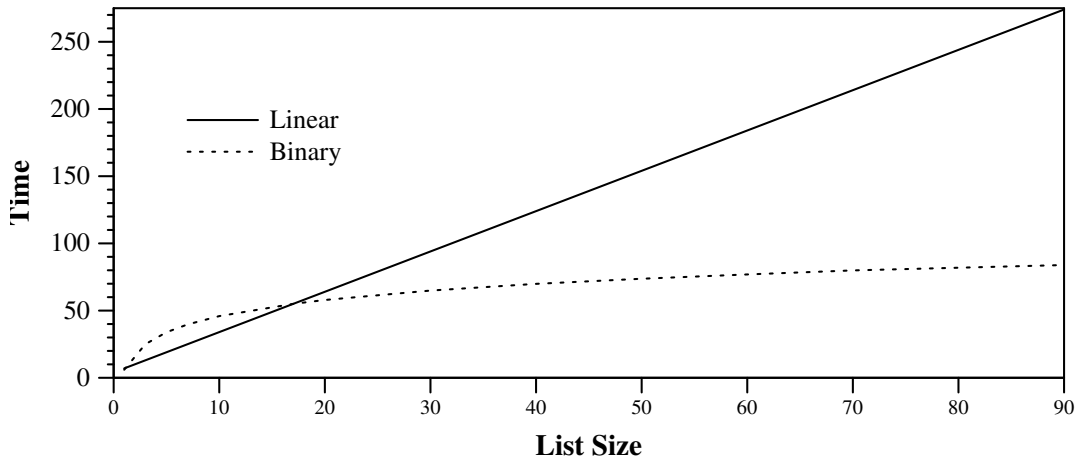
We will call our binary search function $B(n)$. Figure 12.3 shows the plot of the two functions $L(n) = 3n+4$ and $B(n) = 12\log_2 n + 6$.

For $n < 17$, the linear function $3n+4$ is less than the binary function $12\log_2 n + 6$. This means that linear search should perform better than binary search for vector sizes less than 17. This is because the code for linear search is less complicated, and it can complete its work on smaller vectors before the binary search finishes its more sophisticated computations. At $n = 17$, however, the two algorithms should perform about the same because

$$L(17) = 3(17)+4 = 51+4 = 55 \approx 55.05 = 49.05+6 = 12(4.09)+6 = 12\log_2 17 + 6 = B(17)$$

Figure 12.3 shows that for all $n > 17$ binary search outperforms linear search, and the performance gap increases rapidly as n grows. This wide performance discrepancy agrees with our empirical observations

Figure 12.3 A graph of the functions derived from analyzing the linear and binary search routines



we obtained from Listing 12.5 (searchcompare.cpp). Unfortunately we cannot empirically compare the running times of the two searches for vectors small enough to demonstrate that linear search is faster for very small vectors. As the output of Listing 12.5 (searchcompare.cpp) shows, both searches complete their work in time less than the resolution of our timer for vectors with 1,000 elements. Both empirically and analytically, we see that binary search is fast even for very large vectors, while linear search is impractical for large vectors.

12.4 Vector Permutations

Sometimes it is useful to consider all the possible arrangements of the elements within a vector. A sorting algorithm, for example, must work correctly on any initial arrangement of elements in a vector. To test a sort function, a programmer could check to see if it produces the correct result for all arrangements of a relatively small vector. A rearrangement of a collection of ordered items is called a *permutation*. Listing 12.6 (vectorpermutations.cpp) prints all the permutations of the contents of a given vector.

Listing 12.6: vectorpermutations.cpp

```
#include <iostream>
#include <vector>

/*
 *  print
 *     Prints the contents of a vector of integers
 *     a is the vector to print; a is not modified
 */
void print(const std::vector<int>& a) {
    int n = a.size();
    std::cout << "{";
    if (n > 0) {
        std::cout << a[0];  // Print the first element
        for (int i = 1; i < n; i++)
            std::cout << ',' << a[i];  // Print the rest
```

```
    }
    std::cout << "}";
}

/*
 *  Prints all the permutations of vector a in the
 *  index range begin...end, inclusive.  The function's
 *  behavior is undefined if begin or end
 *  represents an index outside of the bounds of vector a.
 */
void permute(std::vector<int>& a, int begin, int end) {
    if (begin == end) {
        print(a);
        std::cout << '\n';
    }
    else {
        for (int i = begin; i <= end; i++) {
            // Interchange the element at the first position
            // with the element at position i
            std::swap(a[begin], a[i]);
            // Recursively permute the rest of the vector
            permute(a, begin + 1, end);
            // Interchange the current element at the first position
            // with the current element at position i
            std::swap(a[begin], a[i]);
        }
    }
}

/*
 *  Tests the permutation functions
 */
int main() {
    // Get number of values from the user
    std::cout << "Please enter number of values to permute: ";
    int number;
    std::cin >> number;
    // Create the vector to hold all the values
    std::vector<int> list(number);
    // Initialize the vector
    for (int i = 0; i < number; i++)
        list[i] = i;

    // Print original list
    print(list);
    std::cout << "\n----------\n";
    // Print all the permutations of list
    permute(list, 0, number - 1);
    std::cout << "\n----------\n";
    // Print list after all the manipulations
    print(list);
}
```

A sample run of Listing 12.6 (vectorpermutations.cpp) when the user enters 4 prints

```
Please enter number of values to permute: 4
{0,1,2,3}
----------
{0,1,2,3}
{0,1,3,2}
{0,2,1,3}
{0,2,3,1}
{0,3,2,1}
{0,3,1,2}
{1,0,2,3}
{1,0,3,2}
{1,2,0,3}
{1,2,3,0}
{1,3,2,0}
{1,3,0,2}
{2,1,0,3}
{2,1,3,0}
{2,0,1,3}
{2,0,3,1}
{2,3,0,1}
{2,3,1,0}
{3,1,2,0}
{3,1,0,2}
{3,2,1,0}
{3,2,0,1}
{3,0,2,1}
{3,0,1,2}

----------
{0,1,2,3}
```

The `permute` function in Listing 12.6 (vectorpermutations.cpp) is a recursive function, as it calls itself inside of its definition. We have seen how recursion can be an alternative to iteration; however, the `permute` function here uses *both* iteration *and* recursion together to generate all the arrangements of a vector. At first glance, the combination of these two algorithm design techniques as used here may be difficult to follow, but we actually can understand the process better if we *ignore* some of the details of the code.

First, notice that in the recursive call the argument `begin` is one larger, and `end` remains the same. This means as the recursion progresses the ending index never changes, and the beginning index keeps increasing until it reaches the ending index. The recursion terminates when `begin` becomes equal to `end`.

In its simplest form the function looks like this:

```
void permute(int *a, int begin, int end) {
    if (begin == end)
        // Print the whole vector
    else
        // Do the interesting part of the algorithm
}
```

Let us zoom in on the interesting part of the algorithm (less the comments):

```
for (int i = begin; i <= end; i++) {
```

```
    swap(a[begin], a[i]);
    permute(a, begin + 1, end);
    swap(a[begin], a[i]);
}
```

If the mixture of iteration and recursion is confusing, eliminate iteration!

If a loop iterates a fixed number of times, you may replace the loop with the statements in its body duplicated that number times; for example, we can rewrite the code

```
for (int i = 0; i < 5; i++)
    std::cout << i << '\n';
```

as

```
std::cout << 0 << '\n';
std::cout << 1 << '\n';
std::cout << 2 << '\n';
std::cout << 3 << '\n';
std::cout << 4 << '\n';
```

Notice that the loop is gone. This process of transforming a loop into the series of statements that the loop would perform is known as *loop unrolling*. Compilers sometimes unroll loops to make the code's execution faster. After unrolling the loop the loop control variable (in this case `i`) is gone, so there is no need to initialize it (done once) and, more importantly, no need to check its value and update it during each iteration of the loop.

Our purpose for unrolling the loop in `perform` is not to optimize it. Instead we are trying to understand better how the algorithm works. In order to unroll `perform`'s loop, we will consider the case for vectors containing exactly three elements. In this case the `for` in the `perform` function would be hardcoded as

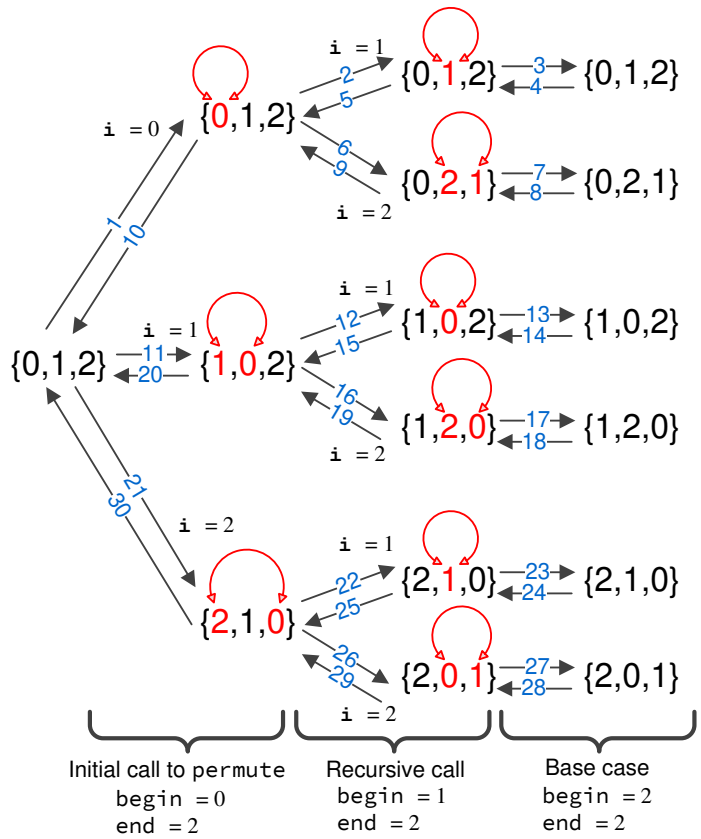
```
for (int i = 0; i <= 2; i++) {
    swap(a[begin], a[i]);
    permute(a, begin + 1, end);
    swap(a[begin], a[i]);
}
```

and we can transform this code into

```
swap(a[begin], a[0]);
permute(a, begin + 1, end);
swap(a[begin], a[0]);

swap(a[begin], a[1]);
permute(a, begin + 1, end);
swap(a[begin], a[1]);

swap(a[begin], a[2]);
permute(a, begin + 1, end);
swap(a[begin], a[2]);
```

Once the loop is gone, we see we have simply a series of recursive calls of `permute` sandwiched by calls to `swap`. The first call to `swap` interchanges an element in the vector with the first element. The second call to `swap` reverses the effects of the first swap. This series of `swap`-`permute`-`swap` operations allows

Figure 12.4 A tree mapping out the recursive process of the `permute` function operating on the vector `{}`.



each element in the vector to have its turn being the first element in the permuted vector. The `permute` recursive call generates all the permutations of the rest of the list. Figure 12.4 traces the recursive process of generating all the permutations of the vector `{0,1,2}`.

The leftmost third of Figure 12.4 shows the original contents of the vector and the initial call of `permute`. The three branches represent the three iterations of the `for` loop: `i` varying from `begin` (0) to `end` (2). The vectors indicate the state of the vector after the first swap but before the recursive call to `permute`.

The middle third of Figure 12.4 shows the state of the vector during the first recursive call to `permute`. The two branches represent the two iterations of the `for` loop: `i` varying from `begin` (1) to `end` (2). The vectors indicate the state of the vector after the first swap but before the next recursive call to `permute`. At this level of recursion the element at index zero is fixed, and the remainder of the processing during this chain of recursion is restricted to indices greater than zero.

The rightmost third of Figure 12.4 shows the state of the vector during the second recursive call to `permute`. At this level of recursion the elements at indices zero and one are fixed, and the remainder of the processing during this chain of recursion is restricted to indices greater than one. This leaves the

element at index two, but this represents the base case of the recursion because `begin` (2) equals `end` (2). The function makes no more recursive calls to itself. The function merely prints the current contents of the vector.

The arrows in Figure 12.4 represent a call to, or a return from, `permute`. They illustrate the recursive call chain. The arrows pointing left to right represent a call, and the arrows pointing from right to left represent a return from the function. The numbers associated with arrow indicate the order in which the calls and returns occur during the execution of `permute`.

The second column from the left shows the original contents of the vector after the first `swap` call but before the first recursive call to `permute`. The swapped elements appear in red. The third column shows the contents of the vector at the second level of recursion. In the third column the elements at index zero are fixed, as this recursion level is using `begin` with a value of one instead of zero. The `for` loop within this recursive call swaps the elements highlighted in red. The rightmost column is the point where `begin` equals `end`, and so the `permute` function does not call itself effectively terminating the recursion.

While Listing 12.6 (vectorpermutations.cpp) is a good exercise in vector manipulation and recursion, the C++ standard library provides a function named `next_permutation` that rearranges the elements of a vector. Listing 12.7 (stlpermutations.cpp) uses `next_permutation` within a loop to print all the permutations of the vector's elements.

Listing 12.7: `stlpermutations.cpp`

```
#include <iostream>
#include <vector>
#include <algorithm>

/*
 *  print
 *     Prints the contents of an int vector
 *     a is the vector to print; a is not modified
 */
void print(const std::vector<int>& a) {
    int n = a.size();
    std::cout << "{";
    if (n > 0) {
        std::cout << a[0];  // Print the first element
        for (int i = 1; i < n; i++)
            std::cout << ',' << a[i];  // Print the rest
    }
    std::cout << "}";
}


int main() {
    std::vector<int> nums { 0, 1, 2, 3 };
    std::cout << "---------------\n";
    do {
        print(nums);
        std::cout << '\n';
    }   // Compute the next ordering of elements
    while (next_permutation(begin(nums), std::end(nums)));
}
```

12.5 Randomly Permuting a Vector

Section 12.4 showed how we can generate all the permutations of a vector in an orderly fashion. More often, however, we need to produce one of those permutations chosen at random. For example, we may need to randomly rearrange the contents of an ordered vector so that we can test a sort function to see if it will produce the original ordered sequence. We could generate all the permutations, put each one in a vector of vectors, and select a permutation at random from that vector of vectors. This approach is very inefficient, especially as the length of the vector to permute grows larger. Fortunately, we can randomly permute the contents of a vector easily and quickly. Listing 12.8 (randompermute.cpp) contains a function named `permute` that randomly permutes the elements of an vector.

Listing 12.8: `randompermute.cpp`

```
#include <iostream>
#include <vector>
#include <ctime>
#include <cstdlib>

/*
 *  print
 *     Prints the contents of an int vector
 *     a is the vector to print; a is not modified
 */
void print(const std::vector<int>& a) {
    int n = a.size();
    std::cout << "{";
    if (n > 0) {
        std::cout << a[0];  // Print the first element
        for (int i = 1; i < n; i++)
            std::cout << ',' << a[i];  // Print the rest
    }
    std::cout << "}";
}

/*
 *  Returns a pseudorandom number in the range begin...end - 1,
 *  inclusive.  Returns 0 if begin >= end.
 */
int random(int begin, int end) {
    if (begin >= end)
        return 0;
    else {
        int range = end - begin;
        return begin + rand()%range;
    }
}

/*
 *  Randomly permute a vector of integers.
 *  a is the vector to permute, and n is its length.
 */
void permute(std::vector<int>& a) {
    int n = a.size();
    for (int i = 0; i < n - 1; i++) {
```

```
        // Select a pseudorandom location from the current
        // location to the end of the collection
        std::swap(a[i], a[random(i, n)]);
    }
}

// Tests the permute function that randomly permutes the
// contents of a vector
int main() {
    // Initialize random generator seed
    srand(static_cast<int>(time(0)));

    // Make the vector {1,2,3,4,5,6,7,8}
    std::vector<int> vec { 1, 2, 3, 4, 5, 6, 7, 8 };

    // Print vector before
    print(vec);
    std::cout << '\n';

    permute(vec);

    // Print vector after
    print(vec);
    std::cout << '\n';
}
```

One run of Listing 12.8 (randompermute.cpp) produces

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

Notice that the `permute` function in Listing 12.8 (randompermute.cpp) uses a simple un-nested loop and no recursion. The `permute` function varies the `i` index variable from 0 to the index of the next to last element in the vector. Within the loop, the function obtains via `rand` (see Section 8.6) a pseudorandom index greater than or equal to `i`. It then exchanges the elements at position `i` and the random position. At this point all the elements at index `i` and smaller are fixed and will not change as the function's execution continues. The loop then increments index `i`, and the process continues until all the `i` values have been considered.

To be correct, our `permute` function must be able to generate any valid permutation of the vector. It is important that our `permute` function is able to produce all possible permutations with equal probability; said another way, we do not want our `permute` function to generate some permutations more often than others. The `permute` function in Listing 12.8 (randompermute.cpp) is fine, but consider a slight variation of the algorithm:

```
// Randomly permute a vector?
void faulty_permute(std::vector<int>& a) {
    int n = a.size()
    for (int i = 0; i < n; i++) {
        // Select a pseudorandom position somewhere in the vector
        swap(a[i], a[random(0, n)]);
    }
}
```

Do you see the difference between `faulty_permute` and `permute`? The `faulty_permute` function chooses the random index from all valid vector indices, whereas `permute` restricts the random index to valid indices greater than or equal to `i`. This means that `faulty_permute` can exchange any element within vector `a` with the element at position `i` during any loop iteration. While this approach superficially may appear to be just as good as `permute`, it in fact produces an uneven distribution of permutations. Listing 12.9 (comparepermutations.cpp) exercises each `permutation` function 1,000,000 times on the vector `{1, 2, 3}` and tallies each permutation. There are exactly six possible permutations of this three-element vector.

Listing 12.9: `comparepermutations.cpp`

```
#include <iostream>
#include <vector>
#include <ctime>
#include <cstdlib>

/*
 *  print
 *     Prints the contents of an int vector
 *     a is the vector to print; a is not modified
 *     n is the number of elements in the vector
 */
void print(const std::vector<int>& a) {
    int n = a.size();
    std::cout << "{";
    if (n > 0) {
        std::cout << a[0];  // Print the first element
        for (int i = 1; i < n; i++)
            std::cout << ',' << a[i];  // Print the rest
    }
    std::cout << "}";
}

/*
 *  Returns a pseudorandom number in the range begin...end - 1,
 *  inclusive.  Returns 0 if begin >= end.
 */
int random(int begin, int end) {
    if (begin >= end)
        return 0;
    else {
        int range = end - begin;
        return begin + rand()%range;
    }
}

/*
 *  Randomly permute a vector of integers.
 *  a is the vector to permute, and n is its length.
 */
void permute(std::vector<int>& a) {
    int n = a.size();
    for (int i = 0; i < n - 1; i++) {
        // Select a pseudorandom location from the current
        // location to the end of the collection
```

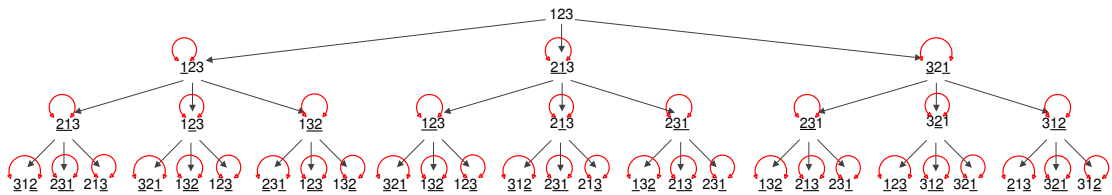
```
        std::swap(a[i], a[random(i, n)]);
    }
}

/*   Randomly permute a vector?  */
void faulty_permute(std::vector<int>& a) {
    int n = a.size();
    for (int i = 0; i < n; i++) {
        // Select a pseudorandom position somewhere in the collection
        std::swap(a[i], a[random(0, n)]);
    }
}

/*  Classify a vector as one of the six permutations */
int classify(const std::vector<int>& a) {
    switch (100*a[0] + 10*a[1] + a[2]) {
        case 123:  return 0;
        case 132:  return 1;
        case 213:  return 2;
        case 231:  return 3;
        case 312:  return 4;
        case 321:  return 5;
    }
    return -1;
}

/*  Report the accumulated statistics */
void report(const std::vector<int>& a) {
    std::cout << "1,2,3: " << a[0] << '\n';
    std::cout << "1,3,2: " << a[1] << '\n';
    std::cout << "2,1,3: " << a[2] << '\n';
    std::cout << "2,3,1: " << a[3] << '\n';
    std::cout << "3,1,2: " << a[4] << '\n';
    std::cout << "3,2,1: " << a[5] << '\n';
}

/*
 * Fill the given vector with zeros.
 * a is the vector, and n is its length.
 */
void clear(std::vector<int>& a) {
    int n = a.size();
    for (int i = 0; i < n; i++)
        a[i] = 0;
}

int main() {
    // Initialize random generator seed
    srand(static_cast<int>(time(0)));

    // permutation_tally vector keeps track of each permutation pattern
    // permutation_tally[0] counts {1,2,3}
    // permutation_tally[1] counts {1,3,2}
    // permutation_tally[2] counts {2,1,3}
    // permutation_tally[3] counts {2,3,1}
```

```
    // permutation_tally[4] counts {3,1,2}
    // permutation_tally[5] counts {3,2,1}
    std::vector<int> permutation_tally { 0, 0, 0, 0, 0, 0 };

    // original always holds the vector {1,2,3}
    const std::vector<int> original { 1, 2, 3 };

    // working holds a copy of original that gets permuted and tallied
    std::vector<int> working;

    // Run each permutation one million times
    const int RUNS = 1000000;

    std::cout << "--- Random permute #1 -----\n";
    clear(permutation_tally);
    for (int i = 0; i < RUNS; i++) { // Run 1,000,000 times
        // Make a copy of the original vector
        working = original;
        // Permute the vector with the first algorithm
        permute(working);
        // Count this permutation
        permutation_tally[classify(working)]++;
    }
    report(permutation_tally);   // Report results

    std::cout << "--- Random permute #2 -----\n";
    clear(permutation_tally);
    for (int i = 0; i < RUNS; i++) { // Run 1,000,000 times
        // Make a copy of the original vector
        working = original;
        // Permute the vector with the second algorithm
        faulty_permute(working);
        // Count this permutation
        permutation_tally[classify(working)]++;
    }
    report(permutation_tally);   // Report results
}
```

In Listing 12.9 (comparepermutations.cpp)'s output, permute #1 corresponds to our original `permute` function, and permute #2 is the `faulty_permute` function. The output of Listing 12.9 (comparepermutations.cpp) reveals that the faulty permutation function favors some permutations over others:

```
--- Random permute #1 -----
1,2,3: 166608
1,3,2: 166820
2,1,3: 166350
2,3,1: 166702
3,1,2: 166489
3,2,1: 167031
--- Random permute #2 -----
1,2,3: 148317
1,3,2: 184756
2,1,3: 185246
2,3,1: 185225
3,1,2: 148476
```

Figure 12.5 A tree mapping out the ways in which `faulty_permute` can transform the vector 1, 2, 3 at each iteration of its `for` loop




```
3,2,1: 147980
```

In one million runs, the `permute` function provides an even distribution of the six possible permutations of `{1, 2, 3}`. The `faulty_permute` function generates the permutations `{1, 3, 2}`, `{2, 1, 3}`, and `{2, 3, 1}` more often than the permutations `{1, 2, 3}`, `{3, 1, 2}`, and `{3, 2, 1}`.

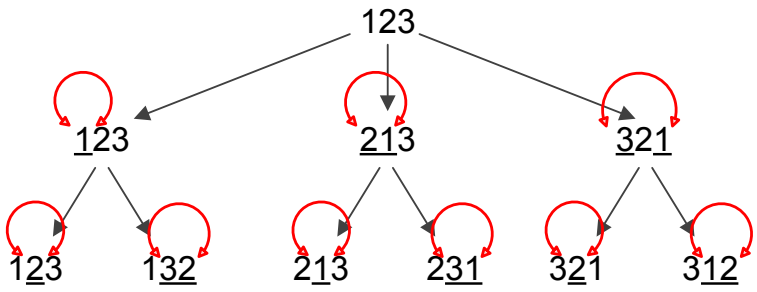
To see why `faulty_permute` misbehaves, we need to examine all the permutations it can produce during one call. Figure 12.5 shows a hierarchical structure that maps out how `faulty_permute` transforms the contents of its vector parameter each time through the `for` loop.

The top of the tree shows the original vector, `{1, 2, 3}`. The second row shows the three possible resulting configurations after the first iteration of the `for` loop. The leftmost 3-tuple represents the element at index zero swapped with the element at index zero (effectively no change). The second 3-tuple on the second row represents the interchange of the elements at index 0 and index 1. The third 3-tuple on the second row results from the interchange of the elements at positions 0 and 2. The underlined elements represent the elements most recently swapped. If only one item in the 3-tuple is underlined, the function merely swapped the item with itself. The bottom row of 3-tuples contains all the possible outcomes of the `faulty_permute` function given the vector `{1, 2, 3}`.

As Figure 12.5 shows, the vector `{1, 3, 2}`, `{2, 1, 3}`, and `{2, 3, 1}` each appear five times in the last row, while `{1, 2, 3}`, `{3, 1, 2}`, and `{3, 2, 1}` each appear only four times. There are a total of 27 possible outcomes, so some permutations appear $\frac{4}{27} = 14.815\%$ of the time, while the others appear $\frac{5}{27} = 18.519\%$ of the time. Notice that these percentages agree with our experimental results from Listing 12.9 (comparepermutations.cpp).

Compare Figure 12.5 to Figure 12.6. The second row of the tree for `permute` is identical to the second row of the tree for `faulty_permute`, but the third row is different. The second time through its loop the `permute` function does not attempt to exchange the element at index zero with any other elements. We see that none of the first elements in the 3-tuples in row three are underlined. The third row contains exactly one instance of each of the possible permutations of `{1, 2, 3}`. This means that the correct `permute` function is not biased towards any of the individual permutations, and so the function can generate all the permutations with equal probability. The `permute` function has a $\frac{1}{6} = 16.667\%$ probability of generating a particular permutation; this number agrees with our the experimental results of Listing 12.9 (comparepermutations.cpp).

Figure 12.6 A tree mapping out the ways in which `permute` can transform the vector 1, 2, 3 at each iteration of its `for` loop



12.6 Exercises

1. Complete the following function that reorders the contents of a vector so they are reversed from their original order. For example, a vector containing the elements 2, 6, 2, 5, 0, 1, 2, 3 would be transformed into 3, 2, 1, 0, 5, 2, 6, 2. Note that your function must physically rearrange the elements within the vector, not just print the elements in reverse order.

```
void reverse(std::vector<int>& v) {
    // Add your code...
}
```

2. Complete the following function that reorders the contents of a vector so that all the even numbers appear before any odd number. The even values are sorted in ascending order with respect to themselves, and the odd numbers that follow are also sorted in ascending order with respect to themselves. For example, a vector containing the elements 2, 1, 10, 4, 3, 6, 7, 9, 8, 5 would be transformed into 2, 4, 6, 8, 10, 1, 3, 5, 7, 9 Note that your function must physically rearrange the elements within the vector, not just print the elements in reverse order.

```
void special_sort(std::vector<int>& v) {
    // Add your code...
}
```

3. Complete the following function that shifts all the elements of a vector backward one place. The last element that gets shifted off the back end of the vector is copied into the first (0th) position. For example, if a vector containing the elements 2, 1, 10, 4, 3, 6, 7, 9, 8, 5 is passed to the function, it would be transformed into 5, 2, 1, 10, 4, 3, 6, 7, 9, 8 Note that your function must physically rearrange the elements within the vector, not just print the elements in the shifted order.

```
void rotate(std::vector<int>& v) {
    // Add your code...
}
```

4. Complete the following function that determines if the number of even and odd values in a vector is the same. The function would return true if the vector contains 5, 1, 0, 2 (two evens and two odds), but it would return false for the vector containing 5, 1, 0, 2, 11 (too many odds). The function should

return true if the vector is empty, since an empty vector contains the same number of evens and odds (0). The function does not affect the contents of the vector.

```
bool balanced(const std::vector<int>& v) {
    // Add your code...
}
```

5. Complete the following function that returns true if vector a contains duplicate elements; it returns false if all the elements in a are unique. For example, the vector 2, 3, 2, 1, 9 contains duplicates (2 appears more than once), but the vector 2, 1, 0, 3, 8, 4 does not (none of the elements appear more than once).

   An empty vector has no duplicates.

   The function does not affect the contents of the vector.

```
bool has_duplicates(const std::vector<int>& v) {
    // Add your code...
}
```

6. Can binary search be used on an unsorted vector? Why or why not?

7. Can linear search be used on an unsorted vector? Why or why not?

8. Complete the following function `is_ascending` that returns true if the elements in a vector of integers appear in ascending order (more precisely, non-descending order, if the vector contains duplicates). For example, the following statement

```
std::cout << is_ascending({3, 6, 2, 1, 7}) << '\n';
```

   would print *false*, but the statement

```
std::cout << is_ascending({3, 6, 7, 12, 27}) << '\n';
```

   would print *true*. The nonexistent elements in an empty vector are considered to be in ascending order because they cannot be out of order.

```
bool is_ascending(std::vector<int>& v) {
    // Add your code...
}
```

9. Consider a sort function that uses the `is_ascending` function from the previous problem. It uses a loop to test the permutations of a vector of integers. When it finds a permutation that contains all of the vector's elements in ascending order it exits the loop. Do you think this would be a good sorting algorithm? Why or why not?

Chapter 13

Standard C++ Classes

In the hardware arena, a desktop computer is built by assembling

- a motherboard (a circuit board containing sockets for a processor and assorted supporting cards),
- a processor,
- memory,
- a video card,
- an input/output card (USB ports, parallel port, and mouse port),
- a disk controller,
- a disk drive,
- a case,
- a keyboard,
- a mouse, and
- a monitor.

(Some of these components like the I/O, disk controller, and video may be integrated with the motherboard.)

The video card is itself a sophisticated piece of hardware containing a video processor chip, memory, and other electronic components. A technician does not need to assemble the card; the card is used as is off the shelf. The video card provides a substantial amount of functionality in a standard package. One video card can be replaced with another card from a different vendor or with another card with different capabilities. The overall computer will work with either card (subject to availability of drivers for the operating system) because standard interfaces allow the components to work together.

Software development today is increasingly *component based*. Software components are used like hardware components. A software system can be built largely by assembling pre-existing software building blocks. C++ supports various kinds of software building blocks. The simplest of these is the *function* that we investigated in Chapter 8 and Chapter 9. A more powerful technique uses built-in and user designed software *objects*.

©2018 Richard L. Halterman

C++ is object-oriented. It was not the first OO programming language, but it was the first OO language that gained widespread use in a variety of application areas. An OO programming language allows the programmer to define, create, and manipulate objects. Variables representing objects can have considerable functionality compared to the primitive numeric variables like `int`s and `double`s. Like a normal variable, every C++ object has a type. We say an object is an instance of a particular *class*, and *class* means the same thing as *type*. An object's type is its class. We have been using the `std::cout` and `std::cin` objects for some time. `std::cout` is an instance of the `std::ostream` class—which is to say `std::cout` is of type `std::ostream`. `std::cin` is an instance of the `std::istream` class.

Code that uses an object is a *client* of that object; for example, the following code fragment

```
std::cout << "Hello\n";
```

uses the `std::cout` object and, therefore, is a client of `std::cout`. Many of the functions we have seen so far have been clients of the `std::cout` and/or `std::cin` objects. Objects provide services to their clients.

13.1 String Objects

A string is a sequence of characters, most often used to represent words and names. The C++ standard library provides the class `string` which specifies *string objects*. In order to use string objects, you must provide the preprocessor directive

```
#include <string>
```

The `string` class is part of the standard namespace, which means its full type name is `std::string`. If you use the

```
using namespace std;
```

or

```
using std::string;
```

statements in your code, you can use the abbreviated name `string`.

You declare a `string` object like any other variable:

```
string name;
```

You may assign a literal character sequence to a `string` object via the familiar string quotation syntax:

```
string name = "joe";
std::cout << name << '\n';
name = "jane";
std::cout << name << '\n';
```

You may assign one `string` object to another using the simple assignment operator:

```
string name1 = "joe", name2;
name2 = name1;
std::cout << name1 << " " << name2 << '\n';
```

In this case, the assignment statement copies the characters making up `name1` into `name2`. After the assignment both `name1` and `name2` have their own copies of the characters that make up the string; they do not share their contents. After the assignment, changing one string will not affect the other string. Code within the `string` class defines how the assignment operator should work in the context of `string` objects.

Like the `vector` class (Section 11.1.3), the `string` class provides a number of methods. Some `string` methods include:

- `operator[]`—provides access to the value stored at a given index within the string
- `operator=`—assigns one string to another
- `operator+=`—appends a string or single character to the end of a `string` object
- `at`—provides bounds-checking access to the character stored at a given index
- `length`—returns the number of characters that make up the string
- `size`—returns the number of characters that make up the string (same as `length`)
- `find`—locates the index of a substring within a `string` object
- `substr`—returns a new `string` object made of a substring of an existing `string` object
- `empty`—returns true if the string contains no characters; returns false if the string contains one or more characters
- `clear`—removes all the characters from a string

The following code fragment

```
string word = "computer";
std::cout << "\"" << word << "\" contains " << word.length()
          << " letters." << '\n';
```

prints

```
"computer" contains 8 letters.
```

The expression:

```
word.length()
```

invokes the `length` method on behalf of the `word` object. The `string` class provides a method named `size` that behaves exactly like the `length` method.

The `string` class defines a method named `operator[]` that allows a programmer to access a character within a `string`, as in

```
std::cout << "The letter at index 3 is " << word.operator[](3) << '\n';
```

Here the `operator[]` method uses the same syntax as the `length` method, but the `operator[]` method expects a single integer parameter. The above code fragment is better written as

```
std::cout << "The letter at index 3 is " << word[3] << '\n';
```

The expression

```
word.operator[](3)
```

is equivalent to the expression

```
word[3]
```

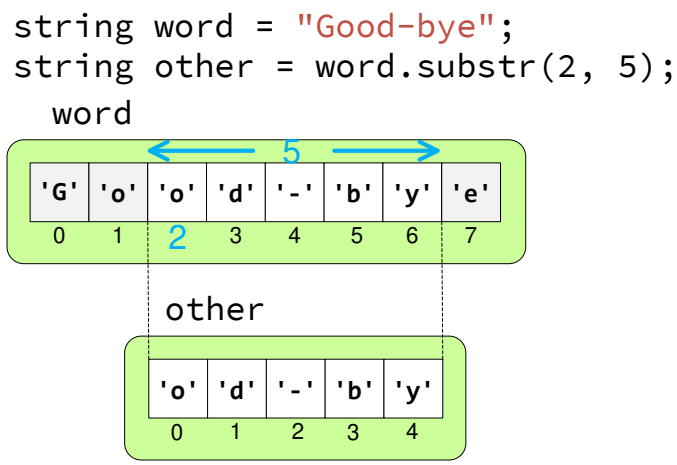
We see that `operator[]` works exactly like its namesake in the `std::vector` class Section 11.1.3.

The following code fragment exercises some of the methods available to string objects:

Listing 13.1: stringoperations.cpp

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

int main() {
    // Declare a string object and initialize it
    std::string word = "fred";
    // Prints 4, since word contains four characters
    std::cout << word.length() << '\n';
    // Prints "not empty", since word is not empty
    if (word.empty())
        std::cout << "empty\n";
    else
        std::cout << "not empty\n";
    // Makes word empty
    word.clear();
    // Prints "empty", since word now is empty
    if (word.empty())
        std::cout << "empty\n";
    else
        std::cout << "not empty\n";
    // Assign a string using operator= method
    word = "good";
    // Prints "good"
    std::cout << word << '\n';
    // Append another string using operator+= method
    word += "-bye";
    // Prints "good-bye"
    std::cout << word << '\n';
    // Print first character using operator[] method
    std::cout << word[0] << '\n';
    // Print last character
    std::cout << word[word.length() - 1] << '\n';
    // Prints "od-by", the substring starting at index 2 of length 5
    std::cout << word.substr(2, 5);
    std::string first = "ABC", last = "XYZ";
    // Splice two strings with + operator
    std::cout << first + last << '\n';
    std::cout << "Compare " << first << " and ABC: ";
    if (first == "ABC")
        std::cout << "equal\n";
    else
        std::cout << "not equal\n";
    std::cout << "Compare " << first << " and XYZ: ";
```

Figure 13.1 Extracting the new string "od-by" from the string "Good-bye"




```
    if (first == "XYZ")
        std::cout << "equal\n";
    else
        std::cout << "not equal\n";
}
```

The statement

```
word = "good";
```

is equivalent to

```
word.operator=("good");
```

Here we see the explicit use of the dot (`.`) operator to invoke the method. Similarly,

```
word += "-bye";
```

is the syntactally sweetened way to write

```
word.operator+=("-bye");
```

The + operator performs *string concatenation*, making a new string by appending one string to the back of another.

With the `substr` method we can extract a new string from another, as shown in Figure 13.1.

In addition to string methods, the standard `string` library provides a number of global functions that process strings. These functions use operator syntax and allow us to compare strings via <, ==, >=, etc. A more complete list of `string` methods and functions can be found at `http://www.cplusplus.com/reference/string/`.

13.2 Input/Output Streams

We have used `iostream` objects from the very beginning. `std::cout` is the output stream object that prints to the screen. `std::cin` is the input stream object that receives values from the keyboard. The precise type of `std::cout` is `std::ostream`, and `std::cin`'s type is `std::istream`.

Like other objects, `std::cout` and `std::cin` have methods. The `<<` and `>>` operators actually are the methods `operator<<` and `operator>>`. (The operators `<<` and `>>` normally are used on integers to perform left and right bitwise shift operations; see Section 4.10 for more information.) The following code fragment

```
std::cin >> x;
std::cout << x;
```

can be written in the explicit method call form as:

```
cin.operator>>(x);
cout.operator<<(x);
```

The first statement calls the `operator>>` method on behalf of the `std::cin` object passing in variable `x` by reference. The second statement calls the `operator<<` method on behalf of the `std::cout` object passing the value of variable `x`. A statement such as

```
std::cout << x << '\n';
```

is a more pleasant way of expressing

```
cout.operator<<(x).operator<<('\n');
```

Reading the statement left to right, the expression `cout.operator<<(x)` prints `x`'s value on the screen and returns the `std::cout` object itself. The return value (simply `std::cout`) then is used to invoke the `operator<<` method again with `'\n'` as its argument.

A statement such as

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

can be written

```
cin.operator>>(x).operator>>(y);
```

As is the case of `operator<<` with `std::cout`, reading left to right, the expression `cin.operator>>(x)` calls the `operator>>` method passing variable `x` by reference. It reads a value from the keyboard and assigns it to `x`. The method call returns `std::cin` itself, and the return value is used immediately to invoke `operator>>` passing variable `y` by reference.

You probably have noticed that it is easy to cause a program to fail by providing input that the program was not expecting. For example, compile and run Listing 13.2 (naiveinput.cpp).

Listing 13.2: `naiveinput.cpp`

```
#include <iostream>

int main() {
    int x;
    // I hope the user does the right thing!
```

```
    std::cout << "Please enter an integer: ";
    std::cin >> x;
    std::cout << "You entered " << x << '\n';
}
```

Listing 13.2 (naiveinput.cpp) works fine as long as the user enters an integer value. What if the user enters the word "five," which arguably is an integer? The program produces incorrect results. We can use some additional methods available to the `std::cin` object to build a more robust program. Listing 13.3 (betterintinput.cpp) detects illegal input and continues to receive input until the user provides an acceptable value.

Listing 13.3: `betterintinput.cpp`

```
#include <iostream>
#include <limits>

int main() {
    int x;
    // I hope the user does the right thing!
    std::cout << "Please enter an integer: ";
    // Enter and remain in the loop as long as the user provides
    // bad input
    while (!(std::cin >> x)) {
        // Report error and re-prompt
        std::cout << "Bad entry, please try again: ";
        // Clean up the input stream
        std::cin.clear();  // Clear the error state of the stream
        // Empty the keyboard buffer
        std::cin.ignore(std::numeric_limits<std::streamsize>::max(),'\n');
    }
    std::cout << "You entered " << x << '\n';
}
```

We learned in Section 6.1 that the expression

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

has a Boolean value that we may use within a conditional or iterative statement. If the user enters a value with a type compatible with the declared type of the variable, the expression evaluates to true; otherwise, it is interpreted as false. The negation

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

is true if the input is bad, so the only way to execute the body of the loop is provide illegal input. As long as the user provides bad input, the program's execution stays inside the loop.

While determining whether of not a user's entry is correct seems sufficient for the programmer to make corrective measures, it is not. Two additional steps are necessary:

- The bad input characters the user provided cause the `std::cin` object to enter an error state. The input stream object remains in an error state until the programmer manually resets it. The call

  ```
  cin.clear();
  ```

  resets the stream object so it can process more input.

- Whatever characters the user typed in that cannot be assigned to the given variable remain in the keyboard input buffer. Clearing the stream object does not remove the leftover keystrokes. Asking the user to retry without clearing the bad characters entered from before results in the same problem—the stream object re-enters the error state and the bad characters remain in the keyboard buffer. The solution is to flush from the keyboard buffer all of the characters that the user entered since the last valid data entry. The statement

  ```
  cin.ignore(numeric_limits<streamsize>::max(),'\n');
  ```

  removes from the buffer all the characters, up to and including the newline character (`'\n'`). The function call

  ```
  numeric_limits<streamsize>::max()
  ```

  returns the maximum number of characters that the buffer can hold, so the `ignore` method reads and discards characters until it reads the newline character (`'\n'`) or reaches the end of the buffer, whichever comes first.

Once the stream object has been reset from its error state and the keyboard buffer is empty, user input can proceed as usual.

The `ostream` and `istream` classes have a number of other methods, but we will not consider them here.

`istream` objects use whitespace (spaces and tabs) as delimiters when they get input from the user. This means you cannot use the `operator>>` to assign a complete line of text from the keyboard if that line contains embedded spaces. Listing 13.4 (faultyreadline.cpp) illustrates.

Listing 13.4: `faultyreadline.cpp`

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

int main() {
    std::string line;
    std::cout << "Please enter a line of text: ";
    std::cin >> line;
    std::cout << "You entered: \"" << line << "\"" << '\n';
}
```

A sample run of Listing 13.4 (faultyreadline.cpp) reveals:

```
Please enter a line of text: Mary had a little lamb.
You entered: "Mary"
```

As you can see, Listing 13.4 (faultyreadline.cpp) does not assign the complete line of text to the sting variable `line`. The text is truncated at the first space in the input.

To read in a complete line of text from the keyboard, including any embedded spaces that may be present, use the global `getline` function. As Listing 13.5 (readline.cpp) shows, the `getline` function accepts an `istream` object and a `string` object to assign.

Listing 13.5: `readline.cpp`

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

```
int main() {
    std::string line;
    std::cout << "Please enter a line of text: ";
    getline(std::cin, line);
    std::cout << "You entered: \"" << line << "\"" << '\n';
}
```

A sample run of Listing 13.5 (readline.cpp) produces:

```
Please enter a line of text: Mary has a little lamb.
You entered: "Mary has a little lamb."
```

13.3 File Streams

Many applications allow users to create and manipulate data. Truly useful applications allow users to store their data to files; for example, word processors can save and load documents.

Vectors would be more useful it they were *persistent*. Data is persistent when it exists between program executions. During one execution of a particular program the user may create and populate a vector. The user then saves the contents of the vector to disk and quits the program. Later, the user can run the program again and reload the vector from the disk and resume work.

C++ `fstream` objects allow programmers to build persistence into their applications. Listing 13.6 (numberlist.cpp) is a simple example of a program that allows the user to save the contents of a vector to a text file and load a vector from a text file.

Listing 13.6: `numberlist.cpp`

```
// File file_io.cpp

#include <iostream>
#include <fstream>
#include <string>
#include <vector>

/*
 *  print_vector(v)
 *    Prints the contents of vector v.
 *    v is a vector holding integers.
 */
void print_vector(const std::vector<int>& vec) {
    std::cout << "{";
    int len = vec.size();
    if (len > 0) {
        for (int i = 0; i < len - 1;  i++)
            std::cout << vec[i] << ",";  // Comma after elements
        std::cout << vec[len - 1];  // No comma after last element
    }
    std::cout << "}\n";
}

/*
```

```
 *  save_vector(filename, v)
 *    Writes the contents of vector v.
 *    filename is name of text file created.  Any file
 *    by that name is overwritten.
 *    v is a vector holding integers.  v is unchanged by the
 *    function.
 */
void save_vector(const std::string& filename, const std::vector<int>& vec) {
    // Open a text file for writing
    std::ofstream out(filename);
    if (out.good()) {  // Make sure the file was opened properly
        int n = vec.size();
        for (int i = 0; i < n;  i++)
            out << vec[i] << " ";   // Space delimited
        out << '\n';
    }
    else
        std::cout << "Unable to save the file\n";
}

/*
 *  load_vector(filename, v)
 *    Reads the contents of vector v from text file
 *    filename.  v's contents are replaced by the file's
 *    contents.  If the file cannot be found, the vector v
 *    is empty.
 *    v is a vector holding integers.
 */
void load_vector(const std::string& filename, std::vector<int>& vec) {
    // Open a text file for reading
    std::ifstream in(filename);
    if (in.good()) {  // Make sure the file was opened properly
        vec.clear();  // Start with empty vector
        int value;
        while (in >> value)  // Read until end of file
            vec.push_back(value);
    }
    else
        std::cout << "Unable to load in the file\n";
}

int main() {
    std::vector<int> list;
    bool done = false;
    char command;
    while (!done) {
        std::cout << "I)nsert <item> P)rint "
                  << "S)ave <filename> L)oad <filename> "
                  << "E)rase Q)uit: ";
        std::cin >> command;
        int value;
        std::string filename;
        switch (command) {
          case 'I':
          case 'i':
```

```
            std::cin >> value;
            list.push_back(value);
            break;
          case 'P':
          case 'p':
            print_vector(list);
            break;
          case 'S':
          case 's':
            std::cin >> filename;
            save_vector(filename, list);
            break;
          case 'L':
          case 'l':
            std::cin >> filename;
            load_vector(filename, list);
            break;
          case 'E':
          case 'e':
            list.clear();
            break;
          case 'Q':
          case 'q':
            done = true;
            break;
        }
    }
}
```

Listing 13.6 (numberlist.cpp) is command driven with a menu, and when the user types S data1.text the program saves the current contents of the vector to a file named data1.text. The user can erase the contents of the vector:

```
I)nsert <item> P)rint S)ave <filename> L)oad <filename> E)rase Q)uit: E
```

and then restore the contents with a load command:

```
I)nsert <item> P)rint S)ave <filename> L)oad <filename> E)rase Q)uit:
L data1.text
```

The user also may quit the program, and later re-run the program and load in the previously saved list of numbers. The user can save different number lists to different files using different file names.

Notice that in the `save_vector` and `load_vector` functions we pass the `std::string` parameter as a `const` reference. We do this for the same reason we do so for `std::vector` objects (see Section 11.1.4)—this technique avoids making a copy of the string to pass the functions. These functions can "see" the caller's actual string via the reference, rather than working with a copy of the string object. The `const` specifier prevents the functions from modifying the string passed. It takes time to copy a string (especially a long string), and the copy would occupy extra memory. In this case the copy really is not necessary, so passing by `const` reference is the ideal approach.

An `std::ofstream` object writes data to files. The statement

```
std::ofstream out(filename);
```

associates the object named `out` with the text file named `filename`. This opens the file as the point of declaration. We also can declare a file output stream object separately from opening it as

```
std::ofstream out;
out.open(filename);
```

The `save_vector` function in Listing 13.6 (numberlist.cpp) passes a `std::string` object to open the file, but file names can be string literals (quoted strings) as well. Consider the following code fragment:

```
std::ofstream fout("myfile.dat");
int x = 10;
if (fout.good()) // Make sure the file was opened properly
    fout << "x = " << x << '\n';
else
    std::cout << "Unable to write to the file \"myfile.dat\"\n";
```

After opening the file, programmers should verify that the method correctly opened the file by calling the file stream object's `good` method. An output file stream may fail for various reasons, including the disk being full or insufficient permissions to create a file in a given folder.

Once we have its associated file open, we can use a `std::ofstream` object like the `std::cout` output stream object, except the data is recorded in a text file instead of being printed on the screen. Just like with `std::cout`, you can use the `<<` operator and send a `std::ofstream` object stream manipulators like `std::setw`. The `std::cout` object and objects of class `std::ofstream` are in the same family of classes and related through a concept known as *inheritance*. We consider inheritance in more detail in Chapter 17. For our purposes at this point, this relationship means anything we can do with the `std::cout` object we can do a `std:ofstream` object. The difference, of course, is the effects appear in the console window for `std::cout` and are written in a text file given a `std::ofstream` object.

After the executing program has written all the data to the file and the `std::ofstream` object goes out of scope, the file object automatically will close the file ensuring that all data the program writes to the file is saved completely on disk. The `std::ofstream` class provides also a `close` method that allows programmers to manually close the file. This sometimes is useful when using the same file object to recreate the same file, as in Listing 13.7 (endltest.cpp).

In Listing 13.6 (numberlist.cpp), a `std::ifstream` object reads data from files. The statement

```
std::ifstream in(filename);
```

associates the object named `in` with the text file named `filename`. This opens the file as the point of declaration. We also can declare a file output stream object separately from opening it as

```
std::ifstream in;
in.open(filename);
```

As with `std::ofstream` objects, `filename` is a string file name identifying the file to read.

After opening the file the program should call `good` to ensure the file was successfully opened. An input stream object often fails to open properly because the file does not exist; perhaps the file name is misspelled, or the path to the file is incorrect. An input stream can also fail because of insufficient permissions or because of bad sectors on the disk.

Once it opens its associated file, an input file stream object behaves like the `std::cin` object, except its data comes from a text file instead the keyboard. This means the familiar `>>` operator and `getline` function are completely compatible with `std::ifstream` objects. The `std::cin` object and `std::ifstream`

objects are related through inheritance simmilar to the way the `std::cout` object and `std::ofstream` objects are related.

As with an output stream object, a `std::ifstream` object automatically will close its associated file when it goes out of scope.

Input and output streams use a technique known as *buffering*. Buffering relies on two facts:

- It is faster to write data to memory than to disk.
- It is faster to write one block of n bytes to disk in a single operation than it is to write n bytes of data one byte at a time using n operations.

A buffer is a special place in memory that holds data to be written to disk. A program can write to the buffer much faster than directly to the disk. When the buffer is full, the program (via the operating system) can write the complete contents of the buffer to the disk.

To understand the concept of buffering, consider the task of building a wall with bricks. Estimates indicate that the wall will require about 1,350 bricks. Once we are ready to start building the wall we can drive to the building supply store and purchase a brick. We then can drive to the job site and place the brick in its appropriate position using mortar as required. Now we are ready to place the next brick, so we must drive back to the store to get the next brick. We then drive back to the job site and set the brick. We repeat this process about 1,350 times.

If this seems very inefficient, it is. It would be better to put as many bricks as possible into the vehicle on the first trip, and then make subsequent trips to the store for more loads of bricks as needed until the wall is complete.

In this analogy, the transport vehicle is the *buffer*. The output stream object uses a special place in memory called a buffer. Like the vehicle used to transport our bricks, the memory buffer has a fixed capacity. A program can write to the buffer much more quickly than directly to the disk. The `<<` operator writes the individual values to save to the buffer, and when the buffer is full, the output stream sends all the data in the buffer out to the disk with one request to the operating system. As with the bricks, this is more efficient than sending just one character at a time to the display. This buffering process can speed up significantly the input and output operations of programs.

After the `std::ofstream` object writes all its data to its buffer and its lifetime is over, it flushes the remaining data from the buffer to disk, even if the buffer is not full. The buffer is a fixed size, so the last part of the data likely will not completely fill the buffer. This is analogous to the last load of bricks needed for our wall that may not make up a full load. We still need to get those remaining bricks to our almost complete wall even though the vehicle is not fully loaded.

In some situations it is necessary to ensure the buffer is flushed before it is full and before closing the file completely. With any output stream object writing text we can use the `std::endl` stream object to flush the buffer without closing the file. We mentioned `std::endl` briefly in Section 3.1. We can use `std::endl` interchangeably with `'\n'` to represent newlines for console printing. Because of the performance advantage buffering provides to file input and output, the choice of `std::endl` and `'\n'` can make a big difference for file processing. Listing 13.7 (endltest.cpp) compares the performance of `std::endl` and `'\n'` in various situations.

Listing 13.7: `endltest.cpp`

```
#include <iostream>
#include <fstream>
#include <ctime>
```

```
#include <vector>
#include <cstdlib>

// Make a convenient alias for the long type name
using Sequence = std::vector<int>;

Sequence make_random_sequence(int size, int max) {
    Sequence result(size);
    for (int i = 0; i < size; i++)
        result[i] = rand() % max;
    return result;
}

void print_with_endl(const Sequence& vs, std::ostream& out) {
    for (auto elem : vs)
        out << elem << std::endl;
}

void print_with_n(const Sequence& vs, std::ostream& out) {
    for (auto elem : vs)
        out << elem << '\n';
}

int main() {
    // Sequence up to 100,000 elements, with each element < 100.
    auto seq = make_random_sequence(100000, 100);

    // Time writing the elements to the console with std::endl newlines
    clock_t start_time = clock();
    print_with_endl(seq, std::cout);
    unsigned elapsed1 = clock() - start_time;


    // Time writing the elements to the console with '\n' newlines
    start_time = clock();
    print_with_n(seq, std::cout);
    unsigned elapsed2 = clock() - start_time;

    // Time writing the elements to a text file with std::endl newlines
    std::ofstream fout("temp.out");
    start_time = clock();
    print_with_endl(seq, fout);
    fout.close();
    unsigned elapsed3 = clock() - start_time;

    // Reopen the file for writing
    fout.open("temp.out");
    // Time writing the elements to a text file with '\n' newlines
    start_time = clock();
    print_with_n(seq, fout);
    fout.close();
    unsigned elapsed4 = clock() - start_time;

    std::cout << "With std::endl (console): " << elapsed1 << '\n';
    std::cout << "With '\\n' (console):     " << elapsed2 << '\n';
```

```
    std::cout << "With std::endl (file):    " << elapsed3 << '\n';
    std::cout << "With '\\n' (file):        " << elapsed4 << '\n';
}
```

Listing 13.7 (endltest.cpp) writes a vector containing 100,000 integers to the console and a text file. Each number appears on its own line. Since `std::endl` flushes the stream object's buffer in addition to printing a `'\n'`, we would expect it to reduce the program's performance since it would minimize the benefit of buffering in this case. Multiple runs of Listing 13.7 (endltest.cpp) on one system revealed that using `'\n'` to terminate lines generally was only slightly faster than `std::endl` (but not always) when writing to the console window. The `'\n'` terminator was consistently about three times faster than `std::endl` when writing to a text file.

Listing 13.7 (endltest.cpp) also exploits the special relationship between `std:cout` and any `std::ofstream` object. The `print_with_endl` and `print_with_n` functions both accept a `std::ostream` object as their second parameter. Note that the caller, `main`, passes both the `std::cout` object and the `fout` object to these printing functions at various times, and the compiler does not complain. We defer an explanation of how this works until Chapter 17.

13.4 Complex Numbers

C++ supports mathematical complex numbers via the `std::complex` class. Recall from mathematics that a complex number has a real component and an imaginary component. Often written as $a+bi$, a is the real part, an ordinary real number, and bi is the imaginary part where b is a real number and $i^2 = -1$.

The `std::complex` class in C++ is a template class like `vector`. In the angle brackets you specify the precision of the complex number's components:

```
std::complex<float> fc;
std::complex<double> dc;
std::complex<long double> ldc;
```

Here, the real component and imaginary coefficient of `fc` are single-precision floating-point values. `dc` and `ldc` have the indicated precisions. Listing 13.8 (complex.cpp) is a small example that computes the product of complex conjugates (which should be real numbers).

Listing 13.8: `complex.cpp`

```
// File complex.cpp
 #include <iostream>
 #include <complex>

 int main() {
     // c1 = 2 + 3i, c2 = 2 - 3i; c1 and c2 are complex conjugates
     std::complex<double> c1(2.0, 3.0), c2(2.0, -3.0);

     // Compute product "by hand"
     double real1 = c1.real(),
            imag1 = c1.imag(),
            real2 = c2.real(),
            imag2 = c2.imag();
     std::cout << c1 << " * " << c2 << " = "
               << real1*real2 + imag1*real2 + real1*imag2 - imag1*imag2
```

```
              << '\n';

    // Use complex arithmetic
    std::cout << c1 << " * " << c2 << " = " << c1*c2 << '\n';
}
```

Listing 13.8 (complex.cpp) prints

```
(2,3) * (2,-3) = 13
(2,3) * (2,-3) = (13,0)
```

Observe that the program displays the complex number $2-3i$ as the ordered pair `(2,-3)`. The first element of the pair is the real part, and the second element is the imaginary coefficient. If the imaginary part is zero, the number is a real number (or, in this case, a `double`).

Imaginary numbers have scientific and engineering applications that exceed the scope of this book, so this concludes our brief into C++'s `complex` class. If you need to solve problems that involve complex numbers, more information can be found at `http://www.cplusplus.com/reference/std/complex/`.

13.5 Better Pseudorandom Number Generation

Listing 12.9 (comparepermutations.cpp) showed that we must use care when randomly permuting the contents of a vector. A naïve approach can introduce accidental bias into the result. It turns out that our simple technique for generating pseudorandom numbers using `rand` and modulus has some issues itself.

Suppose we wish to generate pseudorandom numbers in the range $0\ldots9,999$. This range spans 10,000 numbers. Under Visual C++ `RAND_MAX` is 32,767, which is large enough to handle a maximum value of 9,999. The expression `rand() % 10000` will evaluate to number in our desired range. A good pseudorandom number generator should be just as likely to produce one number as another. In a program that generates one billion pseudorandom values in the range $0\ldots9,999$, we would expect any given number to appear approximately $\frac{1,000,000,000}{10,000} = 100,000$ times. The actual value for a given number will vary slightly from one run to the next, but the average over one billion runs should be very close to 100,000.

Listing 13.9 (badrand.cpp) evaluates the quality of the `rand` with modulus technique by generating one billion pseudorandom numbers within a loop. It counts the number of times the pseudorandom number generator produces 5 and also it counts the number of times 9,995 appears. Note that 5 is near the beginning of the range $0\ldots9,999$, and 9,995 is near the end of that range. To verify the consistency of its results, it repeats this test 10 times. The program reports the results of each individual trial, and in the end it computes the average of the 10 trials.

Listing 13.9: `badrand.cpp`

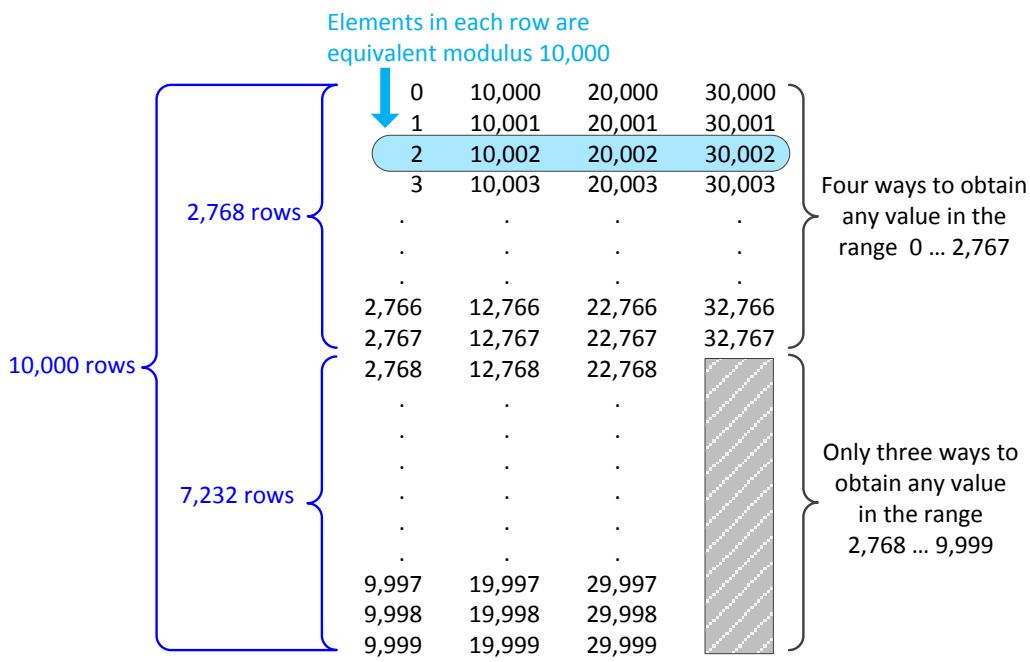
```
#include <iostream>
#include <iomanip>
#include <cstdlib>
#include <ctime>

int main() {
    // Initialize a random seed value
    srand(static_cast<unsigned>(time(nullptr)));
```

```
    // Verify the largest number that rand can produce
    std::cout << "RAND_MAX = " << RAND_MAX << '\n';

    // Total counts over all the runs.
    // Make these double-precision floating-point numbers
    // so the average computation at the end will use floating-point
    // arithmetic.
    double total5 = 0.0, total9995 = 0.0;

    // Accumulate the results of 10 trials, with each trial
    // generating 1,000,000,000 pseudorandom numbers
    const int NUMBER_OF_TRIALS = 10;

    for (int trial = 1; trial <= NUMBER_OF_TRIALS; trial++) {
        // Initialize counts for this run of a billion trials
        int count5 = 0, count9995 = 0;
        // Generate one billion pseudorandom numbers in the range
        // 0...9,999 and count the  number of times 5 and 9,995 appear
        for (int i = 0; i < 1000000000; i++) {
            // Generate a pseudorandom number in the range 0...9,999
            int r = rand() % 10000;
            if (r == 5)
                count5++;     // Number 5 generated, so count it
            else if (r == 9995)
                count9995++;  // Number 9,995 generated, so count it
        }
        // Display the number of times the program generated 5 and 9,995
        std::cout << "Trial #" << std::setw(2) << trial << "  5: " << count5
                  << "    9995: " << count9995 << '\n';
        total5 += count5;         // Accumulate the counts to
        total9995 += count9995;   // average them at the end
    }
    std::cout << "-------------------\n";
    std::cout << "Averages for " << NUMBER_OF_TRIALS << " trials:  5: "
              << total5 / NUMBER_OF_TRIALS << "  9995: "
              << total9995 / NUMBER_OF_TRIALS << '\n';
}
```

The output of Listing 13.9 (badrand.cpp) shows that our pseudorandom number generator favors 5 over 9,995:

```
RAND_MAX = 32767
Trial # 1  5: 122295    9995: 91255
Trial # 2  5: 121789    9995: 91862
Trial # 3  5: 122440    9995: 91228
Trial # 4  5: 121602    9995: 91877
Trial # 5  5: 122599    9995: 91378
Trial # 6  5: 121599    9995: 91830
Trial # 7  5: 122366    9995: 91598
Trial # 8  5: 121839    9995: 91387
Trial # 9  5: 122295    9995: 91608
Trial #10  5: 121898    9995: 91519
-------------------
Averages for 10 trials:  5: 122072  9995: 91554.2
```

Figure 13.2 If shown in full, the table would contain 10,000 rows and 32,768 individual numbers. The values in each row are equivalent modulus 10,000. All the columns except the rightmost column contain 10,000 entries.



The first line verifies that the largest pseudorandom number that Visual C++ can produce through `rand` is 32,767. The next 10 lines that the program show the result of each trial, monitoring the activity of the one billion number generations. Since we are dealing with pseudorandom numbers, the results for each trial will not be exactly the same, but over one billion runs each they should be close. Note how consistent the results are among the runs.

While we expected both 5 and 9,995 to appear about the same number of times—each approximately 100,000 times—in fact the number 5 appeared consistently more that 100,000 times, averaging 122,072 times. The number 9,995 appeared consistently less than 100,000 times, averaging 91,554.2. Note that $\frac{122,072}{91,554.2} = 1.33$; this means the value 5 appeared 1.33 times more often than 9,995. Looking at it another way, $1.33 \approx \frac{4}{3}$, so for every four times the program produced a 5 it produced 9,995 only three times. As we soon will see, this ratio of 4:3 is not accidental.

Figure 13.2 shows why the expression `rand() % 10000` does not produce an even distribution.

Figure 13.2 shows an abbreviated list of all the numbers the `rand` function can produce before applying the modulus operation. If you add the missing rows that the ellipses represent, the table would contain 10,000 rows. All of the four values in each row are equivalent modulus 10,000; thus, for example,

$$2 = 10,002 = 20,002 = 30,002$$

and

$$1,045 = 11,045 = 21,045 = 31,045$$

Since the `rand` function cannot produce any values in the range $32,678 \ldots 39,999$, the rightmost column is not complete. Because the leftmost three columns are complete, the modulus operator can produce values in the range $0 \ldots 2,767$ four different ways; for example, the following code fragment

```
std::cout << 5 % 10000 << ' ' << 10005 & 10000 << ' '
          << 20005 % 10000 << ' ' << 30005 % 10000 << '\n';
```

prints

```
5 5 5 5
```

The `rand` function cannot return a value greater than 32,767; specifically, in our program above, `rand` cannot produce 39,995. Listing 13.9 (badrand.cpp), therefore, using `rand` and modulus we can produce 9,995 in only three different ways: 9,995, 19,995, and 29,995. Based on our analysis, Listing 13.9 (badrand.cpp) can generate the number 5 four different ways and 9,995 three different ways. This 4:3 ratio agrees with our empirical observations of the behavior of Listing 13.9 (badrand.cpp). The consequences of this bias means that values in the relatively small range $0 \ldots 2,767$ will appear disproportionately more frequently than numbers in the larger range $2,768 \ldots 9,999$. Such bias definitely is undesirable in a pseudorandom number generator.

We must use more sophisticated means to produce better pseudorandom numbers. Fortunately C++11 provides several standard classes that provide high-quality pseudorandom generators worthy of advanced scientific and mathematical applications.

The `rand` function itself has another weakness that makes it undesirable for serious scientific, engineering, and mathematical applications. `rand` uses a linear congruential generator algorithm (see `http://en.wikipedia.org/wiki/Linear_congruential_generator`). `rand` has a relatively small period. This means that the pattern of the sequence of numbers it generates will repeat itself exactly if you call `rand` enough times. For Visual C++, `rand`'s period is 2,147,483,648. Listing 13.10 (randperiod.cpp) verifies the period of `rand`.

Listing 13.10: `randperiod.cpp`

```
#include <iostream>
#include <iomanip>
#include <cstdlib>
#include <ctime>

int main() {
    // Set random number seed value
    srand(42);

    // Need to use numbers larger than regular integers; use long long ints
    for (long long i = 1; i < 4294967400LL; i++) {
       int r = rand();
       if (1 <= i && i <= 10)
          std::cout << std::setw(10) << i << ":" << std::setw(6) << r << '\n';
       else if (2147483645 <= i && i <= 2147483658)
          std::cout << std::setw(10) << i << ":" << std::setw(6) << r << '\n';
       else if (4294967293LL <= i && i <= 4294967309LL)
          std::cout << std::setw(10) << i << ":" << std::setw(6) << r << '\n';
    }
}
```

Listing 13.10 (randperiod.cpp) uses the C++ standard `long long int` integer data type because it needs to count above the limit of the `int` type, 2,147,483,647. The short name for `long long int` is just `long long`. Visual C++ uses four bytes to store both `int` and `long` types, so their range of values are identical. Under Visual C++, the type `long long` occupies eight bytes which allows the `long long` data type to span the range $-9,223,372,036,854,775,808 \ldots 9,223,372,036,854,775,807$. To represent a literal `long long` within C++ source code, we append the `LL` suffix, as in `12LL`. The expression `12` represents the `int` (4-byte version) of 12, but `12LL` represents the `long long` (8-byte version) of 12.

Listing 13.10 (randperiod.cpp) prints the first 10 pseudorandom numbers it generates, then it prints numbers 2,147,483,645 through 2,147,483,658. Finally the program prints its 4,294,96,729th through 4,294,967,309th pseudorandom numbers. Listing 13.10 (randperiod.cpp) displays

```
          1:   175
          2:   400
          3: 17869
          4: 30056
          5: 16083
          6: 12879
          7:  8016
          8:  7644
          9: 15809
         10:  1769
 2147483645: 25484
 2147483646: 21305
 2147483647:  6359
 2147483648:     0
 2147483649:   175
 2147483650:   400
 2147483651: 17869
 2147483652: 30056
 2147483653: 16083
 2147483654: 12879
 2147483655:  8016
 2147483656:  7644
 2147483657: 15809
 2147483658:  1769
 4294967293: 25484
 4294967294: 21305
 4294967295:  6359
 4294967296:     0
 4294967297:   175
 4294967298:   400
 4294967299: 17869
 4294967300: 30056
 4294967301: 16083
 4294967302: 12879
 4294967303:  8016
 4294967304:  7644
 4294967305: 15809
 4294967306:  1769
 4294967307: 32409
 4294967308: 29950
 4294967309: 13471
```

Notice that after 2,147,483,648 iterations the program begins to print the same numbers in the same sequen-

tial order as it began. 2,147,483,648 iterations later (after 4,294,967,296 total iterations) the sequence once again repeats. A careful observer could detect this repetition and thus after some time be able to predict the next pseudorandom value that the program would produce. A predictable pseudorandom number generator is not a good random number generator. Such a generator used in a game of chance would render the game perfectly predictable by clever (and patient!) players. A better pseudorandom number generator would have a much longer period.

The Mersenne twister (see `http://en.wikipedia.org/wiki/Mersenne_twister`) is a widely-used, high-quality pseudorandom number generator. It has a very long period, $2^{19,937} - 1$, which is approximately $4.3154 \times 10^{6,001}$. If an implementation of the Mersenne twister could generate 1,000,000,000 (one billion) pseudorandom numbers every second, a program that generated such pseudorandom numbers exclusively and did nothing else would need to run about $1.3684 \times 10^{5,985}$ years before it began to repeat itself. It is safe to assume that an observer will not be able to wait around long enough to be able to witness a repeated pattern in the sequence.

The standard C++ library contains the `mt19937` class from which programmers can instantiate objects used to generate pseudorandom numbers using the Mersenne twister algorithm. Generating the pseudorandom numbers is one thing, but ensuring that the numbers fall uniformly distributed within a specified range of values is another concern. Fortunately, the standard C++ library provides a multitude of classes that allow us to shape the production of an `mt19937` object into a mathematically sound distribution.

Our better pseudorandom generator consists of three pieces:

- an object that produces a random seed value,
- a pseudorandom number generator object that we construct with the random seed object, and
- a distribution object that uses the pseudorandom number generator object to produce a sequence of pseudorandom numbers that are uniformly distributed.

The C++ classes for these objects are

- The seed object is an instance of the `random_device` class.
- The pseudorandom number generator object is an instance of the `mt19937` class.
- The distribution object is an instance of the `uniform_int_distribution` class.

We use the `random_device` object in place of `srand`. The `mt19937` object performs the role of the `rand` function, albeit with much better characteristics. The `uniform_int_distribution` object constrains the pseudorandom values to a particular range, replacing the simple but problematic modulus operator. Listing 13.11 (highqualityrandom.cpp) upgrades Listing 13.9 (badrand.cpp) with improved random number generation is based on these classes.

Listing 13.11: `highqualityrandom.cpp`

```
#include <iostream>
#include <iomanip>
#include <random>

int main() {
    std::random_device rdev;    // Used to establish a seed value
    // Create a Mersenne Twister random number generator with a seed
    // value derived from rd
    std::mt19937 mt(rdev());
```

```
    // Create a uniform distribution object.  Given a random
    // number generator, dist will produce a value in the range
    // 0...9999.
    std::uniform_int_distribution<int> dist(0, 9999);

    // Total counts over all the runs.
    // Make these double-precision floating-point numbers
    // so the average computation at the end will use floating-point
    // arithmetic.
    double total5 = 0.0, total9995 = 0.0;

    // Accumulate the results of 10 trials, with each trial
    // generating 1,000,000,000 pseudorandom numbers
    const int NUMBER_OF_TRIALS = 10;

    for (int trial = 1; trial <= NUMBER_OF_TRIALS; trial++) {
        // Initialize counts for this run of a billion trials
        int count5 = 0, count9995 = 0;
        // Generate one billion pseudorandom numbers in the range
        // 0...9,999 and count the  number of times 5 and 9,995 appear
        for (int i = 0; i < 1000000000; i++) {
            // Generate a pseudorandom number in the range 0...9,999
            int r = dist(mt);
            if (r == 5)
                count5++;     // Number 5 generated, so count it
            else if (r == 9995)
                count9995++;  // Number 9,995 generated, so count it
        }
        // Display the number of times the program generated 5 and 9,995
        std::cout << "Trial #" << std::setw(2) << trial
                  << "  5: " << std::setw(6) << count5
                  << "    9995: " << std::setw(6) << count9995 << '\n';
        total5 += count5;         // Accumulate the counts to
        total9995 += count9995;   // average them at the end
    }
    std::cout << "-------------------" << '\n';
    std::cout << "Averages for " << NUMBER_OF_TRIALS << " trials:  5: "
              << std::setw(6) << total5 / NUMBER_OF_TRIALS << "  9995: "
              << std::setw(6) << total9995 / NUMBER_OF_TRIALS << '\n';
}
```

One run of Listing 13.11 (highqualityrandom.cpp) reports

```
Trial # 1  5:  99786    9995: 100031
Trial # 2  5:  99813    9995:  99721
Trial # 3  5:  99595    9995: 100144
Trial # 4  5: 100318    9995: 100243
Trial # 5  5:  99570    9995: 100169
Trial # 6  5:  99860    9995:  99724
Trial # 7  5:  99821    9995: 100263
Trial # 8  5:  99851    9995:  99887
Trial # 9  5: 100083    9995: 100204
Trial #10  5: 100202    9995:  99943
-------------------
Averages for 10 trials:  5: 99889.9  9995: 100033
```

During this particular program run we see that in 1,000,000,000 attempts the program generates the value 5 on average 99,889.9 times and generates 9,995 on average 100,033 times. Both of these counts approximately equal the expected 100,000 target. Examining the 10 trials individually, we see that neither the count for 5 nor the count for 9,995 is predisposed to be greater than or less than the other. In some trials the program generates 5 slightly less than 100,000 times, and in others it appears slightly greater than 100,000 times. The same is true for 9,995. These multiple trials show that in over 1,000,000,000 iterations the program consistently generates 5 approximately 100,000 times and 9,995 approximately 100,000 times. Listing 13.11 (highqualityrandom.cpp) shows us that the pseudorandom numbers generated by the Mersenne Twister object in conjunction with the distribution object are much better uniformly distributed than those produced by `rand` with the modulus operation.

Notice in Listing 13.11 (highqualityrandom.cpp) how the three objects work together:

- The `uniform_int_distribution` object produces a pseudorandom number from the `mt19937` generator object. The `mt19937` object generates a pseudorandom number, and the `uniform_int_distribution` object constrains this pseudorandom number to the desired range.
- Programmers create an `mt19937` object with a `random_device` object. The `random_device` object provides the seed value, potentially from a hardware source, to the `mt19937` generator object. We also can pass a fixed integer value to `mt19937`'s constructor if we want the generator to produce a perfectly reproducible sequence of values; for example, the following code fragment

  ```
  mt19937 gen(20);  // Use fixed seed instead of random_device
  uniform_int_distribution<int> dist(0, 9999);
  std::cout << dist(gen) << '\n';
  std::cout << dist(gen) << '\n';
  std::cout << dist(gen) << '\n';
  ```

  always prints

  ```
  7542
  6067
  6876
  ```

The use of `random_device`, `mt19937`, and `uniform_int_distribution` is a little more complicated than using `srand` and `rand` with the modulus operator, but the extra effort is worth it for many applications. This object-oriented approach is more modular because it allows us to substitute an object of a different pseudorandom number generator class in place of `mt19937` if we so choose. We also may swap out the normal distribution for a different distribution. Those familiar with probability theory may be familiar with a variety of different probability distributions, such as Bernoulli, Poisson, binomial, chi-squared, etc. The C++ standard library contains distribution classes that model all of these probability distributions and many more. Programmers can mix and match generator objects and distribution objects as needed to achieve specialized effects. While this flexibility is very useful and has its place, the `random_device`, `mt19937`, and `uniform_int_distribution` classes as used in Listing 13.11 (highqualityrandom.cpp) suffice for most applications needing to simulate random processes.

13.6 Exercises

1. Suppose `s` is a `std::string` object.

   (a) What expression represents the number of characters that make up `s`?

(b) What expression represents the first character in `s`?

(c) What statement would insert the character `'x'` onto the front of `s`?

(d) What statement would append the character `'x'` onto the back of `s`?

2. Compared to C strings (see Section 11.2.6), what advantages does the C++ `std::string` type offer to programmers?

3. A palindrome is a string of text that reads the same forwards and backwards. Examples include:

   - "TENET"
   - "STEP ON NO PETS"
   - "RATS LIVE ON NO EVIL STAR"

   All these strings are examples of strict palindromes where spacing and punctuation must reverse exactly. Complete the following function that determines if a given string is a strict palindrome. It should return true if the string is a strict palindrome and false otherwise.

   The empty string reads the same forward and backward, so it is a palindrome. Your function should ignore capitalization (that is, the string "Tenet" is regarded as a palindrome).

```
bool palindrome(const string& s) {
    // Your code goes here . . .
}
```

4. What is the class of the `std::cout` object?

5. What is the class of the `std::cin` object?

6. Suppose the user types in a line of text and presses the enter key. For example, the user might type the following:

   The sky is blue

   What C++ statement could you use to assign to a `std::string` variable named `msg` the complete line of text, including spaces?

7. Consider the following C++ program:

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

int main() {
    std::string w;
    std::ifstream x("results");
    while (x >> w)
        std::cout << '[' << w << "]\n";
}
```

   (a) Explain what the program accomplishes.

   (b) What does the variable `x` represent?

   (c) As it currently is written, what does the program expect that may not be true?

   (d) What would cause the program to print nothing?

   (e) How can you improve the program so that it always provides feedback to the user, even it the expectation in part #7c?

Chapter 14

Custom Objects

In earlier times programmers wrote software in the machine language of the computer system because compilers had yet to be invented. The introduction of variables in association with higher-level programming languages marked a great step forward in the late 1950s. No longer did programmers need to be concerned with the lower-level details of the processor and absolute memory addresses. Named variables and functions allow programmers to abstract away such machine-level details and concentrate on concepts such as integers and characters that transcend computer electronics. Objects provide a level of abstraction above that of simple variables. Objects allow programmers to go beyond simple values—developers can focus on more complex things like geometric shapes, bank accounts, and aircraft wings. Programming objects that represent these real-world things can possess capabilities that go far beyond the simple variables we have studied to this point.

A C++ object typically consists of a collection of data and code. By bundling data and code together, objects store information and provide services to other parts of the software system. An object forms a computational unit that makes up part of the overall computer application. A programming object can model a real-world object more naturally than can a collection of simple variables since it can encapsulate considerable complexity. Objects make it easier for developers to build complex software systems.

C++ is classified as an object-oriented language. Most modern programming languages have some degree of object orientation. This chapter shows how programmers can define, create, and use custom objects.

14.1 Object Basics

Consider the task of dealing with geometric points. Mathematicians represent a single point as an ordered pair of real numbers, usually expressed as (x, y). In C++, the `double` type serves to approximate a subset of the mathematical real numbers. We can model a point with coordinates within the range of double-precision floating-point numbers with two `double` variables. We may consider a point as one thing conceptually, but we here we would be using two variables. As a consequence, a function that computes the distance between two points requires four parameters—x_1, y_1, x_2, and y_2—rather than two points—(x_1, y_1) and (x_2, y_2). Ideally, we should be able to use one variable to represent a point.

One approach to represent a point could use a two-element vector, for example:

```
std::vector<double> pt { 3.2, 0.0 };
```

This approach has several problems:

- We must use numeric indices instead of names to distinguish between the two components of a point object. We may agree that `pt[0]` means the x coordinate of point `pt` and `pt[1]` means the y coordinate of point `pt`, but the compiler is powerless to detect the error if a programmer uses an expression like `pt[19]` or `pt[-3]`.
- We cannot restrict the vector's size to two. A programmer may accidentally push extra items onto the back of a vector meant to represent a point object. The compiler could not defend against a program treating an empty vector as a point object.
- We cannot use a vector to represent objects in general. Consider a bank account object. A bank account object could include, among many other diverse things, an account number (an integer), a customer name (a string), and an interest rate (a double-precision floating-point number). A vector implementation of such an object is impossible because the elements in a vector must all be of the same type.

In addition to storing data, we want our objects to be active agents that can do computational tasks. We need to be able associate code with a class of objects. We need a fundamentally different programming construct to represent objects.

Before examining how C++ specifically handles objects, we first will explore what capabilities are desirable. Consider an automobile. An automobile user—the driver—uses the car for transportation. The user's interface to the car is fairly simple, considering an automobile's overall complexity. A driver provides input to the car via its steering wheel, accelerator and brake pedals, turn signal control, shift lever, etc. The automobile produces output to the driver with its speedometer, tachometer, various instrument lights and gauges, etc. These standardized driver-automobile interfaces enable an experienced driver to drive any modern car without the need for any special training for a particular make or model.

The typical driver can use a car very effectively without understanding the details of how it works. To drive from point A to point B a driver does not need to know the number of cylinders in the engine, the engine's horsepower, or whether the vehicle is front-wheel drive or rear-wheel drive. A driver may look under the hood at the engine, but the driver cannot confirm any details about what is *inside* the engine itself without considerable effort or expense. Many details are of interest only to auto enthusiasts or mechanics. There may be only a select few automotive engineers capable of understanding and appreciating other more esoteric details about the vehicle's design and implementation.

In some ways programming objects as used in object-oriented programming languages are analogous to automobile components. An object may possess considerable capability, but a programmer using the object needs to know only *what* the object can do without needing to know *how* it works. An object provides an interface to any client code that wishes to use that object. A typical object selectively exposes some parts of itself to clients and keeps other parts hidden from clients. The object's designer, on the other hand, must know the complete details of the object's implementation and must be an expert on both the *what* the object does and *how* it works.

Programmers define the structure of a new type of object using one of two keywords: `struct` or `class`. The two constructs are very similar. We will use the `class` construct in this chapter, and we will consider `struct`s in Section 15.9.

A class serves as a pattern or template from which an executing program may produce objects. In this chapter we will concentrate on four things facilitating object-oriented programming with C++ classes:

1. specifying the data that constitute an object's state,

2. defining the code to be executed on an object's behalf that provides services to clients that use the object,
3. defining code that automatically initializes a newly-created object ensuring that it begins its life in a well-defined state, and
4. specifying which parts of objects are visible to clients and which parts are hidden from clients.

A class is a programmer-defined type. An object is an *instance* of a class. The terms *object* and *instance* may be used interchangeably.

14.2 Instance Variables

The simplest kind of object stores only data. We can define a mathematical point type as follows:

```
class Point {
public:
    double x;
    double y;
};
```

Notice the semicolon that follows the close curly brace of the class definition. This semicolon is required, but it is easy to forget. By convention class names begin with a capital letter, but class names are just identifiers like variable names and function names. Here, our class name is `Point`. The body of the class appears within the curly braces.

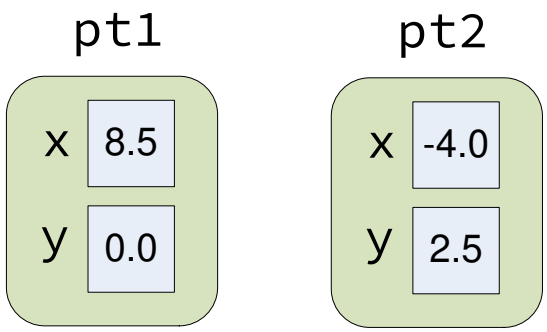
The elements declared within a class are known as *members* of the class. The `Point` class specifies two double-precision floating-point data components named `x` and `y`. These components are known as *instance variables*. The C++ community often refers to these as *member data* or *data members*. Other names for instance variables include *fields* and *attributes*. The declarations for `x` and `y` appear within the class body after the `public` label. We say that `x` and `y` are *public* members of the `Point` class; this means client code using a `Point` object has full access to the object's `x` and `y` fields. Any client may examine and modify the `x` and `y` components of a `Point` object.

Once this `Point` class definition is available, a client can create and use `Point` objects as shown in Listing 14.1 (mathpoints.cpp).

Listing 14.1: `mathpoints.cpp`

```
#include <iostream>

// The Point class defines the structure of software
// objects that model mathematical, geometric points
class Point {
public:
    double x;   // The point's x coordinate
    double y;   // The point's y coordinate
};

int main() {
    // Declare some point objects
    Point pt1, pt2;
    // Assign their x and y fields
```

Figure 14.1 Two Point objects with their individual data fields




```
    pt1.x = 8.5; // Use the dot notation to get to a part of the object
    pt1.y = 0.0;
    pt2.x = -4;
    pt2.y = 2.5;
    // Print them
    std::cout << "pt1 = (" << pt1.x << "," << pt1.y << ")\n";
    std::cout << "pt2 = (" << pt2.x << "," << pt2.y << ")\n";
    // Reassign one point from the other
    pt1 = pt2;
    std::cout << "pt1 = (" << pt1.x << "," << pt1.y << ")\n";
    std::cout << "pt2 = (" << pt2.x << "," << pt2.y << ")\n";
    // Are pt1 and pt2 aliases?  Change pt1's x coordinate and see.
    pt1.x = 0;
    std::cout << "pt1 = (" << pt1.x << "," << pt1.y << ")\n";
    // Note that pt2 is unchanged
    std::cout << "pt2 = (" << pt2.x << "," << pt2.y << ")\n";

}
```

Listing 14.1 (mathpoints.cpp) prints

```
pt1 = (8.5,0)
pt2 = (-4,2.5)
pt1 = (-4,2.5)
pt2 = (-4,2.5)
pt1 = (0,2.5)
pt2 = (-4,2.5)
```

It is important to note that `Point` is *not* an object. It represents a class of objects. It is a type. The variables `pt1` and `pt2` are the objects, or instances, of the class `Point`. Each of the objects `pt1` and `pt2` has its own copies of fields named `x` and `y`. Figure 14.1 provides a conceptual view of point objects `pt1` and `pt2`.

Double-precision floating-point numbers on most systems require eight bytes of memory. Since each `Point` object stores two `double`s, a `Point` object uses at least 16 bytes of memory. In practice, an object may be slightly bigger than the sum its individual components because most computer architectures restrict

how data can be arranged in memory. This means some objects include a few extra bytes for "padding." We can use the `sizeof` operator to determine the exact number of bytes an object occupies. Under Visual C++, the expression `sizeof pt1` evaluates to 16.

A client may use the dot (`.`) operator with an object to access one of the object's members. The expression `pt1.x` represents the *x* coordinate of object `pt1`. The dot operator is a binary operator; its left operand is an expression representing a class instance (object), and its right operand is the name of a member of the class.

The assignment statement in Listing 14.1 (mathpoints.cpp)

```
pt1 = pt2;
```

and the statements that follow demonstrate that we may assign one object to another directly without the need to copy each individual member of the object. The above assignment statement accomplishes the following:

```
pt1.x = pt2.x;   // No need to assignment this way;
pt1.y = pt2.y;   // direct object assignment does this
```

As another example, suppose we wish to implement a simple bank account object. We determine that the necessary information for each account consists of a name, ID number, and a balance (amount of money in the account). We can define our bank account class as

```
class Account {
public:
    std::string name;  // The name of the account's owner
    int id;            // The account number
    double balance;    // The current balance
};
```

The `name`, `id`, and `balance` fields constitute the `Account` class, and the fields represent three different types.

We can define a vector that holds instances of our `Account` class as easily as

```
std::vector<Account> accounts(5000);
```

Here the `accounts` variable represents a sequence of 5,000 bank account objects.

Listing 14.2 (bankaccount.cpp) is a simple program that uses `Account` objects.

Listing 14.2: bankaccount.cpp

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

class Account {
public:
    // String representing the name of the account's owner
    std::string name;
    // The account number
    int id;
    // The current account balance
    double balance;
};
```

```
// Allows the user to enter via the keyboard information
// about an account and adds that account to the database.
void add_account(std::vector<Account>& accts) {
    std::string name;
    int number;
    double amount;

    std::cout << "Enter name, account number, and account balance: ";
    std::cin >> name >> number >> amount;
    Account acct;
    acct.name = name;
    acct.id = number;
    acct.balance = amount;
    accts.push_back(acct);
}

// Print all the accounts in the database
void print_accounts(const std::vector<Account>& accts) {
    int n = accts.size();
    for (int i = 0; i < n; i++)
        std::cout << accts[i].name << "," << accts[i].id
                  << "," << accts[i].balance << '\n';
}

void swap(Account& er1, Account& er2) {
    Account temp = er1;
    er1 = er2;
    er2 = temp;
}

bool less_than_by_name(const Account& e1, const Account& e2) {
    return e1.name < e2.name;
}


bool less_than_by_id(const Account& e1, const Account& e2) {
    return e1.id < e2.id;
}

bool less_than_by_balance(const Account& e1, const Account& e2) {
    return e1.balance < e2.balance;
}

// Sorts a bank account database into ascending order
// The comp parameter determines the ordering
void sort(std::vector<Account>& db,
          bool (*comp)(const Account&, const Account&)) {
    int size = db.size();
    for (int i = 0; i < size - 1; i++) {
        int smallest = i;
        for (int j = i + 1; j < size; j++)
            if (comp(db[j], db[smallest]))
                smallest = j;
        if (smallest != i)
```

```
            swap(db[i], db[smallest]);
    }
}


// Allows a user interact with a bank account database.
int main() {
    // The simple database of bank accounts
    std::vector<Account> customers;

    // User command
    char cmd;

    // Are we done yet?
    bool done = false;

    do {
        std::cout << "[A]dd [N]ame [I]D [B]alance [Q]uit==> ";
        std::cin >> cmd;
        switch (cmd) {
            case 'A':
            case 'a':
                // Add an account
                add_account(customers);
                break;
            case 'P':
            case 'p':
                // Print customer database
                print_accounts(customers);
                break;
            case 'N':
            case 'n':
                // Sort database by name
                sort(customers, less_than_by_name);
                print_accounts(customers);
                break;
            case 'I':
            case 'i':
                // Sort database by ID (account number)
                sort(customers, less_than_by_id);
                print_accounts(customers);
                break;
            case 'B':
            case 'b':
                // Sort database by account balance
                sort(customers, less_than_by_balance);
                print_accounts(customers);
                break;
            case 'Q':
            case 'q':
                done = true;
                break;
        }
    }
    while (!done);
```

```
}
```

Listing 14.2 (bankaccount.cpp) stores bank account objects in a vector (see Section 11.1). Also, a bank account object contains a `std::string` object as a field. This shows that objects can contain other objects and implies that our objects can have arbitrarily complex structures.

A sample run of Listing 14.2 (bankaccount.cpp) prints

```
[A]dd [N]ame [I]D [B]alance [Q]uit==> a
Enter name, account number, and account balance: Sheri 34 100.34
[A]dd [N]ame [I]D [B]alance [Q]uit==> a
Enter name, account number, and account balance: Mary 10 1323.00
[A]dd [N]ame [I]D [B]alance [Q]uit==> a
Enter name, account number, and account balance: Larry 88 55.05
[A]dd [N]ame [I]D [B]alance [Q]uit==> a
Enter name, account number, and account balance: Terry 33 423.50
[A]dd [N]ame [I]D [B]alance [Q]uit==> a
Enter name, account number, and account balance: Gary 11 7.27
[A]dd [N]ame [I]D [B]alance [Q]uit==> n
Gary,11,7.27
Larry,88,55.05
Mary,10,1323
Sheri,34,100.34
Terry,33,423.5
[A]dd [N]ame [I]D [B]alance [Q]uit==> i
Mary,10,1323
Gary,11,7.27
Terry,33,423.5
Sheri,34,100.34
Larry,88,55.05
[A]dd [N]ame [I]D [B]alance [Q]uit==> b
Gary,11,7.27
Larry,88,55.05
Sheri,34,100.34
Terry,33,423.5
Mary,10,1323
[A]dd [N]ame [I]D [B]alance [Q]uit==> q
```

The program allows users to sort the bank account database in several different ways using different comparison functions. Notice that the `less_than_by_name` and similar comparison functions use `const` reference parameters for efficiency (see Section 11.1.4).

Observe that the `add_account` function has a local variable named `name`. All `Account` objects have a field named `name`. The `add_account` function uses the `name` identifier in both ways without a problem. The compiler can distinguish between the two uses of the identifier because one is qualified with an object variable before the dot (`.`) operator and the other is not; that is, `acct.name` refers to the data member `name` in the `acct` object, while the variable `name` by itself is the local variable. Despite their similar names, the data to which `acct.name` and `name` refer live in two completely different memory locations.

14.3 Member Functions

The classes we have developed so far, `Point` and `Account`, have been passive entities that have no built-in functionality. In addition to defining the structure of the data for its objects, a class can define functions that operate on behalf of its objects.

Recall the bank account class, `Account`, from Section 14.2:

```
class Account {
public:
    // String representing the name of the account's owner
    string name;
    // The account number
    int id;
    // The current account balance
    double balance;
};
```

Suppose this design is given to programmer Sam who must write a `withdraw` function. Sam is a careful programmer, so his `withdraw` function checks for overdrafts:

```
/*****************************************************
 *  withdraw(acct, amt)
 *     Deducts amount amt from Account acct, if possible.
 *     Returns true if successful; otherwise, it returns false.
 *     A call can fail if the withdraw would
 *     cause the balance to fall below zero
 *
 *     acct: a bank account object
 *     amt:  funds to withdraw
 *
 *     Author: Sam Coder
 *     Date: April 17, 2018
 *****************************************************/
bool withdraw(Account& acct, double amt) {
    bool result = false;  // Unsuccessful by default
    if (acct.balance - amt >= 0) {
        acct.balance -= amt;
        result = true;  // Success
    }
    return result;
}
```

The following code fragment shows the expected code a client might write when withdrawing funds from the bank:

```
// Good client code
// ----------------
// A simple database of bank customers
std::vector<Account> accountDB;

// Populate the database via some function
```

```
accountDB = initialize_db();

// Get transaction information
int number;
double debit;
std::cout << "Please enter account number and amount to withdraw:";
std::cin >> number >> debit;

// Locate account index
int n = accountDB.size(), i = 0;
while (i < n) {
    if (accountDB[i].id == number)
        break;  // Found the account
    i++;
}

// Perform the transaction, if possible
if (i < n) {
    if (withdraw(accountDB[i], debit))
        std::cout << "Withdrawal successful\n";
    else
        std::cout << "Cannot perform the withdraw\n";
}
else
    std::cout << "Account does not exist\n";
```

Unfortunately, nothing prevents a client from being sloppy:

```
// Bad client code
// ----------------
// A simple database of bank customers
std::vector<Account> accountDB;

// Populate the database via some function
accountDB = initialize_db();


// Get transaction information
int number;
double debit;
std::cout << "Please enter account number and amount to withdraw:";
std::cin >> number >> debit;

// Locate account index
int n = accountDB.size(), i = 0;
while (i < n) {
    if (accountDB[i].id == number)
        break;  // Found the account
    i++;
}
```

```
// Perform the transaction
if (i < n)
    accountDB[i] -= debit;    // What if debit is too big?
else
    std::cout << "Account does not exist\n";
```

Nothing in an `Account` object itself can prevent an overdraft or otherwise prevent clients from improperly manipulating the balance of an account object.

We need to be able to protect the internal details of our bank account objects and yet permit clients to interact with them in a well-defined, controlled manner.

Consider a non-programming example. If I deposit $1,000.00 dollars into a bank, the bank then has custody of my money. It is still my money, so I theoretically can reclaim it at any time. The bank stores money in its safe, and my money is in the safe as well. Suppose I wish to withdraw $100 dollars from my account. Since I have $1,000 total in my account, the transaction should be no problem. What is wrong with the following scenario:

1. Enter the bank.
2. Walk past the teller into a back room that provides access to the safe.
3. The door to the safe is open, so enter the safe and remove $100 from a stack of $20 bills.
4. Exit the safe and inform a teller that you got $100 out of your account.
5. Leave the bank.

This is not the process a normal bank uses to handle withdrawals. In a perfect world where everyone is honest and makes no mistakes, all is well. In reality, many customers might be dishonest and intentionally take more money than they report. Even though I faithfully counted out my funds, perhaps some of the bills were stuck to each other and I made an honest mistake by picking up six $20 bills instead of five. If I place the bills in my wallet with other money already there, I may never detect the error. Clearly a bank needs a more controlled procedure for handling customer withdrawals.

When working with programming objects, in many situations it is better to restrict client access to the internals of an object. Client code should not be able to change directly bank account objects for various reasons, including:

- A withdrawal should not exceed the account balance.
- Federal laws dictate that deposits above a certain amount should be reported to certain government agencies, so a bank would not want customers to be able to add funds to an account in a way to circumvent this process.
- An account number should never change for a given account for the life of that account.

How do we protect the internal details of our bank account objects and yet permit clients to interact with them in a well-defined, controlled manner? The trick is to hide completely from clients the object's fields and provide special functions called *member functions* or *methods* that have access to the hidden fields. These methods provide the only means available to clients of changing the object's internal state.

In the following revised `Account` class:

```
class Account {
    // String representing the name of the account's owner
    string name;
    // The account number
    int id;
    // The current account balance
    double balance;
public:
    // Methods will be added here . . .
};
```

all the fields no longer reside in the public section of the class definition. This makes the following client code impossible:

```
Account acct;
// Set the account's balance
acct.balance = 100;  // Illegal, compiler reports an error
// Withdraw some funds
acct.balance -= 20;  // Illegal, compiler reports an error
```

The `balance` field is in the private area of the class.

You may use the literal `private` label, as in

```
class Account {
private:
    // String representing the name of the account's owner
    string name;
    // The account number
    int id;
    // The current account balance
    double balance;
public:
    // Methods will be added here . . .
};
```

Because any parts of a class not explicitly labeled are implicitly private, the `private` label is not necessary if you place all the private members in the first unlabeled section of the class. Said another way, all members of a class are automatically private unless otherwise labeled. Some programmers like to put the public members before the private members within a class definition, as in

```
class Account {
public:
    // Methods will be added here . . .
private:
    // String representing the name of the account's owner
    string name;
    // The account number
    int id;
    // The current account balance
    double balance;
};
```

In this case the `private` label is necessary.

In order to enforce the spirit of the `withdraw` function, we will make it a method, and add a `deposit` method to put funds into an account. Listing 14.3 (newaccount.cpp) enhances the `Account` class with methods.

Listing 14.3: `newaccount.cpp`

```
class Account {
    // String representing the name of the account's owner
    string name;
    // The account number
    int id;
    // The current account balance
    double balance;
public:
    /*******************************************************
     *  deposit(amt)
     *     Adds amount amt to the account's balance.
     *
     *     Author: Sam Coder
     *     Date: April 17, 2018
     *******************************************************/
    void deposit(double amt) {
        balance += amt;
    }

    /*******************************************************
     *  withdraw(amt)
     *     Deducts amount amt from the account's balance,
     *     if possible.
     *     Returns true if successful; otherwise, it returns false.
     *     A call can fail if the withdraw would
     *     cause the balance to fall below zero
     *
     *     amt:  funds to withdraw
     *
     *     Author: Sam Coder
     *     Date: April 17, 2018
     *******************************************************/
    bool withdraw(double amt) {
        bool result = false;  // Unsuccessful by default
        if (balance - amt >= 0) {
            balance -= amt;
            result = true;  // Success
        }
        return result;
    }
};
```

In this new definition of `Account`, the members named `deposit` and `withdraw` are not fields; they are method definitions. A method definition looks like a function definition, but it appears within a class definition. Because of this, a method is also known as a *member function*.

A client accesses a method with the dot (`.`) operator:

```
// Withdraw money from the Account object named acct
acct.withdraw(100.00);
```

The `withdraw` method definition uses three variables: `amt`, `result`, and `balance`. The variables `amt` and `result` are local to `withdraw`—`amt` is the method's parameter, and `result` is declared within the body of `withdraw`. Where is `balance` declared? It is the field declared in the private section of the class. The `withdraw` method affects the `balance` field of the object upon which it is called:

```
// Affects the balance field of acct1 object
acct1.withdraw(100.00);
// Affects the balance field of acct2 object
acct2.withdraw(25.00);
```

When you see a variable used within the code of a method definition, it can be one of several kinds of variables. The compiler establishes the exact nature of the variable in the following order:

1. Method parameter—If the variable is declared in the method's parameter list, it is a parameter to the method. As in the case of free functions, a method parameter is a variable local to that method.
2. Local variable—If the variable is declared in the body of the method, the variable is local to the method. Do anything you please with the local variable, and it will not affect any variables outside of that function.
3. Instance variable—If the variable is not a parameter to the method, is not declared within the method body, but is declared as an instance variable within the class, it is an instance variable of the object the client used to invoke the method.
4. Global variable—If the variable is not a parameter to the method, is not declared within the method body, and is not declared as an instance variable within the class, it must be a global variable. If no such global variable exists, the variable is undeclared, and the compiler will report the error.

It is important to note that the compiler checks in this order. That means it is legal to give a parameter to a method or a local variable the same name as an instance variable within the class. In this case the method's code cannot access the instance variable or global variable by using its simple name. We say the local variables *hide* the instance or global variables from view. Section 15.3 shows how we can gain access to these hidden variables.

Methods may be overloaded just like global functions (see Section 10.3). This means multiple methods in the same class may have the same names, but their signatures must be different. Recall that a function's signature consists of its name and parameter types; a method's signature too consists of its name and parameter types.

We saw in Section 14.2 that each object provides storage space for its own data fields. An object does not require any space for its methods. This means the only things about an individual object that an executing program must maintain are the object's fields. While the exact organization of memory varies among operating systems, all the data processed by a program appears in one of three sections: stack, heap, or static memory. As with simple data types like `int`s, the fields of an object declared local to a function or method reside in the segment of memory known as the stack. Also like simple data types, the fields of an object allocated with the `new` operator appear on the heap. The fields in global and `static` local objects reside in the static section of the executing program's memory.

Consider the following simple class:

```
class Counter {
```

```
    int count;
public:
    // Allow clients to reset the counter to zero
    void clear() {
        count = 0;
    }

    // Allow clients to increment the counter
    void inc() {
        count++;
    }

    // Return a copy of the counter's current value
    int get() {
        return count;
    }
};
```

For this code we see that each `Counter` object will store a single integer value (its `count` field). Under Visual C++ `sizeof Counter` is the same as `sizeof int`—that is, four. A local `Counter` object consumes four bytes of stack space, a global `Counter` object uses four bytes of static memory, and a dynamically-allocated `Counter` object uses four bytes of heap space.

In addition to static, stack, and heap memory used for data, executing programs reserve a section of memory known as the *code segment* which stores the machine language for all the program's functions and methods. The compiler translates methods into machine language as it does regular functions. Internally, the method `inc` in the `Counter` class is identified by a longer name, `Counter::inc`. Although `Counter::inc` is a method, in the compiled code it works exactly like a normal function unrelated to any class. In the client code

```
Counter ctr1, ctr2;  // Declare a couple of  Counter objects
ctr1.clear();  // Reset the counters to zero
ctr2.clear();
ctr1.inc();    // Increment the first counter
```

the statement

```
ctr1.clear();
```

sets the private `count` field of `ctr1` to zero, while the statement

```
ctr2.clear();
```

sets `ctr2`'s `count` field to zero. Since all `Counter` objects share the same `reset` method, how does each call to `Counter::clear` reset the field of the proper `Counter` object? The trick is this: While it appears that the `reset` method of the `Counter` class accepts no parameters, it actually receives a secret parameter that corresponds to the address of the object on left side of the dot. The C++ source code statement

```
ctr1.clear();   // Actual C++ code
```

internally is treated like

```
Counter::clear(&ctr1);   //  <-- Not real C++ code!  Do not write this!
```

in the compiled code. The code within the method can influence the field of the correct object via the pointer it receives from the caller. The `clear` method contains the single statement

```
count = 0;
```

Since the `count` variable is not declared locally within the `clear` method and it is not a global variable, it must be the field named `count` of the object pointed to by the secret parameter passed to `clear`. This means the call

```
ctr1.clear();    // Actual C++ code
```

thus modifies the `count` instance variable of the `ctr1` counter object.

Section 15.3 shows how programmers can access this secret pointer passed to methods.

14.4 Constructors

One crucial piece still is missing. How can we make sure the fields of an object have reasonable initial values before a client begins using the object? A class may define a *constructor* that ensures an object will begin in a well-defined sate. A constructor definition looks similar to a method definition. The code within a constructor executes on behalf of an object when a client creates the object. For some classes, the client can provide information for the constructor to use when initializing the object. As with functions and methods, class constructors may be overloaded. Listing 14.4 (bankaccountmethods.cpp) exercises an enhanced `Account` class that offers `deposit` and `withdraw` methods, as well as a constructor.

Listing 14.4: `bankaccountmethods.cpp`

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

class Account {
    // String representing the name of the account's owner
    std::string name;
    // The account number
    int id;
    // The current account balance
    double balance;
public:
    // Initializes a bank account object
    Account(const std::string& customer_name, int account_number,
            double amount):
            name(customer_name), id(account_number), balance(amount) {
       if (amount < 0) {
           std::cout << "Warning: negative account balance\n";
           balance = 0.0;
       }
    }

    // Adds amount amt to the account's balance.
    void deposit(double amt) {
        balance += amt;
    }
```

```
    // Deducts amount amt from the account's balance,
    // if possible.
    // Returns true if successful; otherwise, it returns false.
    // A call can fail if the withdraw would
    // cause the balance to fall below zero
    bool withdraw(double amt) {
        bool result = false;  // Unsuccessful by default
        if (balance - amt >= 0) {
            balance -= amt;
            result = true;  // Success
        }
        return result;
    }

    // Displays information about the account object
    void display() {
        std::cout << "Name: " << name << ", ID: " << id
                  << ", Balance: " << balance << '\n';
    }
};


int main() {
    Account acct1("Joe", 2312, 1000.00);
    Account acct2("Moe", 2313, 500.29);
    acct1.display();
    acct2.display();
    std::cout << "---------------------" << '\n';
    acct1.withdraw(800.00);
    acct2.deposit(22.00);
    acct1.display();
    acct2.display();
}
```

Listing 14.4 (bankaccountmethods.cpp) produces

```
Name: Joe, ID: 2312, Balance: 1000
Name: Moe, ID: 2313, Balance: 500.29
---------------------
Name: Joe, ID: 2312, Balance: 200
Name: Moe, ID: 2313, Balance: 522.29
```

The following characteristics differentiate a constructor definition from a regular method definition:

- A constructor has the same name as the class.
- A constructor has no return type, not even `void`.

The constructor in Listing 14.4 (bankaccountmethods.cpp) initializes all the fields with values supplied by the client. The comma-separated list between the colon and the curly brace that begins the body of the constructor is called the *constructor initialization list*. An initialization list contains the name of each field with its initial value in parentheses. All of the fields that make up an object must be initialized before the body of the constructor executes. In this case the code within the constructor adjusts the balance to zero and issues a warning if a client attempts to create an account with an initial negative balance. The constructor

thus ensures an account object's balance can never begin with a negative value. The `withdraw` method ensures that, once created, an `Account` object's balance will never be negative. Notice that the client provides the required constructor parameters at the point of the object's declaration:

```
// Client creating two Account objects
Account acct1("Joe", 2312, 1000.00);
Account acct2("Moe", 2313, 500.29);
```

Since the `Account` class contains a constructor definition which requires arguments, it now is impossible for a client to declare an `Account` object like

```
Account acct3;  // Illegal, must supply arguments for constructor
```

With constructors, unlike with normal methods, we can use curly braces in place of parentheses, as in

```
// Client creating two Account objects
Account acct1{"Joe", 2312, 1000.00};
Account acct2{"Moe", 2313, 500.29};
```

If `acct` is a `Account` object defined as in Listing 14.4 (bankaccountmethods.cpp), the following code is illegal:

```
acct.balance -= 100;  // Illegal, balance is private
```

Clients do not have direct access to the `balance` field since it is private. Clients instead should use the appropriate method call to adjust the balance of an `Account` object:

```
Account acct("Joe", 1033, 50.00); // New bank account object
acct.deposit(1000.00);  // Add some funds
acct.withdraw(2000.00); // Method should disallow this operation
```

The details of depositing and withdrawing funds are the responsibility of the object itself, not the client code. The attempt to withdraw the $2,000 dollars above would not change the account's balance, and the client can check the return value of `withdraw` to provide appropriate feedback to the user about the error. The program then could take steps to correct the situation.

A constructor that specifies no parameters is called a *default constructor*. If the programmer does not specify any constructor for a class, the compiler will provide a default constructor that does nothing. If the programmer defines any constructor for a class, the compiler will not generate a default constructor. See Section 15.2 for the consequences of this constructor policy.

If you do not define a constructor for your class, the compiler automatically will create one for you—a default constructor that accepts no parameters. The compiler-generated constructor does not do anything to affect the state of newly created instances.

If you define any constructor for your class, the compiler will not provide a default constructor.

14.5 Defining a New Numeric Type

C++'s class feature allows us to define our own complete types. We will define a new numeric type that models mathematical rational numbers. In mathematics, a *ratio*nal number is defined as the ratio of two integers, where the second integer must be nonzero. Commonly called a *fraction*, a rational number's two integer components are called *numerator* and *denominator*. Rational numbers possess certain properties; for example, two fractions can have different numerators and denominators but still be considered equal ($\frac{1}{2} = \frac{2}{4}$). Listing 14.5 (simplerational.cpp) shows how we can define and use rational numbers.

Listing 14.5: `simplerational.cpp`

```
#include <iostream>
#include <cstdlib>

// Models a mathematical rational number
class SimpleRational {
    int numerator;
    int denominator;
public:
    // Initializes the components of a Rational object
    SimpleRational(int n, int d): numerator(n), denominator(d) {
        if (d == 0) {
            // Display error message
            std::cout << "Zero denominator error\n";
            exit(1);   // Exit the program
        }
    }

    // The default constructor makes a zero rational number
    // 0/1
    SimpleRational(): numerator(0), denominator(1) {}

    // Allows a client to reassign the numerator
    void set_numerator(int n) {
        numerator = n;
    }

    // Allows a client to reassign the denominator.
    // Disallows an illegal fraction (zero denominator).
    void set_denominator(int d) {
        if (d != 0)
            denominator = d;
        else {
            // Display error message
            std::cout << "Zero denominator error\n";
            exit(1);   // Exit the program
        }
    }

    // Allows a client to see the numerator's value.
    int get_numerator() {
        return numerator;
    }
```

```
    // Allows a client to see the denominator's value.
    int get_denominator() {
        return denominator;
    }
};

// Returns the product of two rational numbers
SimpleRational multiply(SimpleRational  f1, SimpleRational f2) {
    return {f1.get_numerator() * f2.get_numerator(),
            f1.get_denominator() * f2.get_denominator()};
}

void print_fraction(SimpleRational f) {
    std::cout << f.get_numerator() << "/" << f.get_denominator();
}

int main() {
    SimpleRational fract(1, 2);  // The fraction 1/2
    std::cout << "The fraction is ";
    print_fraction(fract);
    std::cout << '\n';
    fract.set_numerator(19);
    fract.set_denominator(4);
    std::cout << "The fraction now is ";
    print_fraction(fract);
    std::cout << '\n';

    // Alternate syntax uses {} with constructor instead of ()
    SimpleRational fract1{1, 2}, fract2{2, 3};
    auto prod = multiply(fract1, fract2);
    std::cout << "The product of ";
    print_fraction(fract1);
    std::cout << " and ";
    print_fraction(fract2);
    std::cout << " is ";
    print_fraction(prod);
    std::cout << '\n';
}
```

Listing 14.5 (simplerational.cpp) prints the following:

```
The fraction is 1/2
The fraction now is 19/4
The product of 1/2 and 2/3 is 2/6
```

The `SimpleRational` class defines a new numeric type that C++ does not natively provide—the *rational number* type. (It is named `SimpleRational` because it is our first cut at a rational number class; a better version is to come in Listing 16.1 (rational.cpp).) This `SimpleRational` class defines two overloaded constructors. One constructor accepts the numerator and denominator values from the client. The other constructor allows a client to declare a `SimpleRational` object as

```
SimpleRational frac;
```

without supplying the initial numerator and denominator values. This default constructor assigns $\frac{0}{1}$ to the object. Both constructors ensure that a `SimpleRational` object's denominator will not be zero.

The function `multiply`:

```
SimpleRational multiply(SimpleRational  f1, SimpleRational f2) {
    return {f1.get_numerator() * f2.get_numerator(),
            f1.get_denominator() * f2.get_denominator()};
}
```

accepts two `SimpleRational` objects as arguments and returns a `SimpleRational` result. It is a free function, not a method (or member function) of the `SimpleRational` class. The function's return statement requires special attention. Because of the function's declaration the compiler knows the function returns a `SimpleRational` object. The expression within curly braces after the `return` keyword:

```
{f1.get_numerator() * f2.get_numerator(),
 f1.get_denominator() * f2.get_denominator()}
```

is exactly the expression we can pass to the compiler to create a `SimpleRational` object. The `return` statement therefore creates a fraction object with a numerator that is the product of the numerators of the objects supplied by the caller. Similarly, the denominator of the returned object is the product of the denominators of the function's parameters.

We can use this curly brace initialization syntax when passing parameters to functions; for example, we can simplify the following statements:

```
SimpleRational fract{2, 3};
print_fraction(fract);
```

into a single statement:

```
print_fraction({2, 3});
```

The compiler knows that `print_fraction` accepts a single `SimpleRational` object as a parameter. The compiler generates code that uses the caller's curly brace initialization list to create a `SimpleRational` object to pass to the function.

Our new numeric type certainly leaves a lot to be desired. We cannot display one of our rational number objects with `std::cout` very conveniently. We cannot use the standard arithmetic operators like `+` or `*`, and we cannot compare two rational numbers using `==` or `<`. In Section 16.1 we address these shortcomings.

14.6 Encapsulation

When developing complex systems, allowing indiscriminate access to an object's internals can be disastrous. It is all too easy for a careless, confused, or inept programmer to change an object's state in such a way as to corrupt the behavior of the entire system. A malicious programmer may intentionally tweak one or more objects to sabotage the system. In either case, if the software system controls a medical device or military missile system, the results can be deadly.

C++ provides several ways to protect the internals of an object from the outside world, but the simplest strategy is the one we have been using: We can qualify fields and methods, generically referred to as class members, as either `public` or `private`.

The compiler enforces the inaccessibility of private members. In Listing 14.5 (simplerational.cpp), for example, client code cannot directly modify the `denominator` instance variable of a `SimpleRational` object making it zero. A client may influence the values of `numerator` and `denominator` only via methods provided by the class designer.

Accessibility rules, also called visibility rules or permissions, determine what parts of a class and/or object are accessible to the outside world. C++ provides a great deal of flexibility in assigning access permissions, but some general principles exist that, if followed, foster programs that are easier to build and extend.

- In general, fields should be `private`. Clients should not be able to arbitrarily change the state of an object. Allowing such might allow client code to put an object into an undefined state (for example, changing the denominator of a fraction to zero). An object's state should only change as a result of calling the object's methods.

  The built-in primitive types like `int` and `double` offer no protection from client access. One exception to the private fields rule applies to simple objects that programmers naturally would treat as primitive types. Recall the geometric `Point` class found in Listing 14.1 (mathpoints.cpp). The x and y fields of a point object safely may assume any legitimate floating-point value, and it may be reasonable in some applications for clients to treat a `Point` object as a primitive type. In this case it is appropriate to make the `x` and `y` fields public.

- Methods that provide a service to client code should be part of the `public` section of the class. The author of the class must ensure that the `public` methods cannot place the object into an illegal state. For example, consider the following method:

  ```
  void set_denominator(int d) {
      denominator = d;
  }
  ```

  If this method were part of the `SimpleRational` class, it would permit client code to sabotage a valid fraction with a simple statement:

  ```
  fract.set_denominator(0);
  ```

- Methods that assist the service methods but that are not meant to be used by the outside world should be in the `private` section of the class. This allows a `public` method to be decomposed into simpler, perhaps more coherent activities without the threat of client code accessing these more primitive methods. These private methods are sometimes called *helper methods* or *auxiliary methods*.

Why would a programmer intentionally choose to limit access to parts of an object? Restricting access obviously limits the client's control over the objects it creates. While this may appear to be a disadvantage at first glance, this access restriction actually provides a number of advantages:

- **Flexibility in implementation**. A class conceptually consists of two parts:
  - **The class interface—the visible part**. Clients see and can use the public parts of an object. The public methods and public variables of a class constitute the *interface* of the class. A class's interface specifies *what* it does.
  - **The class implementation—the hidden part**. Clients cannot see any private methods or private variables. Since this private information is invisible to clients, class developers are free to do whatever they want with the private parts of the class. A class's implementation specifies *how* it accomplishes what it needs to do.

  We would like our objects to be black boxes: clients shouldn't need to know *how* the objects work but merely rely on *what* objects can do.

A good rule of thumb in class design is this: make data `private`, and make methods that provide a service to clients `public`.

Many real-world objects follow this design philosophy. Consider a digital wristwatch. Its display gives its user the current time and date. It can produce different output in different modes; for examples, elapsed time in stopwatch mode or wake up time in alarm mode. It presents to its user only a few buttons for changing modes, starting and stopping stopwatches, and setting the time. *How* it does what is does is irrelevant to most users; most users are concerned with *what* it does. Its user risks great peril by opening the watch and looking at its intricate internal details. The user is meant to interact with the watch only through its interface—the display and buttons.

Similarly, an automobile presents an accelerator pedal to its user. The user knows that pushing the pedal makes the car go faster. That the pedal is connected to the fuel injection system (and possibly other systems, like cruise control) through a cable, wire, or other type of linkage is of concern only to the automotive designer or mechanic. Most drivers prefer to be oblivious to the under-the-hood details.

Changing the interface of a class can disturb client code that already has been written to use objects of that class. For example, what if the maintainers of the `SimpleRational` class decide that `SimpleRational` objects should be immutable; that is, after a client creates a `SimpleRational` object the client cannot adjust the `numerator` or `denominator` values. The `set_numerator` and `set_denominator`, therefore, would have to disappear. Unfortunately, both of these methods are public and thus part of `SimpleRational`'s interface to client. Existing client code may be using these methods, and removing them, making them private, altering the types of their parameters or any other changes to the interface would render existing client code incompatible. Client code that has been written to use `set_numerator` according to its original interface no longer will be correct. We say the change in `SimpleRational`'s interface *breaks* the client code.

Class authors have no flexibility to alter the interface of a class once the class has been released for clients to use; any changes risk breaking existing client code. On the other hand, altering the private information in a class will not break existing client code that uses that class, since private class information is invisible to clients. When the private parts of a class change, clients need only recompile their code; client programmers do not need to modify their source code. A class, therefore, becomes less resilient to change as more of its components become exposed to clients. To make classes as flexible as possible, which means maximizing the ability to make improvements to the class in the future, hide as much information as possible from clients.

- **Reducing programming errors**. Client code cannot misuse the parts of a class that are private since the client cannot see the private parts of a class. Properly restricting client access can make it impossible for client code to put an object into an ill-defined state. In fact, if a client can coax an object into an illegal state via the class interface, then the design and/or implementation of the class is faulty. As an example, if a client can somehow make a `SimpleRational` object's denominator zero, then one of the methods or a constructor must contain a logic error. Clients should never be able to place an object into an illegal state.

- **Hiding complexity**. Objects can provide a great deal of functionality. Even though a class may provide a fairly simple interface to clients, the services it provides may require a significant amount of complex code to accomplish their tasks. One of the challenges of software development is dealing with the often overwhelming complexity of the task.

It is difficult, if not impossible, for one programmer to be able to comprehend at one time all the details of a large software system. Classes with well-designed interfaces and hidden implementations provide a means to reduce this complexity. Since private components of a class are hidden, their details cannot contribute to the complexity the client programmer must manage. The client programmer needs not be concerned with exactly how an object works, but the details that make the object work are present nonetheless. The trick is exposing details only when necessary:

- **Class designer**. The class designer must be concerned with the hidden implementation details of the class. Since objects of the class may be used in many different contexts, the class designer usually does not have to worry about the context in which the class will be used. From the perspective of the class designer, the complexity of the client code that may use the class is therefore eliminated.
- **Applications developer using a class**. The developer of the client code must be concerned with the details of the application code being developed. The application code will use objects. The hidden details of the class these objects represent are of no concern to the client developers. From the perspective of the client code designer, therefore, the complexity of the code that makes the objects work is eliminated.

This concept of information hiding is called *encapsulation*. Details are exposed to particular parties only when appropriate. In sum, the proper use of encapsulation results in

- software that is more flexible and resilient to change,
- software that is more robust and reliable, and
- a software development process that programmers can more easily comprehend and manage.

Finally, the C++ encapsulation model has its limits. It is not possible to protect an object from code within itself. Any method within a class has full access to any member defined within that class. If you believe that parts of a class should be protected from some of its methods, you should split up the class into multiple classes with suitable restrictions among the resulting component classes.

14.7 Exercises

1. Given the definition of the `SimpleRational` number class in Listing 14.5 (simplerational.cpp), complete the function named `add`:

   ```
   SimpleRational add(SimpleRational r1, SimpleRational r2) {
       // Details go here
   }
   ```

   that returns the rational number representing the sum of its two parameters. Recall that adding fractions is more involved than multiplying them; you must find a common denominator, adjust the fractions so they both have the same denominator, and then add just the numerators.

2. Given the definition of the geometric `Point` class in Listing 15.6 (point.cpp), complete the function named `distance`:

   ```
   double distance(Point r1, Point r2) {
       // Details go here
   }
   ```

that returns the distance between the two points passed as parameters.

3. What is the purpose of a class constructor?

4. May a class constructor be overloaded?

5. Given the definition of the `SimpleRational` number class in Section 14.3, complete the following method named `reduce`:

```
class SimpleRational {
     // Other details omitted here ...

     // Returns an object of the same value reduced
     // to lowest terms
     SimpleRational reduce() {
         // Details go here
     }
};
```

that returns the rational number that represents the object reduced to lowest terms; for example, the fraction 10/20 would be reduced to 1/2.

6. Given the definition of the `SimpleRational` number class in Section 14.3, complete the following free function named `reduce`:

```
// Returns a fraction to lowest terms
SimpleRational reduce(SimpleRational f) {
    // Details go here
}
```

that returns the rational number that represents the object reduced to lowest terms; for example, the fraction 10/20 would be reduced to 1/2.

7. Given the definition of the geometric `Point` class in Section 14.2 add a method named `distance`:

```
class Point {
    // Other details omitted

    // Returns the distance from this point to the
    // parameter p
    double distance(Point p) {
        // Details go here
    }
};
```

that returns the distance between the point on whose behalf the method is called and the parameter `p`.

8. Consider the following C++ code:

```
#include <iostream>

class IntPoint {
public:
    int x;
    int y;
```

```
    IntPoint(int x, int y): x(x), y(y) {}
};

class Rectangle {
    IntPoint corner;  // Location of the rectangle's lower-left corner
    int width;        // The rectangle's width
    int height;       // The rectangle's width
public:
    Rectangle(IntPoint pt, int w, int h):
              corner((pt.x < -100) ? -100 : (pt.x > 100 ? 100 : pt.x),
                     (pt.y < -100) ? -100 : (pt.y > 100 ? 100 : pt.y)),
              width((w < 0) ? 0 : w), height((h < 0) ? 0 : h) {}

    int perimeter() {
        return 2*width + 2*height;
    }

    int area() {
        return width * height;
    }

    int get_width() {
        return width;
    }

    int get_height() {
        return height;
    }

    // Returns true if rectangle r overlaps this
    // rectangle object.
    bool intersect(Rectangle r) {
        // Details omitted
    }

    // Returns the length of a diagonal rounded to the nearest
    // integer.
    int diagonal() {
        // Details omitted
    }

    // Returns the geometric center of the rectangle with
    // the (x,y) coordinates rounded to the nearest integer.
    IntPoint center() {
        // Details omitted
    }

    bool is_inside(IntPoint pt) {
        // Details omitted
    }
```

```
};

int main() {
    Rectangle rect1(IntPoint(2, 3), 5, 7),
              rect2(IntPoint(2, 3), 1, 3),
              rect3(IntPoint(2, 3), 15, 3),
              rect4(IntPoint(2, 3), 5, 3);
    std::cout << rect1.get_width() << '\n';
    std::cout << rect1.get_height() << '\n';
    std::cout << rect2.get_width() << '\n';
    std::cout << rect2.get_height() << '\n';
    std::cout << rect3.get_width() << '\n';
    std::cout << rect3.get_height() << '\n';
    std::cout << rect4.get_width() << '\n';
    std::cout << rect4.get_height() << '\n';
    std::cout << rect1.get_perimeter() << '\n';
    std::cout << rect1.get_area() << '\n';
    std::cout << rect2.get_perimeter() << '\n';
    std::cout << rect2.get_area() << '\n';
    std::cout << rect3.get_perimeter() << '\n';
    std::cout << rect3.get_area() << '\n';
    std::cout << rect4.get_perimeter() << '\n';
    std::cout << rect4.get_area() << '\n';
}
```

(a) What does the program print?

(b) With regard to a `Rectangle` object's lower-left corner, what are the minimum and maximum values allowed for the x coordinate? What are the minimum and maximum values allowed for the y coordinate?

(c) What is a `Rectangle` object's minimum and maximum width?

(d) What is a `Rectangle` object's minimum and maximum height?

(e) What happens when a client attempts to create a `Rectangle` object with parameters that are outside the acceptable ranges?

(f) Implement the `diagonal` method.

(g) Implement the `center` method.

(h) Implement the `intersect` method.

(i) Implement the `is_inside` method.

(j) Complete the following function named `corner`:

```
IntPoint corner(Rectangle r) {
    // Details go here
};
```

that returns the lower-left corner of the `Rectangle` object `r` passed to it. You may not modify the `Rectangle` class.

(k)

9. Develop a `Circle` class that, like the `Rectangle` class above, provides methods to compute perimeter and area. The `Rectangle` instance variables are not appropriate for circles; specifically, circles do have corners, and there is no need to specify a width and height. A center point and a radius more naturally describe a circle. Build your `Circle` class appropriately.

10. Given the `Rectangle` and `Circle` classes from questions above, write an `encloses` function:

```
// Returns true if rectangle rect is large enough to
// completely enclose circle circ
bool encloses(Rectangle rect, Circle circ) {
    // Details omitted
}
```

so that it returns true if circle `circ`'s dimensions would allow it to fit completely within rectangle `rect`. If `circ` is too big, the function returns false. The positions of `rect` and `circ` do not influence the result.

11. Consider the following C++ code:

```
class Widget {
    int value;
public:
    Widget();
    Widget(int v);
    int get();
    void bump();
};

Widget::Widget() {
    value = 40;
}

Widget::Widget(int v) {
    if (v >= 40)
        value = v;
    else
        value = 0;
}

int Widget::get() const {
   return value;
}

void Widget::bump() {
    if (value < 50)
        value++;
}

int main() {
    Widget w1, w2(5);
    std::cout << w1.get() << '\n';
    std::cout << w2.get() << '\n';
    w1.bump();  w2.bump();
```

```
    std::cout << w1.get() << '\n';
    std::cout << w2.get() << '\n';
    for (int i = 0; i < 20; i++) {
        w1.bump();
        w2.bump();
    }
    std::cout << w1.get() << '\n';
    std::cout << w2.get() << '\n';
}
```

(a) What does the program print?

(b) If `wid` is a `Widget` object, what is the minimum value the expression `wid.get()` can return?

(c) If `wid` is a `Widget` object, what is the maximum value the expression `wid.get()` can return?

Chapter 15

Fine Tuning Objects

In Chapter 14 we introduced the basics of object-oriented programming: private data, public methods, and automatic initialization. In this chapter examine some details that enable C++ programmers to fine tune class design.

15.1 Passing Object Parameters

Recall the `print_fraction` function from Listing 14.5 (simplerational.cpp):

```
void print_fraction(SimpleRational f) {
    std::cout << f.get_numerator() << "/" << f.get_denominator();
}
```

Observe that the function accepts its parameter `f` using the call by value protocol. This means that during the execution of the following client code:

```
SimpleRational my_rational{3, 4};
print_fraction(my_rational);
```

the program makes a new `SimpleRational` object (the formal parameter `f` in `print_fraction`'s definition) and copies the member data from the actual parameter `my_rational` into `f`. At this point the program has two copies of the fraction 3/4; the client has a 3/4 rational object in its variable `my_rational`, and the executing `print_function` has a 3/4 rational object in its formal parameter `f`. When the `print_fraction` function finishes executing, the formal parameter `f` goes out of scope and its memory is freed. The function returns back to the caller, leaving the caller's 3/4 rational object as the sole remaining one.

Even though `SimpleRational` objects are relatively small (four bytes under Visual C++), this process of creating a new object serving as the formal parameter, copying the contents from the actual parameter into the formal parameter, and finally cleaning up the formal parameter is all unnecessary work on the part of the executing program.

We noted in Section 11.1.4 that declaring `std::vector` objects as `const` reference parameters to functions provides the efficiency of pass by reference with the safety of pass by value:

- Efficiency of pass by reference: The function has access to the caller's object via its location in

memory. Pass by reference sends a single memory address to the function. There is no need to copy the object's data into the function.

- Safety of pass by value: The function cannot modify the caller's actual parameter since the formal parameter is declared `const`.

We revisited this concept of pass by `const` reference for `std::string` objects in Section 13.3.

It turns out that it is a good idea to prefer passing objects by `const` reference instead of pass by value. This means the following function should replace the original `print_fraction` function:

```
void print_fraction(const SimpleRational& f) {
    std::cout << f.get_numerator() << "/" << f.get_denominator();
}
```

Rewriting `multiply` also is in order. The following provides the new implementation:

```
// Returns the product of two rational numbers
SimpleRational multiply(const SimpleRational& f1,
                        const SimpleRational& f2) {
    return {f1.get_numerator() * f2.get_numerator(),
            f1.get_denominator() * f2.get_denominator()};
}
```

In this new version of `multiply` the `const` reference parameters avoid the construction, member data copying, and memory clean up of two `SimpleRational` objects for each call to `multiply`.

If passing object types by `const` reference for object types is the better approach why does C++ not do this automatically for us? C++ defaults to pass by value as the default. This guarantees safety for all types at the expense of reduced efficiency for some types.

When is pass by value warranted for object types? A caller may make an object and then pass a copy of the object to a function that would modify that object and return the modified version back to the caller. The caller then could have both the original object plus the modified object returned by the function. Doing this via passing by `const` reference is possible, but requires a little more code. The following sample code illustrates:

```
SimpleRational zero1(const SimpleRational& f) {
    SimpleRational result{f}; // Make a copy of parameter f
    result.set_numerator(0);
    return result;
}

SimpleRational zero2(SimpleRational f) {
    // Just use f; it is a copy of the caller's actual parameter
    f.set_numerator(0);
    return f;
}
```

Here `zero1` must explicitly copy its formal parameter to a new local variable it eventually will return. The `zero2` function does not need to copy its formal parameter; its formal parameter already is a copy of the caller's actual parameter.

A caller could use the `zero1` and `zero2` functions as shown here:

```
SimpleRational one_third{1, 3};
auto other = zero1(one_third);
// Caller owns two objects: one_third (1/3) and other (0/3).
```

15.2 Pointers to Objects and Object Arrays

Given the `Point` class from Section 14.2, the statement

```
Point pt;
```

declares the variable `pt` to be a `Point` object. As with primitive data, we can declare pointers to objects:

```
Point pt;
Point *p_pt;
```

Here, `p_pt` is a pointer to a `Point` object. Before we use the pointer we must initialize it to point to a valid object. We can assign the pointer to refer to an existing object, as in

```
p_pt = &pt;
```

or use the `new` operator to dynamically allocate an object from the heap:

```
p_pt = new Point;
```

If the class has a constructor that accepts parameters, we need to provide the appropriate arguments when using `new`. Recall the `Account` class from Section 14.4. Given the declarations

```
Account acct("Joe", 3143, 90.00);
Account *acct_ptr;
```

before we use `acct_ptr` it must point to a valid object. As in the `Point` class example, we can assign the pointer to refer to an existing object, as in

```
acct_ptr = &acct;
```

or use the `new` operator to dynamically allocate an object from the heap:

```
acct_ptr = new Account("Moe", 400, 1300.00);
```

Note that we include the arguments expected by the `Account` class constructor. This statement allocates memory for one `Account` object and executes the constructor's code to initialize the newly created object. Figure 15.1 illustrates a pointer to an account object.

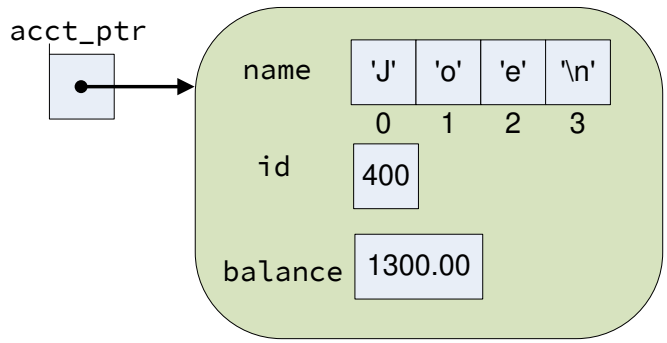
As with any dynamically allocated entity, a programmer must be careful to use the `delete` operator to deallocate any objects created via `new`.

The dot operator syntax to access a field of a object through a pointer is a bit awkward:

```
Point *p = new Point;
(*p).x = 253.7;
(*p).y = -00.5;
```

The parentheses are required since the dot (`.`) has higher precedence than the pointer dereferencing operator (`*`). Without the parentheses, the statement would be evaluated as if the parentheses were placed as shown here

Figure 15.1 A pointer to an `Account` object




```
*(p.x) = 253.7;  // Error
```

The variable `p` is not a `Point` object, but a pointer to a `Point` object, so it must be dereferenced with the `*` operator before applying the `.` operator. C++ provides a simpler syntax to access fields of an object through a pointer. The *pointer operator* eliminates the need for the parentheses:

```
Point *p = new Point;
p->x = 253.7;
p->y = -00.5;
```

The pair of symbols `->` constitute one operator (no space in between is allowed), and the operator is meant to look like an arrow pointing right. There is no associated left pointing arrow in C++.

You can use the `->` operator to access the methods of an object referenced by a pointer:

```
Account *acct_ptr = new Account("Joe", 400, 1300.00);
if (acct_ptr->withdraw(10.00))
    std::cout << "Withdrawal successful\n";
else
    std::cout << "*** Insufficient funds ***\n";
```

An executing program will *not* automatically deallocate the memory allocated via `new`; It is the programmer's responsibility to manually deallocate the memory when the object is no longer needed. The `delete` operator with no `[]` decoration frees up a single dynamically allocated object:

```
// acct_ptr points to an Account object previously allocated
// via new
delete acct_ptr;
```

The following function has a logic error because it fails to free up memory allocated with `new`:

```
void faulty_func() {
    Account *acct_ptr = new Account("Joe", 400, 1300.00);
    if (acct_ptr->withdraw(10.00))
        std::cout << "Withdrawal successful\n";
    else
```

```
        std::cout << "*** Insufficient funds ***\n";
    acct_ptr->display();
}
```

Here, acct_ptr is a local variable, so it occupies space on the stack. It represents a simple address, essentially a number. The space for the acct_ptr variable is automatically deallocated when faulty_func completes. The problem is acct_ptr points to memory that was allocated with new, and this memory is not freed up within the function. The pointer acct_ptr is the only way to get to that memory, so when the function is finished, that memory is lost for the life of the program. The condition is known as a *memory leak*. If the program runs to completion quickly, the leak may go undetected. In a longer running program such as a web server, memory leaks can cause the program to crash after a period of time. The problem arises when code that leaks memory is executed repeatedly and eventually all of available memory becomes used up.

The corrected function would be written

```
void corrected_func() {
    Account *acct_ptr = new Account("Joe", 400, 1300.00);
    if (acct_ptr->withdraw(10.00))
        std::cout << "Withdrawal successful\n";
    else
        std::cout << "*** Insufficient funds ***\n";
    acct_ptr->display();
    delete acct_ptr;
}
```

If we step back and take an honest look at this corrected_func function, we will see that there really is no reason for using pointers and dynamic memory at all. This corrected_func offers no advantage over the following function:

```
void even_better_func() {
    Account acct("Joe", 400, 1300.00);
    if (acct.withdraw(10.00))
        std::cout << "Withdrawal successful\n";
    else
        std::cout << "*** Insufficient funds ***\n";
    acct.display();
}
```

So why consider pointers to objects at all? Programmers commonly use pointers to objects to build elaborate linked data structures. Section 18.3 describes one such data structure.

C++ supports vectors and arrays of objects, but they present special challenges. First, consider a simple situation. The Point class defines no constructor, so the following code is valid:

```
std::vector<Point> pts(100);    // Okay
```

The compiler happily generates code that at run time will allocate enough space for 100 Point objects. No special initializations are needed since Point has no constructor. What if a class defines a constructor that accepts arguments and does not supply also a constructor that requires no arguments? Consider the Account class. The following statement is illegal:

```
std::vector<Account> accts(100);  // Illegal, the Account class has
                                  // no default constructor
```

When creating the space for the `accts` elements, C++ expects a default constructor to properly initialize all of the vector's elements. The only constructor in the `Account` class requires arguments, and so the compiler refuses to accept the declaration of `accts`. The compiler has no means by which it can produce the code needed to initialize the vector's elements before the programmer begins using the vector.

One solution uses a vector of pointers, as in

```
std::vector<Account *> accts(100);  // A vector of account pointers
```

Note that this does not create any `Account` objects. The programmer subsequently must iterate through the vector and use `new` to create individually each account element. An example of this would be

```
std::vector<Account *> accts(100);  // A vector of account pointers
for (int i = 0; i < 100; i++) {
    // Get information from the user
    std::cin >> name >> id >> amount;
    // Create the new account object
    accts[i] = new Account(name, id, amount);
}
```

In this case, when the program no longer needs the `accts` vector, it must execute code like

```
for (int i = 0; i < 100; i++) {
   delete accts[i];
```

to reclaim the dynamic memory held by each of the `Account` objects.

An alternative approach that avoids pointers uses the vector `push_back` method to add objects one at a time, as shown here:

```
std::vector<Account> accts;  // Vector initially empty
for (int i = 0; i < 100; i++) {
    std::string name;
    int id;
    double amount;
    // Get information from the user
    std::cin >> name >> id >> amount;
    // Create the new account object
    accts.push_back({name, id, amount});
}
```

In this case there is no need to use `delete` later.

Recall (see 10.7) that in C++ there is one literal value to which a pointer of any type may be assigned—`nullptr`:

```
Account *p_rec = nullptr;   // p_rec is null
```

A pointer with the value `nullptr` is interpreted to mean a pointer that is pointing to nothing. An attempt to `delete` a null pointer is legal and does nothing.

15.3 The `this` Pointer

Recall our `Counter` class from Section 14.3, reproduced here:

```
class Counter {
    int count;
public:
    // Allow clients to reset the counter to zero
    void clear() {
        count = 0;
    }

    // Allow clients to increment the counter
    void inc() {
        count++;
    }

    // Return a copy of the counter's current value
    int get() {
        return count;
    }
};
```

In Section 14.3 we saw that an expression such as `ctr1.clear()` passes the address of `ctr1` as a secret parameter during the call to the `clear` method. This is how the method determines which `count` field to reset—`ctr1.clear()` resets `ctr1`'s `count` field, and `ctr2.clear()` would reset `ctr2`'s `count` field.

Within a method definition a programmer may access this secret parameter via the reserved word `this`. Within a method body the `this` expression represents a pointer to the object upon which the method was called. We can rewrite the `Counter` class to make the `this` pointer explicit, as shown here:

```
// Uses this directly
class Counter {
    int count;
public:
    // Allow clients to reset the counter to zero
    void clear() {
        this->count = 0;
    }

    // Allow clients to increment the counter
    void inc() {
        this->count++;
    }

    // Return a copy of the counter's current value
    int get() {
        return this->count;
    }
};
```

Within a method of the `Counter` class, `this->count` is an alternate way of writing just `count`. Some programmers always use the `this` pointer as shown here to better communicate to human readers that the `count` variable has to be a field, and it cannot be a local or global variable.

The this pointer is handy when a method parameter has the same name as a field:

```
class Point {
    double x;
    double y;
public:
    void set_x(double x) {
        // Assign the parameter's value to the field
        this->x = x;
    }
    // Other details omitted . . .
};
```

In the set_x method the parameter x has the same name as field x. This is legal in C++; a method parameter or local variable of a method may have the same name as a field within that class. The problem is that the local variable or parameter hides the access to the field. Any unqualified use of the name x refers to the parameter x, not the field x. One solution would be to name the parameter something else: perhaps _x or x_param. A strong argument can be made, though, that x is the best name for the field, *and* x is also the best name for the parameter to the set_x method. To get access to the field in this case, use the this pointer. Since this is a reserved word, the expression this->x cannot be mistaken for anything other than the x field of the object upon which the method was invoked.

Another use of the this pointer involves passing the current object off to another function or method. Suppose a global function named log exists that accepts a Counter object as shown here:

```
void log(Counter c) {
    // Details omitted . . .
}
```

If within the clear method we wish to call the log function passing it the object on whose behalf the clear method was executing, we could write

```
class Counter {
    // . . .
public:
    void clear() {
        count = 0;
        // Pass this object off to the log function
        log(*this);
    }
    // . . .
};
```

We pass the expression *this as the actual parameter to log because the log function expects an object, not a pointer to an object. Remember the syntax of pointers: If this is a pointer, *this is the object to which this points.

Since this serves as an implicit parameter passed to methods, it is illegal to use the this expression outside of the body of a method.

15.4 const Methods

Given the SimpleRational class in Listing 14.5 (simplerational.cpp), the following code produces a compiler error:

```
const SimpleRational fract(1, 2);  // Constant fraction 1/2
fract.set_numerator(2);            // Error, cannot change a constant
```

This behavior is desirable, since the set_numerator method can change the state of a SimpleRational object, and our fract object is supposed to be constant. Unfortunately, this correct behavior is accidental. The compiler does not analyze our methods to determine exactly what they do. Consider the following code that also will not compile:

```
const SimpleRational fract(1, 2);     // Constant fraction 1/2
std::cout << fract.get_numerator();   // Error!
```

Since the get_numerator method does not modify a SimpleRational object, we would expect that invoking it on a constant object should be acceptable, but the compiler rejects it. Again, the compiler cannot understand what get_numerator is supposed to do; specifically, it is not designed to be able to determine that a method will not change the state of an object. The programmer must supply some additional information to help the compiler.

If a method is not supposed to change the state of an object, that method should be declared const. In the SimpleRational class, the methods get_numerator and get_denominator simply return, respectively, copies of the fraction's numerator and denominator. Neither method is intended to modify any instance variables. If we look at the code for those methods, we see that indeed neither method changes anything about a SimpleRational object. What if the programmer made a mistake—perhaps a spurious copy and paste error—and the statement

```
numerator = 0;
```

made its way into the get_numerator method? Unfortunately, the way things stand now, the compiler cannot detect this error, and get_numerator will contain a serious logic error.

We can remove the possibility of such an error by declaring get_numerator (and get_denominator) const:

```
class SimpleRational {
public:
    /* ... stuff omitted ... */
    int get_numerator() const {
        return numerator;
    }
    int get_denominator() const  {
        return denominator;
    }
    /* ... other stuff omitted ... */
};
```

The const keyword goes after the closing parenthesis of the parameter list and before the opening curly brace of the method's body. If we accidentally attempt to reassign an instance variable in a const method, the compiler will report an error.

Declaring a method `const` is not merely a good defensive programming strategy used by a class developer. Methods declared to be `const` can be called with `const` objects, while it is illegal to invoke a non-`const` method with a `const` object. With the new `const` version of `SimpleRational`'s `get_numerator` method, the following code

```
const Rational fract(1, 2);  // Constant fraction 1/2
fract.set_numerator(2);      // Error, cannot change a constant
```

is still illegal, since `Rational::set_numerator` is not `const`, but

```
const Rational fract(1, 2);         // Constant fraction 1/2
std::cout << fract.get_numerator(); // Okay with const get_numerator
```

now is legal.

You should declare `const` any method that has no need to change any field within the object. Similarly, do not declare `const` any method that is supposed to change a field in an object. A `const` method can be used with both `const` and non-`const` objects, but a non-`const` method cannot be used with `const` objects. For maximum flexibility, always declare a method to be `const` unless doing so would prevent the method from doing what it is supposed to do.

You can invoke a `const` method from both `const` and non-`const` objects, but you cannot invoke a non-`const` method from a `const` object. For maximum flexibility, always declare a method to be `const` unless doing so would prevent the method from doing what it is supposed to do.

15.5 Separating Method Declarations and Definitions

It is common in larger C++ projects to separate a method implementation from its declaration. A simple example illustrates how to do this; consider the class `MyClass`:

```
class MyClass {
public:
    void my_method() const {
        std::cout << "Executing \"my_method\"\n";
    }
};
```

This version of `MyClass` uses what is known as an *inline method definition*; that is, the method `my_method` is defined completely with a body in the same place as it is declared. Compare the inline version to the following equivalent representation split across two files: myclass.h and myclass.cpp. The code in myclass.h is shown in Listing 15.1 (myclass.h).

Listing 15.1: `myclass.h`

```
// File myclass.h

class MyClass {
public:
    void my_method() const;   // Method declaration
};
```

No body for `my_method` appears in the declaration of the class `MyClass` in myclass.h; instead, the method implementation appears elsewhere, in myclass.cpp. Listing 15.2 (myclass.cpp).

Listing 15.2: `myclass.cpp`

```
// File myclass.cpp

// Include the class declaration
#include "myclass.h"
#include <iostream>

// Method definition
void MyClass::my_method() const {
    std::cout << "Executing \"my_method\"\n";
}
```

Without the prefix `MyClass::` the definition of `my_method` in myclass.cpp simply would be global function definition like all the ones we have seen since Chapter 9. The class name and scope resolution operator (`::`) binds the definition of the method to the class to which is belongs.

We would `#include` the .h header file in all source files that need to use `MyClass` objects; the myclass.cpp file is compiled separately and linked into the rest of the project's .cpp files.

Listing 15.3 (pointinline.h) shows a point class written inline.

Listing 15.3: `pointinline.h`

```
class Point {
    double x;
    double y;
public:
    Point(double x, double y): x(x), y(y) {}
    double get_x() const { return x; }
    double get_y() const { return y; }
    void set_x(double x) { this->x = x; }
    void set_y(double y) { this->y = y; }
};
```

Listing 15.4 (pointsplit.h) and Listing 15.5 (pointsplit.cpp) show how we can separate the method declarations from their implementations.

Listing 15.4: `pointsplit.h`

```
class Point {
    double x;
    double y;
public:
    Point(double x, double y);   // No constructor implementation
    double get_x() const;        // and no method
    double get_y() const;        // implementations
    void set_x(double x);
    void set_y(double y);
};
```

Listing 15.5: `pointsplit.cpp`

```
// Point constructor
Point::Point(double x, double y): x(x), y(y) {}

// Get the x value
double Point::get_x() const {
    return x;
}

// Get the y value
double Point::get_y() const {
    return y;
}

// Set the x value
void Point::set_x(double x) {
    this->x = x;
}

// Set the y value
void Point::set_y(double v) {
    this->y = y;
}
```

The class name prefix such as `Point::` is necessary not only so the compiler can distinguish a method definition such as `get_x` from a global function definition, but also to distinguish the method from a method with the same name and parameter types that might appear in a different class. A method signature for a method is just like a global function signature, except a method signature includes the class name as well. Each of the following represent distinct signatures:

- `get_x()` is the signature for a global function named `get_x` that accepts no parameters.
- `Point::get_x()` is the signature for a method of the `Point` class named `get_x()` that accepts no parameters.
- `LinearEquation::get_x()` is the signature for a method of the `LinearEquation` class named `get_x()` that accepts no parameters.

Recall from Section 10.3 that the return type is not part of a function's signature. The same is true for methods.

Many C++ programmers avoid the inline style of class declarations and use the separate class declaration and method definition files for several reasons:

- If the class is meant to be used in multiple programs, the compiler must recompile the methods each time the header file is `#include`d by some C++ source file. When the method declarations and definitions are separate, the compiler can compile the code for the definitions once, and the linker can combine this compiled code with the client code that uses it.
- Client programmers can look at the contents of header files. If method definitions are inlined, client programmers can see exactly how the methods work. This can be a disadvantage; for example, a client programmer may make assumptions about how fast a method takes to execute or the particular order in which a method processes data in a vector. These assumptions can influence how the client code calls the method. If the class maintainer changes the implementation of the method, the client

programmer's previous assumptions may be invalid. A certain ordering of the data that before the change resulted in faster processing may now be slower. An improvement in a graphics processing algorithm may cause an animation to run too quickly when the method is rewritten. For the class designer's maximum flexibility, client programmers should not be able to see the details of a method's implementation because then they cannot form such assumptions.

Client programmers need to know *what* the method does, not *how* it accomplishes it.

To enforce this hidden method implementation:

1. Separate the method declarations from their definitions. Put the class declaration in the header file and the method definitions in a `.cpp` file.
2. Compile the `.cpp` file into an object file.
3. Provide the client the `.h` file and the compiled object file but not the source file containing the method definitions.
4. The client code can now be compiled by including the `.h` file and linked with the object file, but the client programmer has no access to the source code of the method definitions.

- Some programmers find the inline style difficult since it provides too much detail. It complicates determining *what* objects of the class are supposed to do because the *how* is sprinkled throughout the class declaration.

Despite the disadvantages mentioned above, inline methods are sometimes appropriate. For simple classes defined in the same file that uses them, the inline style is handy, as Listing 15.6 (point.cpp) shows.

Listing 15.6: `point.cpp`

```
#include <iostream>

class Point {
    double x;
    double y;
public:
    Point(double x, double y): x(x), y(y) {}
    double get_x() const { return x; }
    double get_y() const { return y; }
    void set_x(double x) { this->x = x; }
    void set_y(double y) { this->y = y; }
};

double dist(const Point& pt1, const Point& pt2) {
    // Compute distance between pt1 and pt2 and return it
    // This is a function stub; add the actual code later
    return 0.0;  // Just return zero for now
}

int main() {
    Point p1(2.5, 6), p2(0.0, 0.0);
    std::cout << dist(p1, p2) << '\n';
}
```

Here the `Point` class is very simple, and the constructor and method implementations are obvious. The inline structure makes more sense in this situation.

Listing 15.7 (trafficlight.h) contains the interface for a class used to create objects that simulate stop-caution-go traffic signals.

Listing 15.7: trafficlight.h

```
enum class SignalColor { Red, Green, Yellow };

class Trafficlight {
private:
    SignalColor color;  // The light's current color: Red, Green, or Yellow
public:

    Trafficlight(SignalColor initial_color);
    void change();
    SignalColor get_color() const;
};
```

`SignalColor` is an enumerated type that clients may use. (Section 3.9 introduced enumerated types.) By using an enumerated type, we restrict a traffic light object's color to the three specified by `SignalColor`.

The traffic signal implementation code in Listing 15.8 (trafficlight.cpp) defines the `Trafficlight` methods.

Listing 15.8: trafficlight.cpp

```
#include "trafficlight.h"

// Ensures a traffic light object is in the state of
// red, green, or yellow.  A rogue value makes the
// traffic light red
Trafficlight::Trafficlight(SignalColor initial_color) {
    switch (initial_color) {
      case SignalColor::Red:
      case SignalColor::Green:
      case SignalColor::Yellow:
        color = initial_color;
        break;
      default:
        color = SignalColor::Red;  // Red by default, just in case
    }
}

// Ensures the traffic light's signal sequence
void Trafficlight::change() {
    // Red --> green, green --> yellow, yellow --> red
    if (color == SignalColor::Red)
        color = SignalColor::Green;
    else if (color == SignalColor::Green)
        color = SignalColor::Yellow;
    else if (color == SignalColor::Yellow)
        color = SignalColor::Red;
}

// Returns the light's current color so a client can
// act accordingly
SignalColor Trafficlight::get_color() const {
```

```
    return color;
}
```

The code within Listing 15.8 (trafficlight.cpp) must `#include` the trafficlight.h header so the compiler is exposed to the `Trafficlight` class declaration; otherwise, when compiling trafficlight.cpp the compiler would not know if the method implementations faithfully agreed with declarations. If the method definition of `TrafficLight::change` in trafficlight.cpp specified parameters but its declaration within trafficlight.h did not, that would be a problem. Furthermore, without including trafficlight.h, the type `SignalColor` would be an undefined type within trafficlight.cpp.

Notice that outside the class declaration in Listing 15.7 (trafficlight.h) we must prefix the method names with `TrafficLight::`. Without this prefix the compiler would treat the identifiers as globals. It would interpret them as free functions, not methods. `SignalColor` is declared outside of the `TrafficLight` class, so it does not need the `TrafficLight::` prefix. `SignalColor` is a global type available to any code that `#include`s trafficlight.h.

Listing 15.9 (trafficmain.cpp) shows how a client could use the `Trafficlight` class.

Listing 15.9: `trafficmain.cpp`

```
#include <iostream>
#include "trafficlight.h"

void print(Trafficlight lt) {
    SignalColor color = lt.get_color();
    std::cout << "+-----+\n";
    std::cout << "|     |\n";
    if (color == SignalColor::Red)
        std::cout << "| (R) |\n";
    else
        std::cout << "| ( ) |\n";
    std::cout << "|     |\n";
    if (color == SignalColor::Yellow)
        std::cout << "| (Y) |\n";
    else
        std::cout << "| ( ) |\n";
    std::cout << "|     |\n";
    if (color == SignalColor::Green)
        std::cout << "| (G) |\n";
    else
        std::cout << "| ( ) |\n";
    std::cout << "|     |\n";
    std::cout << "+-----+\n";
}

int main() {
    Trafficlight light(SignalColor::Green);
    while (true) {
        print(light);
        light.change();
        std::cin.get();
    }
}
```

Listing 15.9 (trafficmain.cpp) first prints

```
+-----+
|     |
| ( ) |
|     |
| ( ) |
|     |
| (G) |
|     |
+-----+
```

When the user presses the Enter key the program prints

```
+-----+
|     |
| ( ) |
|     |
| (Y) |
|     |
| ( ) |
|     |
+-----+
```

a second Enter press prints

```
+-----+
|     |
| (R) |
|     |
| ( ) |
|     |
| ( ) |
|     |
+-----+
```

A third Enter press completes the cycle:

```
+-----+
|     |
| ( ) |
|     |
| ( ) |
|     |
| (G) |
|     |
+-----+
```

The program's execution continues in this manner within its infinite loop until the user presses Ctrl C.

Observe that the `color` variable within the `Trafficlight` class is private. That means that once the client creates a traffic light object the only way a client can influence the value of `color` is via the `change` method. This makes it impossible for a client to force a traffic light object to cycle incorrectly; for example, a client cannot make a traffic light change directly from yellow to green (it would have to pass through red on the way from yellow to green).

15.6 Preventing Multiple Inclusion

C++ follows the *one definition rule*, or ODR, which means a variable, function, class, instance variable, or method may not have more than one definition in the same context. The code within the following function definition violates the one definition rule:

```
int sum_and_display(const std::vector<int>& list) {
    int s = 0, size = list.size();
    // Add up the values in the list
    int i = 0;
    while (i < size) {
        s += list[i];
        i++;
    }
    // Print the contents of the list
    int i = 0;  // Illegal, variable i already defined above
    while (i < size) {
        std::cout << std::setw(5) << list[i] << '\n';
        i++;
    }
    std::cout << "--------\n";
    std::cout << std::setw(5) << s << '\n';
}
```

The line

```
int s = 0, size = list.size();
```

is not a problem even though the identifier `size` appears twice. The first appearance denotes the local variable named `size`, and the second use is a call to the `size` method of the `std::vector` class. This statement defines (declares) only the variable `size`; the `size` method already has been defined in the `std::vector` class (and the compiler processed its definition from the <vector> header file).

The variable `i` has two distinct definitions in the same context, so its redeclaration right before the display code is an error.

Like a variable, a class may have only one definition. When we build general purpose classes meant to be used widely in our programs we must take care that we do not violate the one definition rule. In fact, we can violate the one definition rule accidentally even if we define a class only once!

To see how we can accidentally violate the one definition rule, consider the following simple example. Suppose we have a simple counter class declared in count.h:

```
class Counter {
    int count;
public:
    // Allow clients to reset the counter to zero
    void clear();

    // Allow clients to increment the counter
    void inc();

    // Return a copy of the counter's current value
```

```
    int get() const;
};
```

with its implementation in count.cpp:

```
// Allow clients to reset the counter to zero
void Counter::clear() {
    count = 0;
}

// Allow clients to increment the counter
void Counter::inc() {
    count++;
}

// Return a copy of the counter's current value
int Counter::get() const {
    return count;
}
```

Next, consider the following two header files defining objects that themselves use `Counter` objects, widget.h:

```
// File widget.h

#include "count.h"

class Widget {
    Counter meter;
public:
    // Other stuff omitted
};
```

and gadget.h:

```
// File gadget.h

#include "count.h"

class Gadget {
    Counter ticker;
public:
    // Other stuff omitted
};
```

The preprocessor directive

```
#include "count.h"
```

is required in the file widget.h, because the compiler must be acquainted with the type `Counter` to properly handle the instance variable `meter`. Similarly, the count.h header must be included in gadget.h in order for the compiler to accept `Gadget`'s declaration of its `ticker` field.

Client code that wishes to use `Widget` objects must use the appropriate `#include` statement:

```
#include "widget.h"

int main() {
    Widget myWidget;
    // Use the myWidget object
}
```

Similarly, client code that uses `Gadget` objects must include at the top of its file:

```
#include "gadget.h"

int main() {
    Gadget myGadget;
    // Use the myGadget object
}
```

Both of these client programs will build without any problems. Sometimes, however, a program may need to use both `Widget` objects *and* `Gadget` objects. Since the widget.h header file does not know anything about the `Gadget` class and the gadget.h header file has no information about `Widget`s, we must include both header files in code that uses both classes. The appropriate include directives would be

```
#include "widget.h"
#include "gadget.h"

int main() {
    Widget myWidget;
    Gadget myGadget;
    // Use the myWidget and myGadget objects
}
```

Including one of the header files without the other is insufficient since the compiler must be able to check if the client is using both types correctly. This client code, however, will not compile. In this case the problem is with count.h. Including both widget.h and gadget.h includes the definition of the `Counter` class twice, so the compiler sees two definitions of `Counter`. Even though the two definitions are identical, this violates the one definition rule and so results in a compiler error.

The issue here is known as *multiple inclusion*, and it can arise when header files `#include` other header files. Multiple inclusion is a problem, but it often is necessary because the programmer may have a legitimate need for both types of objects within the same program. Fortunately, the solution is simple. The standard way to prevent multiple inclusion is to wrap a class definition as follows:

```
#ifndef COUNT_H_
#define COUNT_H_

class Counter {
    int count;
public:
    // Allow clients to reset the counter to zero
    void clear();

    // Allow clients to increment the counter
    void inc();
```

```
    // Return a copy of the counter's current value
    int get() const;
};

#endif
```

Do not use semicolons at the end of the lines beginning with `#ifndef`, `#define`, and `#endif` because these are preprocessor directives, not C++ statements. The word following the `#ifndef` and `#define` preprocessor directives can be any valid identifier, but best practice produces a unique word tied to the name of the header file in which it appears. The convention is to use all capital letters and underscores as shown above (an identifier cannot contain a dot). Putting an underscore after the header file name further makes it less likely that name will be used elsewhere within the program. If the header file is named myheader.h, the preprocessor wrapper would be

```
#ifndef MYHEADER_H_
#define MYHEADER_H_

/*  Declare your classes here */

#endif
```

Like a C++ program, the preprocessor can maintain a collection of variables. These preprocessor-defined variables can influence the compilation process. Preprocessor-defined variables are merely an artifact of compilation and are unavailable to an executing C++ program. The preprocessor directive `#ifndef` evaluates to true if the preprocessor has not seen the definition of a given preprocessor variable; `#ifndef` returns false if it has seen the definition of the preprocessor variable. The `#define` directive defines a preprocessor variable. The net effect of the `#ifndef`/`#define`/`#endif` directives is that the preprocessor will not include the header file more than once when it is processing a C++ source file. This means the compiler will see the class definition exactly once, thus satisfying the one definition rule.

The `#ifndef`...`#define`...`#endif` preprocessor directives should be used to wrap the class declaration in the header file and do not appear in the `.cpp` file containing the method definitions. Since you cannot always predict how widespread the use of a class will become, it is good practice to use this preprocessor trick for all general purpose classes you create. By doing so you will avoid the problem of multiple inclusion.

15.7 Overloaded Operators

We can overload many of the C++ operators to work with programmer-defined types. Table 15.1 shows some operators that have been defined to work with several of the standard object classes.

We can define how specific operators work for the types we devise. We may express operators for classes either global functions or member functions.

15.7.1 Operator Functions

Consider a simple mathematical point class:

```
class Point {
public:
```

Operator	Class	Function	Example	Chapter
`operator<<`	`std::ostream`	Right-hand operand is sent to the stream specified by the left-hand operand	`std::cout << x;`	Chapter 2
`operator>>`	`std::istream`	Right-hand operand is extracted from the stream specified by the left-hand operand	`std::cin >> x;`	Chapter 4
`operator[]`	`std::vector`	Right-hand operand (the integer within the square brackets) is used to locate an element within the left-hand operand (a `vector`)	`x = v[2];`	Section 11.1
`operator==`	`std::string`	Right-hand operand (a `string`) is compared with the left-hand operator (also a `string`) to determine if they contain exactly the same characters in exactly the same order	`if (word == "Please") proceed();`	Chapter 13

Table 15.1: Some Overloaded Operators for Objects

```
    double x;
    double y;
};
```

Suppose we wish to define the operation of addition on points as follows:

$$(x_1, y_1) + (x_2, y_2) = (x_1 + y_1, x_2 + y_2)$$

Thus, the x coordinate of the sum of two points is the sum of the x coordinates, and the y coordinate is the sum of the y coordinates.

We can overload the addition operator to work on `Point` objects with the following global function:

```
Point operator+(const Point& p1, const Point& p2) {
    Point result;
    result.x = p1.x + p2.x;
    result.y = p1.y + p2.y;
    return result;
}
```

With a slightly more sophisticated `Point` definition:

```
class Point {
    double x;  // Fields are now private
    double y;
public:
    // Add a constructor
    Point(double x, double y);
    // Add some methods to access the private data members
    double get_x() const;
    double get_y() const;
};
```

with the expected constructor and method implementations:

```
// Initialize fields
Point::Point(double x, double y): x(x), y(y) {}

double Point::get_x() const {
    return x;  // Return a copy of the x member
}
```

```
double Point::get_y() const {
    return y;  // Return a copy of the y member
}
```

we can write the operator function in one line:

```
Point operator+(const Point& p1, const Point& p2) {
    return {p1.get_x() + p2.get_x(), p1.get_y() + p2.get_y()};
}
```

It often is convenient to overload the output stream << operator for custom classes. The `std::cout` object (and, therefore, the `std::ostream` class) has overloaded methods named `operator<<` that allow us to print the primitive types like integers and floating-point numbers. If we create a new type, such as `Point` or `Rational`, the `std::ostream` class has no `operator<<`method built in to handle objects of our new type. In order to use `std::cout`'s << operator with programmer-defined objects we must define a global operator function of the form:

```
std::ostream& operator<<(std::ostream& os, const X& x)
```

where `X` represents a programmer-defined type. Notice that the function returns type `std::ostream&`. This is because the first parameter is also an `std::ostream&`, and the function returns the same object that was passed into it. If `x` is a programmer-defined type, the expression

```
operator<<(std::cout, x)
```

thus evaluates to a reference to `std::cout`. This process of returning the object that was passed allows us to chain together the << operator, as in

```
std::cout << x << y << '\n';
```

This is the beautified syntax for

```
operator<<(operator<<(operator<<(std::cout, x), y), '\n');
```

If you examine this expression carefully, you will see that the first argument to all the calls of `operator<<` is simply `std::cout`.

For our enhanced `Point` class, `operator<<` could look like:

```
std::ostream& operator<<(std::ostream& os, const Point& pt) {
    os << '(' << pt.get_x() << ',' << pt.get_y() << ')';
    return os;
}
```

The function simply returns the same `std::ostream` object reference that was passed into it via the first parameter.

Given the above definitions of `operator+` and `operator<<`, clients can write code such as

```
Point my_point(1, 2), your_point(0.45, 0);
std::cout << "Point 1: " << my_point << '\n';
std::cout << "Point 2: " << my_point << '\n';
std::cout << my_point << " + " << your_point
          << " = " << my_point + your_point << '\n';
```

which displays

```
Point 1: (1.0, 2.0)
Point 2: (0.45, 0.0)
(1.0, 2.0) + (0.45, 0.0) = (1.45, 2.0)
```

When class developers provide such an `operator<<` function, clients can print out objects just as easily as printing the basic data types.

The `<<` operator is not overloaded for the `std::vector` class, but we now easily can do it ourselves as needed. For a vector of integers, the following function

```
ostream& operator<<(ostream& os, const std::vector<int>& vec) {
    os << '{';
    int n = vec.size();
    if (n != 0) {
        os << vec[0]
        for (int i = 1; i < n; i++)
            os << ',' << vec[i];
    }
    os << '}';
    return os;
}
```

provides the necessary functionality. Given this `operator<<` function, clients can display vectors as they would a built in type:

```
std::vector<int> list(10, 5);
std::cout << list << '\n';
```

This code fragment prints

```
{5,5,5,5,5,5,5,5,5,5}
```

15.7.2 Operator Methods

A class may define operator methods. A method for a unary operator accepts no parameters, and a method for a binary operator accepts only one parameter. The "missing" parameter is the object upon which the operator is applied; that is, the `this` pointer. To see how operator methods work, consider an enhanced rational class:

```
class EnhancedRational {
    int numerator;
    int denominator;
public:
    /*  Other details omitted */

    // Unary plus returns the double-precision floating-point
    // equivalent of the rational number.
    double operator+() const {
        return static_cast<double>(numerator)/denominator;
    }
```

```
    // Binary plus returns the rational number that results
    // from adding another rational number (the parameter) to
    // this object.
    Rational operator+(const Rational& other) const {
        int den = denominator * other.denominator,
            num = numerator * other.denominator
                  + other.numerator * denominator;
        return {num, den};
    }
};
```

The following code fragment shows how clients can use these operator methods:

```
Rational fract1(1, 2), fract2(1, 3), fract3;
double value = +fract1;    // Assigns 0.5
fract3 = fract1 + fract2;  // fract3 is 5/6
```

The statement

```
fract3 = fract1 + fract2;
```

is syntactic sugar for the equivalent statement

```
fract3 = fract1.operator+(fract2);
```

Here we see how the righthand operand of the plus operator, `fract2`, becomes the single parameter to the binary `operator+` method of `EnhancedRational`.

The statement

```
double value = +fract1;
```

is the syntactically sugared version of

```
double value = fract1.operator+();
```

While programmers may change the meaning of many C++ operators in the context of objects, the precedence and associativity of all C++ operators are fixed. In the `Rational` class, for example, it is impossible to enable the binary `+` operator to have a higher precedence than the binary `*` in the context of `Rational` objects or any other classes of objects.

15.8 `static` Members

Variables declared in a class declaration are known as instance variables because each instance (object) of that class maintains its own copy of the variables. This means, for example, that changing the numerator of one `Rational` object will not affect the numerator of any other `Rational` object.

Sometimes it is convenient to have variables or constants that all objects of a class share. Global variables and constants certainly will work, but globals are not tied to any particular class. C++ uses the `static` keyword within a class to specify that all objects of that class share a field; for example,

```
class Widget {
    int value;
    static int quantity;
};
```

Each individual `Widget` object has its own `value` variable, but only one `quantity` variable exists and is shared by all `Widget` objects.

One unusual thing about static fields is that must be *defined* outside of the class declaration. For the `Widget` class above, we must supply the statement

```
int Widget::quantity;
```

somewhere in the source code outside of the class declaration.

Consider a widget factory in which each widget object must have a unique serial number. Serial numbers are sequential, and a new widget object's serial number is one larger than the widget produced immediately before. Important for warranty claims, a client should not be able to alter a serial number of a widget object. The `Widget` class in Listing 15.10 (serialnumber.cpp) shows how to use a `static` variable to manage serial numbers for widget objects.

Listing 15.10: `serialnumber.cpp`

```
#include <iostream>

class Widget {
    // All Widget objects share serial_number_source
    static int serial_number_source;
    // Each Widget object is supposed to have its own
    // unique serial_number
    int serial_number;
public:
    // A Widget object's serial number is initialized from
    // the shared serial_number_source variable which is
    // incremented each time a Widget object is created
    Widget(): serial_number(serial_number_source++) {}
    // Client's can look at the serial number but not touch
    int get_serial_number() const {
        return serial_number;
    }
};

// Initialize the initial serial number; the first
// Widget made will have serial number 1
int Widget::serial_number_source = 1;

// Make some widgets and check their serial numbers
int main() {
    Widget w1, w2, w3, w4;
    std::cout << "w1 serial number = " << w1.get_serial_number() << '\n';
    std::cout << "w2 serial number = " << w2.get_serial_number() << '\n';
    std::cout << "w3 serial number = " << w3.get_serial_number() << '\n';
    std::cout << "w4 serial number = " << w4.get_serial_number() << '\n';
}
```

The output of Listing 15.10 (serialnumber.cpp) is

```
w1 serial number = 1
w2 serial number = 2
w3 serial number = 3
w4 serial number = 4
```

Each `Widget` object has its own `serial_number` variable, but all `Widget` objects have access to the shared `serial_number_source` variable. The executing program initializes `serial_number_source` one time at the beginning of the program's execution before it calls the `main` function. This means `serial_number_source` is properly initialized to 1 before the program creates any `Widget` objects. Each time the client creates a new `Widget` object, the constructor assigns the individual object's serial number from the `static` variable. The constructor also increments `serial_number_source`'s value, so the next object created will have a serial number one higher than the previous `Widget`.

C++ programmers often use class static fields to provide public constants available to clients. Consider Listing 15.11 (trafficsignal.h) that models a simple traffic light a little differently from Listing 15.7 (trafficlight.h). It uses symbolic integer constants instead of enumeration types.

Listing 15.11: `trafficsignal.h`

```
#ifndef TRAFFICSIGNAL_H_
#define TRAFFICSIGNAL_H_

class TrafficSignal {
    int color;  // The light's current color: RED, GREEN, or YELLOW
public:
    // Class constants available to clients
    static const int RED = 0;
    static const int GREEN = 1;
    static const int YELLOW = 2;

    TrafficSignal(int initial_color);
    void change();
    int get_color() const;
};

#endif
```

The state of a traffic light object—which of its lamps is illuminated—is determined by an integer value: 0 represents *red*, 1 stands for *green*, and 2 means *yellow*. It is much more convenient for clients to use the symbolic constants `RED`, `GREEN`, and `YELLOW` than to try to remember which integer values stand for which colors. These constants are `public`, so clients can freely access them, but, since they are constants, clients cannot alter their values. An additional benefit to being `const` is this: You may initialize a `static const` field within the class body itself. You do not need to re-define a `static const` field outside the class body as you do for a `non-const static` field.

Listing 15.12 (trafficsignal.cpp) provides the implementation of the methods of `TrafficSignal` class.

Listing 15.12: `trafficsignal.cpp`

```
#include "trafficsignal.h"

// Ensures a traffic light object is in the state of
// red, green, or yellow.  A rogue integer value makes the
// traffic light red
```

```
TrafficSignal::TrafficSignal(int initial_color) {
    switch (initial_color) {
      case RED:
      case GREEN:
      case YELLOW:
        color = initial_color;
        break;
      default:
        color = RED;  // Red by default
    }
}

// Ensures the traffic light's signal sequence
void TrafficSignal::change() {
    // Red --> green, green --> yellow, yellow --> red
    color = (color + 1) % 3;
}

// Returns the light's current color so a client can
// act accordingly
int TrafficSignal::get_color() const {
    return color;
}
```

Within the methods of the `TrafficSignal` class we can use the simple names `RED`, `GREEN`, and `YELLOW`. Code outside of the `TrafficSignal` class can access the color constants because they are public but must use the fully-qualified names `TrafficSignal::RED`, `TrafficSignal::GREEN`, and `TrafficSignal::YELLOW`.

A client can create a traffic light object as

```
TrafficSignal light(TrafficSignal::YELLOW);
```

Since the client code is outside the `TrafficSignal` class it must must use the full `TrafficSignal::YELLOW` name. This statement makes an initially yellow traffic light. Since the `RED`, `GREEN`, and `YELLOW` public fields are constants, clients cannot modify them to subvert the behavior of a traffic light object.

It may not be obvious, but the color constants in the `TrafficSignal` class *must* be declared `static`. To see why, consider a minor change to `TrafficSignal` in the class we will call `TrafficSignalAlt`:

```
class TrafficSignalAlt {
    int color;  // The light's current color: RED, GREEN, or YELLOW
public:
    // Constant instance fields available to clients
    const int RED;      // Note: NOT static
    const int GREEN;
    const int YELLOW;

    TrafficSignalAlt(int initial_color);
    void change();
    int get_color() const;
};
```

The `TrafficSignalAlt` class is identical to `TrafficSignal`, except that the color constants are constant instance fields instead of constant class (`static`) fields. Observe that just like the `TrafficSignal`

class, the `TrafficSignalAlt` class has no default constructor. Recall from Section 14.4 that a default constructor accepts no arguments, and that if a programmer provides any constructor for a class, the compiler will not automatically provide a default constructor. This means the client cannot write code such as

```
TrafficSignal light;
```

or

```
TrafficSignalAlt light2;
```

During the object's creation the client *must* provide an integer argument representing a traffic light color. If `RED`, `GREEN`, and `YELLOW` are constant instance variables (that is, constant non-`static` fields), every `TrafficSignalAlt` object has its own copy of the fields, and the `RED`, `GREEN`, and `YELLOW` fields cannot exist outside of any traffic light object. This leads to a chicken-and-egg problem—how can we create the first `TrafficSignalAlt` object using the symbolic constants `RED`, `GREEN`, or `YELLOW`? These constants do not exist unless we have a traffic light object, yet we need a traffic light object to have any of these constants!

An executing program initializes `static` class fields before it invokes the `main` function. This means any data pertaining to a class that must exist before any object of that class is created must be declared `static`. A `static` class variable exists outside of any instance of its class.

C++ allows methods to be declared `static`. A `static` method executes on behalf of the class, not an instance of the class. This means that a `static` method may not access any instance variables (that is non-`static` fields) of the class, nor may they call other non-`static` methods. Since a `static` method executes on behalf of the class, it has no access to the fields of any particular instance of that class. That explains the restriction against `static` methods accessing non-`static` data members. Since a non-`static` method may access instance variables of an object upon which it is called, a `static` method may not call a non-`static` method and thus indirectly have access to instance variables. The restriction goes only one way—any class method, `static` or non-`static`, may access a `static` data member or call a `static` method.

Looking at it from a different perspective, all non-`static` methods have the `this` implicit parameter (Section 15.3). No `static` method has the `this` parameter. This means it is a compile-time error to use `this` within a `static` method.

15.9 Classes vs. structs

All members of a C++ class are `private` by default. The class

```
class Point {
public:
    double x;
    double y;
};
```

must use the `public` label so clients can access a `Point` object's fields. The `struct` keyword is exactly like the `class` keyword, except that the default access to members is `public`:

```
struct Point {
    double x;    // These fields now are public
    double y;
```

```
};
```

The C language supports the `struct` feature, but not the `class` keyword. C `struct`s do not support methods and constructors. In C++ (unlike in C), a `struct` can contain methods and constructors. By default, members in `struct`s are public, but you can apply the `private` and `public` labels as in a class to fine tune client access.

Despite their similarities, C++ programmers favor `class`es over `struct`s for programmer-defined types with methods. The `struct` construct is useful for declaring simple composite data types that are meant to be treated like primitive types. Consider the `int` type, for example. We can manipulate directly integers, and integers do not have methods or any hidden parts. Likewise, a geometric point object consists of two coordinates that can assume any valid floating-point values. It makes sense to allow client code to manipulate directly the coordinates, rather than forcing clients to use methods like `set_x` and `set_y`. On the other hand, it is unwise to allow clients to modify directly the denominator of a `Rational` object, since a fraction with a zero denominator is undefined.

In C++, by default everything in an object defined by a `struct` is accessible to clients that use that object. In contrast, clients have no default access to the internals of an object that is an instance of a `class`. The default member access for `struct` instances is `public`, and the default member access for `class` instances is `private`.

The `struct` feature is, in some sense, redundant. It is a carryover from the C programming language. By retaining the `struct` keyword, however, C++ programs can use C libraries that use C `struct`s. Any C++ program that expects to utilize a C library using a `struct` must restrict its `struct` definitions to the limited form supported by C. Such `struct` definitions may not contain non-`public` members, methods, constructors, etc.

15.10 Friends

The private members of a class by default are inaccessible to code outside of that class. Ordinarily only methods within the class itself have permission to see and modify the private instance variables and invoke the private methods within that class. This access protection allows programmers to modify the implementation of the hidden internals of a class without affecting existing client code that uses the class.

For some class designs this all-or-nothing access dictated by the `public` and `private` labels within a class or `struct` is too limiting. At times it can be advantageous to design a class that grants special access to some precisely specified functions or classes of objects outside of the class.

As an example, consider a variation of the `SimpleRational` class from Listing 14.5 (simplerational.cpp). Our new class, named `OpaqueRational` is shown here:

```
// Models a mathematical rational number
class OpaqueRational {
    int numerator;
    int denominator;
public:
    // Initializes the components of an OpaqueRational object
    OpaqueRational(int n, int d): numerator(n), denominator(d) {
```

```
        if (d == 0) {
            // Display error message
            std::cout << "Zero denominator error\n";
            exit(1);   // Exit the program
        }
    }

    // The default constructor makes a zero rational number
    // 0/1
    OpaqueRational(): OpaqueRational(0, 1) {}
};
```

Note that clients can make an `OpaqueRational` object, but after its creation clients can neither see nor change the internals of the object. Gone are the `set_numerator`, `set_denominator`, `get_numerator`, and `get_denominator` methods of the `SimpleRational` class.

We would like to able to print an `OpaqueRational` object as easily as the following:

```
OpaqueRational frac{1, 2};  // Make the fraction 1/2
std::cout << frac << '\n';
```

We would like the `std::cout` statement to print the text

```
1/2
```

This would require an `operator<<` function overloaded to accept a `OpaqueRational` object, and this `operator<<` function would need to be able to access the private instance variables of the `OpaqueRational` object.

A class can grant access to an outside function or another class via the `friend` reserved word. In Listing 15.13 (printonlyrational.cpp) we specify a `friend` function that allows clients to display in human-readable form an opaque rational number object via `operator<<` with an output stream object.

Listing 15.13: `printonlyrational.cpp`

```
#include <iostream>
#include <cstdlib>

// Models a mathematical rational number
class PrintOnlyRational {
    int numerator;
    int denominator;
public:
    // Initializes the components of an PrintOnlyRational object
    PrintOnlyRational(int n, int d): numerator(n), denominator(d) {
        if (d == 0) {
            // Display error message
            std::cout << "Zero denominator error\n";
            exit(1);   // Exit the program
        }
    }

    // The default constructor makes a zero rational number
    // 0/1
    PrintOnlyRational(): PrintOnlyRational(0, 1) {}
```

```
    // This operator can access the internal details of an
    // object of this class.
    friend std::ostream& operator<<(std::ostream& os,
                                    const PrintOnlyRational& f);
};

std::ostream& operator<<(std::ostream& os, const PrintOnlyRational& f) {
    os << f.numerator << '/' << f.denominator;
    return os;
}

int main() {
    PrintOnlyRational fract{1, 2};  // The fraction 1/2
    std::cout << "The fraction is " << fract << '\n';
    PrintOnlyRational fract2{2, 3};  // The fraction 2/3
    std::cout << "The fraction is " << fract2 << '\n';
}
```

Listing 15.13 (printonlyrational.cpp) prints the following:

```
The fraction is 1/2
The fraction is 2/3
```

Observe that the `operator<<` function is not a member function of the `PrintOnlyRational` class. Ordinarily `operator<<` would be unable to access the `numerator` and `denominator` fields of a `PrintOnlyRational` object. We can see, however, that the compiler allows the `operator<<` function to access the private instance variables of its `f` parameter.

We chose in Listing 15.13 (printonlyrational.cpp) to implement the `operator<<` function outside of the `PrintOnlyRational` class; however, we just as well could have implemented it inline within the body of the class, as shown here:

```
class PrintOnlyRational {
    // Other details about the private members of the class omitted here . . .
public:
    // Other details about the public members of the class omitted here . . .
    friend std::ostream& operator<<(std::ostream& os,
                                    const PrintOnlyRational& f) {

        os << f.numerator << '/' << f.denominator;
        return os;
    }
};
```

Implementing `operator<<` inline in this way makes it look very much like an inline method implementation, but it is not a member function. The reserved word `friend` at the front indicates to the compiler that is it indeed a free function; C++ does not allow member functions to be declared as friends.

Listing 15.14 (friendclass.cpp) provides an example of a friend class.

Listing 15.14: `friendclass.cpp`

```
#include <iostream>
```

```
class Widget {
    int data;
public:
    Widget(int d): data(d) {}
    friend class Gadget;
};

class Gadget {
    int value;
public:
    // A Gadget objects copies the private data from a Widget object
    Gadget(const Widget& w): value(w.data) {}
    int get() const {
        return value;
    }
    bool compare(const Widget& w) const {
        return value == w.data;
    }
};

int main() {
    Widget wid{45};
    Gadget gad{wid};
    std::cout << gad.get() << '\n';
    if (gad.compare(wid))
        std::cout << "They are the same" << '\n';
}
```

In Listing 15.14 (friendclass.cpp) the `Gadget` constructor accesses the private `data` element of a `Widget` object to initialize a `Gadget` object. The `Gadget::compare` method accesses the `data` of a `Widget` parameter passed to it. Neither of these accesses would be possible if the `Widget` class did not declare `Gadget` to be a friend. Note that while a `Gadget` object may freely access the private members of any `Widget` object, no `Widget` object has special access to any `Gadget` object. Friendship is not automatically *symmetric*; that is, the `friend` declaration is one directional. For the relationship to be mutual, the programmer would have to declare the `Widget` class to be a friend within the `Gadget` class; then objects of both classes could freely access the internal details of each other.

Note that a class grants friendship unilaterally to an outside class or function; there is no way for an outside class or function to become a friend of a class that has not specified it as a `friend`. This gives the class designer full control over the code that accesses the private members of the class.

Suppose A, B, and C are classes. Further suppose that class B is a friend of class A and that class C is a friend of class B. The friendship is not *transitive*; this means that C being a friend of B and B being a friend of A does not automatically make C a friend of A. C++ is very strict about the friendship relationship. The designer of class A would need to declare C as an additional friend of A. A class may have as many friend functions and classes as needed.

Good object-oriented design avoids friends as much as possible. Ideally, if class A grants friendship to class B, the design of classes A and B should be under the control of the same developers. Otherwise, friendship weakens encapsulation (see Section 14.6). If class A grants friendship to class B and class B is not under the control of the developer of class A, one or more of the methods in class B could manipulate the internals of an object of type A and potentially place it in an ill-defined state.

Some would argue that the Listing 15.13 (printonlyrational.cpp) example could be better written as

shown in Listing 15.15 (readonlyrational.cpp).

Listing 15.15: readonlyrational.cpp

```
#include <iostream>
#include <cstdlib>

// Models a mathematical rational number
class ReadOnlyRational {
    int numerator;
    int denominator;
public:
    // Initializes the components of a ReadOnlyRational object
    ReadOnlyRational(int n, int d): numerator(n), denominator(d) {
        if (d == 0) {
            // Display error message
            std::cout << "Zero denominator error\n";
            exit(1);   // Exit the program
        }
    }

    // The default constructor makes a zero rational number
    // 0/1
    ReadOnlyRational(): ReadOnlyRational(0, 1) {}

    // Allows a client to see the numerator's value.
    int get_numerator() const {
        return numerator;
    }

    // Allows a client to see the denominator's value.
    int get_denominator() const {
        return denominator;
    }
};

// This operator sends a fraction object to the output stream in
// a human-readable form.
std::ostream& operator<<(std::ostream& os, const ReadOnlyRational& f) {
    os << f.get_numerator() << '/' << f.get_denominator();
    return os;
}

int main() {
    ReadOnlyRational fract{1, 2};  // The fraction 1/2
    std::cout << "The fraction is " << fract << '\n';
    ReadOnlyRational fract2{2, 3};  // The fraction 2/3
    std::cout << "The fraction is " << fract2 << '\n';
}
```

The `get_numerator` and `get_denominator` methods eliminate the need for a friend function. Note that a `ReadOnlyRational` object is subtly different from an `PrintOnlyRational` object. Clients can see the `numerator` and `denominator` fields of a `ReadOnlyRational` object as desired, but `PrintOnlyRational` objects are "print only;" clients readily can print an `PrintOnlyRational` object but cannot easily discern the individual `numerator` and `denominator` values.

Both the `PrintOnlyRational` and `ReadOnlyRational` classes represent read-only fraction objects—once a client creates one these objects the client cannot change its value. In truth, the `ReadOnlyRational` class is more versatile, as programmers could incorporate `ReadOnlyRational` objects into an application that displays its output via a graphical interface. `PrintOnlyRational` objects limit their access to output streams. The `PrintOnlyRational` class may better serve applications that need such limited access.

15.11 Exercises

1. Suppose `Widget` is a class of objects, and function `proc` accepts a single `Widget` object as a parameter. Without knowing anything about class `Widget`, which of the following definitions of function `proc` is considered better, and why?

   Option #1:

   ```
   void proc(Widget obj) { /* Details omitted . . . */ }
   ```

   Option #2:

   ```
   void proc(const Widget& obj) { /* Details omitted . . . */ }
   ```

2. Suppose you have the following definition for class `Assembly`:

   ```
   class Assembly {
   public:
       int value;
   };
   ```

   and the variable `ptr_a` declared and initialized as so:

   ```
   Assembly *ptr_a = new Assembly;
   ```

   (a) What statement using the dot operator (`.`) could you use to assign 5 to the `value` field of the object to which `ptr_a` points?

   (b) What statement using the arrow operator (`->`) could you use to assign 5 to the `value` field of the object to which `ptr_a` points?

3. Suppose you have the following definition for class `Gadget`:

   ```
   class Gadget {
       int value;
   public:
       Gadget(int v): value(v) {}
       int get() const {
           return value;
       }
   };
   ```

   Consider the following code fragment:

   ```
   Gadget x(5);
   int y = x.get();
   ```

(a) What value does the second statement assign to `y`?

(b) Rewrite `Gadget::get` so it explicitly uses the `this` pointer.

(c) During the execution of `Gadget::get` on the second line, what is the value of the method's `this` pointer?

(d) What can a client do to alter the `value` field of a `Gadget` object after the object's creation?

4. What are the consequences of declaring a method to be `const`?

5. Why are `const` methods necessary in C++?

6. Given the following `Counter` class declaration:

```
class Counter {
    int value;
public:
    Counter(): value(0) {}
    void increment() {
       value++;
    }
    int get() const {
        return value;
    }
};
```

and `Counter` objects declared as shown here:

```
Counter c1;
const Counter c2;
```

determine if each of the following statements is legal.

(a)
```
c1.increment();

c2.increment();
```

(b)
```
int x = c1.get();

int y = c2.get();
```

(d) Given the following `Counter` class declaration:

```
class Counter {
    int value;
public:
    Counter(): value(0) {}
    void increment() {
       value++;
    }
    int get() {
        return value;
    }
};
```

and `Counter` objects declared as shown here:

```
Counter c1;
const Counter c2;
```

determine if each of the following statements is legal.

(a)
```
c1.increment();

c2.increment();
```

(b)
```
int x = c1.get();

int y = c2.get();
```

(8) Consider the following class declaration:

```
class Counter {
    int value;
public:
    Counter(): value(0) {}
    void increment() {
       value++;
    }
    int decrement() {
       value--;
    }
    int get() const {
        return value;
    }
};
```

(a) Show how you would properly separate the code of the `Counter` class into two source files: its declaration in a counter.h file and its method implementations in a counter.cpp file.

(b) Would client code ordinarily `#include` both the counter.h and counter.cpp files in its source code?

(c) How can you protect your counter.h file to prevent clients from accidentally `#include`-ing the contents of the header file more than once?

(d) What advantages does separating the class declaration and method implementations provide?

9. What are the consequences of declaring a method to be `static`?

10. Consider the following class declaration:

```
class ValType {
public:
    int value1;
    static int value2;
};
```

(a) If a client creates 100 `ValType` objects, how many `value1` fields will exist in memory?

(b) If a client creates 100 `ValType` objects, how many `value2` fields will exist in memory?

11. How is a C++ `struct` similar to a `class`?

12. How does a C++ `struct` differ from a `class`?

13. Given the following custom type declarations:

```
class X {
    int value1;
public:
    static int value2;
    int value3;
    X(): value1(5) {}
    void f() {
        value1 = 0;
    }
    static void g() {
        value2 = 0;
    }
};
int X::value2 = 3;

struct Y {
    int quantity1;
    Y(): quantity1(5) {}
    void f() {
        quantity1 = 0;
    }
private:
    int quantity2;
};
```

and the following variable declarations:

```
X x_obj;
Y y_obj;
```

determine if each of the following statements is legal.

(a) `x_obj.value1 = 0;`

(b) `x_obj.value2 = 0;`

(c) `x_obj.value3 = 0;`

(d) `X::value1 = 0;`

(e) `X::value2 = 0;`

(f) `X::value3 = 0;`

(g) `y_obj.quantity1 = 0;`

(h) `y_obj.quantity2 = 0;`

(i) `Y::quantity1 = 0;`

(j) `Y::quantity2 = 0;`

(k) `x_obj.f();`

(l) `x_obj.g();`

(m) `X::f();`

(n) `X::g();`

14. What privileges does function `f` have with respect to the class that declares `f` to be its `friend`?

15. Consider the following class declaration:

```
class ValType {
    int value;
public:
    ValType(): value(0) {}
    void set(int v1) {
        value = v1;
    }
    void show() const {
        std::cout << value << '\n';
    }
    friend int f(const ValType& x);
};
```

and determine the legality of each of the following function definitions:

(a)
```
int f(const ValType& x) {
    return 2 * x.value;
}
```

(b)
```
int g(const ValType& x) {
    return 2 * x.value;
}
```

(c)
```
int f(const ValType& x) {
   x.show();
   return 0;
}
```

(d)
```
int g(const ValType& x) {
   x.show();
   return 0;
}
```

(e)
```
int f(const ValType& x, int n) {
   return n * x.value;
}
```

16. What are the risks associated with using the `friend` construct?

Chapter 16

Building some Useful Classes

This chapter uses the concepts from the past few chapters to build some complete, practical classes.

16.1 A Better Rational Number Class

Listing 16.1 (rational.cpp) enhances the `SimpleRational` class (Listing 14.5 (simplerational.cpp)) providing a more complete type.

Listing 16.1: `rational.cpp`

```
#include <iostream>

class Rational {
    int numerator;
    int denominator;

    // Compute the greatest common divisor (GCD) of two integers
    static int gcd(int m, int n) {
        if (n == 0)
            return m;
        else
            return gcd(n, m % n);
    }
    // Compute the least common multiple (LCM) of two integers
    static int lcm(int m, int n) {
        return m * n / gcd(m, n);
    }

public:
    Rational(int n, int d): numerator(n), denominator(d) {
        if (d == 0) {   // Disallow an undefined fraction
            std::cout << "*****Warning---Illegal Rational\n";
            numerator = 0;    // Make up a reasonable default fraction
            denominator = 1;
        }
    }
```

```
    // Default fraction is 0/1
    Rational(): numerator(0), denominator(1) {}

    int get_numerator() const {
        return numerator;
    }

    int get_denominator() const {
        return denominator;
    }

    Rational reduce() const {
        // Find the factor that numerator and denominator have in common...
        int factor = gcd(numerator, denominator);
        // ...then divide it out in the new fraction
        //return Rational(numerator/factor, denominator/factor);
        return {numerator/factor, denominator/factor};
    }

    // Equal fractions have identical numerators and denominators
    bool operator==(const Rational& fract) const {
        // First, find the reduced form of this fraction and the parameter...
        Rational f1 = reduce(),
                 f2 = fract.reduce();
        // ...then see if their components match.
        return (f1.numerator == f2.numerator)
               && (f1.denominator == f2.denominator);
    }

    // Unequal fractions are not equal
    bool operator!=(const Rational& other) {
        return !(*this == other);
    }

    // Compute the sum of fract and the current rational number
    Rational operator+(const Rational& fract) const {
        // Find common denominator
        int commonDenominator = lcm(denominator, fract.denominator);
        // Add the adjusted numerators
        int newNumerator = numerator * commonDenominator/denominator
                        + fract.numerator * commonDenominator/fract.denominator;
        return {newNumerator, commonDenominator};
    }

    // Compute the product of fract and the current rational number
    Rational operator*(const Rational& fract) const {
        return Rational(numerator * fract.numerator,
                        denominator * fract.denominator).reduce();
    }
};

// Allow a Rational object to be displayed in a nice
// human-readable form.
std::ostream& operator<<(std::ostream& os, const Rational& r) {
```

```
    os << r.get_numerator() << "/" << r.get_denominator();
    return os;
}

int main() {
    Rational f1(1, 2), f2(1, 3);
    std::cout << f1 << " + " << f2 << " = " << (f1 + f2) << '\n';
    std::cout << f1 << " * " << f2 << " = " << (f1 * f2) << '\n';
}
```

Listing 16.1 (rational.cpp) produces

```
1/2 + 1/3 = 5/6
1/2 * 1/3 = 1/6
```

This rational number type has some notable features:

- **Constructors**. The overloaded constructors permit convenient initialization and make it impossible to create an undefined fraction.
- **Private `static` methods**. `Rational` provides two `private static` methods: `gcd` and `lcm`. The algorithm for the recursive `gcd` (greatest common divisor) method was introduced in Section 10.5. The `lcm` (least common multiple) method is derived from the mathematical relationship:

  $$\gcd(m,n) \times \mathrm{lcm}(m,n) = m \times n$$

  These two methods are declared `private` because they are not meant to be used directly by client code. *Greatest common divisor* and *least common multiple* are concepts from number theory of which `Rational` clients have no direct need. Client code expects functionality typical of rational numbers, such as addition and reduction; these two private methods are used by other, public, methods that provide functionality more closely related to rational numbers. These two private methods are `static` methods because they do not use instance variables. An object is not required to compute the greatest common divisor of two integers. It is legal for `gcd` and `lcm` to be instance methods, but instance methods should be used only where necessary, since they have the power to alter the state of an object. Faulty coding that accidentally modifies an instance variable can be difficult to track down. If a class method is used, however, the compiler can spot any attempt to access an instance variable immediately.
- **Public instance methods**. None of the instance methods (`operator==`, `reduce`, `operator+`, and `operator*`) modify the state of the object upon which they are invoked. Thus, the `Rational` class still produces immutable objects. The methods `operator+`, `operator*`, and `reduce` use the `private` helper methods to accomplish their respective tasks.
- **Global « operator**. `operator<<` is a global function that allows a `Rational` object to be sent to the `std::cout` object to be displayed as conveniently as a built-in type.

16.2 Stopwatch

The linear search vs. binary search comparison program (Listing 12.5 (searchcompare.cpp)) accessed the system clock in order to time the execution of a section of code. The program used the `clock` function from the standard C library and an additional variable to compute the elapsed time. The following skeleton code fragment:

```
clock_t seconds = clock();    // Record starting time

/*
 *  Do something here that you wish to time
 */

clock_t other = clock();      // Record ending time
std::cout << static_cast<double>(other - seconds)/CLOCKS_PER_SEC
          << " seconds\n";
```

certainly works and can be adapted to any program, but it has several drawbacks:

- A programmer must take care to implement the timing code correctly for each section of code to be timed. This process is error prone:
  - `clock_t` is a specialized type that is used infrequently. It is not obvious from its name that `clock_t` is equivalent to an unsigned integer, so a programmer may need to consult a library reference to ensure its proper use.
  - The programmer must specify the correct arithmetic:

    ```
    (other - seconds)/CLOCKS_PER_SEC
    ```

  - The type cast to `double` of the time difference is necessary but easily forgotten or applied incorrectly. If a programmer incorrectly applies parentheses as so

    ```
    static_cast<double>((other - seconds)/CLOCKS_PER_SEC)
    ```

    the result will lose precision. Worse yet, the following parenthetical grouping

    ```
    static_cast<double>(other) - seconds/CLOCKS_PER_SEC
    ```

    will result in an incorrect value, since division has precedence over subtraction.
- The timing code is supplemental to the actual code that is being profiled, but it may not be immediately obvious by looking at the complete code which statements are part of the timing code and which statements are part of the code to be timed.

Section 10.6 offered a solution to the above shortcomings of using the raw types, constants, and functions available from the C time library. Listing 10.14 (timermodule.cpp) hides the details of the C time library and provides a convenient functional interface to callers. Unfortunately, as mentioned in Section 10.6, the functional approach has a serious limitation. The code in Listing 10.14 (timermodule.cpp) uses global variables to maintain the state of the timer. There is only one copy of each global variable. This means programmers using the timer functions cannot independently measure the elapsed time of overlapping events; for example, you cannot measure how long it takes for a function to execute and simultaneously measure how long a section of code within that function takes to execute.

A programmer could time multiple, simultaneous activites by using the raw C library `clock` function directly, but then we are back to where we began: messy, potentially error-prone code.

Objects provide a solution. Consider the following client code that uses a stopwatch object to keep track of the time:

```
Stopwatch timer;  // Declare a stopwatch object

timer.start();    // Start timing
```

```
/*
 *  Do something here that you wish to time
 */

timer.stop();     // Stop the clock
std::cout << timer.elapsed() << " seconds\n";
```

This code using a `Stopwatch` object is as simple as the code that uses the timer functions from Listing 10.14 (timermodule.cpp). As an added benefit, a developer can think of a `Stopwatch` object as if it is a real physical stopwatch object: push a button to start the clock (call the `start` method), push a button to stop the clock (call the `stop` method), and then read the elapsed time (use the result of the `elapsed` method). What do you do if you need to time two different things at once? You use two stopwatches, of course, so a programmer would declare and use two `Stopwatch` objects. Since each object maintains its own instance variables, each `Stopwatch` object can keep track of its own elapsed time independently of all other active `Stopwatch` objects.

Programmers using a `Stopwatch` object in their code are much less likely to make a mistake because the details that make it work are hidden and inaccessible. With objects we can wrap all the messy details of the timing code into a convenient package. Given our experience designing our own types though C++ classes, we now are adequately equipped to implement such a `Stopwatch` class. Listing 16.2 (stopwatch.h) provides the header file defining the structure and capabilities of our `Stopwatch` objects.

Listing 16.2: `stopwatch.h`

```
#ifndef STOPWATCH_H_DEFINED_
#define STOPWATCH_H_DEFINED_

#include <ctime>

class Stopwatch {
    clock_t start_time;
    bool running;
    double elapsed_time;
public:
    Stopwatch();
    void start();             // Start the timer
    void stop();              // Stop the timer
    void reset();             // Reset the timer
    double elapsed() const;   // Reveal the elapsed time
    bool is_running() const;  // Is the stopwatch currently running?
};

#endif
```

From this class declaration we see that when clients create a `Stopwatch` object a constructor is available to take care of any initialization details. Four methods are available to clients: `start`, `stop`, `reset`, and `elapsed`. The `reset` method is included to set the clock back to zero to begin a new timing. Note that the "messy" detail of the `clock_t` variable is private and, therefore, clients cannot see or directly affect its value within a `Stopwatch` object.

This `Stopwatch` class (Listing 16.2 (stopwatch.h)) addresses the weaknesses of the non-object-oriented approach noted above:

- The timing code can be implemented in methods of the `Stopwatch` class. Once the methods are correct, a programmer can use `Stopwatch` objects for timing the execution of sections of code without worrying about the details of how the timing is actually done. Client code cannot introduce errors in the timing code if the timing code is hidden within the `Stopwatch` class.
- The details of the timing code no longer intertwine with the code to be timed, since the timing code is located in the `Stopwatch` class. This makes it easier for programmers to maintain the code they are timing.
- The `Stopwatch` class provides a convenient interface for the programmer that replaces the lower-level details of calling system time functions.

Listing 16.3 (stopwatch.cpp) provides the `Stopwatch` implementation.

Listing 16.3: `stopwatch.cpp`

```
#include <iostream>
#include "Stopwatch.h"

// Creates a Stopwatch object
// A newly minted object is not running and is in a "reset" state
Stopwatch::Stopwatch(): start_time(0), running(false), elapsed_time(0.0) {}

// Starts the stopwatch to begin measuring elapsed time.
// Starting a stopwatch that already is running has no effect.
void Stopwatch::start() {
    if (!running) {
        running = true;            // Set the clock running
        start_time = clock();      // Record start time
    }
}

// Stops the stopwatch.  The stopwatch will retain the
// current elapsed time, but it will not measure any time
// while it is stopped.
// If the stopwatch is already stopped, the method has
// no effect.
void Stopwatch::stop() {
    if (running) {
        clock_t stop_time = clock();  // Record stop time
        running = false;
        // Accumulate elapsed time since start
        elapsed_time += static_cast<double>((stop_time - start_time))
                                                /CLOCKS_PER_SEC;
    }
}

// Reports the cummulative time in seconds since the
// stopwatch was last reset.
// This method does not affect the state of the stopwatch.
double Stopwatch::elapsed() const {
    if (running) {   // Compute time since last reset
        clock_t current_time = clock();  // Record current time
        // Add time from previous elapsed to the current elapsed
        // since the latest call to the start method.
```

```
        return elapsed_time
                  + static_cast<double>((current_time - start_time))
                                                  /CLOCKS_PER_SEC;
    }
    else   // Timer stopped; elapsed already computed in the stop method
        return elapsed_time;
}

// Returns the stopwatch's status (running or not) to the client.
// This method does not affect the state of the stopwatch.
bool Stopwatch::is_running() const {
    return running;
}

// Resets the stopwatch so a subsequent start begins recording
// a new time.   Stops the stopwatch if it currently is running.
void Stopwatch::reset() {
    running = false;
    elapsed_time = 0.0;
}
```

Note that our design allows a client to see the running time of a `Stopwatch` object without needing to stop it. An alternate design might print an error message and perhaps exit the program's execution if a client attempts to see the elapsed time of a running stopwatch.

Some aspects of the `Stopwatch` class are notable:

- `Stopwatch` objects use three instance variables:
  - The `start_time` instance variable records the time when the client last called the `start` method.
  - The `elapsed_time` instance variable keeps track of the time since the latest call to the `reset` method.
  - The `running` Boolean instance variable indicates whether or not the clock is running.
- The constructor sets the initial values of the instance variables `start_time`, `elapsed_time`, and `running`.
- The `start` method notes the system time *after* the assignment to `running`. If these two statements were reversed, the elapsed time would include the time to do the assignment to `running`. The elapsed time should as closely as possible just include the statements in the client code between the `start` and `stop` method calls.

  Notice that `start_time` is not assigned if the stopwatch is running.
- In the `stop` method, the system time is noted *before* the assignment to `running` so the elapsed time does not include the assignment to `running`. This provides a more accurate accounting of the client code execution time.

  The `stop` method computes the accumulated elapsed time. This design allows a client to stop the stopwatch and restart it later without losing an earlier segment of time.
- The `elapsed` method either returns the elapsed time computed by the `stop` method or computes the current running time without altering the `elapsed_time` variable. Clients should avoid calling `elapsed` when a `Stopwatch` object is running since doing so would interfere with the accurate timing of client code execution.

Compare the `main` function of Listing 12.5 (searchcompare.cpp) to that of Listing 16.4 (bettersearchcompare.cpp):

Listing 16.4: bettersearchcompare.cpp

```
#include <iostream>
#include <iomanip>
#include <ctime>
#include <vector>
#include "Stopwatch.h"

/*
 *   binary_search(v, seek)
 *       Returns the index of element seek in vector v;
 *       returns -1 if seek is not an element of v
 *       v is the vector to search; v's contents must be
 *       sorted in ascending order.
 *       seek is the element to find
 */
int binary_search(const std::vector<int>& v, int seek) {
    int first = 0,            // Initially the first element in vector
        last = v.size() - 1,  // Initially the last element in vector
        mid;                  // The middle of the vector
    while (first <= last) {
        mid = first + (last - first + 1)/2;
        if (v[mid] == seek)
            return mid;       // Found it
        else if (v[mid] > seek)
            last = mid - 1;   // continue with 1st half
        else  // v[mid] < seek
            first = mid + 1;  // continue with 2nd half
    }
    return -1;    // Not there
}

// This version requires vector v to be sorted in
// ascending order.
/*
 *   linear_search(v, seek)
 *       Returns the index of element seek in vector v;
 *       returns -1 if seek is not an element of a
 *       v is the vector to search; v's contents must be
 *       sorted in ascending order.
 *       seek is the element to find
 */
int linear_search(const std::vector<int>& v, int seek) {
    size_t n = v.size();
    for (size_t i = 0;  i < n && v[i] <= seek;  i++)
        if (v[i] == seek)
            return i;   // Return position immediately
    return -1;  // Element not found
}

int main() {
   const size_t SIZE = 30000;
   std::vector<int> list(SIZE);
```

```
    Stopwatch timer;

    // Ensure the elements are ordered low to high
    for (size_t i = 0; i < SIZE; i++)
        list[i] = i;
    // Search for all the elements in list using linear search
    timer.start();
    for (size_t i = 0; i < SIZE; i++)
        linear_search(list, i);
    // Print the elapsed time
    timer.stop();
    std::cout << "Linear elapsed: " << timer.elapsed() << " seconds\n";
    // Prepare for a new timing
    timer.reset();
    // Search for all the elements in list using binary search
    timer.start();
    for (size_t i = 0; i < SIZE; i++)
        binary_search(list, i);
    // Print the elapsed time
    timer.stop();
    std::cout << "Binary elapsed: " << timer.elapsed() << " seconds\n";
}
```

This new, object-oriented version is simpler and more readable.

The design of the `Stopwatch` class allows clients to create multiple `Stopwatch` instances, and each instance will keep track of its own time. In practice when profiling executing programs, such generality usually is unnecessary. Rarely do developers need to time overlapping code, so one timer object per program usually is enough. Multiple sections of code can be checked with the same `Stopwatch` object; simply start it, stop it, check the time, and then reset it and start it again when another section of code is to be timed.

16.3 Sorting with Logging

Section 12.2 shows how to use function pointers to customize the ordering that selection sort performs on a vector of integers. The `selection_sort` function in Listing 12.2 (flexibleintsort.cpp) accepts a function pointer parameter in addition to the vector. The function pointer points to a function that accepts two integer parameters and returns true or false. The function is supposed to use some kind of ordering rule to determine if its first integer parameter precedes its second integer parameter.

Suppose we wish to analyze the number of comparisons and the number of swaps the sort function performs on a given vector with a particular ordering strategy. One way to do this is have the sort function itself keep track of the number of times it calls the comparison function and `swap` function and return this information when it finishes. To do so we would have to define an object to hold the two pieces of data (comparisons and swaps) since a function can return only one value, not two. Also if we do this, we must significantly alter the code of the sort algorithm itself. We would prefer to keep the sort algorithm focused on its task of sorting and remain uncluttered from this additional logging code.

If instead of passing a function pointer to our sort function we pass a specially crafted object. We can design our object to do whatever we want; specifically, we can design our special object to perform the necessary comparisons and keep track of how many comparisons it performs. We could let the object do the swap, and it could log the swaps it performs.

Listing 16.5 (loggingflexiblesort.cpp) is a variation of Listing 12.2 (flexibleintsort.cpp) that uses a comparison *object* instead of a comparison *function*.

Listing 16.5: loggingflexiblesort.cpp

```
#include <iostream>
#include <vector>

/*
 *  Comparer objects manage the comparisons and element
 *  interchanges on the selection sort function below.
 */
class Comparer {
    // Keeps track of the number of comparisons
    // performed
    int compare_count;
    // Keeps track of the number of swaps performed
    int swap_count;
    // Function pointer directed to the function to
    // perform the comparison
    bool (*comp)(int, int);
public:
    // The client must initialize a Comparer object with a
    // suitable comparison function.
    Comparer(bool (*f)(int, int)):
             compare_count(0), swap_count(0), comp(f) {}

    // Resets the counters to make ready for a new sort
    void reset() {
        compare_count = swap_count = 0;
    }

    // Method that performs the comparison.  It delegates
    // the actual work to the function pointed to by comp.
    // This method logs each invocation.
    bool compare(int m, int n) {
        compare_count++;
        return comp(m, n);
    }

    // Method that performs the swap.
    // Interchange the values of
    // its parameters a and b which are
    // passed by reference.
    // This method logs each invocation.
    void swap(int& m, int& n) {
        swap_count++;
        int temp = m;
        m = n;
        n = temp;
    }

    // Returns the number of comparisons this object has
    // performed since it was created.
    int comparisons() const {
```

```
        return compare_count;
    }

    // Returns the number of swaps this object has
    // performed since it was created.
    int swaps() const {
        return swap_count;
    }
};

/*
 *   selection_sort(a, compare)
 *       Arranges the elements of vector a in an order determined
 *       by the compare object.
 *       a is a vector of ints.
 *       compare is a function that compares the ordering of
 *            two integers.
 *       The contents of a are physically rearranged.
 */
void selection_sort(std::vector<int>& a, Comparer& compare) {
    int n = a.size();
    for (int i = 0;  i < n - 1;  i++) {
        // Note: i,small, and j represent positions within a
        // a[i], a[small], and a[j] represents the elements at
        // those positions.
        // small is the position of the smallest value we've seen
        // so far; we use it to find the smallest value less
        // than a[i]
        int small = i;
        // See if a smaller value can be found later in the array
        for (int j = i + 1;  j < n;  j++)
            if (compare.compare(a[j], a[small]))
                small = j;  // Found a smaller value
        // Swap a[i] and a[small], if a smaller value was found
        if (i != small)
            compare.swap(a[i], a[small]);
    }
}

/*
 *  print
 *     Prints the contents of an integer vector
 *     a is the vector to print.
 *     a is not modified.
 */
void print(const std::vector<int>& a) {
    int n = a.size();
    std::cout << '{';
    if (n > 0) {
        std::cout << a[0];  // Print the first element
        for (int i = 1; i < n; i++)
            std::cout << ',' << a[i];  // Print the rest
    }
    std::cout << '}';
}
```

```
/*
 *  less_than(a, b)
 *     Returns true if a < b; otherwise, returns
 *     false.
 */
bool less_than(int a, int b) {
    return a < b;
}

/*
 *  greater_than(a, b)
 *     Returns true if a > b; otherwise, returns
 *     false.
 */
bool greater_than(int a, int b) {
    return a > b;
}

int main() {
   // Make a vector of integers from an array
   std::vector<int> original { 23, -3, 4, 215, 0, -3, 2, 23, 100, 88, -10 };

   // Make a working copy of the original vector
   std::vector<int> working = original;
   std::cout << "Before:   ";
   print(working);
   std::cout << '\n';
   Comparer lt(less_than), gt(greater_than);
   selection_sort(working, lt);
   std::cout << "Ascending:  ";
   print(working);
   std::cout << "  (" << lt.comparisons() << " comparisons, "
             << lt.swaps() << " swaps)\n";
   std::cout << "----------------------------\n";
   // Make another copy of the original vector
   working = original;
   std::cout << "Before:   ";
   print(working);
   std::cout << '\n';
   selection_sort(working, gt);
   std::cout << "Descending: ";
   print(working);
   std::cout << "  (" << gt.comparisons() << " comparisons, "
             << gt.swaps() << " swaps)\n";
   std::cout << "----------------------------\n";
   // Sort a sorted vector
   std::cout << "Before:   ";
   print(working);
   std::cout << '\n';
   // Reset the greater than comparer so we start counting at
   // zero
   gt.reset();
   selection_sort(working, gt);
   std::cout << "Descending: ";
```

```
    print(working);
    std::cout << "  (" << gt.comparisons() << " comparisons, "
              << gt.swaps() << " swaps)\n";
}
```

Notice that a `Comparison` object wraps a comparison function pointer, contains a `swap` method, and maintains two integer counters. The comparison object passed to the sort routine customizes the sort's behavior (via its function pointer) and keeps track of the number of comparisons and swaps it performs (via its integer counters). As in Listing 12.2 (flexibleintsort.cpp), the basic structure of the sorting algorithm remains the same regardless of the ordering determined by the comparison object.

The output of Listing 16.5 (loggingflexiblesort.cpp) is

```
Before:   {23,-3,4,215,0,-3,2,23,100,88,-10}
Ascending:  {-10,-3,-3,0,2,4,23,23,88,100,215}  (55 comparisons, 7 swaps)
----------------------------
Before:   {23,-3,4,215,0,-3,2,23,100,88,-10}
Descending: {215,100,88,23,23,4,2,0,-3,-3,-10}  (55 comparisons, 5 swaps)
----------------------------
Before:   {215,100,88,23,23,4,2,0,-3,-3,-10}
Descending: {215,100,88,23,23,4,2,0,-3,-3,-10}  (55 comparisons, 0 swaps)
```

We see from the results that the number of comparisons is dictated by the algorithm itself, but the number of element swaps depends on the ordering of the elements and the nature of the comparison. Sorting an already sorted array with selection sort does not reduce the number of comparisons the function must perform, but, as we can see, it requires no swaps.

16.4 Automating Testing

We know that a clean compile does not imply that a program will work correctly. We can detect errors in our code as we interact with the executing program. The process of exercising code to reveal errors or demonstrate the lack thereof is called testing. The informal testing that we have done up to this point has been adequate, but serious software development demands a more formal approach. As you gain more experience developing software you will realize that good testing requires the same skills and creativity as programming itself.

Until recently testing was often an afterthought. Testing was not seen to be as glamorous as designing and coding. Poor testing led to buggy programs that frustrated users. Also, tests were written largely after the program's design and coding were complete. The problem with this approach is major design flaws may not be revealed until late in the development cycle. Changes late in the development process are invariably more expensive and difficult to deal with than changes earlier in the process.

Weaknesses in the standard approach to testing led to a new strategy: test-driven development. In test-driven development the testing is automated, and the design and implementation of good tests is just as important as the design and development of the actual program. In pure TDD, developers write the tests before writing any application code and immediately test all application code they write.

Listing 16.6 (tester.h) defines the structure of a rudimentary test object.

Listing 16.6: `tester.h`

```
#ifndef TESTER_H_
#define TESTER_H_
```

```
#include <vector>
#include <string>

class Tester {
    int error_count;  // Number of errors detected
    int total_count;  // Number of tests executed

    // Determines if double-precision floating-point
    // values d1 and d2 are "equal."
    // Returns true if their difference is less than tolerance.
    bool equals(double d1, double d2, double tolerance) const;

    // Displays vector a in human-readable form
    void print_vector(const std::vector<int>& a);
public:
    // Initializes a Tester object
    Tester();

    // Determines if an expected integer result (expected)
    // matches the actual result (actual).  msg is the message
    // that describes the test.
    void check_equals(const std::string& msg, int expected, int actual);

    // Determines if an expected double result (expected)
    // matches the actual result (actual) or they differ by at
    // most tolerance.  msg is the message that describes the test.
    void check_equals(const std::string& msg, double expected,
                      double actual, double tolerance);

    // Determines if an expected string result (expected)
    // matches the actual result (actual).  msg is the message
    // that describes the test.
    void check_equals(const std::string& msg,
                      const std::vector<int>& expected,
                      const std::vector<int>& actual);

    // Reports the final results: number of tests passed and
    // failed and the total number of tests run.
    void report_results() const;
};

#endif
```

A simple test object keeps track of the number of tests performed and the number of failures. The client uses the test object to check the results of a computation against a predicted result. Notice that the `equals` method, which checks for the equality of two double-precision floating-point numbers is private, as it is meant to be used internally by the other methods within the class. The `equals` method works the same way as the `equals` function we examined in Listing 9.17 (floatequals.cpp).

Listing 16.7 (tester.cpp) implements the `Tester` methods.

Listing 16.7: `tester.cpp`

```
#include <iostream>
```

```
#include <cmath>
#include "tester.h"

Tester::Tester(): error_count(0), total_count(0) {
    std::cout << "+----------------------------------------\n";
    std::cout << "|  Testing \n";
    std::cout << "+----------------------------------------\n";
}

// d1 and d2 are "equal" if their difference is less than
// a specified tolerance
bool Tester::equals(double d1, double d2, double tolerance) const {
    return d1 == d2 || abs(d1 - d2) < tolerance;
}

// Prints the contents of a vector of integers.
void Tester::print_vector(const std::vector<int>& a) {
    int n = a.size();
    std::cout << '{';
    if (n > 0) {
        std::cout << a[0];  // Print the first element
        for (int i = 1; i < n; i++)
            std::cout << ',' << a[i];  // Print the rest
    }
    std::cout << '}';
}

// Compare integer outcomes
void Tester::check_equals(const std::string& msg, int expected,
                          int actual) {
    std::cout << "[" << msg << "] ";
    total_count++;   // Count this test
    if (expected == actual)
        std::cout << "OK\n";
    else {
        error_count++;  // Count this failed test
        std::cout << "*** Failed!  Expected: " << expected
                  << ", actual: " << actual << '\n';
    }
}

// Compare double-precision floating-point outcomes
void Tester::check_equals(const std::string& msg, double expected,
                          double actual, double tolerance) {
    std::cout << "[" << msg << "] ";
    total_count++;   // Count this test
    if (equals(expected, actual, tolerance))
        std::cout << "OK\n";
    else {
        error_count++;  // Count this failed test
        std::cout << "*** Failed!  Expected: " << expected
                  << ", actual: " << actual << '\n';
    }
}
```

```
// Compare vector outcomes
void Tester::check_equals(const std::string& msg,
                          const std::vector<int>& expected,
                          const std::vector<int>& actual) {
    std::cout << "[" << msg << "] ";
    total_count++;   // Count this test
    if (expected == actual)
        std::cout << "OK\n";
    else {
        error_count++;  // Count this failed test
        std::cout << "*** Failed!  Expected: ";
        print_vector(expected);
        std::cout << "   Actual:   ";
        print_vector(actual);
        std::cout << '\n';
    }
}

// Display final test statistics
void Tester::report_results() const {
    std::cout << "+-------------------------------------\n";
    std:cout << "| " << total_count << " tests run, "
              << total_count - error_count << " passed, "
              << error_count << " failed\n";
    std::cout << "+-------------------------------------\n";

}
```

Listing 16.8: testvectorstuff.cpp

```
#include <iostream>
#include <vector>
#include "tester.h"

// sort has a bug (it does not do anyting)
void sort(std::vector<int>& vec) {
    // Not yet implemented
}

// sum has a bug (misses first element)
int sum(const std::vector<int>& vec) {
    int total = 0;
    for (size_t i = 1; i < vec.size(); i++)
        total += vec[i];
    return total;
}

int main() {
    Tester t;  // Declare a test object
    // Some test cases to test sort
    std::vector<int> vec { 4, 2, 3 };
    sort(vec);
    t.check_equals("Sort test #1", {2, 3, 4}, vec);
    vec = {2, 3, 4};
```

```
    sort(vec);
    t.check_equals("Sort test #2", {2, 3, 4}, vec);
    // Some test cases to test sum
    t.check_equals("Sum test #1", sum({0, 3, 4}), 7);
    t.check_equals("Sum test #2", sum({-3, 0, 5}), 2);
}
```

The program's output is

```
+---------------------------------------
|  Testing
+---------------------------------------
[Sort test #1] *** Failed!  Expected: {2,3,4}   Actual:   {4,2,3}
[Sort test #2] OK
[Sum test #1] OK
[Sum test #2] *** Failed!  Expected: 5, actual: 2
```

Notice that the `sort` function has yet to be implemented, but we can test it anyway. The first test is bound to fail. The second test checks to see if our `sort` function will not disturb an already sorted vector, and we pass this test with no problem. This is an example of *coincidental correctness*.

In the `sum` function, the programmer was careless and used 1 as the beginning index for the vector. Notice that the first test does not catch the error, since the element in the zeroth position (zero) does not affect the outcome. A tester must be creative and devious to try and force the code under test to demonstrate its errors.

16.5 Convenient High-quality Pseudorandom Numbers

In Section 13.5 we used some classes from the standard C++ library to generate high-quality pseudorandom numbers. Listing 13.11 (highqualityrandom.cpp) used three kinds of objects—`random_device`, `mt19937`, and `uniform_int_distribution`—to produce good pseudorandom sequences.

The C++ class construct allows us to creatively combine multiple sources of functionality into one convenient package. Listing 16.9 (uniformrandom.h) contains a custom `UniformRandomGenerator` class.

Listing 16.9: `uniformrandom.h`

```
#ifndef UNIFORM_RANDOM_DEFINED_
#define UNIFORM_RANDOM_DEFINED_

#include <random>

class UniformRandomGenerator {

    // A uniform distribution object
    std::uniform_int_distribution<int> dist;
    // A Mersenne Twister random number generator with a seed
    // obtained from a random_device object
    std::mt19937 mt;
public:
    // The smallest pseudorandom number this generator can produce
    const int MIN;
```

```
    // The largest pseudorandom number this generator can produce
    const int MAX;

    // Create a pseudorandom number generator that produces values in
    // the range low...high
    UniformRandomGenerator(int low, int high) : dist(low, high),
        mt(std::random_device()()),
        MIN(low), MAX(high) {}

    // Return a pseudorandom number in the range MIN...MAX
    int operator()() {
        return dist(mt);
    }
};
#endif
```

The `UniformRandomGenerator` class provides a simplified interface to programmers who need access to high-quality pseudorandom numbers. Behind the scenes, every `UniformRandomGenerator` object contains its own `uniform_int_distribution` object and `mt19937` object. The constructor accepts the minimum and maximum values in the range of pseudorandom numbers desired. The constructor uses this range to construct the appropriate `uniform_int_distribution` object for this range. The constructor also initializes the `mt19937` object field. The `UniformRandomGenerator` constructor passes to the constructor of the `mt19937` class a temporary `random_device` object. Since a `UniformRandomGenerator` object uses the `random_device` only for creating its `mt19937` field and does not need it later, `UniformRandomGenerator` objects do not contain a `random_device` field.

To create a `UniformRandomGenerator` object that produces pseudorandom integers in the range $-100 \ldots 100$, a client need only write

```
UniformRandomGenerator gen(-100, 100);
```

The `UniformRandomGenerator` class also provides an `operator()` method. This allows a client to "call" an object as if it were a function. Given a `UniformRandomGenerator` object named `gen`, we can assign a pseudorandom number to an integer variable `x`, with the statement

```
int x = gen();
```

This statement may appear to be calling a global function named `gen`; in fact, it is calling the `operator()` method of the `UniformRandomGenerator` class on behalf of object `gen`. The expression `gen()` is syntactic sugar for `gen.operator()()`. The expression `gen.operator()()` may look unusual, but it simply is invoking the `UniformRandomGenerator::operator()` method on behalf of `gen` passing no arguments in the empty last pair of parentheses.

Listing 16.10 (testuniformrandom.cpp) is a simplified remake of Listing 13.11 (highqualityrandom.cpp). In Listing 16.10 (testuniformrandom.cpp) we see that with our `UniformRandomGenerator` class we can generate high-quality pseudorandom numbers with a single object that is simple to use.

Listing 16.10: `testuniformrandom.cpp`

```
#include <iostream>
#include <iomanip>
#include "uniformrandom.h"
```

```
int main() {
    // Pseudorandom number generator with range 0...9,999
    UniformRandomGenerator rand(0, 9999);

    // Total counts over all the runs.
    // Make these double-precision floating-point numbers
    // so the average computation at the end will use floating-point
    // arithmetic.
    double total5 = 0.0, total9995 = 0.0;

    // Accumulate the results of 10 trials, with each trial
    // generating 1,000,000,000 pseudorandom numbers
    const int NUMBER_OF_TRIALS = 10;

    for (int trial = 1; trial <= NUMBER_OF_TRIALS; trial++) {
        // Initialize counts for this run of a billion trials
        int count5 = 0, count9995 = 0;
        // Generate one billion pseudorandom numbers in the range
        // 0...9,999 and count the  number of times 5 and 9,995 appear
        for (int i = 0; i < 1000000000; i++) {
            // Generate a pseudorandom number in the range 0...9,999
            int r = rand();
            if (r == 5)
                count5++;     // Number 5 generated, so count it
            else if (r == 9995)
                count9995++;  // Number 9,995 generated, so count it
        }
        // Display the number of times the program generated 5 and 9,995
        std::cout << "Trial #" << std::setw(2) << trial << "  5: "
                  << std::setw(6) << count5
                  << "    9995: " << std::setw(6) << count9995 << '\n';
        total5 += count5;         // Accumulate the counts to
        total9995 += count9995;   // average them at the end
    }
    std::cout << "-------------------\n";
    std::cout << "Averages for " << NUMBER_OF_TRIALS << " trials:  5: "
              << std::setw(6) << total5 / NUMBER_OF_TRIALS << "  9995: "
              << std::setw(6) << total9995 / NUMBER_OF_TRIALS << '\n';
}
```

Note that in Listing 16.10 (testuniformrandom.cpp), the statement

```
int r = rand();
```

is not a call to our familiar `std::rand` function; rather, `rand` here is a `UniformRandomGenerator` object, and the statement is invoking its `operator()` method.

16.6 Exercises

1. Create a large unsigned integer type named `BigUnsigned`. `BigUnsigned` objects represent unsigned integers with arbitrary precision; that is, unlike the standard C++ unsigned integer primitive types, a `BigUnsigned` object can represent an unsigned integer as large as necessary. Unlike the floating-point types, a `BigUnsigned` value retains all its digits of precision.

Internally, the `BigUnsigned` class should hold a `std::vector` of integers. Each integer in the vector is one of 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9. Each element in the vector represents a digit in a place value within the integer; for example, if the internal vector contains the following elements in the following order:

$$4,9,1,1,4,3,0,5$$

we would interpret the associated `BigUnsigned` object as the mathematical nonnegative integer 49,114,305.

Your `BigUnsigned` class implementation should provide the following features:

- The class should provide a constructor that accepts no arguments that initializes the `BigUnsigned` object's vector to contain a single element equal to zero.
- The class should provide a constructor that accepts a single unsigned integer. This constructor should populate its internal vector with the appropriate elements to correspond to the value of its parameter.
- The class should provide a constructor that accepts a `BigUnsigned` argument. Clients use this constructor to create a new `BigUnsigned` object from an exiting `BigUnsigned` object.
- The class should provide a constructor that accepts a `std::string` object representing an integer. Clients use this constructor when they need to create a large integer whose value exceeds the range of `unsigned long long`. The string argument should contain only digits.
- The class should provide access to a `friend` function named `operator+` that adds two `BigUnsigned` objects and returns the `BigUnsigned` result.
- The class should provide access to a `friend` function named `operator<<` that allows clients to print a `BigUnsigned` value as easily as a built-in integer type.

Chapter 17

Inheritance and Polymorphism

In Chapter 16 we saw how it is possible to design classes from which clients can produce objects that exhibit somewhat sophisticated behavior. We built each of the classes from scratch. C++ provides a way to create a class from an existing class by a process known as *inheritance*. Through this process the new class inherits all the characteristics of the existing class, and the class developer can extend and customize the inherited functionality as well as add new features.

17.1 I/O Stream Inheritance

Recall from Section 13.3 that writing data to a file is almost as easy as printing it on the screen. Once a `std::ofstream` object is set up, we can use the `<<` operator in the same way we use it with the `std::cout` object:

```
std::ofstream fout("myfile.dat");
int x = 10;
if (fout.good()) {   // Make sure the file was opened properly
    fout << "x = " << x << '\n';
}
else
    std::cout << "Unable to write to the file \"myfile.dat\"\n";
```

Section 15.7 showed how `operator<<` may be overloaded for a programmer-defined type. For a vector of integers we can write the function

```
ostream& operator<<(ostream& os, const std::vector<int>& vec) {
    os << '{';
    int n = vec.size();
    if (n > 0) {     // Is the vector non-empty?
        os << vec[0];  // Send first element to the strean
        for (int i = 1; i < n; i++)
            os << ',' << vec[i];   // Send remaining elements
    }
    os << '}';
    return os;
}
```

©2018 Richard L. Halterman

Can we easily adapt our << operator so that we can use it with `std::ofstream` objects as well?

The answer, perhaps surprisingly, is we do not have to adapt our `operator<<` function; it works as is with `std::ofstream` objects. How can this be, since the `std::cout` object has type `std::ostream` which is not the same class as `std::ofstream`?

C++ allows some automatic conversions among the built in numeric types; for example, an `int` is widened to a `double`, and a `double` is truncated to an `int`. `bool`s and `int`s are freely interconvertible. These types are built into the language, so standard conversions apply. When it comes to programmer-defined types, however, the compiler is unable to generate automatically code to convert from one type to another. Most of the time it would make no sense to do so—imagine attempting to convert an object of type `Stopwatch` (Listing 16.2 (stopwatch.h)) to an object of type `Tester` (Listing 16.6 (tester.h)).

So, how is it that an object of type `std::ofstream` can be converted automatically to an object of type `ostream`? You might think that it is because they are both part of the standard library, so the conversion has been programmed into them. The fact is, no conversion takes place! Any object of type `std::ofstream` object is automatically treated as if it were an `ostream` object because the `ostream` and `std::ofstream` classes are related in a special way. The `std::ofstream` class is *derived* from the `ostream` class. We say that `ostream` is the *base class* and `std::ofstream` is the *derived class*. Sometimes the term *superclass* is used for the base class, in which case the derived class is called the *subclass*. The base/derived class terminology is used interchangeably with super/subclass, although the C++ community tends to prefer the terms base/derived class to super/subclass. The process is known as *inheritance* because the derived class inherits all the characteristics of its base class. The terms *derivation*, *subclassing* and *specialization* are sometimes used in place of the term *inheritance*.

As a consequence of the derivation process, an instance of the derived class can be treated as if it were an instance of the base class. Listing 17.1 (printstreams.cpp) shows how a function can process instances of two different classes related by inheritance.

Listing 17.1: `printstreams.cpp`

```
#include <fstream>

void print(std::ostream& os, int n) {
    os.width(10);  // Right justified in 10 spaces
    os.fill('*');  // Fill character is *
    os << n << '\n';
}

int main() {
    // Pretty print to screen
    print(std::cout, 35);

    // Pretty print to text file
    std::ofstream fout("temp.data");
    if (fout.good()) {
        print(fout, 36);
    }
}
```

Observe that the `print` function in Listing 17.1 (printstreams.cpp) expects a client to pass an `ostream` reference as the first parameter, but the `main` function can send it both `ostream` and `std::ofstream` object references with equal ease.

This ability to allow a subclass object to be used in any context that expects a superclass object is known as the *Liskov Substitution Principle*, after computer scientist Barbara Liskov.

The term *is a* has a special meaning in the context of inheritance. Suppose we have class B and class D derived from B. B is the base class and D is the derived class. Since we can treat an instance of a derived class as if it were an instance of its base class, if we declared a D object as

```
D d;  // d is a D object
```

we can say d *is a* D (as it is declared), and we also can say d *is a* B.

17.2 Inheritance Mechanics

The developers of the `std::ofstream` class did not begin with a blank slate. The `ostream` class existed first, and the developers specified the `std::ofstream` class in such a way so any `std::ofstream` object would be treated as a specific kind of `ostream`. In this section we will examine a very simple example that illustrates the mechanics of class inheritance in C++.

Suppose we have class B defined as shown here:

```
class B {
    // Details omitted
};
```

To derive a new class D from B we use the following syntax:

```
class D: public B {
    // Details omitted
};
```

Here B is the base class and D is the derived class. B is the pre-existing class, and D is the new class based on B.

To see how inheritance works, consider classes B and D with additional detail:

```
class B {
    // Other details omitted
public:
    void f();
};

void B::f() {
    std::cout << "In function 'f'\n";
}

class D: public B {
    // Other details omitted
public:
    void g();
};

void D::g() {
    std::cout << "In function 'g'\n";
```

```
}
```

The client code

```
B myB;
D myD;
myB.f();
myD.f();
myD.g();
```

prints

```
In function 'f'
In function 'f'
In function 'g'
```

Even though the source code for class D does not explicitly show the definition of a method named f, it has such a method that it inherits from class B.

Note that inheritance works in one direction only. Class D inherits method f from class B, but class B cannot inherit D's g method. Given the definitions of classes B and D above, the following code is illegal:

```
B myB;
myB.g();  // Illegal, a B object is NOT a D object
```

If we omit the word public from class D's definition, as in

```
class D: B {
    // Details omitted
};
```

all the public members of B inherited by D objects will be *private* by default; for example, if base class B looks like the following:

```
class B {
public:
    void f();
};

void B::f() {
    std::cout << "In function 'f'\n";
}
```

the following code is not legal:

```
D myD;
myD.f();  // Illegal, method f now is private!
```

This means a client may not treat a D object exacty as if it were a B object. This violates the Liskov Substitution Principle, and the *is a* relationship does not exist.

While this *private inheritance* is useful in rare situations, the majority of object-oriented software design uses public inheritance. C++ is one of the few object-oriented languages that supports private inheritance.

There is no limit to the number of classes a developer may derive from a single base class. In the following code:

```
class D1: public B { /* Details omitted */ };
class D2: public B { /* Details omitted */ };
class D3: public B { /* Details omitted */ };
```

the classes D1, D2, and D3 are all derived from class B.

A developer may derive a class from more than one base class in a process known as *multiple inheritance*. In the following code:

```
class B1 { /* Details omitted */ };
class B2 { /* Details omitted */ };

class D: public B1, public B2 { /* Details omitted */ };
```

Here class D has two base classes, B1 and B2. If a particular function or method expects a B1 object as an argument, the caller may pass a D object; similarly, any context that requires a B2 object will accept a D object. In object-oriented design, multiple inheritance is not as common as single inheritance (one base class).

The next section provides a simple example that shows how inheritance works.

17.3 Uses of Inheritance

Inheritance is a design tool that allows developers to take an existing class and produce a new class that provides enhanced behavior or different behavior. The enhanced or new behavior does not come at the expense of existing code; that is, when using inheritance programmers do not touch any source code in the base class. Also, developers can leverage existing code (in the base class) without duplicating it in the derived classes.

We can demonstrate the use of inheritance to enhance the behavior of a simple class. Listing 17.2 (text.h) contains the declaration of a class named Text.

Listing 17.2: text.h

```
#ifndef TEXT_H_INCLUDED
#define TEXT_H_INCLUDED

#include <string>

// Base class for all Text derived classes
class Text {
    std::string text;
public:
    // Create a Text object from a client-supplied string
    Text(const std::string& t);

    // Allow clients to see the text field
    virtual std::string get() const;

    // Concatenate another string onto the
    // back of the existing text
```

```
    virtual void append(const std::string& extra);
};

#endif
```

A `Text` object wraps a `std::string` object, and, since the string object is private to the `Text` class, clients cannot get to the string object directly. The clients may see the string via the `get` method, and can modify the wrapped string only in a limited way via the `append` method. Notice that the two methods, `get` and `append`, are marked `virtual`. The `virtual` keyword indicates that the designer of this `Text` class intends that classes derived from `Text` will be able to customize the behavior of their `get` and `append` methods.

The implementation of the `Text` methods as shown in Listing 17.3 (text.cpp) contains nothing unusual.

Listing 17.3: `text.cpp`

```
#include "text.h"

// Create a Text object from a client-supplied string
Text::Text(const std::string& t): text(t) {}

// Allow clients to see the text field
std::string Text::get() const {
    return text;
}

// Concatenate another string onto the
// back of the existing text
void Text::append(const std::string& extra) {
    text += extra;
}
```

Next we introduce a new class, `FancyText`, derived from the `Text` class. Its interface appears in Listing 17.4 (fancytext.h)

Listing 17.4: `fancytext.h`

```
#ifndef FANCYTEXT_H_INCLUDED
#define FANCYTEXT_H_INCLUDED

// Compiler needs to know specifics of the Text class in order to
// derive a new class from it
#include "text.h"

// Provides minimal decoration for the text
class FancyText: public Text {
    std::string left_bracket;
    std::string right_bracket;
    std::string connector;
public:
    // Client supplies the string to wrap plus some extra
    // decorations
    FancyText(const std::string& t, const std::string& left,
              const std::string& right, const std::string& conn);
```

```
    // Allow clients to see the decorated text field
    std::string get() const override;

    // Concatenate another string onto the
    // back of the existing text, inserting the connector
    // string
    void append(const std::string& extra) override;
};

#endif
```

Observe that the declarations of the `get` and `append` methods include the `override` specifier. This keyword indicates that the `FancyText` class overrides, or provides custom behavior for, the methods declared `virtual` in its base class.

Listing 17.5 (fancytext.cpp) provides the implementation for the `FancyText` methods.

Listing 17.5: `fancytext.cpp`

```
#include "fancytext.h"

// Client supplies the string to wrap plus some extra
// decorations
FancyText::FancyText(const std::string& t, const std::string& left,
                     const std::string& right, const std::string& conn):
           Text(t), left_bracket(left),
           right_bracket(right), connector(conn) {}

// Allow clients to see the decorated text field
std::string FancyText::get() const {
    return left_bracket + Text::get() + right_bracket;
}

// Concatenate another string onto the
// back of the existing text, inserting the connector
// string
void FancyText::append(const std::string& extra) {
    Text::append(connector + extra);
}
```

While the constructor for the `Text` class requires a single std::string parameter, the constructor for `FancyText` requires four string arguments:

```
FancyText::FancyText(const std::string& t, const std::string& left,
                     const std::string& right, const std::string& conn):
           Text(t), left_bracket(left),
           right_bracket(right), connector(conn) {}
```

The body of the constructor is empty, so all initialization happens in the constructor initialization list. We want to assign the constructor's first parameter, `t`, to the inherited member `text`, but `text` is private in the base class. This means the `FancyText` constructor cannot initialize it directly. Since the constructor of its base class knows what to do with this parameter, the first expression in the constructor initialization list (the part between the : and the):

```
... Text(t) ...
```

explicitly calls the base class constructor, passing it `t`. This base class initialization expression must be the first thing in the initialization list because C++ requires that the parts of an object inherited from its base class must be initialized before any new parts (added by the derived class) are initialized. The next three expressions in the initialization list:

```
... left_bracket(left), right_bracket(right), connector(conn) ...
```

initialize the `left_bracket`, `right_bracket`, and `connector` fields of the `FancyText` object under construction. The body of the constructor is empty as no other initialization work is necessary.

The `FancyText` class alters the way `get` works. We say the `FancyText` class *overrides* its inherited `get` method. The keyword `override` emphasizes the fact that the code in the `get` method in `FancyText` intends to do something differently from the code in `Text`'s `get` method. In this case the `FancyText::get` method builds a string result by concatenating three other strings: the first string is a front bracketing string, the second is the wrapped string, and the third string is a trailing bracketing string. Notice that the second string is obtained with the expression

```
Text::get()
```

This calls the base class version of the `get` method. We say that the `FancyText::get` method delegates some of its work to the base class version, `Text::get`. Notice that the statement

```
return left_bracket + text + right_bracket;  // Illegal
```

would be illegal because `text` is private to the base class. The member specifier `private` means inaccessible outside of the class, period. Derived classes have no special privilege in this regard. In order to perform the delegation we must use `Text::get()` and not simply `get()`; the unqualified expression `get()` is equivalent to `this->get`, which calls `FancyText::get`. This means invoking the unqualified `get` within the definition of `FancyText::get` is a recursive call, which is not what we want.

A method declared `virtual` in a base class is automatically virtual in its derived classes. You may repeat the `virtual` specifier in derived classes, but the use of `override` makes this redundant.

The `override` keyword was added to the language in C++11. Prior to C++11 when a method in a derived class had the same signature as a virtual method in its base class, the method implicitly overrode its base class version. The problem was that a programmer could intend to override a method in the derived class but get the signature wrong. The resulting method *overloaded* the original method rather than overriding it. If a programmer uses the `override` specifier and uses a signature that does not match the base class version, the compiler will report an error. The `override` specifier provides a way for programmers to explicitly communicate their intentions.

For backwards compatibility the `override` keyword is optional. Its presence enables the compiler to verify that the method is actually overriding a `virtual` method in the base class. Without it, the programmer must take care to faithfully reproduce the signature of the method to override.

The `override` keyword is a *context-sensitive keyword*, meaning it is a keyword only when appearing as it does here in the declaration of a method header. In other contexts it behaves like an identifier.

The `FancyText::append` method overrides the inherited `append` method by inserting a special separator string in between the existing wrapped text string and the string to append. Like `FancyText::get`, the `FancyText::append` method delegates the actual concatenation operation to its base class by calling `Text::append`. Again, this is because code within the `FancyText` class cannot directly influence its `text` field.

The class `FixedText`, declared and defined in Listing 17.6 (fixedtext.h) and Listing 17.7 (fixedtext.cpp), is another class derived from `Text`.

Listing 17.6: fixedtext.h

```
#ifndef FIXEDTEXT_H_INCLUDED
#define FIXEDTEXT_H_INCLUDED

// Compiler needs to know specifics of the Text class in order to
// derive a new class from it
#include "text.h"

// The text is always the word FIXED
class FixedText: public Text {
public:
    // Client does not provide a string argument; the
    // wrapped text is always "FIXED"
    FixedText();

    // Nothing may be appended to a FixedText object
    void append(const std::string&) override;
};

#endif
```

The `FixedText` class is somewhat simpler than the `FancyText` class. It adds no fields and does not override the `get` method. Its constructor accepts no parameters because clients are not permitted to determine the contents of the wrapped string—the text field is always the word *FIXED*. It inherits the `text` field and `get` method (as is) from the `Text` class.

Listing 17.7: fixedtext.cpp

```
#include "fixedtext.h"

// Client does not provide a string argument; the
// wrapped text is always "FIXED"
FixedText::FixedText(): Text("FIXED") {}

// Nothing may be appended to a FixedText object
void FixedText::append(const std::string&) {
    // Disallow concatenation
}
```

The `FixedText` constructor implementation invokes the base class constructor to assign the string `"FIXED"` to its inherited `text` field. The `get` method needs no customization. The custom `append` method does nothing, as clients are not allowed to add any extra text.

Listing 17.8 (testtext.cpp) exercises these classes.

Listing 17.8: testtext.cpp

```
#include <iostream>
#include "text.h"
#include "fancytext.h"
#include "fixedtext.h"

int main() {
    Text t1("plain");
    FancyText t2("fancy", "<<", ">>", "***");
    FixedText t3;
    std::cout << t1.get() << '\n';
    std::cout << t2.get() << '\n';
    std::cout << t3.get() << '\n';
    std::cout << "------------------------\n";
    t1.append("A");
    t2.append("A");
    t3.append("A");
    std::cout << t1.get() << '\n';
    std::cout << t2.get() << '\n';
    std::cout << t3.get() << '\n';
    std::cout << "------------------------\n";
    t1.append("B");
    t2.append("B");
    t3.append("B");
    std::cout << t1.get() << '\n';
    std::cout << t2.get() << '\n';
    std::cout << t3.get() << '\n';
}
```

The output of Listing 17.8 (testtext.cpp) is

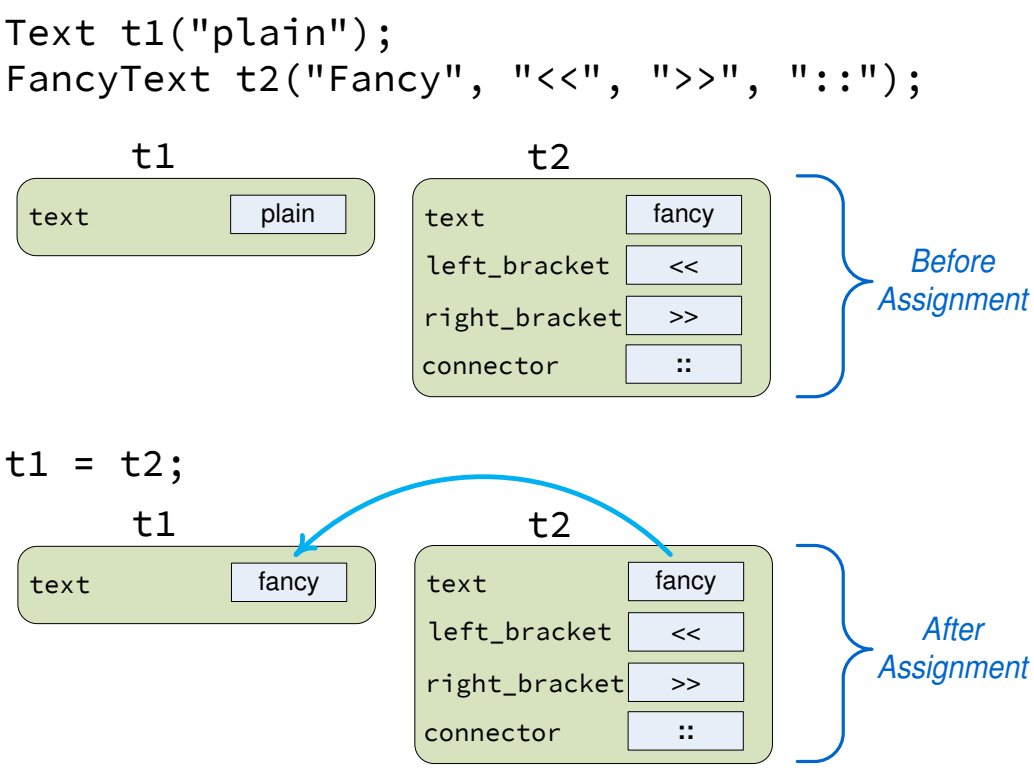
```
plain
<<fancy>>
FIXED
------------------------
plainA
<<fancy***A>>
FIXED
------------------------
plainAB
<<fancy***A***B>>
FIXED
```

Given the `Text` and `FancyText` classes as declared in Listing 17.2 (text.h) and Listing 17.5 (fancytext.cpp), the following client code is legal:

```
Text t1("ABC");
FancyText t2("XYZ", "[", "]", ":");
std::cout << t1.get() << "  " << t2.get() << '\n';
t1 = t2;
std::cout << t1.get() << "  " << t2.get() << '\n';
```

Since `t2`'s declared type is `FancyText`, by virtue of inheritiance, `t2` *is a* `Text` object as well. The above code fragment prints

Figure 17.1 Object slicing during assignment of a derived class instance to a base class variable




```
ABC  <<XYZ>>
XYZ  <<XYZ>>
```

Notice that the assignment

```
t1 = t2;
```

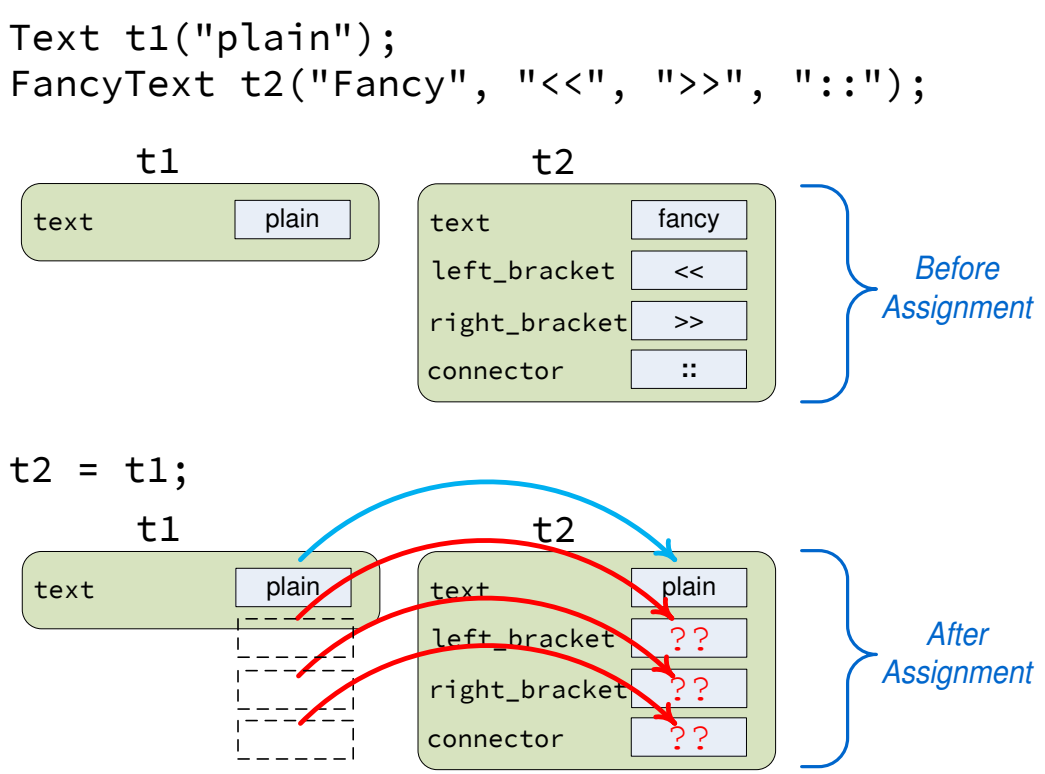
copied into object `t1` only the fields that `FancyText` objects have in common with `Text` objects; that is, the `text` field. Since `t1` is a plain `Text` object, it does not have the `left_bracket`, `right_bracket`, and `connector` fields capable of storing the additional data contained in a `FancyText` object. This process of losing derived class data when assigning to a base class instance is known as *object slicing*. The parts that will not fit into the base class instance are "sliced" off. Figure 17.1 illustrates object slicing.

Note that the assignment in the other direction:

```
t2 = t1;  // Illegal
```

is not possible; even though any `FancyText` object *is a* `Text` object, we cannot say any `Text` object is a `FancyText` object (it could be a `FixedText` object or just a `Text` object without any special decoration). Figure 17.2 shows how such an attempted assignment would be meaningless because the base class instance has missing information needed by the derived class instance. Failure to properly assign all

Figure 17.2 Assigning a base class instance to a derived class variable is not possible and is illegal in C++



the fields in the derived class instance would produce an object that is not well defined, so C++ does not allow the assignment.

`FixedText` class instances do not contain any additional data that plain `Text` instances do not have, but the assignment rules remain the same:

```
Text t1("ABC");
FixedText t3;
std::cout << t1.get() << "  " << t3.get() << '\n';
t1 = t3;
std::cout << t1.get() << "  " << t3.get() << '\n';
```

is legal and produces

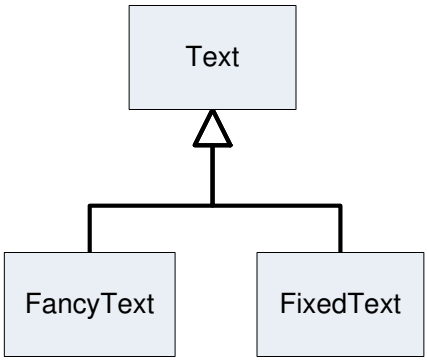
```
ABC  FIXED
FIXED  FIXED
```

but the statement

```
t3 = t1;  //  Illegal
```

is illegal and will not compile.

Figure 17.3 The Unified Modeling Language class hierarchy diagram for the family of text classes



It always is legal to assign a derived class instance to a variable of a base type. This is because a derived class instance *is a* specific kind of base class instance. In contrast, it is not legal to assign a base class instance to a variable of a derived type. This is because the *is a* relationship is only one directional, from a derived class to its base class.

The `Text`, `FancyText`, and `FixedText` classes form a small *class hierarchy*. We can represent the relationships in this class hierarchy in a graphical form using the Unified Modeling Language (UML). Figure 17.3 shows the UML diagram for our simple `Text` class hierarchy.

A rectangle represents a class. `Text`, the base class, appears at the top of the Figure 17.3. The two derived classes appear below `Text`. The inheritance arrow points from the derived classes to the base class. The arrow represents the *is a* relationship which flows upward. This visual layout and level of detail provided by the UML diagram more succinctly communicates the relationships amongst the classes than does the C++ source code.

The UML is a large, complex graphical language that may be used to model many facets of the software development process. More information about the UML is available at `http://www.uml.org`.

17.4 Polymorphism

Recall the `Text` class hierarchy from Listing 17.2 (text.h), Listing 17.4 (fancytext.h), and Listing 17.6 (fixedtext.h) illustrated in Figure 17.3. Consider the following client code fragment:

```
Text t1("ABC");
FancyText t2("XYZ", "[", "]", ":");
std::cout << t1.get() << "  " << t2.get() << '\n';
```

What is the nature of the machine code generated by the expression `t1.get()`? The compiler translates this higher-level expression into the machine language instruction that causes the program's execution to jump to another part of the compiled code. Where does the program's execution jump to? The variable `t1`'s declared type is `Text`, the program's execution jumps to the compiled `Text::get` method (also the address of `t1` is passed as the implicit `this` parameter to the method so the compiled code can access `t1`'s `text` field).

For the expression `t2.get()`, the program's execution jumps to `FancyText`'s compiled `get` method, passing the address of object `t2` as the implicit `this` parameter.

When the compiler can determine which method to execute based on the declared type of an object, the process is known as *static binding* or *early binding*. Static binding is used in all cases for non-virtual methods and in the cases we have seen so far for virtual methods.

The situation is different if we use pointers to objects instead of the objects themselves. Consider the following code:

```
Text t1("ABC");
FancyText t2("XYZ", "[", "]", ":");
std::cout << t1.get() << "  " << t2.get() << '\n';
Text *p1, *p2;
p1 = &t1;
p2 = &t2;
```

The variables `p1` and `p2` are declared to be pointers to `Text` objects. The variable `t1` is a `Text` object, so the assignment

```
p1 = &t1;
```

makes perfect sense. The variable `t2` has a declared type of `FancyText`, but a `FancyText` object *is a* `Text` object, so the assignment

```
p2 = &t2;
```

is legal since `p2` is indeed pointing to a special kind of `Text` object.

What does the following code fragment print?

```
Text t1("ABC");
FancyText t2("XYZ", "[", "]", ":");
std::cout << t1.get() << "  " << t2.get() << '\n';
Text *p1, *p2;
p1 = &t1;
p2 = &t2;
std::cout << p1->get() << "  " << p2->get() << '\n';
```

This code displays

```
ABC <<XYZ>>
ABC <<XYZ>>
```

Even though `p2`'s declared type is "pointer to a `Text` object" not "pointer to a `FancyText` object," the expression `p2->get()` calls `FancyText::get`, not `Text::get`. How does the compiler determine which method to call in this case? The answer may be surprising: The compiler does *not* determine which method to call!

In the case of a virtual method invoked via a pointer, the running program, not the compiler, determines exactly which code to execute. The process is known as *dynamic binding* or *late binding*. Static binding is relatively easy to understand: the method to execute depends on the declared type of the variable upon which the method is invoked. The compiler keeps track of the declared type of every variable, so the choice is easy. Inheritance and the *is a* relationship make things more complicated. In the example above, `p2`'s declared type is "pointer to `Text`." If the compiler were given the authority to select the method to call for the expression `p2->get()`, its only choice would be `Text::get`; however, `p2` actually is pointing to a `FancyText` object. How does the executing program know which code to execute?

We know that every instance of a class contains its own copy of the fields declared within the class. In the code

```
Text word1("Wow"), word2("Wee");
```

the `word1` object's `text` field is the `std::string` object representing the string `"Wow"`, and `word2`'s `text` field is `"Wee"`. In reality, if a class contains at least one virtual method, all instances of that class will contain one extra "hidden" field, a pointer to an array of virtual method pointers. This array of method pointers commonly is called the *vtable*. One vtable exists for each class, but all instances of that class must store a pointer to that shared vtable. The compiler assigns an index to each virtual method in a class. In our example, the `Text` class contains two virtual methods, so the index of `get` might be 0, and the index of `append` might be 1.

Figure 17.4 illustrates a typical scenario with vtables. In this case the compiler translates the expression `p1->get()` into the machine language equivalent of `p1->`*vtable*`[0]()`, which invokes the correct `get` method for the instance pointed to by `p1`. Similarly for the `append` method, `p1->append("suffix")` internally becomes `p1->`*vtable*`[1]("suffix")`. If `p1` points to a `Text` instance, `p1->`*vtable* points to the vtable array for the `Text` class. If `p1` points instead to a `FancyText` instance, `p1->`*vtable* points to the vtable array of the `FancyText` class. In either case the correct method is selected during the program's execution.

You may argue that perhaps the compiler could be made to be able to determine in all situations which virtual method to call. In the code from above:

```
Text t1("ABC");
FancyText t2("XYZ", "[", "]", ":");
std::cout << t1.get() << "  " << t2.get() << '\n';
Text *p1, *p2;
p1 = &t1;
p2 = &t2;
std::cout << p1->get() << "  " << p2->get() << '\n';
```

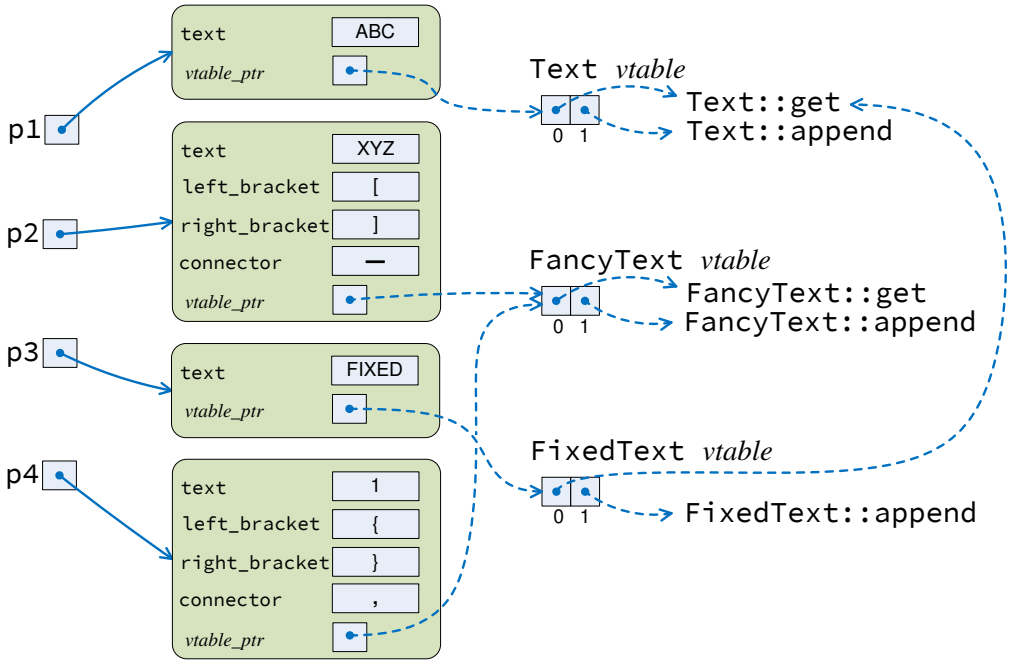
cannot the compiler deduce `p2`'s exact type since `p2` is assigned to point to `t2`, and it knows `t2`'s type is `FancyText`?

Such extended analysis capabilities would further complicate the compiler, and C++ compilers are already very complicated pieces of software. Attempting to add this ability would prove futile anyway because the compiler cannot in general always determine which method to invoke. To see why, consider the following code fragment:

```
Text *p;
if (std::rand() % 2 == 0)
    p = new Text("ABC");
else
    p = new FancyText("XYZ", "[", "]", ":");
std::cout << p->get() << '\n';
```

Figure 17.4 Several objects and their associated vtables. You will not find any evidence in the C++ source code of the dashed pointers and vtables shown in this diagram. When virtual methods are involved the compiler produces executable code that builds these hidden data structures behind the scenes. When a client calls a virtual method via a pointer to an object, the run-time environment locates the exact method to execute by following the vtable pointer stored in the object itself. Observe that since the FixedText class does not override the inherited get method, its vtable references the same get code as that of Text's vtable.

```
Text *p1, *p2, *p3, *p4;
p1 = new Text("ABC");
p2 = new FancyText("XYZ", "[", "]", "-");
p3 = new FixedText;
p4 = new FancyText("1", "{", "}", ",");
```



This code generates a pseudorandom number at run time. If it generates an even number, it directs `p` to point to a plain `Text` object; otherwise, it directs `p` to point to a `FancyText` object. Which method will this code call, `Text::get` or `FancyText::get`? Since the code generates the pseudorandom number at run time, the compiler is powerless to determine the exact type of the object to which `p` points; thus, the compiler cannot determine which `get` method to call. The compiler can, however, generate code that jumps to the method found at the address stored in `p`'s associated vtable at the index corresponding to the `get` method.

Listing 17.9 (vtablesize.cpp) demonstrates that the vtable pointer does occupy space within an object that contains a virtual method.

Listing 17.9: `vtablesize.cpp`

```
#include <iostream>

// A class with no virtual methods
class NoVTable {
    int data;
public:
    void set(int d) { data = d; }
    int get() { return data; }
};

// A class with virtual methods
class HasVTable {
    int data;
public:
    virtual void set(int d) { data = d; }
    virtual int get() { return data; }
};

int main() {
    NoVTable no_vtable;
    HasVTable has_vtable;
    no_vtable.set(10);
    has_vtable.set(10);
    std::cout << "no_vtable size = " << sizeof no_vtable << '\n';
    std::cout << "has_vtable size = " << sizeof has_vtable << '\n';
}
```

The output of Listing 17.9 (vtablesize.cpp) is

```
no_vtable size = 4
has_vtable size = 8
```

A `NoVTable` object requires four bytes for its integer field, but a `HasVTable` object occupies eight bytes—four bytes for its integer field and four bytes for its secret vtable pointer.

Dynamic binding enables a powerful technique in object-oriented programming called *polymorphism*. A polymorphic method behaves differently depending on the actual type of object upon which it is invoked. Consider Listing 17.10 (polymorphicvector.cpp).

Listing 17.10: `polymorphicvector.cpp`

```
#include <string>
```

```
#include <vector>
#include <iostream>
#include "fancytext.h"
#include "fixedtext.h"

int main() {
    std::vector<Text *> texts { new Text("Wow"),
                                new FancyText("Wee", "[", "]", "-"),
                                new FixedText,
                                new FancyText("Whoa", ":", ":", ":") };
    for (auto t : texts)
        std::cout << t->get() << '\n';
    for (auto t : texts) {
        t->append("A");
        t->append("B");
        t->append("C");
    }
    for (auto t : texts)
        std::cout << t->get() << '\n';
    for (auto t : texts)
        delete t;
}
```

We know from Section 11.1 that a vector is a collection of homogeneous elements. *Homogeneous* means the elements in a vector must all be of the same type. Homogeneity takes on a deeper meaning when inheritance and the *is a* relationship is involved. In Listing 17.10 (polymorphicvector.cpp) the declared type of the `texts` vector is `std::vector<Text *>`. With inheritance, not only can the `texts` vector hold pointers to simple `Text` objects, it also simultaneously can hold pointers to `FixedText` and `FancyText` objects. Listing 17.10 (polymorphicvector.cpp) prints

```
Wow
[Wee]
FIXED
:Whoa:
WowABC
[Wee-A-B-C]
FIXED
:Whoa:A:B:C:
```

As we can see, the expression `t->get()` in the `main` function is polymorphic; the actual `get` method invoked—`Text::get`, `FancyText::get`, or `FixedText::get`—depends on the exact type of the object to which `t` points. The `append` method is polymorphic as well. As `t` assumes the value of each element in the vector during the loop's execution, the exact type of object that `t` points to varies. Even though all the elements of the `texts` vector are pointers to `Text` objects, only one of the elements points to a pure `Text` object; the rest of the elements point to `FancyText` or `FixedText` objects.

Why must we use pointers to objects rather than the objects themselves to achieve polymorphism? Remember that a pointer stores a memory address (see Section 10.7). All pointers, no matter what type they point to, are all the same size (4 bytes on 32-bit systems and 8 bytes on 64-bit systems). `Text` objects and `FancyText` objects are not the same size (see Figure 17.1 for a conceptual picture); `FancyText` objects are bigger, containing three extra string fields. All the elements of a vector must be the same size. If we made `texts` a vector of `Text` objects rather than a vector of pointers to `Text` objects, when we assign a `FancyText` object to an element in the `texts` vector, the assignment would slice the extra fields in the `FancyText` object. Pointer assignment avoids the slicing problem.

The main reason for using pointers is that C++ uses static binding for all methods (virtual and non-virtual) invoked on behalf of an object; the compiler chooses the method based on the declared type of the object. In contrast, C++ uses dynamic binding for virtual method calls made via pointers to objects; the exact type of the object determines the method selection.

A polymorphic method in C++ requires four key ingredients:

1. The method must appear in a class that is part of an inheritance hierarchy.
2. The method must declared `virtual` in the base class at the top of the hierarchy.
3. Derived classes override the behavior of the inherited virtual methods as needed.
4. Clients must invoke the method via a pointer (or reference) to an object, not directly through the object itself.

In summary, polymorphism requires inheritance and the *is a* relationship. The base class defines the method signature, and the derived classes override the method with their own custom behavior.

17.5 Alternative to Inheritance and Polymorphism

Consider would life would be like without inheritance and polymorphism. Listing 17.11 (alttext.h) provides an alternative to the `Text` class hierarchy.

Listing 17.11: `alttext.h`

```
#ifndef ALT_TEXT_H_INCLUDED
#define ALT_TEXT_H_INCLUDED

#include <string>

// The kinds of text objects supported
enum class TextType { Plain, Fancy, Fixed };

class AltText {
    TextType type;
    std::string text;
    std::string left_bracket;
    std::string right_bracket;
    std::string connector;
public:
    AltText(TextType type, const std::string& t);
    AltText(TextType type, const std::string& t, const std::string& left,
            const std::string& right, const std::string& conn);
    AltText(TextType type);
    std::string get() const;
    void append(const std::string& extra);
};
```

```
#endif
```

Listing 17.12 (alttext.cpp) provides the method implementations for the `AltText` class.

Listing 17.12: `alttext.cpp`

```
#include "alttext.h"

AltText::AltText(TextType type, const std::string& t): type(type), text(t) {}

AltText::AltText(TextType type, const std::string& t, const std::string& left,
           const std::string& right, const std::string& conn):
     type(type), text(t), left_bracket(left), right_bracket(right),
     connector(conn) {}

AltText::AltText(TextType type): type(type), text("FIXED") {}

std::string AltText::get() const {
    switch (type) {
        case TextType::Plain:
        case TextType::Fixed:
            return text;
        case TextType::Fancy:
            return left_bracket + text + right_bracket;
        default:
            return "UNKNOWN TYPE";
    }
}

void AltText::append(const std::string& extra) {
    switch (type) {
        case TextType::Plain:
           text += extra;  // Just concatenate
           break;
        case TextType::Fancy:
           text += connector + extra;  // Join with connector
           break;
        case TextType::Fixed:
           break;     // Ignore attempt to append
    }
}
```

Listing 17.13 (testalttext.cpp) tests the new `AltText` class.

Listing 17.13: `testalttext.cpp`

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

#include "alttext.h"

int main() {
    std::vector<AltText> texts { {TextType::Plain, "Wow"},
```

```
                                {TextType::Fancy, "Wee", "[", "]", "-"},
                                {TextType::Fixed},
                                {TextType::Fancy, "Whoa", ":", ":", ":"} };
    for (auto t : texts)
        std::cout << t.get() << '\n';
    for (auto& t : texts) {
        t.append("A");
        t.append("B");
        t.append("C");
    }
    for (auto t : texts)
        std::cout << t.get() << '\n';
}
```

From the user's perspective Listing 17.13 (testalttext.cpp) works exactly like Listing 17.10 (polymorphicvector.cpp). From the programmer's perspective, however, the two programs are very different. Listing 17.13 (testalttext.cpp) packages within a single class all the functionality for the three different classes in Listing 17.10 (polymorphicvector.cpp). This non-polymorphic code differentiates among text objects with the `TextType` custom enumeration type. Note that each of the methods that were polymorphic in our earlier example now use conditional logic (a `switch` statement) to execute the code appropriate for actual kind of text object we wish to model.

This approach that avoids inheritance and polymorphism has several different issues:

1. Every `Text` object carries the burdens of every possible kind of text object. A plain text object holds the left bracket, right bracket and separator strings that fancy text objects require. The amount of memory a string occupies depends on its length (number of characters it holds), but even an empty `std:string` object requires some memory. An application managing many mainly plain text objects could be consuming much more memory than it otherwise needs.

2. With inheritance, a developer can introduce a new text type without touching the code in existing text classes. A programmer wishing to introduce a new kind of text object in this example that avoids inheritance must make considerable changes to the code, involving the following:

   - augmenting the enumeration type with a new element identifying the new kind of text object,
   - adding new fields as may be needed by the new kind of text object, and
   - augmenting the `switch` statement in *every* method that" was polymorphic in the inheritance version, adding a `case` label that matches the new kind of text object.

   This is not a small point. Modifying existing code can introduce errors. This can cause existing applications that depend on the `Text` class to fail. With inheritance, programmers add new classes without touching existing code. Pre-existing applications will continue to function as usual because they do not use the new code introduced via inheritance.

3. Without inheritance we need access to the original source code in order to augment the data type. If it is part of a pre-compiled library, we cannot make the necessary modifications. With inheritance, however, we can derive a new class from an existing class even if we do not have access to, or control over, the source code of its implementation.

To illustrate these issues, suppose we wish to add a new kind of text object that bounds the length of the text. Listing **??** (alttext2.h) augments the non-polymorphic `Text` class.

Listing 17.14: `alttext2.cpp`

```
#ifndef ALT_TEXT2_H_INCLUDED
#define ALT_TEXT2_H_INCLUDED

#include <string>

// The kinds of text objects supported (added Bounded)
enum class TextType { Plain, Fancy, Fixed, Bounded };

class AltText2 {
    TextType type;
    std::string text;
    std::string left_bracket;
    std::string right_bracket;
    std::string connector;
    size_t length_limit;   // Maximum number of characters in the text
public:
    AltText2(TextType type, const std::string& t);
    AltText2(TextType type, const std::string& t, const std::string& left,
            const std::string& right, const std::string& conn);
    AltText2(TextType type);
    // Constructor for a length-bounded text object
    AltText2(TextType type, const std::string& t, size_t limit);
    std::string get() const;
    void append(const std::string& extra);
};

#endif
```

We extended the enumeration to include `TextType::Bounded`. We added one more field (`limit`) to the `Text` class, which means every `Text` object will have a `limit` field whether they need it or not. We also added one more constructor overload to properly initialize a bounded text object.

Listing 17.14 (alttext2.cpp) implements the methods in the new `Text` class.

Listing 17.15: `name.cpp`

```
#include "alttext2.h"

AltText2::AltText2(TextType type, const std::string& t): type(type), text(t) {}

AltText2::AltText2(TextType type, const std::string& t, const std::string& left,
           const std::string& right, const std::string& conn):
     type(type), text(t), left_bracket(left), right_bracket(right),
     connector(conn) {}

AltText2::AltText2(TextType type): type(type), text("FIXED") {}

AltText2::AltText2(TextType type, const std::string& t, size_t limit):
      type(type),
      text(t.substr(0, limit)), // Ensure text initially does not exceed bound
      length_limit(limit) {}

std::string AltText2::get() const {
    switch (type) {
        case TextType::Plain:
```

```
        case TextType::Fixed:
        case TextType::Bounded:
            return text;
        case TextType::Fancy:
            return left_bracket + text + right_bracket;
        default:
            return "UNKNOWN TYPE";
    }
}

void AltText2::append(const std::string& extra) {
    switch (type) {
        case TextType::Plain:
           text += extra;  // Just concatenate
           break;
        case TextType::Fancy:
           text += connector + extra;  // Join with connector
           break;
        case TextType::Fixed:
           break;     // Ignore attempt to append
        case TextType::Bounded:
           // Ensure the length does not exceed the established limit
           int avail = length_limit - text.length();
           text += extra.substr(0, avail);
           break;
    }
}
```

Listing 17.16 (testalttext2.cpp) demonstrates that our new bounded text object does indeed limit the length of its enclosed string field.

Listing 17.16: testalttext2.cpp

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

#include "alttext2.h"

int main() {
    std::vector<AltText2> texts { {TextType::Plain, "Wow"},
                                  {TextType::Fancy, "Wee", "[", "]", "-"},
                                  {TextType::Fixed},
                                  {TextType::Bounded, "XYZ", 5},
                                  {TextType::Fancy, "Whoa", ":", ":", ":"} };
    for (auto t : texts)
        std::cout << t.get() << '\n';
    for (auto& t : texts) {
        t.append("A");
        t.append("B");
        t.append("C");
        t.append("D");
        t.append("E");
        t.append("F");
    }
```

```
    for (auto t : texts)
        std::cout << t.get() << '\n';
}
```

Listing 17.16 (testalttext2.cpp) displays

```
Wow
[Wee]
FIXED
XYZ
:Whoa:
WowABCDEF
[Wee-A-B-C-D-E-F]
FIXED
XYZAB
:Whoa:A:B:C:D:E:F:
```

Note that even though our new bounded text objects require no special `get` method, we still must add the `case` label to match the `TextType::Bounded` type. Failure to add this `case` label would mean every bounded text object would respond with the `"UNKNOWN TYPE"` string. We definitely must add new code to the `append` to ensure the concatenation operation does not make bounded text objects too long.

Even if the `get` and `append` methods were known to work flawlessly before we introduced this new kind of text object, they now are suspect. Our act of modifying them could have introduced a subtle bug, and so they must be thoroughly retested.

Our non-polymorphic `Text` class is relatively small, containing only two methods. Consider a more substantial class that provides many more methods. The introduction of a variation of that type could require significant code modification.

Now suppose we do not have access the source code within Listing 17.14 (alttext2.cpp). It may be compiled and part of a library. In this case we cannot modify the `Text` class to introduce the custom behavior we need.

In contrast to these challenges, consider added a bounded text type to our polymorphic `Text` class hierarchy. Listing 17.17 (boundedtext.h) provides the interface for our new class.

Listing 17.17: `boundedtext.h`

```
#ifndef BOUNDEDTEXT_H_INCLUDED
#define BOUNDEDTEXT_H_INCLUDED

// Compiler needs to know specifics of the Text class in order to
// derive a new class from it
#include "text.h"

// Provides minimal decoration for the text
class BoundedText: public Text {
    size_t length_limit;
public:
    // Client supplies the string to wrap plus the limit on
    // how many characters it can hold
    BoundedText(const std::string& t, size_t limit);

    // Concatenate another string onto the back of the existing text,
    // but do not exceed the predetermined limit on the text's length
```

```
    void append(const std::string& extra) override;
};

#endif
```

Listing 17.18 (boundedtext.cpp) implements the `BoundedText` constructor and `append` method.

Listing 17.18: `boundedtext.cpp`

```
#include "boundedtext.h"


BoundedText::BoundedText(const std::string& t, size_t limit):
      Text(t.substr(0, limit)), // Ensure text initially does not exceed bound
      length_limit(limit) {}

void BoundedText::append(const std::string& extra) {
    // Ensure the length does not exceed the established limit
    int avail = length_limit - get().length();
    Text::append(extra.substr(0, avail));
}
```

Listing 17.19 (testbounded.cpp) contains client code to exercise the new `BoundedText` class.

Listing 17.19: `testbounded.cpp`

```
#include <iostream>
#include <vector>
#include "text.h"
#include "fancytext.h"
#include "fixedtext.h"
#include "boundedtext.h"

int main() {
   std::vector<Text *> texts { new Text("Wow"),
                               new FancyText("Wee", "[", "]", "-"),
                               new FixedText,
                               new BoundedText("XYZ", 5),
                               new FancyText("Whoa", ":", ":", ":") };

   for (auto t : texts)
       std::cout << t->get() << '\n';
   for (auto t : texts) {
       t->append("A");
       t->append("B");
       t->append("C");
       t->append("D");
       t->append("E");
       t->append("F");
   }
   for (auto t : texts)
       std::cout << t->get() << '\n';
   // Clean up memory
   for (auto t : texts)
       delete t;
```

```
}
```

This BoundedText class addresses all the concerns we have about the non-polymorphic Text class:

1. A pure Text object will have a string field, period. A Fixed text object as well contains only a single string field. A FancyText object will contain four string fields, and each BoundedText object will maintain a string and integer value. Each particular type of text object will contain only the data it requires, not all the data that may be required for any possible kind of text object.
2. This derived class leaves the original code in the base class untouched. We can do nothing in our BoundedText class code to damage the Text, FancyText, or FixedText classes.
3. We can compile the code within Listing 17.17 (boundedtext.h) and Listing 17.18 (boundedtext.cpp) and build an executable program using BoundedText objects without access to the source code in Listing 17.3 (text.cpp). All we need is the header file Listing 17.2 (text.h) that specifies the interface of the Text class. This header must be available to any clients that wish to use a Text object, but the source code for the implementation of the Text class is not needed to compile client code that uses Text objects. The linker would need the compiled Listing 17.3 (text.cpp) in order to link and produce an executable program, but at no point in the build process is the source code in Listing 17.3 (text.cpp) necessary if its compiled code is available.

The inheritance with polymorphism approach is more modular; that is, if our BoundedText code does not work or is not as efficient as it could be, we or someone else easily can supply a different version to compile and link into an application. This modularity enables multiple programmers to work independently implementing separate classes without interfering with each other.

17.6 Adapter Design Pattern

Suppose a third-party library provides a class named Message that is similar to the classes in our Text hierarchy in Listing 17.10 (polymorphicvector.cpp). A Message object contains a private piece of text and a method named text that reveals the text that it holds. The following code fragment:

```
Message msg("My message");
std::cout << msg.text() << '\n';
```

would print

```
My message
```

In addition to revealing its contained text, each time a client calls a Message object's text method the object records the access. A Message object thus keeps track of the number of times clients have called its text method. The Message class provides a method named access_count that returns the number of text calls. The following code:

```
Message msg("My message");
std::cout << msg.text() << '\n';
std::cout << msg.text() << '\n';
std::cout << msg.text() << '\n';
std::cout << msg.access_count() << '\n';
```

would print

```
My message
My message
My message
3
```

We cannot modify the class `Message`, either because we do not have the source code to this library, or its license prohibits modifying or reverse engineering its code. This means we have no control over the `Message` class or how it works.

What if would like to maintain a collection of `Text` and `Message` objects, storing them all in a single vector? Is this possible? All of the elements in a particular C++ vector must all be of the same type. The `Message` class is not part of the `Text` class hierarchy, so no *is a* relationship exists between the two classes. Let us experiment with two possible solutions to this problem.

Make a copy of a `Message` object. We could try to simply copy the contents of a `Message` object to a `Text` object and put this new `Text` object into the vector. The following code illustrates:

```
// get_message is a function that provides a Message object
Message msg = get_message();
// Put the text inside msg into a new Text object
Text txt(msg.text());
// Insert a pointer to the text object into vector
std::vector<Text *> texts;
texts.push_back(&txt);
std::cout << texts[0]->get() << '\n';
std::cout << texts[0]->get() << '\n';
std::cout << msg.access_count() << '\n';
```

The last line this code fragment prints is

```
1
```

We would rather it printed

```
2
```

because `Text::get` is analogous to `Message::text`, and the code calls the `text` method twice. Unfortunately as you may have noticed, the code calls `text` only once, when creating the `txt` object from the `msg` object. It is this single call that makes the access count become one. The subsequent calls to `get` are on a `Text` object, not a `Message` object. This means we really do not have a `Message` object in our vector, so none of the objects in our vector can take advantage of the services that a `Message` object provides.

Use inheritance and polymorphism to encapsulate `Message` objects within `Text` objects. We can derive a new `Text` class in a clever way to allow us to place `Message` objects into a vector of `Text` objects. Consider the following new class:

```
class MessageText: public Text {
    Message *msg;
public:
    MessageText(Message *m): Text(""), msg(m) {}
    std::string get() const override {
        return msg->text();
    }
```

```
};
```

Instances of this `MessageText` class wrap a pointer to a `Message` object. The constructor does not call the wrapped `Message` object's `text` method, so making a `MessageText` object from a `Message` object does not artificially elevate the access count for the `Message` object. Further, any call to a `MessageText` object's `get` method actually calls the `text` method of the `Message` object it is managing. The following code:

```
// get_message is a function that provides a Message object
Message msg = get_message();
// Put the text inside msg into a new Text object
//Text txt(msg.text());
MessageText txt(&msg);
// Insert a pointer to the text object into vector
std::vector<Text *> texts;
texts.push_back(&txt);
std::cout << texts[0]->get() << '\n';
std::cout << texts[0]->get() << '\n';
std::cout << msg.access_count() << '\n';
```

behaves exactly as desired, reporting

```
2
```

as the last line of its output. This proves that we really did sneak a `Message` object into our vector of `Text` objects.

This technique of adapting the interface of one class to that of another class has a name: the *adapter design pattern*. The adapter design pattern is useful when you have a class with properties similar to another class but with incompatible interfaces. You can use this technique of inheritance and polymorphism to derive a new class with the desired interface

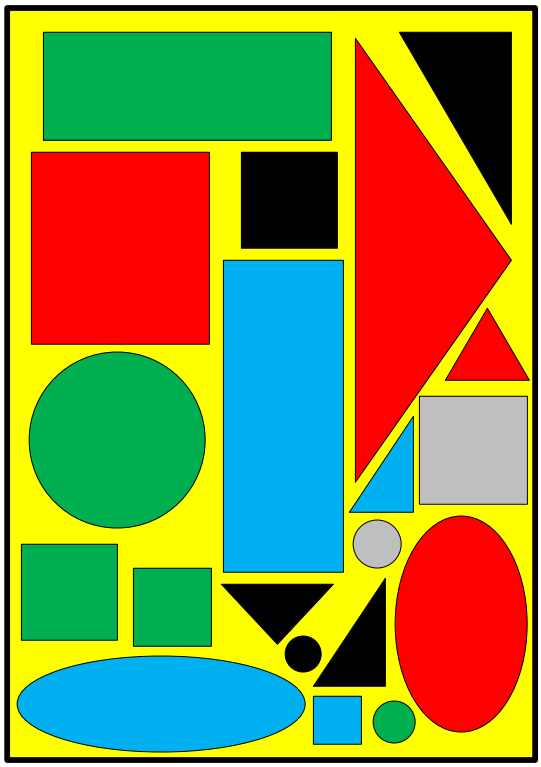
Suppose you have class `X` and class `Y` that describe objects with similar functionality but different interfaces (different methods), and you need to treat a `Y` object as if it were an `X` object. You may be able to use the adapter design pattern as follows:

1. Derive a new class, `Z`, from `X`.
2. Store a pointer to the `Y` object in an instance of class `Z`.
3. Override methods in class `Z` to invoke the corresponding `Y` methods on the enclosed `Y` object.

17.7 Protected Members

As a more practical, yet still relatively simple example, suppose we need to model two-dimensional shape objects, such as rectangles, squares, triangles, ellipses, and circles. Our application will determine how to

Figure 17.5 Packing Two-dimensional Shapes into a Two-dimensional Container



pack parts into a containing area, so it is important to be able to determine an individual shape object's area. Figure 17.5 illustrates packing some two-dimensional shapes into a two-dimensional container.

We thus want to determine the minimum area of the container that can hold a given collection of parts. A real-world application could be computing the size of the smallest circuit board that can hold a collection of electronic components. Our program will be much simpler and give us only a rough approximation; it does not take into account the geometry and orientation of the individual shapes but merely computes the total area of the components. The actual answer generally will be larger than the result computed by our program.

An interesting problem arises if a method in a derived class needs to use an inherited variable. If the variable is declared `private` in the base class, a derived class method cannot access it. The `private` specifier means "not available outside the class, period." We know it is unwise in general to make instance variables public, since that breaks encapsulation. Since encapsulation is a desirable property, C++ provides a third level of access protection within a class—`protected`. A `protected` member, whether it be data or a method, is inaccessible to all code outside the class, *except* for code within a derived class. The use of `protected` is illustrated in the shapes code.

We begin by defining the base class for all shape objects: `Shape`. Listing 17.20 (shape.h) provides the code for `Shape`.

Listing 17.20: shape.h

```
#ifndef SHAPE_H_
#define SHAPE_H_

/*
 *  Shape is the base class for all shapes
 */
class Shape {
public:
    // Longest distance across the shape
    virtual double span() const = 0;
    // The shape's area
    virtual double area() const = 0;
};

#endif
```

The "assignment to zero" of the two virtual methods in the `Shape` class make them special; they are called *pure virtual methods* or *pure virtual functions*. This special syntax signifies that these methods have no implementations. It is not possible to provide a body for these methods. A class that contains at least one pure virtual method is an *abstract class*. It is not possible to create an instance of an abstract class. The compiler enforces this restriction; the following statement;

```
Shape myShape;    // Illegal
```

is illegal. A non-abstract class is called a *concrete class*. All the classes we have seen to this point except for `Shape` have been concrete classes.

An abstract class represents an abstract concept. *Shape* is an abstract concept. We can have circles, rectangles, and lots of other kinds of shapes, but can we have something that is "just a shape" without being a particular kind of shape? Even though we cannot create instances of abstract classes, abstract classes are useful for organizing a class hierarchy.

We can derive a concrete class from our `Shape` class as shown in Listing 17.21 (rectangle.h).

Listing 17.21: rectangle.h

```
#ifndef RECTANGLE_H_
#define RECTANGLE_H_

#include "shape.h"

class Rectangle: public Shape {
protected:
    double length;
    double width;
public:
    Rectangle(double len, double wid);
    // Length of the longer side
    double span() const override;
    double area() const override;
};
```

```
#endif
```

Note the appearance of the `protected` access specifier. The methods in any class derived directly or indirectly from `Rectangle` will be able to access the `length` and `width` fields. These fields will be inaccessible to code outside these classes. Listing 17.22 (rectangle.cpp) provides the implementation of `Rectangle`'s methods.

Listing 17.22: `rectangle.cpp`

```
// File rectangle.cpp

#include "rectangle.h"
#include <algorithm>  // For max function

// Generally for rectangles length >= width, but the
// constructor does not enforce this.
Rectangle::Rectangle(double len, double wid): length(len), width(wid) {}

// Length of the longer side--determine which is longer
double Rectangle::span() const {
    return std::max(length, width);
}

double Rectangle::area() const {
    return length * width;
}
```

From mathematics we know that a square is a special kind of rectangle, so Listing 17.23 (square.h) and Listing 17.24 (square.cpp) specify a `Square` class.

Listing 17.23: `square.h`

```

#ifndef SQUARE_H_
#define SQUARE_H_

#include "rectangle.h"

// A square is a special case of a rectangle

class Square: public Rectangle {
public:
    // Length and width are equal in a square, so specify the
    // length of only one side
    Square(double side);
    // The inherited methods work as is; no need to
    // change their behavior.
};

#endif
```

Listing 17.24: `square.cpp`

```
// File square.cpp
```

```
#include "square.h"

// Defer the work of initialization to the base class constructor
Square::Square(double side): Rectangle(side, side) {}
```

Next, we add a triangle shape (Listing 17.25 (triangle.h) and Listing 17.26 (triangle.cpp)).

Listing 17.25: `triangle.h`

```
#ifndef TRIANGLE_H_
#define TRIANGLE_H_

#include "shape.h"

class Triangle: public Shape {
protected:
    double side1;   // Triangles have three sides
    double side2;
    double side3;
public:
    Triangle(double s1, double s2, double s3);
    double span() const override;
    double area() const override;
};

#endif
```

Listing 17.26: `triangle.cpp`

```
// File triangle.cpp

#include "triangle.h"
#include <algorithm>  // For max function

Triangle::Triangle(double s1, double s2, double s3):
          side1(s1), side2(s2), side3(s3)  {}

// The span of a triangle is the length of the longest side
double Triangle::span() const {
    return std::max(side1, std::max(side2, side3));
}

// Not having the base and height of the triangle explicitly, we
// use Heron's formula to compute the area
double Triangle::area() const {
    // Compute semiperimeter
    double s = (side1 + side2 + side3)/2;
    // Compute area using Heron's formula
    return s*(s - side1)*(s - side2)*(s - side3);
}
```

An ellipse is a simple curved shape that we can add to our class hierarchy. Listing 17.27 (ellipse.h) and Listing 17.28 (ellipse.cpp) define the ellipse class.

Listing 17.27: ellipse.h

```
#ifndef ELLIPSE_H_
#define ELLIPSE_H_

#include "shape.h"

class Ellipse: public Shape {
protected:
    double major_radius;  // The longer radius of the ellipse
    double minor_radius;  // The shorter radius of the ellipse
public:
    Ellipse(double major, double minor);
    double span() const override;
    double area() const override;
};

#endif
```

Listing 17.28: ellipse.cpp

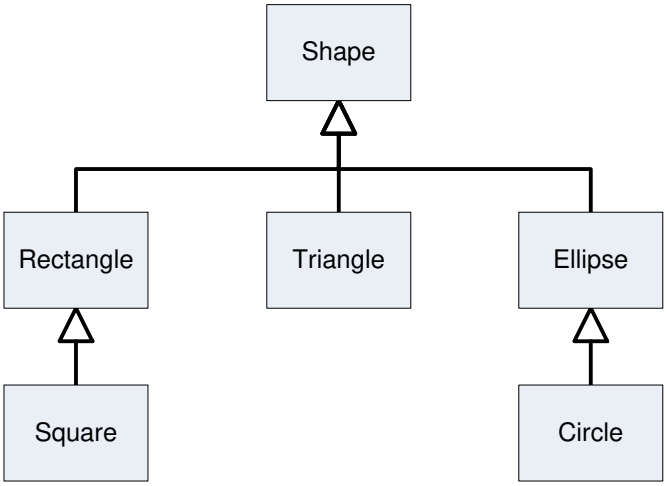
```
// File ellipse.cpp

#include "ellipse.h"
#include <algorithm>  // For max function

// PI is local to this file
static const double PI = 3.14159;

// Note: This constructor does not enforce
// major_axis >= minor_axis
Ellipse::Ellipse(double major, double minor):
        major_radius(major), minor_radius(minor) {}

// Greatest distance across is the length of the longer radius
double Ellipse::span() const {
    return std::max(major_radius, minor_radius);
}

double Ellipse::area() const {
    return PI * major_radius * minor_radius;
}
```

A circle is just an ellipse with equal major and minor radii. Listing 17.29 (circle.h) and Listing 17.30 (circle.cpp) define the `Circle` class.

Listing 17.29: circle.h

```
#ifndef CIRCLE_H_
#define CIRCLE_H_

#include "ellipse.h"
```

Figure 17.6 Inheritance hierarchy of shape classes




```
// A circle is a special case of an ellipse

class Circle: public Ellipse {
public:
    // In a circle the major and minor radii are the same, so
    // we need specify only one value when creating a circle.
    Circle(double radius);
    // Inherited methods work as is, no need to change their
    // behavior.
};

#endif
```

Listing 17.30: `circle.cpp`

```
// File circle.cpp

#include "circle.h"

// PI is local to this file
static const double PI = 3.14159;

Circle::Circle(double radius): Ellipse(radius, radius) {}
```

These shape classes form the class hierachy shown in Figure 17.6.

Listing 17.31: `testshapes.cpp`

```
// File testshapes.cpp
```

```
#include <iostream>
#include <vector>

#include "rectangle.h"
#include "triangle.h"
#include "circle.h"
#include "ellipse.h"


int main() {
    Rectangle rect(3, 4);
    Circle circ(4.5);
    Triangle tri(3, 4, 5);
    Ellipse elli(3, 4);
    std::vector<Shape *> shape_list;

    shape_list.push_back(&circ);
    shape_list.push_back(&tri);
    shape_list.push_back(&rect);
    shape_list.push_back(&elli);

    int n = shape_list.size();
    double area_total = 0.0, max_span = 0.0;

    for (int i = 0; i < n; i++) {
        // Examine the area each shape
        std::cout << "Area = " << shape_list[i]->area() << '\n';
        // Accumulate the areas of all the shapes
        area_total += shape_list[i]->area();
        // Account for the longest object
        if (max_span < shape_list[i]->span())
            max_span = shape_list[i]->span();
    }
    // Report the total area of all the shapes combined
    std::cout << "Total shape area is " << area_total << '\n';
    // Report the minimum length of the container
    std::cout << "Longest shape is " << max_span << '\n';
}
```

Observe that neither `Square` nor `Circle` needed access to the protected fields of their base classes. Consider Listing 17.32 (drawablerectangle.cpp) that derives a new class from `Rectangle` to allow clients to draw rectangle objects in a console window using text graphics.

Listing 17.32: `drawablerectangle.cpp`

```
#include <iostream>
#include "rectangle.h"

class DrawableRectangle: public Rectangle {
public:
    // Delegate construction to the base class
    DrawableRectangle(double length, double width):
         Rectangle(length, width) {}
```

```
    // Draw a rectangle using text graphics
    void draw() const {
        // Access the inherited protected fields
        int rows = static_cast<int>(length + 0.5),
            columns = static_cast<int>(width + 0.5);
        // Draw the rectangle
        for (int r = 0; r < rows; r++) {
            for (int c = 0; c < columns; c++)
                std::cout << '#';
            std::cout << '\n';
        }
    }
};

int main() {
    DrawableRectangle rec1(3, 2),
                      rec2(10, 5),
                      rec3(4, 8);
    rec1.draw();
    std::cout << "----------------\n";
    rec2.draw();
    std::cout << "----------------\n";
    rec3.draw();
}
```

Listing 17.32 (drawablerectangle.cpp) displays

```
##
##
##
----------------
#####
#####
#####
#####
#####
#####
#####
#####
#####
#####
----------------
########
########
########
########
```

The `DrawableRectangle::draw` method needs access to the fields `length` and `width` to be able to draw the rectangle. It is important to remember that every `DrawableRectangle` object contains these fields because it inherits them from `Rectangle`; however, if the `length` and `width` fields were declared private instead of protected in `Rectangle`, `DrawableRectangle::draw` would not be able to access these fields.

A class supports three levels of protection for its members:

1. `private`: This is the default. Private members are accessible to any code within the class itself but are inaccessible to code outside of the class.
2. `protected`: Protected members are accessible to code within the class itself and to code within derived classes that use public inheritance. Protected members are inaccessible to other (non-derived) classes.
3. `public`: Public members are accessible to code anywhere.

In sum, private means not accessible outside the class, period. Public means open to everyone. Protected means "public" to subclasses and "private" to all other classes.

17.8 Fine Tuning Inheritance

Recall the flexible sorting code found in Listing 16.5 (loggingflexiblesort.cpp). It uses objects to enable the following functionality:

- The ordering imposed by a selection sort function can be varied by using custom comparison functions.
- The object used to determine the sort's ordering collects data about the number of comparisons and element interchanges the sort function performs.

Listing 17.35 (polymorphicsort.cpp) provides a slight variation of Listing 16.5 (loggingflexiblesort.cpp). Besides splitting the code up into multiple source files, it adds two virtual methods that enable future developers to customize the `compare` and `swap` methods in derived classes.

Listing 17.33 (comparer.h) contains the declaration of the `Comparer` class.

Listing 17.33: `comparer.h`

```
#ifndef COMPARER_H_
#define COMPARER_H_
/*
 *  Comparer objects manage the comparisons and element
 *  interchanges on the selection sort function below.
 */
class Comparer {
    // The object's data is private, so it is inaccessible to
    // clients and derived classes

    // Keeps track of the number of comparisons
    // performed
    int compare_count;

    // Keeps track of the number of swaps performed
    int swap_count;
```

```
    // Function pointer directed to the function to
    // perform the comparison
    bool (*comp)(int, int);

protected:
    // Method that actually performs the comparison
    // Derived classes may customize this method
    virtual bool compare_impl(int m, int n);

    // Method that actually performs the swap
    // Derived classes may customize this method
    virtual void swap_impl(int& m, int& n);

public:
    // The client must initialize a Comparer object with a
    // suitable comparison function.
    Comparer(bool (*f)(int, int));

    // Resets the counters to make ready for a new sort
    void reset();

    // Method that performs the comparison.  It delegates
    // the actual work to the function pointed to by comp.
    // This method logs each invocation.
    bool compare(int m, int n);

    // Method that performs the swap.
    // Interchange the values of
    // its parameters a and b which are
    // passed by reference.
    // This method logs each invocation.
    void swap(int& m, int& n);

    // Returns the number of comparisons this object has
    // performed since it was created.
    int comparisons() const;

    // Returns the number of swaps this object has
    // performed since it was created.
    int swaps() const;
};
#endif
```

Listing 17.34 (comparer.cpp) provides the implementation of the `Comparer` class methods.

Listing 17.34: `comparer.cpp`

```
#include "comparer.h"

// Method that actually performs the comparison
// Derived classes may customize this method
bool Comparer::compare_impl(int m, int n) {
    return comp(m, n);
}
```

```
// Method that actually performs the swap
// Derived classes may customize this method
void Comparer::swap_impl(int& m, int& n) {
    int temp = m;
    m = n;
    n = temp;
}

// The client must initialize a Comparer object with a
// suitable comparison function.
Comparer::Comparer(bool (*f)(int, int)):
      compare_count(0), swap_count(0), comp(f) {}

// Resets the counters to make ready for a new sort
void Comparer::reset() {
    compare_count = swap_count = 0;
}

// Method that performs the comparison.  It delegates
// the actual work to the function pointed to by comp.
// This method logs each invocation.
bool Comparer::compare(int m, int n) {
    compare_count++;
    return compare_impl(m, n);
}

// Method that performs the swap.
// Interchange the values of
// its parameters a and b which are
// passed by reference.
// This method logs each invocation.
void Comparer::swap(int& m, int& n) {
    swap_count++;
    swap_impl(m, n);
}

// Returns the number of comparisons this object has
// performed since it was created.
int Comparer::comparisons() const {
    return compare_count;
}

// Returns the number of swaps this object has
// performed since it was created.
int Comparer::swaps() const {
    return swap_count;
}
```

Notice that even though the `virtual` keyword appears before the method declarations in comparer.h, it does not appear before the method implementations in comparer.cpp.

Listing 17.35 (polymorphicsort.cpp) provides the client code that tests the new `Comparer` class.

Listing 17.35: `polymorphicsort.cpp`

```
#include <iostream>
#include <vector>
#include "comparer.h"

/*
 *   selection_sort(a, compare)
 *       Arranges the elements of vector a in an order determined
 *       by the compare object.
 *       a is a vector of ints.
 *       compare is a function that compares the ordering of
 *            two integers.
 *       The contents of a are physically rearranged.
 */
void selection_sort(std::vector<int>& a, Comparer& compare) {
    int n = a.size();
    for (int i = 0;  i < n - 1;  i++) {
        // Note: i,small, and j represent positions within a
        // a[i], a[small], and a[j] represents the elements at
        // those positions.
        // small is the position of the smallest value we've seen
        // so far; we use it to find the smallest value less
        // than a[i]
        int small = i;
        // See if a smaller value can be found later in the array
        for (int j = i + 1;  j < n;  j++)
            if (compare.compare(a[j], a[small]))
                small = j;  // Found a smaller value
        // Swap a[i] and a[small], if a smaller value was found
        if (i != small)
            compare.swap(a[i], a[small]);
    }
}

/*
 *  print
 *     Prints the contents of an integer vector
 *     a is the vector to print.
 *     a is not modified.
 */
void print(const std::vector<int>& a) {
    int n = a.size();
    std::cout << '{';
    if (n > 0) {
        std::cout << a[0];  // Print the first element
        for (int i = 1; i < n; i++)
            std::cout << ',' << a[i];  // Print the rest
    }
    std::cout << '}';
}

/*
 *  less_than(a, b)
 *     Returns true if a < b; otherwise, returns
 *     false.
 */
```

```
bool less_than(int a, int b) {
    return a < b;
}

/*
 *  greater_than(a, b)
 *     Returns true if a > b; otherwise, returns
 *     false.
 */
bool greater_than(int a, int b) {
    return a > b;
}

int main() {
   // Make a vector of integers
   std::vector<int> original { 23, -3, 4, 215, 0, -3, 2, 23, 100, 88, -10 };

   // Make a working copy of the original vector
   std::vector<int> working = original;
   std::cout << "Before:   ";
   print(working);
   std::cout << '\n';
   Comparer lt(less_than), gt(greater_than);
   selection_sort(working, lt);
   std::cout << "Ascending:  ";
   print(working);
   std::cout << "  (" << lt.comparisons() << " comparisons, "
             << lt.swaps() << " swaps)\n";
   std::cout << "----------------------------\n";
   // Make another copy of the original vector
   working = original;
   std::cout << "Before:   ";
   print(working);
   std::cout << '\n';
   selection_sort(working, gt);
   std::cout << "Descending: ";
   print(working);
   std::cout << "  (" << gt.comparisons() << " comparisons, "
             << gt.swaps() << " swaps)\n";
   std::cout << "----------------------------\n";
   // Sort a sorted vector
   std::cout << "Before:   ";
   print(working);
   std::cout << '\n';
   // Reset the greater than comparer so we start counting at
   // zero
   gt.reset();
   selection_sort(working, gt);
   std::cout << "Descending: ";
   print(working);
   std::cout << "  (" << gt.comparisons() << " comparisons, "
             << gt.swaps() << " swaps)\n";
}
```

The functions in Listing 17.35 (polymorphicsort.cpp) are identical to the functions in Listing 16.5 (loggingflexiblesort.cpp), and, not surprisingly, the output of Listing 17.35 (polymorphicsort.cpp) is identical to the output of Listing 16.5 (loggingflexiblesort.cpp):

```
Before:   {23,-3,4,215,0,-3,2,23,100,88,-10}
Ascending:  {-10,-3,-3,0,2,4,23,23,88,100,215}  (55 comparisons, 7 swaps)
----------------------------
Before:   {23,-3,4,215,0,-3,2,23,100,88,-10}
Descending: {215,100,88,23,23,4,2,0,-3,-3,-10}  (55 comparisons, 5 swaps)
----------------------------
Before:   {215,100,88,23,23,4,2,0,-3,-3,-10}
Descending: {215,100,88,23,23,4,2,0,-3,-3,-10}  (55 comparisons, 0 swaps)
```

The question arises: What advantage does the new `Comparer` class have over the original one?

The design of this new `Comparer` class is interesting:

- Since its counter fields are private, only methods in the `Comparer` class itself can access a `Comparer` object's data.
- The two virtual methods, `compare_impl` and `swap_impl`, are protected, so clients cannot access them directly. Derived classes, however, can see them and override them. The suffix `_impl` stands for "implementation," so `compare_impl` represents the implementation details of the *compare* method and `swap_impl` represents the implementation details of the *swap* method.
- The public `compare` and `swap` methods both delegate part of their work to the protected `compare_impl` and `swap_impl` methods.
- The `compare` and `swap` methods are not declared `virtual`, so derived classes cannot override them.
- The `compare` and `swap` methods manage the `compare_count` and `swap_count` counter fields. Since derived classes cannot see these fields, there is nothing that a designer of a derived class can do when overriding `compare_impl` or `swap_impl` to disturb the correct accounting of the number of times a client calls `compare` or `swap`.
- The `comparisons` and `swaps` methods that report the results to the client are non-virtual, so derived classes may not override their behavior.

Observe that the designer of the `Comparer` class allows the nature of the comparisons and swaps in derived classes to be flexible, but it is rigid on the enforcement of how the accounting is performed and reported. The proper use of `protected` and `private` specifiers in a base class as shown in `Comparer` affords class designers a great deal of control over exactly what derived class designers can do. Derived classes may adapt some behaviors, but other behaviors are non-negotiable.

What kind of customization would a programmer want to do to the `Comparer` class beyond changing how the comparison is performed? Consider the `LogComparer` class declared in Listing 17.36 (logcomparer.h).

Listing 17.36: `logcomparer.h`

```
#ifndef LOGCOMPARER_H_
#define LOGCOMPARER_H_

#include <fstream>
```

```
#include <string>
#include "comparer.h"

/*
 *  Comparer objects manage the comparisons and element
 *  interchanges on the selection sort function below.
 */
class LogComparer: public Comparer {
    // Output stream to which logging messages are directed
    std::ofstream fout;

protected:
    // Method that actually performs the comparison
    bool compare_impl(int m, int n) override;

    // Method that actually performs the swap
    void swap_impl(int& m, int& n) override;

public:
    // The client must initialize a LogComparer object with a
    // suitable comparison function and the file name of a text
    // file to which the object will direct logging messages
    LogComparer(bool (*f)(int, int), const std::string& filename);

    // The destructor must close the log file
    ~LogComparer();
};

#endif
```

The `LogComparer` class overrides the `compare_impl` and `swap_impl` methods. Instances of the `LogComparer` class use a `std::ofstream` object to record logging information to a text file.

The implementation of the `LogComparer` methods can be found in Listing 17.37 (logcomparer.cpp).

Listing 17.37: `logcomparer.cpp`

```
#include "logcomparer.h"
#include <cstdlib>
#include <iostream>

// Method that actually performs the comparison
// Derived classes may override this method
bool LogComparer::compare_impl(int m, int n) {
    fout << "Comparing " << m << " to " << n << '\n';
    // Base class method does the comparision
    return Comparer::compare_impl(m, n);
}

// Method that actually performs the swap
// Derived classes may override this method
void LogComparer::swap_impl(int& m, int& n) {
    fout << "Swapping " << m << " and " << n << '\n';
    int temp = m;
    m = n;
    n = temp;
```

```
}

// The client must initialize a LogComparer object with a
// suitable comparison function and the file name of the
// text file to receive logging messages.
LogComparer::LogComparer(bool (*f)(int, int), const std::string& filename):
      Comparer(f) {
    fout.open(filename);
    if (!fout.good()) {
        std::cout << "Could not open log file " << filename
                  << " for writing\n";
        exit(1);  // Terminate the program
    }
    // fout is an instance variable, not a local variable,
    // so the file stays open when the constructor finishes
}
```

The constructor attempts to open a file output stream to write logging information to a text file. If the constructor cannot for any reason open the text file for writing, it terminates the program's execution. The destructor closes the file stream object when the `LogComparer` object's life is over.

The two overridden methods, `compare_impl` and `swap_impl`, write text to the log file. Both methods use the `LogComparer` object's `fout` field rather than the `std::cout` console output stream object.

Notice that even though the `override` keyword appears after the method declarations in logcomparer.h, it does not appear after the method implementations in logcomparer.cpp.

The client code in Listing 17.38 (loggingsort.cpp) uses a `LogComparer` object to create a text file named sort.log.

Listing 17.38: `loggingsort.cpp`

```
#include <iostream>
#include <vector>
#include "logcomparer.h"

/*
 *   selection_sort(a, compare)
 *       Arranges the elements of vector a in an order determined
 *       by the compare object.
 *       a is a vector of ints.
 *       compare is a function that compares the ordering of
 *            two integers.
 *       The contents of a are physically rearranged.
 */
void selection_sort(std::vector<int>& a, Comparer& compare) {
    int n = a.size();
    for (int i = 0;  i < n - 1;  i++) {
        // Note: i,small, and j represent positions within a
        // a[i], a[small], and a[j] represents the elements at
        // those positions.
        // small is the position of the smallest value we've seen
        // so far; we use it to find the smallest value less
        // than a[i]
        int small = i;
```

```
        // See if a smaller value can be found later in the array
        for (int j = i + 1;  j < n;  j++)
            if (compare.compare(a[j], a[small]))
                small = j;  // Found a smaller value
        // Swap a[i] and a[small], if a smaller value was found
        if (i != small)
            compare.swap(a[i], a[small]);
    }
}

/*
 *  print
 *     Prints the contents of an integer vector
 *     a is the vector to print.
 *     a is not modified.
 */
void print(const std::vector<int>& a) {
    int n = a.size();
    std::cout << '{';
    if (n > 0) {
        std::cout << a[0];  // Print the first element
        for (int i = 1; i < n; i++)
            std::cout << ',' << a[i];  // Print the rest
    }
    std::cout << '}';
}

/*
 *  less_than(a, b)
 *     Returns true if a < b; otherwise, returns
 *     false.
 */
bool less_than(int a, int b) {
    return a < b;
}

int main() {
   // Make a vector of integers from an array
   int a[] = { 23, -3, 4, 215, 0, -3, 2 };
   int len = (sizeof a)/(sizeof a[0]);
   std::vector<int> vec(a, a + len);

   // Make a working copy of the original vector
   std::cout << "Before:   ";
   print(vec);
   std::cout << '\n';
   LogComparer lt(less_than, "sort.log");
   selection_sort(vec, lt);
   std::cout << "Ascending:  ";
   print(vec);
   std::cout << "  (" << lt.comparisons() << " comparisons, "
             << lt.swaps() << " swaps)\n";
}
```

Listing 17.38 (loggingsort.cpp) prints to the console

```
Before:   {23,-3,4,215,0,-3,2}
Ascending:  {-3,-3,0,2,4,23,215}  (21 comparisons, 4 swaps)
```

and writes the file sort.log the text

```
Comparing -3 to 23
Comparing 4 to -3
Comparing 215 to -3
Comparing 0 to -3
Comparing -3 to -3
Comparing 2 to -3
Swapping 23 and -3
Comparing 4 to 23
Comparing 215 to 4
Comparing 0 to 4
Comparing -3 to 0
Comparing 2 to -3
Swapping 23 and -3
Comparing 215 to 4
Comparing 0 to 4
Comparing 23 to 0
Comparing 2 to 0
Swapping 4 and 0
Comparing 4 to 215
Comparing 23 to 4
Comparing 2 to 4
Swapping 215 and 2
Comparing 23 to 4
Comparing 215 to 4
Comparing 215 to 23
```

If during development the selection sort function has a problem, the programmer can review the contents of the log file to examine how the sort progresses and perhaps determine where the problem lies.

The design of the `Comparer` class provides a overall structure that allows inheriting classes to fine tune the details without disturbing the overarching framework. Inheritance and polymorphism allow us to add functionality to an existing code base. Virtual methods provide extension points by which derived classes can add custom behavior.

17.9 Exercises

1. Consider the following C++ code:

```
class Widget {
public:
    virtual int f() { return 1; }
};
```

```
class Gadget: public Widget {
public:
    virtual int f() { return 2; }
};

class Gizmo: public Widget {
public:
    virtual int f() { return 3; }
};

void do_it(Widget *w) {
    std::cout << w->f() << " ";
}

int main() {
    std::vector<Widget *> widgets;
    Widget wid;
    Gadget gad;
    Gizmo giz;
    widgets.push_back(&wid);
    widgets.push_back(&gad);
    widgets.push_back(&giz);
    for (size_t i = 0; i < widgets.size(); i++)
        do_it(widgets[i]);
}
```

(a) What does the program print?

(b) Would the program still compile and run if the `f` method within the `Widget` class were a pure virtual function?

(c) How would the program run differently if the `virtual` keyword were removed from all the code?

(d) Would the program behave the same if the `virtual` keyword were removed from all the classes except for `Widget`?

Chapter 18

Memory Management

CAUTION! CHAPTER UNDER CONSTRUCTION

The C++ programming language provides many options to programmers when it comes to managing the memory used by an executing program. This chapter explores some of these frequently used options and introduces modern techniques aimed at reducing the memory management problems that have plagued C++ projects in the past.

18.1 Memory Available to C++ Programs

A modern operating system reserves a section of memory for an executing program. This allows the operating system to manage multiple program executions simultaneously. Different operating systems layout the memory of executing programs in different ways, but the layout will include the following four sections:

- **Code**. The code section of memory holds the program's compiled executable instructions. The contents of the code section should never change while the program executes, and the size of the code segment does not change during the program's execution.
- **Data**. The data section of memory contains global variables (see Section 10.1) and persistent local variables (`static` locals, see Section 10.2). The variables in the data section exist for the life of the executing program, but, unless they are constants (see Section 3.6), the executing program may freely change their values. Even though the values stored in the variables found in the data segment may change during the program's execution, the size of the data segment does not change while the program executes. This is because the program's source code precisely defines the number of global and `static` local variables. The compiler can compute the exact size of the data segment.
- **Heap**. The heap is where an executing program obtains dynamic memory. The `new` operator gets its memory from the heap, and `delete` returns previously allocated memory back to the heap. The size of the heap grows and shrinks during the program's execution as the program allocates and deallocates dynamic memory using `new` and `delete`.
- **Stack**. The stack is where local variables and function parameters live. Space for local variables and parameters appears when a function is called and disappears when the function returns. The size of the stack grows and shrinks during the program's execution as various functions execute.

©2018 Richard L. Halterman

Operating systems generally limit the size of the stack. Deep recursion can consume a considerable amount of stack space. An improperly written recursive function, for example one that omits the base case and thus exhibits "infinite" recursion, will consume all the space available on the stack. Such a situation is known as a *stack overflow*. Modern operating systems will terminate a process that consumes all of its stack space, but on some embedded systems this stack overflow may go undetected. Heap space typically is much more plentiful, and operating systems can use *virtual memory* to provide a executing program more space than is available in real memory. The extra space for virtual memory comes from a disk drive, and the operating system shuttles data from disk to real memory as needed by the executing program. Programs that use a lot of virtual memory run much slower than programs that use little virtual memory. Virtual memory is not unlimited, however, so a program with memory leaks eventually can run out of memory.

Because of the way function and method calls and their subsequent returns work, the stack grows and shrinks in a very regular fashion. It expands during a function call to make room for the executing function's local variables and parameters (and it expands even more if that function calls other functions), and when the function returns, the stack contracts back to the original size it had before the function invocation. Variables are removed from the stack in the reverse order of their creation on the stack. The stack always consists of one contiguous chunk of memory with no areas of unused space within that chunk.

The heap grows and shrinks as the program executes `new` and `delete`, but its expansion and contraction is not regular. One function may allocate an object or dynamic array with `new`, and a different function may much later in the program's execution deallocate the object or array with `delete`. An executing program may `delete` dynamically allocated memory in a very different order from its allocation. This means that memory allocated on the heap is not contiguous; that is, space for deallocated objects can be interspersed with space for allocated objects. The the available memory on the heap thus can become fragmented during the program's execution.

Global variables and `static` local variables in the data segment live for the life of the executing program. The run-time environment initializes globals before `main` begins executing and cleans them up when `main` returns. The run-time environment initializes `static` locals during the function's or method's first invocation. Non-`static` local variables on the stack exist only when a function is executing. If a program calls and returns from the same function 20 times, that function's local variables appear and disappear 20 times. An executing program can create dynamic memory as needed, hold onto it as long as necessary, and finally release it when it is no longer needed.

The quantity of global data is fixed when the program begins executing, and stack data is not persistent for the life of the executing program. This means an executing program that must manage a varying amount of data for an arbitrary amount of time must use the heap as a source of memory for that data. The run-time environment automatically manages global and local memory. Unfortunately, programmers must write code that manually manages heap data. The problem is this: Manual memory management in all but very simple systems turns out to be a difficult task. Development of large software systems with C++ in the early days often was a frustrating experience.

18.2 Manual Memory Management

Memory management issues with `new` and `delete` frequently are the source of difficult to find and fix logic errors. Programmers must adhere strictly to the following tenets:

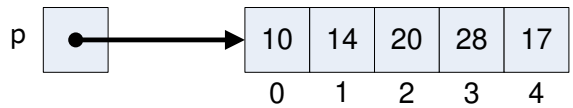
- **Every call to `new` should have an associated call to `delete` when the allocated memory is no longer needed.** It sounds simple enough, but it is not always clear when `delete` should be used. The following function exhibits a memory leak:

```
void calc(int n) {
```

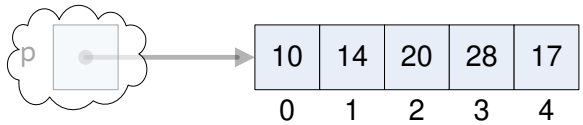
Figure 18.1 Memory leak

1. Before the call to `calc`, variable `p` does not exist

2. During the execution of `calc`, variable `p` is alive and points to dynamically allocated memory



3. After `calc` returns, variable `p` goes away, but the memory allocated on its behalf remains




```
    // ...
    // Do some stuff
    // ...
    int *p = new int[n];
    // ...
    // Do some stuff with p
    // ...
    // Exit function without deleting p's memory
}
```

If a program calls function `calc` enough times, the program will eventually run out of memory. Figure 18.1 shows how memory is stranded when local pointers go out of scope at the end of a function's execution.

In the `calc` function `p` is a local variable. As such, `p` lives on the stack. When a particular call to `calc` completes, the function's clean up code automatically releases the space help by *the variable* `p`. This is because all functions automatically manage the memory for their parameters and local variables. The problem is `p` is assigned via `new` to point to memory allocated from the heap, not the stack. Function executions manage the stack memory only. When the function's execution completes, `p` is gone, and the memory to which `p` pointed is not deallocated automatically. Worse yet, the heap memory formerly referenced by the local variable `p` now is unreachable because no other variables that remain viable know anything about it.

A correctly written `calc` function has three options in this case:

1. free up `p`'s allocated memory with `delete` before returning,
2. assign a global or `static` local pointer to the allocated memory, or

3. return p to the function's caller and let the caller call delete when finished with the object calc created.

Option 1 is the easiest solution. The calc function needs a dynamically allocated array, creates it, uses it, and then deallocates it before returning. Within this single function definition we can see a new and later its corresponding delete—a perfect matched pair. Unfortunately, this is not a common case. If calc needs local working space, it is best handled on the stack through local variables. That way memory clean up is automatic.

Option 2 is possible only if callers invoke the calc function exactly once and, additionally in the case of a global pointer, invoke calc before attempting to use the global pointer.

Option 3 is the more common occurrence. The heap enables an executing program to add extra data at run time that persists across function calls. In theory the process is simple:

1. call calc to obtain a dynamically allocated object and
2. delete the object when finished with it.

The problem here is we cannot see the new paired with the delete. The new is hidden in calc, and the caller must exercise delete.

- **The delete operator never should be used to free up memory that was not allocated by a previous call to new**. This code fragment illustrates one such example:

```
int list[10], *p = list;  // p points to list
// ...
// Do some stuff
// ...
delete [] p;  // Logic error, attempt to deallocate p's memory
```

The space referenced by pointer p was not allocated by new, so delete should not be used to attempt to free up its memory.

Attempting to delete memory not allocated with new results in undefined behavior and represents a logic error.

- **delete must not be used to deallocate the same memory more than once.** This can happen when two pointers refer to the same memory. Such pointers are called *aliases*. The following code fragment illustrates the situation:

```
int *p = new int[10], *q = p;  // q aliases p
// ...
// Do some stuff with p and/or q
// ...
delete [] p;   // Free up p's memory
// ...
// Do some other stuff
// ...
delete [] q;   // Logic error, q's memory already freed!
```

Since pointer p and pointer q point to the same memory, they are aliases of each other. Deallocate the memory referenced by one of them, and the other's memory is also deallocated, since it is the same memory.

Multiple deletes of the same memory results in undefined behavior and represents a logic error.

- **Memory previously deallocated via `delete` should never be accessed.** Attempting to access `delete`d memory results in undefined behavior and represents a logic error. The code fragment

```
int *list = new int[10];
// ...
// Use list, then
// ...
delete [] list;  // Deallocate list's memory
// ...
// Sometime later
// ...
int x = list[2];  // Logic error, but sometimes works!
```

  illustrates how such a situation can arise. For efficiency reasons the `delete` operator generally marks heap space as "available" without modifying the contents of that memory. Careless programmers can accidentally use the memory of a `delete`d pointer obliviously as if it were still live. The problem manifests itself when the freed up memory eventually gets reallocated elsewhere via a call to `new`. The result is that programs seem to "work" for a while and mysteriously fail at unpredictable times. Debugging such situations can be very difficult. The problem often happens because of aliasing:

```
int *list = new int[10];
// ...
int *arr = list;  // arr aliases list
// ...
delete [] list;  // Deallocate list's memory
// ...
// Sometime later
// ...
int x = arr[2];  // Same problem!
```

  We `delete`d `list`'s memory, not `arr`'s memory, didn't we? No, since `arr` is an alias for `list`, `arr` references memory previously deallocated with `delete`.

Aliasing is a problem because of our concept of variables. When we have two variables with different names it is natural to assume they represent two different objects. This is known as *value semantics*. Pointers and references introduce the possibility of aliasing; they use *reference semantics*. Reference semantics enable useful techniques such as call-by-reference and the traversal of dynamic data structures like linked lists, but reasoning about the identity of objects using reference semantics requires extra caution.

It seems simple enough to make sure every `new` has exactly one corresponding `delete`, but in practice it can be very difficult to determine exactly when to use `delete`. Suppose, for example, you obtain a dynamically allocated object from a function call, as in the following:

```
Widget *p = get_widget();
```

Quite possibly the `get_widget` function allocates a `Widget` object via `new` and simply returns a pointer to the object. In this case we can use

```
delete p;
```

when we are finished using the object to which `p` points. What if, however, the `get_widget` function is managing shared resources; that is, if `get_widget` already has created a `Widget` object with a particular

value, it does not create a new `Widget` object. Instead it returns a pointer to the existing object it previously created. Listing 18.1 (howtodelete.cpp) provides a rudimentary example.

Listing 18.1: howtodelete.cpp

```
#include <iostream>
#include <vector>

struct Widget {
    int value;
    Widget(int value): value(value) {
        std::cout << "Creating widget " << value << '\n';
    }
    ~Widget() {
        std::cout << "Destroying widget " << value << '\n';
    }
};

Widget *get_widget() {
    static int pos = 0;
    static std::vector<Widget *> widget_pool {new Widget(23), new Widget(45),
                                              new Widget(16), new Widget(12),
                                              new Widget(3), new Widget(20),
                                              new Widget(10)};
    pos = (pos + 1) % widget_pool.size();
    return widget_pool[pos];
}

void process(int n) {
    std::vector<Widget *> vec;
    while (n-- > 0)
        vec.push_back(get_widget());
    // Clean up vector?
    for (auto& elem : vec)
        delete elem;
}

int main() {
    std::cout << "Entering main\n";
    process(10);
    std::cout << "Leaving main\n";
}
```

Listing 18.1 (howtodelete.cpp) prints

```
Entering main
Creating node 23
Creating node 45
Creating node 16
Creating node 12
Creating node 3
Creating node 20
Creating node 10
Destroying node 45
Destroying node 16
Destroying node 12
```

```
Destroying node 3
Destroying node 20
Destroying node 10
Destroying node 23
Destroying node 45
Destroying node 16
Destroying node 12
Leaving main
```

Observe widgets 45, 16, and 12 being destroyed twice. On some systems the program crashes before printing *Leaving main*. Double deletion is classified as undefined behavior, so it represents a bug in our program.

Before returning from `process`, how can you traverse the vector releasing the dynamically allocated memory and ensuring that you do not `delete` the same object more than once? The answer is that you cannot easily do so. Even if you are careful and account for each allocation of `Widget` objects of a particular value, you really do not know if the program elsewhere earlier called `get_widget()` and stored the result in a global variable that outlives this call to `process`. You would need to implement within your code a complex global accounting infrastructure that keeps track of all memory allocations.

18.3 Linked Lists

An object in C++ can hold just about any type of data, but there are some limitations. Consider the following `struct` definition:

```
struct Node {
    int data;
    Node next;    // Error, illegal self reference
};
```

(Here we use a `struct` instead of a `class` since we will consider a `Node` object a primitive data type that requires no special protection from clients.) How much space should the compiler set aside for a `Node` object? A `Node` contains an integer and a `Node`, but this contained `Node` field itself would contain an integer and a `Node`, and the nested containment would go on forever. Such a structure understandably is illegal in C++, and the compiler will issue an error. You are not allowed to have a `class` or `struct` field of the same type within the `class` or `struct` being defined.

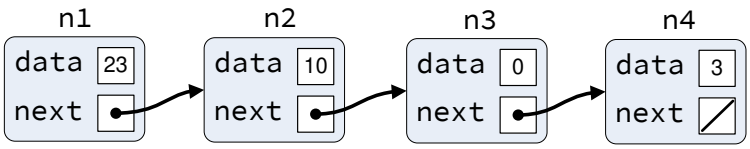
Another object definition looks similar, but it is a legal structure:

```
struct Node {
    int data;
    Node *next;    // Self reference via pointer is legal
};
```

The reason this second version is legal is because the compiler now can compute the size of a `Node` object. A pointer is simply a memory address under the hood, so all pointer variables are the same size regardless of their declared type. The pointer solves the infinitely nested containment problem.

This ability of a object to refer to an object like itself is not merely an interesting curiosity; it has practical applications. Suppose we wish to implement a sequence structure like a vector. We can use the self-referential structure defined above to build a list of `Node` objects linked together via pointers. Listing 18.2 (manuallinkedlist.cpp) builds a small linked list "by hand."

Figure 18.2 A null-terminated linked list



Listing 18.2: manuallinkedlist.cpp

```
#include <iostream>

using namespace std;

struct Node {
    int data;    // The element of interest
    Node *next;  // Link to successor node in the link
    // Constructor
    Node(int data, Node *next): data(data), next(next) {}
};

int main() {
    // Node objects
    Node n4(3, nullptr),  // Make the last node
         n3(0, &n4),      // Make the next to last node and link to last node
         n2(10, &n3),     // Make the second node and link to third node
         n1(23, &n2);     // Make the first node and link to second node

    // Print the linked list built from the Node objects
    for (Node *cursor = &n1; cursor != nullptr; cursor = cursor->next)
        std::cout << cursor->data << ' ';
    std::cout << '\n';
}
```

Listing 18.2 (manuallinkedlist.cpp) creates the null-terminated linked list illustrated in Figure 18.2. The program prints

```
23 10 0 3
```

In Figure 18.2 the line in the `next` field of node `n4` represents the null pointer.

Listing 18.2 (manuallinkedlist.cpp) augments the simple `Node` struct by adding a constructor. The program uses no dynamic memory (no calls to `new`); this is possible only because we know the nodes that are in the list ahead of time. The `for` loop uses a pointer named `cursor` to visit each node in the list. The `cursor` variable first points to node `n1`. The statement `cursor = cursor->next` reassigns `cursor` to point to the node that follows the current node. This enables `cursor` to visit nodes `n1`, `n2`, `n3`, and finally `cursor` will point to `n4`, the last node in the list. The `next` field of `n4` is `nullptr`, so when the loop reassigns `cursor` to `n4`'s `next` field, `cursor` will be `nullptr`, and the loop will terminate.

The value `nullptr` in a Boolean context is considered `false`, so we can simplify the Boolean expression `cursor != nullptr` to just `cursor`; this allows us to simplify the loop somewhat, as

```
// Print the linked list built from the Node objects
for (Node *cursor = &n1; cursor; cursor = cursor->next)
    std::cout << cursor->data << ' ';
```

Since we hard-coded the four list nodes into the program's source, the loop really is not necessary. The following statement:

```
std::cout << n1.data << ' ' << n2.data << ' ' << n3.data
          << n4.data << '\n';
```

is simpler, but why stop there? Since we know the nodes and their values ahead of time, the following statement is even simpler:

```
std::cout << "23 10 0 3\n";
```

The hard-coded list in Figure 18.2 and Listing 18.2 (manuallinkedlist.cpp) does not demonstrate the utility possible due to the dynamic nature of linked lists. It makes sense to use a linked list only when its elements are unknown ahead of time. We then can dynamically allocate space for the elements as they become available.

Armed with our knowledge of C++ classes, encapsulation, and methods, we can build a client-friendly, dynamic linked list type. The code found in Listing 18.3 (intlist1.h), Listing 18.4 (intlist1.cpp), and Listing 18.5 (listmain.cpp) demonstrates the power of linked lists. Listing 18.3 (intlist1.h) is the header file for a simple integer linked list class.

Listing 18.3: `intlist1.h`

```
class IntList1 {
    /*
     *  An object that holds an element in a linked list.
     *  This type is local to the IntList1 class and inaccessible
     *  to outside this class.
     */
    struct Node {
        int data;      // A data element of the list
        Node *next;    // The node that follows this one in the list
        Node(int d);   // Constructor
    };

    Node *head;  // Points to the first item in the list
    Node *tail;  // Points to the last item in the list

    /*
     *  Returns the length of the linked list pointed to by p.
     */
    int length(Node *p) const;

    /*
     *  dispose(p)
     *    Deallocate the memory held by the list pointed to by p.
     */
    void dispose(Node *p);

public:
```

```
    /*
     *  The constructor makes an initially empty list
     */
    IntList1();

    /*
     *  insert(n)
     *    Inserts n onto the back of the list.
     *    n is the element to insert.
     */
    void insert(int n);

    /*
     *  print()
     *    Prints the contents of the linked list of integers.
     */
    void print() const;

    /*
     *  Returns the length of the linked list.
     */
    int length() const;

    /*
     *  clear()
     *    Removes all the elements in the linked list.
     */
    void clear();
};
```

The `Node` struct is declared within the `IntList1` class. We say that `Node` is a *nested struct*. Since the declaration of `Node` appears in the private section of `IntList1`, `Node` is a type known only to code within the `IntList1` class. Because its declaration is nested within the `IntList1` class, the complete name of the `Node` type is `IntList1::Node`. Recall that a struct is equivalent to a class, except the default access to its members is `public`. We can define nested a class in same manner as we defined our nested struct.

Notice that the `IntList1` class has several private methods in addition to its public methods. Code outside the class cannot execute these private methods directly. These private methods are helper methods that several of the public methods invoke to accomplish their tasks. We say that a public method *delegates* the work to its private helper methods. Why is this delegation necessary here? The private methods use recursion that requires a parameter of type `IntList1::Node` which is unknown outside the `IntList1` class. A client is therefore unable to use the private methods directly, even it they were made public. The public methods do not expose to the client any details about the class's implementation; specifically they keep the `IntList1::Node` type and the `head` and `tail` instance variables hidden from clients.

Observe that the overloaded `length` methods (both private and public) and `print` method of the `IntList1` class are declared `const`. Neither printing a list nor requesting its length should modify an `IntList1` object. Clients can, therefore, use the `print` and `length` methods with a constant `IntList1` object. An attempt to use `insert` or `clear` on a constant `IntList1` object will yield a compiler error. The error here makes sense because `insert` definitely will modify a list object, and `clear` potentially will modify a list object (we say *potentially* here because `clear` will not modify an empty list).

Listing 18.4 (intlist1.cpp) implements the methods declared in Listing 18.3 (intlist1.h).

Listing 18.4: intlist1.cpp

```
// intlist1.cpp

#include "intlist1.h"
#include <iostream>

// Private IntList1 operations

/*
 *  Node constructor
 */
IntList1::Node::Node(int n): data(n), next(nullptr) {}

/*
 *  Returns the length of the linked list pointed to by p.
 */
int IntList1::length(IntList1::Node *p) const {
    if (p)
        return 1 + length(p->next);  // 1 + length of rest of list
    else
        return 0;                    // Empty list has length zero
}

/*
 *  dispose(p)
 *    Deallocate the memory held by the list pointed to by p.
 */
void IntList1::dispose(IntList1::Node *p) {
    if (p) {
        dispose(p->next);   // Free up the rest of the list
        delete p;           // Deallocate this node
    }
}

// Public IntList1 operations

/*
 *  The constructor makes an initially empty list.
 *  The list is empty when head and tail are null.
 */
IntList1::IntList1(): head(nullptr), tail(nullptr) {}

/*
 *  insert(n)
 *    Inserts n onto the back of the list.
 *    n is the element to insert.
 */
void IntList1::insert(int n) {
    // Make a node for the new element n
    IntList1::Node *new_node = new Node(n);
    if (tail) {   // Is tail non-null?
       tail->next = new_node;  // Link the new node onto the back
       tail = new_node;        // The new node is the new tail of the list
```

```
    }
    else // List is empty, so make head and tail point to new node
        head = tail = new_node;
}

/*
 *  print()
 *    Prints the contents of the linked list of integers.
 */
void IntList1::print() const {
    for (auto cursor = head; cursor; cursor = cursor->next)
        std::cout << cursor->data << ' ';
    std::cout << '\n';
}

/*
 *  Returns the length of the linked list.
 */
int IntList1::length() const {
    return length(head);  // Delegate work to private helper method
}

/*
 *  clear()
 *    Removes all the elements in the linked list.
 */
void IntList1::clear() {
    dispose(head);          // Deallocate space for all the nodes
    head = tail = nullptr;  // Null head signifies list is empty
}
```

Since the code in Listing 18.4 (intlist1.cpp) appears outside of the class declaration, any use of the `Node` type requires its full name: `IntList1::Node`. Note the `::` use in the `Node` constructor:

```
IntList1::Node::Node(int n): data(n), next(nullptr) {}
```

The two private methods in Listing 18.4 (intlist1.cpp) (`length` and `dispose`) are recursive. The `insert` method of the `IntList1` class uses `new` to dynamically allocate `Node` objects when adding elements to the collection. The `clear` method is responsible for deallocating the list nodes, effectively cleaning up the memory held by the list.

Listing 18.5 (listmain.cpp) provides some sample client code that exercises a linked list.

Listing 18.5: `listmain.cpp`

```
// list_main.cpp

#include "intlist1.h"
#include <iostream>

int main() {
    bool done = false;
    char command;
    int value;
    IntList1 list;
```

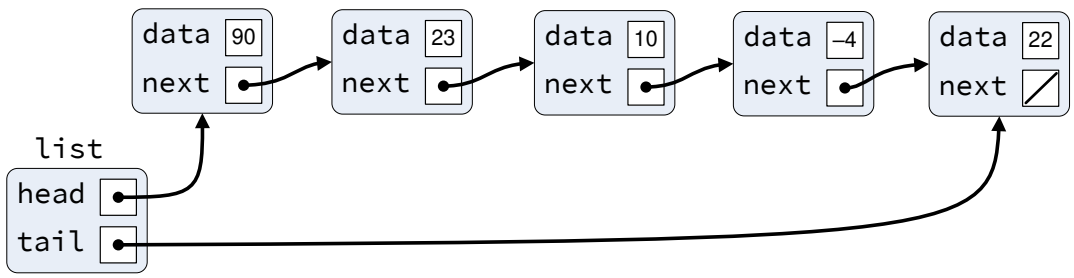
```
    while (!done) {
        std::cout << "I)nsert <item>  P)rint  L)ength  E)rase Q)uit >>";
        std::cin >> command;
        switch (command) {
          case 'I':   // Insert a new element into the list
          case 'i':
            if (std::cin >> value)
                list.insert(value);
            else
                done = true;
            break;
          case 'P':  // Print the contents of the list
          case 'p':
            list.print();
            break;
          case 'L':  // Print the list's length
          case 'l':
            std::cout << "Number of elements: " << list.length() << '\n';
            break;
          case 'E':  // Erase the list
          case 'e':
            list.clear();
            break;
          case 'Q':  // Exit the loop (and the program)
          case 'q':
            done = true;
            break;
        }
    }
    list.clear();  // Free up the space held by the linked list
}
```

The client code in Listing 18.5 (listmain.cpp) allows a user to interactively add items to a list, print the list, determine the list's size, and clear the list. The following shows a sample run:

```
I)nsert <item>  P)rint  L)ength  E)rase Q)uit >>p

I)nsert <item>  P)rint  L)ength  E)rase Q)uit >>i 90
I)nsert <item>  P)rint  L)ength  E)rase Q)uit >>i 23
I)nsert <item>  P)rint  L)ength  E)rase Q)uit >>i 10
I)nsert <item>  P)rint  L)ength  E)rase Q)uit >>i -4
I)nsert <item>  P)rint  L)ength  E)rase Q)uit >>p
90 23 10 -4
I)nsert <item>  P)rint  L)ength  E)rase Q)uit >>i 22
I)nsert <item>  P)rint  L)ength  E)rase Q)uit >>p
90 23 10 -4 22
I)nsert <item>  P)rint  L)ength  E)rase Q)uit >>l
Number of elements: 5
I)nsert <item>  P)rint  L)ength  E)rase Q)uit >>e
I)nsert <item>  P)rint  L)ength  E)rase Q)uit >>p

I)nsert <item>  P)rint  L)ength  E)rase Q)uit >>q
```

Figure 18.3 visualizes the conceptual state of the linked list midway throught this sample run.

Figure 18.3 Conceptual view of a typical `IntList1` object. The linked list object here is named `list`.



Observe that the client code does not use pointers at all. All the pointer manipulations are hidden within the `IntList1` class. Pointer programming can be tricky, and it is easy to introduce subtle, hard to find bugs; thus, encapsulation once again results in easier and more robust application development. The `Node` struct itself is private within `IntList1`, out of the reach of programmers who use these linked lists.

As shown in the `IntList::dispose` method, a pointer variable by itself can be used as a condition within a conditional statement or loop. A null pointer is interpreted as `false`, and any non-null pointer is `true`. This means if `p` is a pointer, the statement

```
if (p)
    /*  Do something  */
```

is a shorthand for

```
if (p != nullptr)
    /*  Do something  */
```

and the statement

```
if (!p)
    /*  Do something  */
```

is a shorthand for

```
if (p == nullptr)
    /*  Do something  */
```

Most C++ programmers use the shorter syntax.

In order to better understand how the recursive methods work, think about the structure of a `Node` object with this interpretation: A `Node` object holds a data item and a pointer to rest of the list that follows. A pointer to a `Node` is either null or non-null. A null pointer represents the empty list. A non-null pointer represents a non-empty list consisting of two parts: the first element in the list and the rest of the list. If the `next` field of a `Node` object is null, the rest of the list is empty.

Armed with this view of lists we can now examine the behavior of the recursive linked list methods in more detail:

- `length`— Lists are either empty or non-empty. The length of the empty list is zero. The length of a non-empty list is at least one because a non-empty list contains at least one element. Symbolically

(not in C++), we can let $\varnothing$ stand for the empty list and $x \rightarrow \varnothing$ stand for the list containing just the element x.

Writing a recursive list `length` method is as simple as this:

- If the list is empty, its length is zero. Symbolically, we can write `length`($\varnothing$) = 0.
- If the list is non-empty, its length is one (counting its first element) plus the `length` of the rest of the list.

Symbolically, we can write `length`($x \rightarrow rest$) = 1 + `length`($rest$).

We can visualize the recursion process for determining the length of $2 \rightarrow 10 \rightarrow 7 \rightarrow \varnothing$ as:

$$\begin{aligned}
\texttt{length}(2 \rightarrow 10 \rightarrow 7 \rightarrow \varnothing) &= 1 + \texttt{length}(10 \rightarrow 7 \rightarrow \varnothing) \\
&= 1 + 1 + \texttt{length}(7 \rightarrow \varnothing) \\
&= 1 + 1 + 1 + \texttt{length}(\varnothing) \\
&= 1 + 1 + 1 + 0 \\
&= 3
\end{aligned}$$

- `dispose`— The `dispose` method behaves similarly to `length`. Notice, however, that it makes the recursive call deleting the nodes in the rest of the list *before* it deletes the current node. An attempt to access data via a pointer after using `delete` to deallocate that data results in undefined behavior; therefore, it is a logic error to attempt to do so. This means the code in `dispose` should not be written as

```
if (p) {   // Logic error!  Do not do it this way!
    delete p;           // Deallocate this node first
    dispose(p->next);   // Then free up the rest of the list
}
```

  Here we are attempting to use `p->next`, which itself uses `p`, after deleting `p`'s memory.

While the recursive methods provided a good review of recursion, recursion is not strictly necessary for these functions. We could instead express the public `length` and `clear` functions as shown in the following code snippets:

```
// Iterative versions of length and clear

int IntList1::length() const {
    int len = 0;
    for (auto cursor = head; cursor; cursor = cursor->next)
        len++;
    return len;
}

void IntList1::clear() {
    auto cursor = head;
    while (cursor) {
        auto temp = cursor;
        cursor = cursor->next;
        delete temp;
    }
    head = tail = nullptr;  // Null head signifies list is empty
}
```

As mentioned in Section 10.5, given the same basic algorithm implemented recursively or iteratively, the iterative version will be more efficient. This is because each recursive call requires additional space on the stack to store fresh local variables (if any) and parameters. Not only does a recursive method or function require more memory, it takes more time to complete because the executing program must perform extra work to set up the new space needed by a recursive invocation and restore the context when a recursive invocation returns. If we build a very large linked list, a call to our recursive `length` or `clear` method could consume considerable space on the stack and impose a significant performance overhead. For future versions of our linked list code we will use the iterative version of the `clear` method.

To optimize the `length` method, we will choose a different route. If you have an application that manages rather large lists and makes frequent calls to the `length` method, even iteration can take considerable time. We can add an integer instance variable to the `IntList1` class that keeps track of the number of elements in the list. This new instance instance variable affects the rest of the class members as follows:

- The constructor would initialize this variable to zero.
- The `insert` method would increment this variable by one each time the client adds an element to the list.
- The `clear` method would reset this variable to zero.
- The `length` method would avoid the loop completely and simply return the value of this variable.

The time spent looping over all the elements in a list counting them can be considerable for large lists, so the additional space required by a single extra integer is a good trade off for speeding up the `length` method.

The `IntList1` class code as defined in Listing 18.3 (intlist1.h) and Listing 18.4 (intlist1.cpp) is useful for introducing the concepts of implementing linked data structures. Our `IntList1` class is, however, fundamentally different from all the earlier custom classes we examined in Section 16. With the optimizations to `length` and `clear` mentioned above, it may appear to be ready for clients to use as an alternative to `std::vector`, but it has some servere limitations and pitfalls that make it practically useless for most applications. Section 18.4 exposes its weaknesses and introduces the modifications required to make it a viable, high-quality class worthy for use in any applications requiring a dynamic linked list data structure.

18.4 Resource Management

Section 18.3 provided our first attempt at a dynamic linked list class, `IntList1`, defined in Listing 18.3 (intlist1.h) and Listing 18.4 (intlist1.cpp). The `IntList1::insert` method dynamically allocates memory for list elements from the heap. It is essential that the programmer intentionally call `clear` when finished with a linked list object. Consider the following function definition:

```
void f() {
    IntList1 my_list;    // Constructor called here
    // Add some numbers to the list
    my_list.insert(22);
    my_list.insert(5);
    my_list.insert(-44);
    // Print the list
    my_list.print();
}   // Oops! Forgot to call my_list.clear!
```

The variable `my_list` is local to function `f`. When function `f` finishes executing the variable `my_list` goes out of scope. At this point the space on the stack allocated for the local `IntList1` *variable* named `my_list` is reclaimed; however, the space for the list's heap-allocated elements remains. The only access the program could have to that memory is via `my_list.head` (or `my_list.tail` to the last element in the list), but `my_list` no longer exists. This presents a classic memory leak.

Observe that none of the classes we have designed thus far except for `IntList1` have this potential problem.

Fortunately C++ provides a way for class designers to specify actions that must occur at the end of an object's lifetime. Analogous to a constructor that executes code at the beginning of an object's existence, a *destructor* is a special method that executes immediately before an object ceases to exist. A destructor has the same name as its class, with a tilde ~ prefix. A destructor accepts no arguments. Listing 18.6 (intlist2.h) adds a destructor to Listing 18.3 (intlist1.h) and also adds the previously suggested optimizations of the `length` and `clear` methods.

Listing 18.6: `intlist2.h`

```
// intlist2.h

class IntList2 {
    // The nested private Node class from before
    struct Node {
        int data;      // A data element of the list
        Node *next;    // The node that follows this one in the list
        Node(int d);   // Constructor
    };

    Node *head;  // Points to the first item in the list
    Node *tail;  // Points to the last item in the list

    int len;     // The number of elements in the list

public:
    // The constructor makes an initially empty list
    IntList2();

    // The destructor that reclaims the list's memory
    ~IntList2();

    // Inserts n onto the back of the list.
    void insert(int n);

    // Prints the contents of the linked list of integers.
    void print() const;

    // Returns the length of the linked list.
    int length() const;

    // Removes all the elements in the linked list.
    void clear();
};
```

Listing 18.7 (intlist2.cpp) provides the implementation of the `IntList2` class.

Listing 18.7: intlist2.cpp

```
// intlist2.cpp

#include "intlist2.h"
#include <iostream>

// Private IntList2 operations

// Node constructor
IntList2::Node::Node(int n): data(n), next(nullptr) {}


// The constructor makes an initially empty list.
// The list is empty when head and tail are null.
// The list's length initially is zero.
IntList2::IntList2(): head(nullptr), tail(nullptr), len(0) {}

// The destructor deallocates the memory held by the list
IntList2::~IntList2() {
    clear();
}

// Inserts n onto the back of the list.
// n is the element to insert.
void IntList2::insert(int n) {
    // Make a node for the new element n
    IntList2::Node *new_node = new Node(n);
    if (tail) {   // Is tail non-null?
       tail->next = new_node;  // Link the new node onto the back
       tail = new_node;        // The new node is the new tail of the list
    }
    else // List is empty, so make head and tail point to new node
        head = tail = new_node;
    len++;  // List now has one more element
}

// Prints the contents of the linked list of integers.
void IntList2::print() const {
    for (auto cursor = head; cursor; cursor = cursor->next)
        std::cout << cursor->data << ' ';
    std::cout << '\n';
}

// Returns the length of the linked list.
int IntList2::length() const {
    return len;
}

// Removes all the elements in the linked list.
void IntList2::clear() {
    auto cursor = head;
    while (cursor) {
        auto temp = cursor;
        cursor = cursor->next;
```

```
        delete temp;
    }
    head = tail = nullptr;  // Null head signifies list is empty
    len = 0;
}
```

The destructor implementation:

```
IntList2::~IntList2() {
    clear();  // Free up the space held by the nodes in the list
}
```

simply invokes the services of the `clear` method to deallocate the space held by the list.

When do constructors and destructors execute? A local or global object definition calls the class constructor to properly initialize the object; for example,

```
IntList2 seq;
```

defines the object `seq`. This definition invokes the `IntList2` constructor to set both `seq.head` and `seq.tail` to `nullptr`. If `seq` is a local variable, its destructor executes at the end of the function's execution. If `seq` is global, its destructor executes when the program finishes.

A pointer is not an object; it points to an object. This means the definition

```
IntList2 *p;
```

does not invoke `IntList2`'s constructor. Given this definition of `p`, the statement

```
p = new IntList2;
```

actually creates an `IntList2` object and therefore calls the `IntList2` constructor for the newly created object. The object to which `p` points is not destroyed until the programmer uses `delete`, as in

```
delete p;
```

It is at this point `IntList2`'s destructor executes for `p`'s object.

The addition of the destructor to `IntList2` removes the memory leak from the `f` function we saw earlier. Listing 18.8 (testf.cpp) is a complete program that uses `IntList2` in function `f`.

Listing 18.8: `testf.cpp`

```
#include <iostream>
#include "intlist2.h"

void f() {
    IntList2 my_list;    // Constructor called here
    // Add some numbers to the list
    my_list.insert(22);
    my_list.insert(5);
    my_list.insert(-44);
    // Print the list
    my_list.print();
}   // my_list goes out of scope; destructor automatically frees its memory

int main() {
```

```
    f();
}
```

We can verify that our revised linked list class properly frees up its dynamic memory by augmenting the internal `Node` struct with a destructor, as shown here:

```
// This is the modified nested, private Node class
struct Node {
    int data;       // A data element of the list
    Node *next;     // The node that follows this one in the list
    Node(int d);    // Constructor
    ~Node();        // Destructor
};
```

We also modify the constructor implementation and add the destructor implementation as shown here:

```
// Node constructor
IntList2::Node::Node(int n): data(n), next(nullptr) {
    std::cout << "Creating node " << data
              << "  (" << reinterpret_cast<uintptr_t>(this) << ")\n";
}

// Node destructor
IntList2::Node::~Node() {
    std::cout << "Destroying node " << data
              << "  (" << reinterpret_cast<uintptr_t>(this) << ")\n";
}
```

Listing 18.9 (intlist3.h) provides the header file updated with this modified `Node` nested class, and Listing 18.10 (intlist3.cpp) contains the implementation.

Listing 18.9: `intlist3.h`

```
// intlist3.h

class IntList3 {
    // The nested private Node class from before
    struct Node {
        int data;       // A data element of the list
        Node *next;     // The node that follows this one in the list
        Node(int d);    // Constructor
        ~Node();        // Destructor
    };

    Node *head;  // Points to the first item in the list
    Node *tail;  // Points to the last item in the list

    int len;     // Number of elements in the list

public:
    // The constructor makes an initially empty list
    IntList3();

    // The destructor that reclaims the list's memory
```

```
    ~IntList3();

    // Inserts n onto the back of the list.
    void insert(int n);

    // Prints the contents of the linked list of integers.
    void print() const;

    // Returns the length of the linked list.
    int length() const;

    // Removes all the elements in the linked list.
    void clear();
};
```

Listing 18.10: `intlist3.cpp`

```
// intlist3.cpp

#include "intlist3.h"
#include <iostream>

// Private IntList3 operations

// Node constructor
IntList3::Node::Node(int n): data(n), next(nullptr) {
    std::cout << "Creating node " << data
              << "  (" << reinterpret_cast<uintptr_t>(this) << ")\n";
}

IntList3::Node::~Node() {
    std::cout << "Destroying node " << data
              << "  (" << reinterpret_cast<uintptr_t>(this) << ")\n";
}

// The constructor makes an initially empty list.
// The list is empty when head and tail are null.
// The list's length initially is zero.
IntList3::IntList3(): head(nullptr), tail(nullptr), len(0) {}

// The destructor deallocates the memory held by the list
IntList3::~IntList3() {
    clear();
}

// Inserts n onto the back of the list.
// n is the element to insert.
void IntList3::insert(int n) {
    // Make a node for the new element n
    IntList3::Node *new_node = new Node(n);
    if (tail) {   // Is tail non-null?
       tail->next = new_node;  // Link the new node onto the back
       tail = new_node;        // The new node is the new tail of the list
    }
    else // List is empty, so make head and tail point to new node
```

```
        head = tail = new_node;
    len++;  // List now has one more element
}

// Prints the contents of the linked list of integers.
void IntList3::print() const {
    for (auto cursor = head; cursor; cursor = cursor->next)
        std::cout << cursor->data << ' ';
    std::cout << '\n';
}

// Returns the length of the linked list.
int IntList3::length() const {
    return len;
}

// Removes all the elements in the linked list.
void IntList3::clear() {
    auto cursor = head;
    while (cursor) {
        auto temp = cursor;
        cursor = cursor->next;
        delete temp;
    }
    head = tail = nullptr;  // Null head signifies list is empty
    len = 0;
}
```

Here we have the `Node` constructor identify which element it is creating by printing its `data` field and the address of where it resides in memory. C++ is very strict about conversions between pointer and non-pointer types—it is easy to do by mistake and almost never intended—so a simple `static_cast` will not work here. The `reinterpret_cast` removes the safeguard and treats the bits that make up the pointer as a `uintptr_t`, an integer type guaranteed to represent the same range of values as a pointer. The destructor indicates when a `Node` object is destroyed. Listing 18.11 (testf2.cpp) is same program as Listing 18.8 (testf.cpp), except it uses an `IntList3` object instead of an `IntList2` object.

Listing 18.11: `testf2.cpp`

```
#include <iostream>
#include "intlist3.h"

void f() {
    IntList3 my_list;    // Constructor called here
    // Add some numbers to the list
    my_list.insert(22);
    my_list.insert(5);
    my_list.insert(-44);
    // Print the list
    my_list.print();
}   // my_list goes out of scope; destructor automatically frees its memory

int main() {
    f();
}
```

If we remove the destructor from the `IntList3` class, effectively making it the `IntList1` class without the enhanced `Node` struct, Listing 18.11 (testf2.cpp) will print the following:

```
Creating node 22  (16846704)
Creating node 5  (16823408)
Creating node -44  (16822992)
22 5 -44
```

(The actual memory addresses will vary from run to run, but the `Node` data values will be the same.) The function creates three nodes but `delete`s none of them. When this function returns, `my_list` no longer exists, so we no longer can access the nodes it allocated. If we leave `IntList3` as is, Listing 18.8 (testf.cpp) prints the following:

```
Creating node 22  (19337072)
Creating node 5  (19313328)
Creating node -44  (19313696)
22 5 -44
Destroying node -44  (19313696)
Destroying node 5  (19313328)
Destroying node 22  (19337072)
```

This shows that the `IntList3` destructor properly deallocates the dynamic memory held by the linked list object. The `Node` constructor executes when declaring a `Node` object or when using `new` to create a `Node` object to assign to a `Node` pointer. The `insert` method uses `new` to create a `Node` object, so this invokes the constructor. The `Node` destructor executes when an object goes out of scope or when `delete` deallocates a dynamically-allocated object. The `IntList2` destructor uses `delete` to free up each node in the list.

Consider the case of dynamically allocating the linked list itself, not just its nodes:

```
void f2() {
    IntList3 *lptr;       // Pointer, constructor NOT called yet
    lptr = new IntList3;  // Constructor called here
    // Add some numbers to the list
    lptr->insert(22);
    lptr->insert(5);
    lptr->insert(-44);
    // Print the list
    lptr->print();
    delete lptr;         // Destructor called here
}
```

In this `f2` function `lptr` is not an object; it is a pointer to an object. Declaring `lptr` does not create an object, and, therefore, the `IntList3` constructor does not execute. The statement

```
lptr = new IntList3;
```

does create an object, and so the `IntList3` constructor executes on behalf of the object to which `lptr` points. When `lptr` goes out of scope at the end of function `f2`'s execution, the stack variable `lptr` goes away, but since it is a pointer, not an object, no destructor code executes. The client must explicitly free up memory with `delete`:

```
delete lptr;         // Destructor called here
```

unless the programmer intends for the function to return the pointer to the client (in which case the client is responsible for calling `delete` when finished with the object).

As you can see, dynamically allocating the list itself (apart from the nodes that store its elements) brings us back to the situation we were in before: We must remember to do something when we are finished with the object. In this case we must remember to `delete` the list itself. A destructor cannot help us here. This is a good example that demonstrates that while dynamic memory enables us to do many interesting things, it is best to avoid it unless absolutely necessary to achieve the behavior we need.

A destructor performs the reverse role of a constructor: A constructor ensures that a new object begins its life in a well-defined state, while a destructor is responsible for performing actions required when an object's life is over. A destructor is unnecessary for most classes. For a linked list object, however, a destructor is essential because the nodes that comprise the linked list are dynamically allocated with the `new` operator. Any memory allocated with `new` needs a corresponding call to `delete` when the memory no longer will be used. The `clear` method will take care of the memory deallocation, but, without a destructor, a programmer must remember to intentionally call `clear` when finished with a linked list object. The destructor relieves the programmer of this responsibility and removes the possibility of a memory leak. The destructor simply calls `clear` to clean up the resources held by the list.

Listing 18.12 (testdestructors.cpp) demonstrates that executing program destroys local and global objects in the reverse order of their creation. For objects allocated via the `new` operator, as in the function `test_widget_pointers`, destructors execute only when applying `delete` to the associated pointers.

Listing 18.12: `testdestructors.cpp`

```
#include <iostream>

class Widget {
    int data;
public:
    Widget(int n): data(n) {
        std::cout << "Creating widget " << data
                  << "  (" << reinterpret_cast<uintptr_t>(this) << ")\n";
    }
    ~Widget() {
        std::cout << "Destroying widget " << data
                  << "  (" << reinterpret_cast<uintptr_t>(this) << ")\n";
    }
};

// Global widgets
Widget global1(100);
Widget global2(200);

void test_widget_objects() {
    std::cout << "Entering test_widget_objects" << '\n';
    Widget w1(1);
    Widget w2(2);
    Widget w3(3);
    Widget w4(4);
    std::cout << "Leaving test_widget_objects" << '\n';
}

void test_widget_pointers() {
    std::cout << "Entering test_widget_pointers" << '\n';
```

```
    Widget *p1 = new Widget(10);
    Widget *p2 = new Widget(20);
    Widget *p3 = new Widget(30);
    Widget *p4 = new Widget(40);
    delete p2;
    delete p1;
    delete p4;
    // Not deleting p3, introducing a memory leak
    std::cout << "Leaving test_widget_pointers" << '\n';
}

int main() {
    std::cout << "Entering main" << '\n';
    test_widget_objects();
    test_widget_pointers();
    std::cout << "Leaving main" << '\n';
}
```

Listing 18.12 (testdestructors.cpp) prints the following:

```
Creating widget 100  (2707068)
Creating widget 200  (2707064)
Entering main
Entering test_widget_objects
Creating widget 1  (14024180)
Creating widget 2  (14024184)
Creating widget 3  (14024188)
Creating widget 4  (14024192)
Leaving test_widget_objects
Destroying widget 4  (14024192)
Destroying widget 3  (14024188)
Destroying widget 2  (14024184)
Destroying widget 1  (14024180)
Entering test_widget_pointers
Creating widget 10  (14790752)
Creating widget 20  (14790816)
Creating widget 30  (14790832)
Creating widget 40  (14791760)
Destroying widget 20  (14790816)
Destroying widget 10  (14790752)
Destroying widget 40  (14791760)
Leaving test_widget_pointers
Leaving main
Destroying widget 200  (2707064)
Destroying widget 100  (2707068)
```

Adding a destructor to the `IntList1` class is a significant step in correcting its deficiencies, but its successors, `IntList2` and `IntList3`, still are not ready for general release. Consider Listing 18.13 (listassign.cpp) that uses the `IntList3` class that properly automatically cleans up its memory.

Listing 18.13: `listassign.cpp`

```
#include "intlist3.h"
#include <iostream>
```

```
void f() {
    IntList3 seq1, seq2;  // Create two empty integer linked lists

    seq1.insert(10);  // Build the list [10]-->[-2]-->[8]
    seq1.insert(-2);
    seq1.insert(8);
    seq1.print();

    seq2.insert(5);   // Build the list [5]-->[4]
    seq2.insert(4);
    seq2.print();

    std::cout << "--------------\n";

    seq1 = seq2;      // What does this do?

    seq1.print();
}

int main() {
    f();
    std::cout << "All done\n";
}
```

Listing 18.13 (listassign.cpp) builds two separate linked lists and then assigns one list to the other. On one system the program prints

```
Creating node 10  (14457304)
Creating node -2  (14457592)
Creating node 8  (14456152)
10 -2 8
Creating node 5  (14456200)
Creating node 4  (14456248)
5 4
--------------
5 4
Destroying node 4  (14456248)
Destroying node 5  (14456200)
Destroying node 4  (14456248)
Destroying node 5  (14456200)
All done
```

During this particular run the program ran to completion; however, sometimes the program will crash before printing *All done*. On some systems the program crashes before printing the last three lines of output.

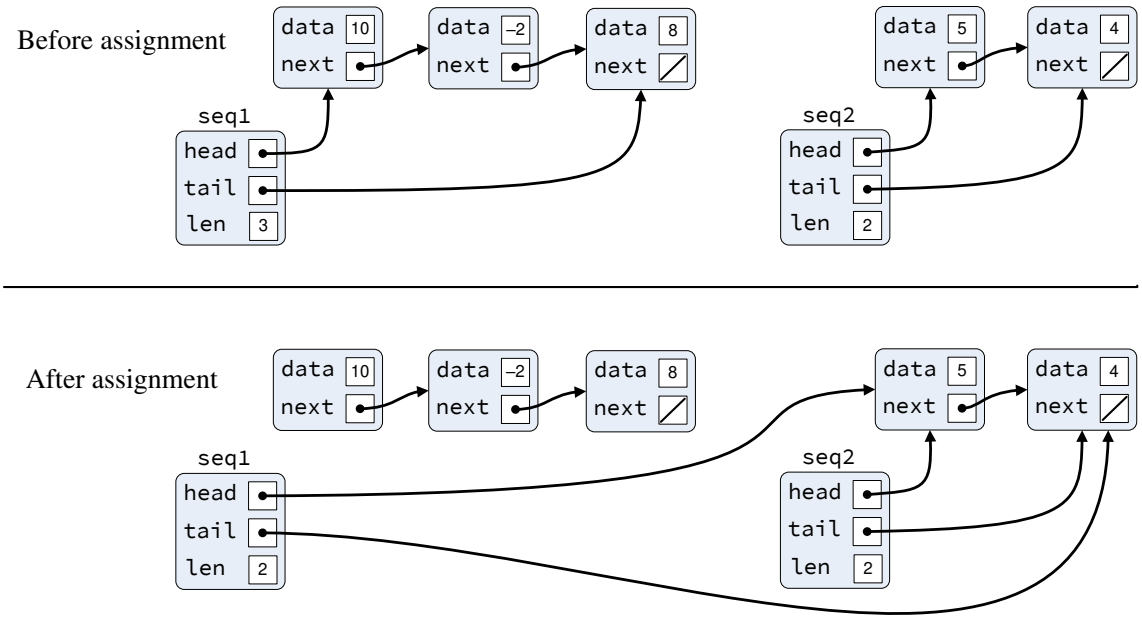
During this run of Listing 18.13 (listassign.cpp) the output shows that the program never destroys the nodes containing 10, -2, and 8. Further, the program destroys the nodes containing 4 and 5 twice! We can see the program `delete`s the same memory twice by comparing the memory addresses in parentheses. This undefined behavior of double deletion is what causes the program to crash at times.

The problem is in this statement:

```
seq1 = seq2;
```

What exactly does this statement do?

Figure 18.4 Conceptual view of `IntList3` assignment. After the assignment the two linked list objects alias the same list of nodes, and `seq1`'s original list becomes unreachable.



With no additional guidance from the programmer, the assignment operator for a programmer-defined type simply copies the bits in memory that make up the object to right of the assignment operator into the memory reserved for the object on the left side of the assignment operator. Under Visual C++producing a 32-bit executable, for example, an `IntList3` object occupies 12 bytes of memory—four bytes for its `head` pointer plus four bytes for its `tail` pointer plus four bytes for `len`. The actual size of an `IntList3` object will vary from system to system and will be larger on 64-bit computers. You can verify the actual size of an `IntList3` object on your system by adding the following statement to Listing 18.13 (listassign.cpp):

```
std::cout << sizeof seq1 << '\n';  // Prints 8 or 16 under Visual C++
```

The assignment

```
seq1 = seq2;
```

simply copies `seq2.head` to `seq1.head` and copies `seq2.tail` to `seq1.tail` and copies `seq2.len` to `seq1.len`. This produces three undesirable results:

- **Aliasing**. After the assignment the objects `seq1` and `seq2` refer to the same list of nodes. See Figure 18.4 for an illustration of the assignment process.

  This means any actions to modify the list managed by `seq1` will modify `seq2` identically. The assignment operator does not make a copy of the contents of `seq2`. This means that our linked list objects behave differently than `std::vector` objects with respect to assignment; assigning one vector to another makes a copy of the contained elements. Clients will expect our linked list objects to behave similarly. This aliasing problem, however, is the least of our problems.

- **Memory leak**. Simply redirecting `seq1`'s `head` and `tail` pointers to `seq2`'s list renders the nodes in `seq1`'s original list unreachable from any variables within the program. The program has no way to `delete` the nodes in `seq1`'s original list. This constitutes a memory leak.
- **Memory corruption**. Both `seq1` and `seq2` are objects local to the `main` function. Based on their declaration order, `seq1`'s constructor executes before `seq2`'s constructor. This means when `main` is finished executing `seq2`'s destructor will execute before `seq1`'s destructor, as object destruction occurs in the reverse order of object construction for local objects. The destructor for `seq2` will free up all the nodes in `seq2`'s list. Next, `seq1`'s destructor will attempt to `delete` the memory occupied by its list. Unfortunately this involves referencing and deleting dynamic memory previously deleted by `seq2`. Any attempts to `delete` already `delete`d memory results in *undefined behavior* that usually results in memory corruption. That is why Listing 18.13 (listassign.cpp) crashes during some runs.

Fortunately C++ provides a way for programmers to customize how assignment works for a custom type. Before we tackle assignment itself, we must distinguish between *initialization* and *assignment*. Consider the following statement:

```
int x = 3;
```

This statement defines and *initializes* the variable `x` to 3. This statement is fundamentally different from the following two statements:

```
int x;
x = 3;
```

For simple types like integers, floating-point numbers, and characters, this pair of statements on the surface behave identically to the single statement above. In fact, these two statements involve a declaration followed by assignment. Initialization and assignment are not the same thing. Recall from Section 3.2 the alternate syntax for initialization:

```
int x{3};
```

To the compiler, this statement is equivalent to the initialization statement above. Note, however, that the following code sequence is not legal:

```
int x;
x{3};    // Illegal
```

C++ gives programmers full control over initialization and assignment of custom types. We have seen how we can specify initialization via a constructor. In order to legitimize our linked list class and make it safe for clients to use, we need to provide an additional constructor, called the *copy constructor*, and define an assignment operator for the class.

The copy constructor for a class named `X` has the following form:

```
class X {
    // Other stuff for class X

    X::X(const X& other);
};
```

The parameter is a reference to an object of the same type.

To specify assignment, we supply an `operator=` method of the form

```
class X {
    // Other stuff for class X

    X& operator=(const X& other);
};
```

Note that the assignment operator accepts a constant reference parameter of the type of the class and returns a reference to the type of the class.

Listing 18.14 (intlist4.h) extends Listing 18.9 (intlist3.h) providing copy construction and assignment.

Listing 18.14: intlist4.h

```
// intlist4.h

class IntList4 {
    // The nested private Node class from before
    struct Node {
        int data;       // A data element of the list
        Node *next;     // The node that follows this one in the list
        Node(int d);    // Constructor
        ~Node();        // Destructor
    };

    Node *head;  // Points to the first item in the list
    Node *tail;  // Points to the last item in the list

    int len;     // Number of elements in the list

public:
    // The constructor makes an initially empty list
    IntList4();

    // The destructor that reclaims the list's memory
    ~IntList4();

    // Copy constructor
    IntList4(const IntList4& other);

    // Assignment operator
    IntList4& operator=(const IntList4& other);

    // Inserts n onto the back of the list.
    void insert(int n);

    // Prints the contents of the linked list of integers.
    void print() const;

    // Returns the length of the linked list.
    int length() const;

    // Removes all the elements in the linked list.
    void clear();
};
```

The implementation of the copy constructor is straightforward:

```
IntList4::IntList4(const IntList4& other): IntList4() {
    // Walk through other's list inserting each of its elements
    // into this list
    for (auto cursor = other.head; cursor; cursor = cursor->next)
        insert(cursor->data);
}
```

The expression `IntList4()` in the constructor initialization list following the colon is calling the overloaded version of the constructor that accepts no arguments. This is known as *constructor delegation*. The no-argument constructor simply initializes `head` and `tail` to `nullptr`. We could have done the same thing here without calling the other constructor, but, in general, delegation is a good idea. Delegation can avoid code duplication, and while there currently is not much code in the no-argument constructor, we may decide to add more functionality in the future. By delegating, we can add the extra activity to the no-argument constructor without having to add it also to this copy constructor.

Constructors always begin with a brand new, non-preexisting object. In the case of our linked list objects, that means our constructor does not have to worry about cleaning up any preexisting list of nodes. All our copy constructor needs to do is ensure that `head` and `tail` instance variables initially are null (the constructor delegation takes care of that) and then visit each node in the other list, inserting that node's data value into its own list as it goes.

Next we will consider assignment. Suppose `lst1` and `lst2` are two linked list objects. In the assignment

```
lst1 = lst2;
```

we will refer to `lst1` as the *assigned-to* object and `lst2` as the *assigned-from* object. Note that the syntactically desugared equivalent of this statement is

```
lst1.operator=(lst2);  // Same as lst1 = lst2;
```

Unlike the copy constructor, the assignment operator works with a preexisting object. This means the assignment operator in the course of its operation must deallocate the original list of nodes, if any, managed by the assigned-to list. Failure to do so would introduce a memory leak. Also, the assignment operator must make a copy of all the values in the assigned-from list to avoid aliasing.

There are various ways we can implement the assignment operator to ensure it works correctly, but since assignment of this nature is such a common operation, C++ programmers have developed a standard idiom that guarantees the correctness of assignment. Best practices dictates that we implement our assignment as a process. This entails making a local temporary copy of the assigned-from list and swapping the list of nodes from the temporary list with list of nodes from the assigned-to list. Note that swapping the list of nodes simply requires swapping the `head` and `tail` pointers between the two objects. This copy-and-swap process depends on a correctly implemented copy constructor and a correctly implemented destructor.

The following code provides the necessary assignment operator:

```
IntList4& IntList4::operator=(const IntList4& other) {
    // Make a local, temporary copy of other
    IntList4 temp{other};
    // Exchange the head and tail pointers and len from this list
    // with those of the new, temporary list
    std::swap(head, temp.head);
    std::swap(tail, temp.tail);
```

```
    std::swap(len, temp.len);
    // The temporary list now points to this list's original contents,
    // and this list now points to the copy of other's list
    // The temporary list will be destroyed when this constructor returns
    return *this;
}
```

Surprising, perhaps, is what we do **not** see in this assignment operator:

- Missing is a loop or any indication of visiting each node in the assign-from list (`other`).
- Missing is any calls to `delete` to free up any preexisting list nodes in the assign-to list.

How does this assignment operator magically meet our requirements for a correct assignment? This is how it works:

- The first executable line:

  ```
  IntList4 temp{other};
  ```

  invokes the copy constructor to make a copy of the assign-from list (`other`) to a temporary list object (`temp`). Recall that the copy constructor uses a loop to traverse the assign-from list, making a copy of each node in that list. This solves the aliasing issue and explains how this assignment operator does actually visit every node in `other`'s list.

- The next two statements:

  ```
  std::swap(head, temp.head);
  std::swap(tail, temp.tail);
  std::swap(len, temp.len);
  ```

  exchange the `head` and `tail` pointers and `len` between the assign-to list object and the temporary list object. This means the assign-to list object now holds the copy of the `assign-from` list, and the temporary list object holds the list originally managed by the assign-to list object.

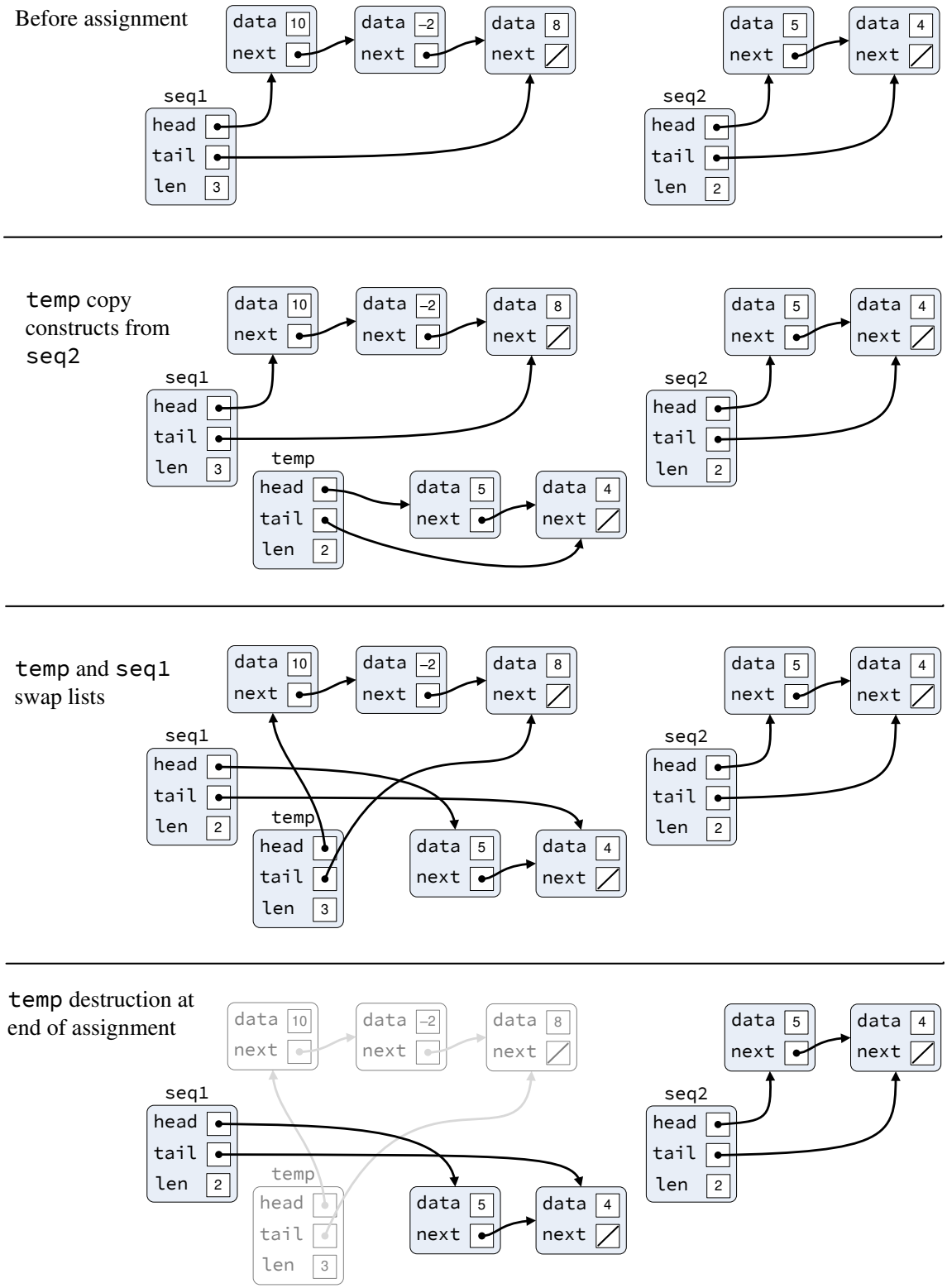
- Since it is a local variable the temporary list object (`temp`) goes out of scope at the end of the function's execution. The `temp` object's destructor executes at this time. This temporary object now holds the `assign-to` object's original list, so the destructor properly deallocates all the nodes in the `assign-to` object's original list. The calls to `delete` to prevent the memory leak are found in the linked list class destructor.

Figure 18.5 illustrates the copy-and-swap process of the assignment operator.

Our assignment operator leverages the code already written in the copy constructor and destructor. Any other implementation of assignment that correctly addresses the aliasing and memory management issues necessarily would duplicate the functionality of either the copy constructor or the destructor.

Note that the `IntList4::operator=` method's declaration indicates it returns a reference to an `IntList4` object, and we see in the last line of `IntList4::operator=`'s body that the method actually returns a reference to the object that appears on the left side of the assignment operator during its invocation. Recall that the `this` keyword represents a pointer to the object upon which the caller invoked the method (in this case the assign-to list object). If `lst1` and `lst2` are list objects, when executing the statement

Figure 18.5 The copy-and-swap process performed by the assignment operator



```
lst1 = lst2;    // Syntactic sugar for lst1.operator=(lst2);
```

the `this` pointer in points to `lst1` (and `lst2` is the parameter named `other`).. The method must return an object *reference*, not a *pointer* to the object, so we must dereference it as `*this`. This is the typical C++ idiom used when defining `operator=` for programmer-defined types.

We could instead define the `operator=` method to return nothing (that is, declare its return type to be `void`). Instead we could define the method to return some type of object other than `IntList4`. If we do so, however, our `IntList4` objects will not behavior properly with *chained assignment* (see Section 4.3). To illustrate, suppose `lst1`, `lst2`, and `lst3` are all properly declared `IntList4` objects. If we define the `IntList4::operator=` method to return `void`, the following statement:

```
lst1 = lst2;
```

is perfectly acceptible and behaves as intended, but the compiler will not accept the following statement:

```
lst1 = lst2 = lst3;
```

This is because the assignment operator associates right-to-left; the compiler interprets the chained assignment statement above as if we had parentheses placed as shown in the following statement:

```
lst1 = (lst2 = lst3);
```

If our assignment operator returns `void`, this statement produces a compile-time error because the subexpression `(lst2 = lst3)` would evaluate to `void`, and `void` is a degenerate type that we cannot assign to `lst1`.

If we use the original definition above of the `IntList4::operator=` method that returns a reference to the `IntList4` object on the left side of the assignment operator (`*this`), the compiler does not complain and the chained assignment works flawlessly in the same manner as chained assignment involving the built-in numeric types; that is, after executing the chained assignment statement above, `lst1` and `lst2` each will contain exact copies of the elements in `lst3`.

Listing 18.15 (intlist4.cpp) provides the complete implementation of our `IntList4` class.

Listing 18.15: `intlist4.cpp`

```
// intlist4.cpp

#include "intlist4.h"
#include <iostream>
#include <utility>   // For std::swap

// Private IntList4 operations

// Node constructor
IntList4::Node::Node(int n): data(n), next(nullptr) {
    std::cout << "Creating node " << data
              << "  (" << reinterpret_cast<uintptr_t>(this) << ")\n";
}

IntList4::Node::~Node() {
    std::cout << "Destroying node " << data
              << "  (" << reinterpret_cast<uintptr_t>(this) << ")\n";
}
```

```
// The constructor makes an initially empty list.
// The list is empty when head and tail are null.
// The list's size initially is zero.
IntList4::IntList4(): head(nullptr), tail(nullptr), len(0) {}

// Copy constructor makes a copy of the other object's list
IntList4::IntList4(const IntList4& other): IntList4() {
    // Walk through other's list inserting each of its elements
    // into this list
    for (auto cursor = other.head; cursor; cursor = cursor->next)
        insert(cursor->data);
}

// Assignment operator
IntList4& IntList4::operator=(const IntList4& other) {
    // Make a local, temporary copy of other
    IntList4 temp{other};
    // Exchange the head and tail pointers and len from this list
    // with those of the new, temporary list
    std::swap(head, temp.head);
    std::swap(tail, temp.tail);
    std::swap(len, temp.len);
    // The temporary list now points to this list's original contents,
    // and this list now points to the copy of other's list
    // The temporary list will be destroyed when this constructor returns
    return *this;
}

// The destructor deallocates the memory held by the list
IntList4::~IntList4() { clear(); }

// Inserts n onto the back of the list.
// n is the element to insert.
void IntList4::insert(int n) {
    // Make a node for the new element n
    IntList4::Node *new_node = new Node(n);
    if (tail) {   // Is tail non-null?
       tail->next = new_node;  // Link the new node onto the back
       tail = new_node;        // The new node is the new tail of the list
    }
    else // List is empty, so make head and tail point to new node
        head = tail = new_node;
    len++;  // List now has one more element
}

// Prints the contents of the linked list of integers.
void IntList4::print() const {
    for (auto cursor = head; cursor; cursor = cursor->next)
        std::cout << cursor->data << ' ';
    std::cout << '\n';
}

// Returns the length of the linked list.
int IntList4::length() const {
    return len;
```

Figure 18.6 An annotated output of Listing 18.13 (listassign.cpp) updated to use our `IntList4` class. Examine carefully the node memory addresses (in parentheses) to convince yourself that the implementation of correct copy constructor, assignment operator, and destructor for the class solved the aliasing, memory leak, and undefined behavior problems.

```
Creating node 10  (17433976)
Creating node -2  (17477728)
Creating node 8  (17477744)
10 -2 8
Creating node 5  (17477760)
Creating node 4  (17478592)
5 4
--------------
Creating node 5  (17478752)
Creating node 4  (17478560)
Destroying node 8  (17477744)
Destroying node -2  (17477728)
Destroying node 10  (17433976)
5 4
Destroying node 4  (17478592)
Destroying node 5  (17477760)
Destroying node 4  (17478560)
Destroying node 5  (17478752)
All done
```

Creating `seq1`'s original list

Creating `seq2`'s list

Creating a copy of `seq2`'s list for `seq1`

Destroying `seq1`'s original list

Destroying `seq2`'s list

Destroying `seq1`'s list (the copy of `seq2`'s list)

```
}

// Removes all the elements in the linked list.
void IntList4::clear() {
    auto cursor = head;
    while (cursor) {
        auto temp = cursor;
        cursor = cursor->next;
        delete temp;
    }
    head = tail = nullptr;  // Null head signifies list is empty
    len = 0;
}
```

See Figure 18.6 provides an annotated output of Listing 18.13 (listassign.cpp) updated to use our `IntList4` class.

The output demonstrates that assignment now works correctly for our linked list objects. Correctly implementing the copy constructor, assignment operator, and destructor for the `IntList4` class solved the aliasing, memory leak, and undefined behavior concerns.

The copy constructor, assignment operator, and destructor have a special relationship in C++. These three methods are involved in the *Rule of Three*. The Rule of Three is this: If a class designer feels the need to add a copy constructor, assignment operator, and/or destructor to a class, the class should have all three

methods. Said another way, it almost always is a mistake to have just one or two of these special methods defined for a class without including all three.

The Rule of Three is a design guideline; it is not requirement of the C++ language that the compiler enforces. Consider why the rule exists. If an object must be guaranteed to perform some action at the end of its existence and failure to do so would result in undesirable consequences as the program continues to execute, the object's class must contain a destructor. This essential action usually involves some releasing some resource it owns. If the destructor is responsible for releasing a resource, the copy constructor must somehow acquire a resource that is in some way related to the resource of an existing object. The assignment operator must both release an owned resource and acquire a resource related to the resource of an existing object. What if you leave out one of the three?

- If you omit the destructor, the default destructor does no resource clean up. Objects can acquire resources through copy construction and assignment but cannot release them. Most resources (like memory) are finite, so lack of an appropriate destructor will limit the number objects available to the executing application.
- If you omit the copy constructor, the default copy constructor will copy the bits of the existing object into the new object. This means the copy-constructed object will have the exact resources managed by another object. The destruction of exactly one of the objects will release the resources held by the other object. The non-destroyed object then will not have access to the resources it needs to function.
- If you omit the assignment operator, the default assignment will copy the bits of one existing object into another existing object. This means the assigned object will have the exact resources managed by another object. It also means that the resources the assigned object originally held are not reclaimed. The destruction of exactly one of the objects will release the resources held by the other object. The non-destroyed object then will not have access to the resources it needs to function.

In sum, you should have a very good reason for designing a class that violates the Rule of Three.

18.5 Rvalue References

Consider the following code fragment:

```
int x = 5, y;
y = x + 2;
```

In the assignment statement

```
y = x + 2;
```

The variable `y` is known as an *lvalue*. The *l* in lvalue stands for *left*, as `y` can appear all by itself on the *left* side or right side of the assignment operator. The expression `x + 2` is classified as an *rvalue*, since it can appear only on the *right* side of the assignment operator but cannot appear on the left side; for example, the following statement is illegal:

```
x + 2 = y;  // Illegal!
```

The expression `x + 2` represents a temporary value, as it has meaning and existence only during the execution of the statement in which it appears. The executing program stores the result of the computation of `x + 2` temporarily in memory, but the program ordinarily cannot reference that memory after the statement

finishes. The term *temporary* is appropriate, as the program may reuse that same memory for other purposes after the statement executes.

Consider the following variable declarations:

```
int x = 5;
int& r = x + 3;   // Illegal!
```

The second declaration is illegal; C++ does not allow us to assign a temporary to a reference variable. In this case the variable `r` would be an alias for a quantity that ceases to exist immediately after the statement executes. The variable `r`, however, would continue to exist, but it would not be aliasing any valid quantity.

C++ does permit us to make a `const` reference to a temporary. The following code fragment is legal:

```
int x = 5;
const int& cr = x + 3;   // Legal
```

The `const` reference places the value of the temporary into a memory location referenced via `cr`. The temporary's value persists until `cr` goes out of scope. In this case `cr` behaves as if it were simply a `const int`.

Next, consider the following function definition:

```
int f(int n) {
    return 10 * n;
}
```

In the following statement:

```
std::cout << f(x + 2) << '\n';
```

the actual parameter is a temporary value passed to the function `f`. Everything works as expected because the executing program copies the temporary's value to the formal parameter `n` in this pass by value example.

Now consider a similar but slightly different function definition:

```
int g(int& n) {
    return 10 * n;
}
```

The following statement is illegal:

```
std::cout << g(x + 2) << '\n';
```

The function `g` accepts a reference to a variable, but the caller is attempting to pass a temporary. Once function `g` begins executing, the memory reserved for the original temporary is invalid, so the compiler rightly does not permit this.

As with the simple declarations we saw above, given function `h`:

```
int h(const int& n) {
    return 10 * n;
}
```

the following statement is perfectly acceptable:

```
std::cout << h(x + 2) << '\n';
```

Since n is a `const int` reference in h, the executing program will make a copy of the temporary's value to send to h.

While these examples may seem a bit esoteric, we are paving the way for a more efficient linked list custom type. Consider Listing 18.16 (templist.cpp) which uses our `IntList4` type from Listing 18.15 (intlist4.cpp) and Listing 18.15 (intlist4.cpp).

Listing 18.16: `templist.cpp`

```
#include <iostream>
#include "intlist4.h"


IntList4 make_list(int n) {
    IntList4 result;
    for (int i = 0; i < n; i++)
        result.insert(i);
    return result;
}

int main() {
    auto my_list = make_list(4);
    my_list.print();
}
```

The output of Listing 18.16 (templist.cpp) indicates the creation and destruction of two `IntList4` objects when we would expect only one. Figure 18.7 annotates the output of Listing 18.16 (templist.cpp) to explain the program's behavior.

The statement

```
auto my_list = make_list(4);
```

invokes the `IntList4` copy constructor. This statement is equivalent to

```
auto my_list{make_list(4)};
```

Either way, the `make_list(4)` expression represents the `IntList4` object returned by the `make_list` function when passed the actual parameter 4. The expression `make_list(4)` is an rvalue, a temporary, because it cannot appear on the left side of the assignment operator. Since the `IntList4` copy constructor accepts a `const IntList4&` parameter, it will accept the temporary object, but it makes a copy of the temporary to send to the copy constructor.

If is wasteful to make a copy of this transient object. If the list object contained millions of nodes, this additional construction and destruction could be time consuming.

C++ does permit a special kind of reference for rvalues known as an *rvalue reference*. An rvalue reference enables programmers to alias a temporary without making a copy.

In the following code:

```
int x = 5;
int&& r = x + 3;   // Legal, note the two ampersands
std::cout << "x = " << x << "  r = " << r << '\n';
```

we see that variable r provides access to the temporary's value.

We can use rvalue references as function parameters, as the following function demonstrates:

Figure 18.7 An annotated output of Listing 18.16 (templist.cpp). Examine carefully the node memory addresses (in parentheses). The executing program first creates the temporary `IntList4` object returned by `make_list`. It the copy constructs the `my_list` object from the temporary and then destroys the temporary. After printing the linked list, the program finally destroys the `my_list` object.

```
Creating node 0  (18154856)
Creating node 1  (18199696)
Creating node 2  (18199872)
Creating node 3  (18199680)
Creating node 0  (18199440)
Creating node 1  (18199760)
Creating node 2  (18199808)
Creating node 3  (18199584)
Destroying node 3  (18199680)
Destroying node 2  (18199872)
Destroying node 1  (18199696)
Destroying node 0  (18154856)
0 1 2 3
Destroying node 3  (18199584)
Destroying node 2  (18199808)
Destroying node 1  (18199760)
Destroying node 0  (18199440)
```

Creating the temporary `make_list(5)`

Copy constructing `my_list` from the temporary

Destroying the temporary

Destroying `my_list`

```
int h(int&& n) {
    return 10 * n;
}
```

Given the definition of function h, the following invocation is legal:

```
std::cout << h(x + 3) << '\n';
```

We can overload functions and methods to accept both normal references and rvalue references. The compiler can determine which overloaded function to invoke based on the actual parameters that the caller passes. Consider Listing 18.17 (rvalueparams.cpp).

Listing 18.17: rvalueparams.cpp

```
#include <iostream>

int twice(const int& n) {
    std::cout << "Calling with reference parameter, result is ==> ";
    return 2 * n;
}

int twice(int&& n) {
    std::cout << "Calling with rvalue reference parameter, result is ==> ";
    return 2 * n;
}

int main() {
    int x = 6;
    std::cout << twice(x) << '\n';
    std::cout << twice(x + 2) << '\n';
    std::cout << twice(6) << '\n';
}
```

Listing 18.17 (rvalueparams.cpp) prints

```
Calling with reference parameter, result is ==> 12
Calling with rvalue reference parameter, result is ==> 16
Calling with rvalue reference parameter, result is ==> 12
```

The expression `twice(x)` calls the `twice` function with an lvalue, so it calls the first version of `twice`. The expressions `x + 2` and `2` are rvalues—neither can appear by itself on the left side of the assignment operator—so both `twice(x + 2)` and `twice(2)` invoke the second version of the `twice` function.

These examples with integers we have seen so far help us better understand the mechanics of rvalue references, but, honestly, rvalue references offer little value to the simple types like integers. The C++ language designers introduced rvalue references into C++ for the specific purpose of making copy construction and assignment from temporaries more efficient. Rvalue references are exactly what we need to make our linked list of integers more efficient when dealing with temporary lists.

Listing 18.18 (intlist5.h) declares the fifth generation of our integer linked list class. It contains a new constructor overload and adds an assignment operator overload, both of which accept rvalue references.

Listing 18.18: intlist5.h

```
// intlist5.h
```

```
class IntList5 {
    // The nested private Node class from before
    struct Node {
        int data;      // A data element of the list
        Node *next;    // The node that follows this one in the list
        Node(int d);   // Constructor
        ~Node();       // Destructor
    };

    Node *head;  // Points to the first item in the list
    Node *tail;  // Points to the last item in the list

    int len;     // Number the elements in the list

public:
    // The constructor makes an initially empty list
    IntList5();

    // The destructor that reclaims the list's memory
    ~IntList5();

    // Copy constructor
    IntList5(const IntList5& other);

    // Move constructor
    IntList5(IntList5&& other);

    // Assignment operator
    IntList5& operator=(const IntList5& other);

    // Move assignment operator
    IntList5& operator=(IntList5&& other);

    // Inserts n onto the back of the list.
    void insert(int n);

    // Prints the contents of the linked list of integers.
    void print() const;

    // Returns the length of the linked list.
    int length() const;

    // Removes all the elements in the linked list.
    void clear();
};
```

The additional constructor is known as a *move constructor*, and the additional assignment operator is known as the *move assignment* operator. To make the discourse clearer, we will refer to the original assignment operator as *copy assignment*. The compiler can disambiguate between the overloaded methods by virtue of the actual parameters passed by the caller:

- If the caller passes an lvalue, the compiler generates code that invokes the copy constructor or copy assignment operator.

- If the caller passes an rvalue, the compiler generates code that invokes the move constructor or move assignment operator.

The purpose of move construction is to efficiently move the resources held by the temporary object into the new object being created. We want to avoid making a copy of the temporary. Similarly, the purpose of move assignment is to efficiently move the resources held by the temporary object into an existing object without creating a new object. The copy constructor and copy assignment operator are designed to make copies.

It is safe to "steal" the resources of the temporary object because a temporary object is transient and cannot be used later within the program. Even though temporary objects are transient, they nonetheless are real, nameless objects. If the class of the temporary has a destructor, the object's destructor will execute when the temporary's ephemeral life is over. This means that even though the move constructor and move assignment operator may move a temporary's resources into another object, they must leave the temporary in a well-defined state so that the temporary's destructor can do its job.

The following implements the move constructor for the `IntList5` class:

```
// Move constructor takes possession of the temporary's list
IntList5::IntList5(IntList5&& temp): IntList5() {
    // Swap contents with the temporary
    std::swap(head, temp.head);
    std::swap(tail, temp.tail);
    std::swap(len, temp.len);
}
```

This move constructor delegates to the constructor that accepts no arguments. This initializes the `head` and `tail` pointers of this new object to `nullptr`. Note that this represents an empty, but valid linked list object. The move constructor then exchanges the `head` and `tail` pointers of the object it is creating with those of the temporary. This effectively moves the temporary's list of nodes to this object and moves the this object's empty, but valid, list of nodes to the temporary. When this copy constructor finishes, the temporary's life is over. Since the temporary still contains a valid list of nodes, its destructor can execute successfully. Observe that no where in this copy constructor did we traverse a list of nodes copying each element individually. This means move construction of a list with 1,000,000 nodes requires no more time than move constructing an empty list.

Note that the parameter to the move constructor is not declared `const`. If it were declared `const`, it would not be possible to swap its contents with new object being created.

The move assignment operator is similar to the move constructor, except assignment, of course, does not begin with an initially empty object. We must ensure our move assignment operator does not leak memory from any preexisting list of nodes the object may contain.

The implementation of the move constructor for the `IntList5` class looks very similar to the move constructor:

```
// Move assignment operator
IntList5& IntList5::operator=(IntList5&& temp) {
    // Exchange the head and tail pointers and len from this list
    // with those of the new, temporary list
    std::swap(head, temp.head);
    std::swap(tail, temp.tail);
    std::swap(len, temp.len);
    // The temporary list now points to this list's original contents,
```

```
    // and this list now points to the temporary's list
    // The temporary list will be destroyed since it is a temporary
    return *this;
}
```

The only difference is the move constructor always will given the empty list to the temporary. The move assignment operator exchange the object's list for the temporary's list. The object's list could be empty, but in general it will not be empty. When the move assignment operator returns, the temporary passed to it will cease to exist. The temporary's destructor then will properly dispose of the object's original list of nodes.

Listing 18.19 (intlist5.cpp) contains the complete implementation of the `IntList5` class.

Listing 18.19: `intlist5.cpp`

```
// intlist5.cpp

#include "intlist5.h"
#include <iostream>
#include <utility>

// Private IntList5 operations

// Node constructor
IntList5::Node::Node(int n): data(n), next(nullptr) {
    std::cout << "Creating node " << data
              << "  (" << reinterpret_cast<uintptr_t>(this) << ")\n";
}

IntList5::Node::~Node() {
    std::cout << "Destroying node " << data
              << "  (" << reinterpret_cast<uintptr_t>(this) << ")\n";
}

// The constructor makes an initially empty list.
// The list is empty when head and tail are null.
// The list's size initially is zero.
IntList5::IntList5(): head(nullptr), tail(nullptr), len(0) {}

// Copy constructor makes a copy of the other object's list
IntList5::IntList5(const IntList5& other): IntList5() {
    // Walk through other's list inserting each of its elements
    // into this list
    for (auto cursor = other.head; cursor; cursor = cursor->next)
        insert(cursor->data);
}

// Move constructor takes possession of the temporary's list
IntList5::IntList5(IntList5&& temp): IntList5() {
    // Swap contents with the temporary
    std::swap(head, temp.head);
    std::swap(tail, temp.tail);
    std::swap(len, temp.len);
}

// Assignment operator
```

```
IntList5& IntList5::operator=(const IntList5& other) {
    // Make a local, temporary copy of other
    IntList5 temp{other};
    // Exchange the head and tail pointers and len from this list
    // with those of the new, temporary list
    std::swap(head, temp.head);
    std::swap(tail, temp.tail);
    std::swap(len, temp.len);
    // The temporary list now points to this list's original contents,
    // and this list now points to the copy of other's list
    // The temporary list will be destroyed since it is a temporary
    return *this;
}

// Move assignment operator
IntList5& IntList5::operator=(IntList5&& temp) {
    // Exchange the head and tail pointers and len from this list
    // with those of the new, temporary list
    std::swap(head, temp.head);
    std::swap(tail, temp.tail);
    std::swap(len, temp.len);
    // The temporary list now points to this list's original contents,
    // and this list now points to the temporary's list
    // The temporary list will be destroyed since it is a temporary
    return *this;
}

// The destructor deallocates the memory held by the list
IntList5::~IntList5() { clear(); }

// Inserts n onto the back of the list.
// n is the element to insert.
void IntList5::insert(int n) {
    // Make a node for the new element n
    IntList5::Node *new_node = new Node(n);
    if (tail) {   // Is tail non-null?
       tail->next = new_node;  // Link the new node onto the back
       tail = new_node;        // The new node is the new tail of the list
    }
    else // List is empty, so make head and tail point to new node
        head = tail = new_node;
    len++;  // List now contains one more element
}

// Prints the contents of the linked list of integers.
void IntList5::print() const {
    for (auto cursor = head; cursor; cursor = cursor->next)
        std::cout << cursor->data << ' ';
    std::cout << '\n';
}

// Returns the length of the linked list.
int IntList5::length() const {
    return len;
}
```

```
// Removes all the elements in the linked list.
void IntList5::clear() {
    auto cursor = head;
    while (cursor) {
        auto temp = cursor;
        cursor = cursor->next;
        delete temp;
    }
    head = tail = nullptr;  // Null head signifies list is empty
    len = 0;
}
```

Listing 18.20 (templist2.cpp) is the same code as Listing 18.16 (templist.cpp), except it uses the newest iteration of our integer linked list class.

Listing 18.20: `templist2.cpp`

```
#include <iostream>
#include "intlist5.h"


IntList5 make_list(int n) {
    IntList5 result;
    for (int i = 0; i < n; i++)
        result.insert(i);
    return result;
}

int main() {
    auto my_list = make_list(5);
    my_list.print();
}
```

The output of Listing 18.20 (templist2.cpp) confirms that our added move constructor avoids creating an extra linked list object just to pass the temporary.

```
Creating node 0  (21169512)
Creating node 1  (21214272)
Creating node 2  (21214512)
Creating node 3  (21214304)
Creating node 4  (21214208)
0 1 2 3 4
Destroying node 4  (21214208)
Destroying node 3  (21214304)
Destroying node 2  (21214512)
Destroying node 1  (21214272)
Destroying node 0  (21169512)
```

The `make_list` function creates the temporary object; this accounts for all statements that begin with *Creating node*. . . . The move constructor exchanges the temporary object's list of nodes with the empty list found in the newly created `my_list` object. At this point `my_list` contains the temporary's original list and the temporary contains the empty list. The temporary's destructor executes but finds no nodes to destroy (`head` is null). At the end of the `main` function `my_list`'s destructor `delete`'s all the nodes in `my_list`.

As an aside, note that the compiler can detect that the `make_list` function creates a local object (`result`) but merely returns it to the caller. The compiler can use return value optimization (see Section 11.1.4) to avoid creating an extra object within the function to return to the caller. Instead, the compiler generates code that enables the `make_list` function to create the linked list object directly in the environment of the caller.

The addition of move construction and move assignment brings us to C++'s *Rule of Five*. The Rule of Five involves copy construction, move construction, destruction, copy assignment, and move assignment. If a programmer defines none of these special methods for a class, the compiler automatically will provide its own versions of each of them that work properly for classes that support value semantics; for example, all the classes in Chapter 16 support value semantics. Classes that use value semantics avoid the memory management and aliasing issues we encountered with our linked list classes.

If the programmer defines one or more of a copy constructor, destructor, or copy assignment operator, the compiler will not automatically supply a move constructor or move assignment operator. This is not a problem as far as program correctness is concerned; move construction and move assignment merely provide an optimization that avoids the unnecessary creation of an extra object when constructing or assigning from a rvalue (temporary). A missing move constructor means a temporary is copied by the copy constructor, and a missing move assignment operator means a temporary is copied by the copy assignment operator.

The programmer can precisely control which of these special methods the compiler supplies by explicitly marking these methods with a `default` or `delete` label. We will not go into the details here, but merely provide a few examples.

Consider class `X1`:

```
class X1 {
public:
    int data;
};
```

Class `X1` has no explicit constructor; therefore, the compiler will supply one that allows callers to create an `X1` object:

```
X1 my_x1;  // Okay; default constructed
std::cout << my_x1.data << '\n';  // Prints some random integer value
```

In this case a client needs to reassign `my_x1.data` for the object to be remotely useful.

Next, consider class `X2`:

```
class X2 {
public:
    int data;
    X2(int n): data(n) {}
};
```

Here the programmer provides an explicit constructor, so the compiler will not generate a default constructor. This mean the following statement is illegal:

```
X2 my_x2;  // Illegal; the client must provide an integer
```

If we really want a client to be able to create an object both with or without an integer argument, we can overload the constructor (provide our own default constructor) or force the compiler to generate one for us using the `default` label, as shown here:

```
class X3 {
public:
    int data;
    X3() = default;   // Allow compiler to generate default constructor
    X3(int n): data(n) {}  // and provide our own
};
```

This makes both of the following statements possible:

```
X3 my_x3a;
X3 my_x3b{4};
```

We can prevent the compiler from generating its default constructor by marking the default constructor it ordinarily would provide with the `delete` label, as in the following:

```
class X4 {
public:
    int data;
    X4() = delete;
};
```

Did you notice that we violated the Rule of Three in our `Node` nested struct that we used in our linked list examples? We added a destructor but did not supply a copy constructor and did not define a copy assignment operator. How can we justify this decision?

In reality, the `Node` struct does not need a destructor. The linked list class will operate flawlessly with a `Node` inner struct that has no destructor. No code within the `Node` struct allocates any resources. The outer linked list class is responsible for allocating and deallocating memory resources. The linked list methods manipulate `Node` objects as passive data. We added the destructor only so we could see exactly when the program destroyed a `Node` object. This enabled us to detect memory leaks and multiple deletions and refine our linked list class into a robust useful type. The `Node` destructor cleans up no resources nor does anything else that would impact the work the compiler-generated copy constructor or the compiler-generated copy assignment operator would do. Remember, you should follow the Rule of Three unless you have a very good reason not to do so. In our case, our good reason is our destructor does nothing that would prompt us to write our own copy constructor and copy assignment operator to properly coordinate with destructor.

In order to emphasize our intentional decision to violate the Rule of Three, we can rewrite the `Node` struct as

```
struct Node {
    int data;                   // A data element of the list
    Node *next;                 // The node that follows this one
    Node(int d);                            // Constructor
    Node(const Node&) = default;            // Copy constructor
    Node(Node&&) = default;                 // Move constructor
    ~Node();                                // Destructor
    Node& operator=(const Node&) = default; // Copy assignment
    Node& operator=(Node&&) = default;      // Move assignment
};
```

By explicitly defaulting the copy constructor and copy assignment operator, we indicate that we did not omit them by accident. The inclusion of the destructor prevents the compiler from automatically generating a move constructor and move assignment operator. By declaring these two methods as `default`, the compiler will generate them for us.

Since our `Node` object contains only simple types (an `int` and a pointer) we can omit all the explicit `default`ed methods in the declaration above, and nothing will be different. This is because C++ "moves" an integer merely by copying it. A simple integer has no resources we need to steal from a temporary; a simple assignment is all that is needed. Similarly, a pointer may refer to an elaborate object of some sort, but the pointer itself in simply an address. Copy and move construction is the same, and and copy and move assignment is handled the same as well.

18.6 Smart Pointers

C++ programmers that manually manage dynamic memory with `new` and `delete` follow a style inherited directly from the C programming language. Many modern programming languages like Python, Java, and C# manage dynamic memory through a technique called *garbage collection*. Garbage collection takes care of the accounting necessary to avoid multiple `delete`s and memory leaks. In garbage collected languages programmers need only call the equivalent of `new`; the garbage collector takes care of freeing up the space later when the executing program no longer uses the dynamically-allocated object. Garbage collection works well, but it does add some overhead to an executing program. This overhead consumes some extra memory and can affect a program's run-time efficiency. C++ strives to be as efficient as possible, so it does not provide an automatic garbage collector. Garbage collected languages typically allocate all objects on the heap, thereby managing objects in a uniform manner. C++ supports statically-allocated and stack-allocated objects, as well as heap-allocated objects. Heap allocation is sightly slower, and heap fragmentation can further degrade program performance. C++ follows the mantra *you only pay for what you use*, meaning programs that do not need the heap do not incur the run-time cost of using the heap.

C++ provides the efficiency benefits of a non-garbage-collected language with the convenience of garbage collection when desired. C++ achieves this in various ways. One way we have seen and have been enjoying its benefits for some time. The `std::vector` class manages a dynamic array, using `new` and `delete` behind the scenes. Software engineers implemented `std::vector` carefully to do the right thing, avoiding aliasing, memory leaks, and multiple deletions.

Modern C++ makes it possible to write programs that manage dynamic memory using neither `new` nor `delete` directly. Such a style uses *smart pointers*; that is, pointers that "know" exactly when to deallocate the memory they reference.

C++ smart pointers eliminate the need for such manual intervention on the part of the programmer. A smart pointer automatically `delete`s its associated memory at the proper time. The `std::shared_ptr` type is one example of a standard C++ smart pointer.

Suppose we have the following `Widget` type:

```
struct Widget {
    int data;
    Widget(int n): data(n) {}
};
```

The following code creates a `Widget` object on the heap and assigns it to a `std::shared_ptr` object named `p`:

```
std::shared_ptr<Widget> p(new Widget(12));
```

The generic `std::make_shared` function provides a more convenient way to make a `std::shared_ptr` object:

```
std::shared_ptr<Widget> p = std::make_shared<Widget>(12);
```

The `std::make_shared` function eliminates the need of the `new` operator. It uses `new` behind the scenes to allocate the necessary memory for the object it creates, but the programmer does not see the `new`. On the other end of the object's lifetime, programmers never call `delete` when using smart pointers; smart pointers automatically call `delete` at the proper time. The use of the `std::make_shared` function enables programmers to work with dynamically-allocated memory without ever using `new` and `delete` directly. This removes the possibility of memory leaks and multiple deletions that haunt manual memory management.

We can use the type inference capabilities of `auto` to express smart pointer creation even more simply:

```
auto p = std::make_shared<Widget>(12);
```

In the expression `std::make_shared<Widget>(12)` the name inside the angle brackets is the type of the object to create, and the value in the parentheses is the argument to pass to that type's constructor. The compiler knows that the function will return type `std::shared_ptr<Widget>`, so there is no need to spell out the exact type of p on the left side.

Once we have p defined in one of these ways we can treat it syntactically as if it were a raw pointer; for example:

```
// Make p point to a dynamically created a Widget object
auto p = std::make_shared<Widget>(12);
std::cout << p->data << '\n';    // Prints 12
p->data = 5;
std::cout << p->data << '\n';    // Prints 5
```

Listing 18.21 (simplesharedptrtest.cpp) provides a program that performs some simple tests with `std::shared_ptr` objects.

Listing 18.21: simplesharedptrtest.cpp

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

struct Widget {
    static unsigned id_source;  // Source of unique IDs
    unsigned id;
    Widget(): id(id_source++) {
        std::cout << "Creating a widget #" << id << " ("
                  << reinterpret_cast<uintptr_t>(this)
                  << ")\n";
    }
    ~Widget() {
        std::cout << "Destroying a widget #" << id << " ("
                  << reinterpret_cast<uintptr_t>(this)
                  << ")\n";
    }
};

unsigned Widget::id_source = 0;

// Global shared pointer
auto global_ptr = std::make_shared<Widget>();

std::shared_ptr<Widget> make_widget() {
```

```
    std::cout << "---- Entering make_widget ----\n";
    std::cout << "---- Leaving make_widget ----\n";
    return std::make_shared<Widget>();
}

void test1() {
    std::cout << "---- Entering Test 1 ----\n";
    // Make p point to a dynamically created a widget object
    auto p = std::make_shared<Widget>();
    std::cout << p->id << '\n';
    p->id = 25;
    std::cout << p->id << '\n';
    std::cout << "---- Leaving Test 1 ----\n";
}

void test2() {
    std::cout << "---- Entering Test 2 ----\n";
    // Make q point to a dynamically created a widget object
    auto q = std::make_shared<Widget>();
    std::cout << q->id << '\n';
    q = nullptr;  // Make q point to nothing
    std::cout << "---- Leaving Test 2 ----\n";
}

void test3() {
    std::cout << "---- Entering Test 3 ----\n";
    // Make p point to a dynamically created integer
    auto p = std::make_shared<int>(55);
    std::cout << *p << '\n';        // Prints 55
    *p = -4;                        // Reassign
    std::cout << *p << '\n';        // Prints -4
    std::cout << "---- Leaving Test 3 ----\n";
}

void test4() {
    std::cout << "---- Entering Test 4 ----\n";
    static auto p = make_widget();
    std::cout << p->id << '\n';
    std::cout << "---- Leaving Test 4 ----\n";
}

void test5() {
    std::cout << "---- Entering Test 5 ----\n";
    auto p = make_widget();
    std::cout << p->id << '\n';
    std::cout << "---- Leaving Test 5 ----\n";
}

void test6() {
    std::cout << "---- Entering Test 6 ----\n";
    // Aliasing
    auto q = std::make_shared<Widget>();
    auto r = q;     // r aliases q, no new memory allocated
    auto s = q;     // s aliases q, no new memory allocated
    std::cout << q->id << ' '
```

```
              << r->id << ' '
              << s->id << '\n';
    q = nullptr;
    std::cout << r->id << ' '
              << s->id << '\n';
    r = nullptr;
    std::cout << s->id << '\n';
    s = nullptr;     // Deallocates the widget object
    std::cout << "---- Leaving Test 6 ----\n";
}


int main() {
    std::cout << "---- Entering main ----\n";
    test1();
    test2();
    test3();
    test4();
    test5();
    test6();
    std::cout << "---- Leaving main ----\n";
}
```

The output of Listing 18.21 (simplesharedptrtest.cpp) shows chronologically when the program creates and destroys `Widget` objects:

```
Creating a widget #0 (18566044)
---- Entering main ----
---- Entering Test 1 ----
Creating a widget #1 (18566092)
1
25
---- Leaving Test 1 ----
Destroying a widget #25 (18566092)
---- Entering Test 2 ----
Creating a widget #2 (18566380)
2
Destroying a widget #2 (18566380)
---- Leaving Test 2 ----
---- Entering Test 3 ----
55
-4
---- Leaving Test 3 ----
---- Entering Test 4 ----
---- Entering make_widget ----
---- Leaving make_widget ----
Creating a widget #3 (18566404)
3
---- Leaving Test 4 ----
---- Entering Test 5 ----
---- Entering make_widget ----
---- Leaving make_widget ----
Creating a widget #4 (18566092)
4
---- Leaving Test 5 ----
```

```
Destroying a widget #4 (18566092)
---- Entering Test 6 ----
Creating a widget #5 (18566092)
5 5 5
5 5
5
Destroying a widget #5 (18566092)
---- Leaving Test 6 ----
---- Leaving main ----
Destroying a widget #3 (18566404)
Destroying a widget #0 (18566044)
```

The `Widget` constructor in Listing 18.21 (simplesharedptrtest.cpp) ensures that each widget object it creates has a unique `id` number. The first widget will have `id` 0, the second has `id` 1, etc.

The executing program initializes global objects before calling the `main` function. The first two lines of the program's output:

```
Creating a widget #0 (18566044)
---- Entering main ----
```

shows the creation of the global object `global_ptr`. The next few lines:

```
---- Entering Test 1 ----
Creating a widget #1 (18566092)
1
25
---- Leaving Test 1 ----
Destroying a widget #25 (18566092)
```

show the creation of the local widget #1 in the `test1` function. The code in `test1` modifies the `id` value of the `Widget` object. (Recall that clients can freely modify fields of a `struct`.) At the end of the function's execution the `std::shared_ptr` object automatically destroys widget #1, now known as widget #25, as we can see the destructor's output.

It is important to note that `p` is a local variable in `test1`, so the space for `std::shared_ptr p` itself appears on the stack. The call to `std::make_shared` creates via `new` a widget object on the heap. When `p` goes out of scope at the end of `test1`'s execution, `p`'s desructor automatically calls `delete` to deallocate the widget from the heap. The usual function return mechanism automatically removes `p` from the stack.

The next few lines of output:

```
---- Entering Test 2 ----
Creating a widget #2 (18566380)
2
Destroying a widget #2 (18566380)
---- Leaving Test 2 ----
```

shows that when a `std::shared_ptr` previously assigned to a widget is reassigned, in this case to `nullptr`, the space for the original widget object is freed up (its destructor called) immediately; the object is cleaned up before reaching the end of the function.

In the next few lines of output:

```
---- Entering Test 3 ----
55
-4
---- Leaving Test 3 ----
```

the `test3` function does not reveal any clues about the automatic memory management of `std:shared_ptr`, as the `int` type has no constructors or destructor that indicate the lifetime of the dynamically allocated `int` value. The program includes the `test3` function simply to demonstrate that `std::shared_ptr` objects can manage the dynamic memory of simple types just as well as fancier types. The simple `int` type does not report its destruction when its memory is freed, but rest assured that the `std::shared_ptr` object will deallocate its memory correctly.

In the next few lines of output:

```
---- Entering Test 4 ----
---- Entering make_widget ----
---- Leaving make_widget ----
Creating a widget #3 (18566404)
3
---- Leaving Test 4 ----
```

we see the creation of widget #3, but we do not see its destruction. This is because it is the first invocation of `test4`. The first call initializes the `static` local `p`, hence the construction of a `std::shared_ptr` and its associated `Widget` object. Since `p` is a `static` local variable it must survive between calls to `test4`; therefore, `p` cannot be deallocated until the program finishes executing (see below).

In the next few lines of output:

```
---- Entering Test 5 ----
---- Entering make_widget ----
---- Leaving make_widget ----
Creating a widget #4 (18566092)
4
---- Leaving Test 5 ----
Destroying a widget #4 (18566092)
```

we see the exact situation as in `test4`, except that `p` is not `static`. This means `p` destroys its widget object when the function returns.

In the output of the final test function, `test6`, we see

```
---- Entering Test 6 ----
Creating a widget #5 (18566092)
5 5 5
5 5
5
Destroying a widget #5 (18566092)
---- Leaving Test 6 ----
```

The `test6` function allocates just one `Widget` object. It assigns three `std::shared_ptr` objects to the same widget object. This provides a clear justification for the name *shared_ptr*—all three pointers share access to the same object. If `p` and `q` are `std::shared_ptrs` of the same type, the statement

```
q = p;
```

make p and q aliases to the same object. Regardless of the aliasing, the code within test6 allocates exactly one Widget object and deallocates that same Widget object when the function completes its execution.

The final lines in the program's output:

```
---- Leaving main ----
Destroying a widget #3 (18566404)
Destroying a widget #0 (18566044)
```

show the destruction of test4's static local std::shared_ptr (widget #3) and global_ptr (widget #0).

If you analyze the output of Listing 18.21 (simplesharedptrtest.cpp) carefully, you will see that every constructor call has an associated destructor call for the same object (remember test1 renamed widget #1 to #25, so compare widget #1's creation address with widget #25's destruction address, and you will see they match). The heap provided all the space for Widget objects in Listing 18.21 (simplesharedptrtest.cpp), yet we see neither new nor delete anywhere in the code.

This ability to deallocate its referenced memory at the proper time is what makes a std::shared_ptr object a smart pointer.

Recall Listing 18.1 (howtodelete.cpp) that demonstrated the difficulty of managing dynamic memory manually. Listing 18.22 (smartdelete.cpp) converts all the raw pointers of Listing 18.1 (howtodelete.cpp) into std::shared_ptrs. The result? We no longer need to worry about who "owns" the memory of the heap allocated objects; std::shared_ptr objects manage all those details.

Listing 18.22: smartdelete.cpp

```
#include <iostream>
#include <vector>
#include <memory>

struct Widget {
    int value;
    Widget(int value): value(value) {
        std::cout << "Creating widget " << value << '\n';
    }
    ~Widget() {
        std::cout << "Destroying widget " << value << '\n';
    }
};

std::shared_ptr<Widget> get_widget() {
    static int pos = 0;
    static std::vector<std::shared_ptr<Widget>> widget_pool {
            std::make_shared<Widget>(23), std::make_shared<Widget>(45),
            std::make_shared<Widget>(16), std::make_shared<Widget>(12),
            std::make_shared<Widget>(3), std::make_shared<Widget>(20),
            std::make_shared<Widget>(10)};
    pos = (pos + 1) % widget_pool.size();
    return widget_pool[pos];
}

void process(int n) {
    std::vector<std::shared_ptr<Widget>> vec;
```

```
    while (n-- > 0)
        vec.push_back(get_widget());
    // No need to clean up vector; smart pointers do it for us
}

int main() {
    std::cout << "Entering main\n";
    process(10);
    std::cout << "Leaving main\n";
}
```

The output of Listing 18.22 (smartdelete.cpp) shows that the program destroys each `Widget` object it creates exactly once.

```
Entering main
Creating widget 23
Creating widget 45
Creating widget 16
Creating widget 12
Creating widget 3
Creating widget 20
Creating widget 10
Leaving main
Destroying widget 23
Destroying widget 45
Destroying widget 16
Destroying widget 12
Destroying widget 3
Destroying widget 20
Destroying widget 10
```

How does a `std::shared_ptr` smart pointer "know" when to call `delete` to free up its object? memory it manages by *reference counting*.

Given the following `Widget` type:

```
struct Widget {
    int value;
    Widget(int value): value(value) {}
};
```

the following statement:

```
auto p = std::make_shared<Widget>(3);
```

creates a smart pointer referencing a `Widget` object containing the value 3. What happens to the object associated with `p` when we reassign the variable `p`? The following statement:

```
//  p is the Widget object created above
p = nullptr;
```

redirects `p` to point to nothing. At this point *no* variable references the `Widget` object created earlier. This means the object effectively is cut off from the remainder of the program's execution. This abandoned object is classified as *garbage*. The term *garbage* is a technical term used in computer science that refers to memory allocated by an executing program that the program no longer can access. The `std::shared_ptr`

type uses a technique known as *reference counting* to automatically reclaim the space occupied by abandoned objects.

Reference counting garbage collection works as follows. All objects dynamically created via the `std::shared_ptr` constructor or the `std::make_shared` convenience function have an associated *reference count*. When the executing program creates a new object managed by a `std::shared_ptr` object, it sets the shared pointer object's reference count to 1. After executing the following statement:

```
auto p = std::make_shared<Widget>(3);
```

the reference count of `p` is 1. A reference count of 1 means that exactly one shared pointer object (in this case `p`) is assigned to the object. Making an alias, as in

```
q = p;
```

increments `p`'s reference count by one. Both `p` and `q` share the same object and share the same reference count. If we make another alias, as in

```
r = p;
```

the assignment statement increases the object's reference count to 3. If reassign `p`, `q`, or `r`, as in

```
q = std::make_shared<Widget>(22);
```

the reference count of the widget containing 3 decreases by one, and the reference count of the new widget object containing 22 becomes 1. If we then reassign `p`, for example:

```
p = nullptr;
```

this leaves only variable `r` referencing the widget containing 3, its reference count is 1. If we finally reassign `r`:

```
r = nullptr;
```

the reference count of the widget containing 3 drops to zero. The `std::shared_ptr` class has a custom assignment operator. The process of assignment alters the reference count for objects managed via these smart pointers. When an assignment would cause a shared pointer's reference count for an object to become zero, the shared pointer can `delete` that object before attaching to the object being assigned. The assignment would increase the reference count of the object being assigned by one.

The `std::shared_ptr` class has a method named `use_count` that returns the reference count for the current object managed by the shared pointer. Listing 18.23 (refcount.cpp) exploits this `use_count` method to demonstrate how reference counting garbage collection works.

Listing 18.23: `refcount.cpp`

```
#include <iostream>
#include <memory>

class Widget {
public:
    Widget() {
        std::cout << "Creating a widget ("
                  << reinterpret_cast<uintptr_t>(this)
                  << ")\n";
    }
    ~Widget() {
```

```
        std::cout << "Destroying a widget ("
                  << reinterpret_cast<uintptr_t>(this)
                  << ")\n";
    }
};

int main() {
    auto p = std::make_shared<Widget>();
    std::cout << p.use_count() << '\n';    // Prints 1
    auto q = p;
    std::cout << p.use_count() << '\n';    // Prints 2
    auto r = p;
    std::cout << p.use_count() << '\n';    // Prints 3
    r = nullptr;
    std::cout << p.use_count() << '\n';    // Prints 2
    q = nullptr;
    std::cout << p.use_count() << '\n';    // Prints 1
    p = nullptr;
    std::cout << p.use_count() << '\n';    // Prints 0
}
```

Listing 18.23 (refcount.cpp) prints

```
Creating a widget (20338980)
1
2
3
2
1
Destroying a widget (20338980)
0
```

Observe how the object's transition from a reference count of one to a reference count of zero automatically `delete`s the widget object, invoking its destructor.

In Listing 18.24 (intlist6.h) we retool our linked list class from Listing 18.18 (intlist5.h) to use `std::shared_ptr` objects instead of raw pointers.

Listing 18.24: `intlist6.h`

```
// intlist6.h

#include <memory>  // For std::shared_ptr

class IntList6 {
    // The nested private Node class from before
    struct Node {
        int data;                    // A data element of the list
        std::shared_ptr<Node> next; // The node that follows this one
        Node(int d);                              // Constructor
        Node(const Node&) = default;              // Copy constructor
        Node(Node&&) = default;                   // Move constructor
        ~Node();                                  // Destructor
        Node& operator=(const Node&) = default; // Copy assignment
        Node& operator=(Node&&) = default;      // Move assignment
```

```
    };


    std::shared_ptr<Node> head;  // Points to the first item in the list
    std::shared_ptr<Node> tail;  // Points to the last item in the list

    int len;                     // Number of elements in the list

public:
    // The constructor makes an initially empty list
    IntList6();

    // The destructor that reclaims the list's memory
    ~IntList6();

    // Copy constructor
    IntList6(const IntList6& other);

    // Move constructor
    IntList6(IntList6&& other);

    // Assignment operator
    IntList6& operator=(const IntList6& other);

    // Move assignment operator
    IntList6& operator=(IntList6&& other);

    // Inserts n onto the back of the list.
    void insert(int n);

    // Prints the contents of the linked list of integers.
    void print() const;

    // Returns the length of the linked list.
    int length() const;

    // Removes all the elements in the linked list.
    void clear();
};
```

Listing 18.25 (intlist6.cpp) implements methods declared in Listing 18.24 (intlist6.h).

Listing 18.25: `intlist6.cpp`

```
// intlist6.cpp

#include "intlist6.h"
#include <iostream>
#include <utility>

// Private IntList6 operations

// Node constructor
IntList6::Node::Node(int n): data(n), next(nullptr) {
    std::cout << "Creating node " << data
```

```
              << "  (" << reinterpret_cast<uintptr_t>(this) << ")\n";
}

IntList6::Node::~Node() {
    std::cout << "Destroying node " << data
              << "  (" << reinterpret_cast<uintptr_t>(this) << ")\n";
}

// The constructor makes an initially empty list.
// The list is empty when head and tail are null.
IntList6::IntList6(): head(nullptr), tail(nullptr), len(0) {}

// Copy constructor makes a copy of the other object's list
IntList6::IntList6(const IntList6& other): IntList6() {
    // Walk through other's list inserting each of its elements
    // into this list
    for (auto cursor = other.head; cursor; cursor = cursor->next)
        insert(cursor->data);
}

// Move constructor takes possession of the temporary's list
IntList6::IntList6(IntList6&& temp): IntList6() {
    // Swap contents with the temporary
    std::swap(head, temp.head);
    std::swap(tail, temp.tail);
    std::swap(len, temp.len);
}

// Assignment operator
IntList6& IntList6::operator=(const IntList6& other) {
    // Make a local, temporary copy of other
    IntList6 temp{other};
    // Exchange the head and tail pointers and len from this list
    // with those of the new, temporary list
    std::swap(head, temp.head);
    std::swap(tail, temp.tail);
    std::swap(len, temp.len);
    // The temporary list now points to this list's original contents,
    // and this list now points to the copy of other's list
    // The temporary list will be destroyed since it is a temporary
    return *this;
}

// Move assignment operator
IntList6& IntList6::operator=(IntList6&& temp) {
    // Exchange the head and tail pointers and len from this list
    // with those of the new, temporary list
    std::swap(head, temp.head);
    std::swap(tail, temp.tail);
    std::swap(len, temp.len);
    // The temporary list now points to this list's original contents,
    // and this list now points to the temporary's list
    // The temporary list will be destroyed since it is a temporary
    return *this;
}
```

```
// The destructor deallocates the memory held by the list
IntList6::~IntList6() { clear(); }

// Inserts n onto the back of the list.
// n is the element to insert.
void IntList6::insert(int n) {
    // Make a node for the new element n
    std::shared_ptr<IntList6::Node> new_node = std::make_shared<Node>(n);
    if (tail) {   // Is tail non-null?
       tail->next = new_node;  // Link the new node onto the back
       tail = new_node;        // The new node is the new tail of the list
    }
    else // List is empty, so make head and tail point to new node
        head = tail = new_node;
    len++;
}

// Prints the contents of the linked list of integers.
void IntList6::print() const {
    for (auto cursor = head; cursor; cursor = cursor->next)
        std::cout << cursor->data << ' ';
    std::cout << '\n';
}

// Returns the length of the linked list.
int IntList6::length() const {
    return len;
}

// Removes all the elements in the linked list.
void IntList6::clear() {
    auto cursor = head;
    while (cursor) {
        auto temp = cursor;     // Remember where we are
        cursor = cursor->next;  // Move next node
        temp->next = nullptr;   // Sever link from previous node
    }
    head = tail = nullptr;  // Null head signifies list is empty
    len = 0;
}
```

Nowhere in the `IntList6` class can we see the `new` or `delete` operators. The `std::make_shared` function and `std::shared_ptr` class hides those details from us.

We could have expressed the code within the `clear` method as simply as the following:

```
void IntList6::clear() {
    head = tail = nullptr;  // Null head signifies list is empty
    len = 0;
}
```

Note that it uses no loop to iterate over the nodes of the list. When we set `head`, which points to the first element in the list, to `nullptr`, the reassignment automatically will destroy the first node on the list. This is because no other smart pointers currently point to that node. The destruction of the first node will destroy

its next pointer that points to the second node on the list. This results in the destruction of the second node. The destruction of the second node destroys its next pointer which leads to the distruction of the third node, followed by the fourth node, etc. This cascading process ultimately destroys all the nodes in the list to which head points.

If this simple implementation of clear achieves the same result as the more involved clear method in Listing 18.25 (intlist6.cpp) that uses a loop, why even consider the more complicated code? As it turns out, the simpler method has a problem with very long lists. To see how, suppose we have a long linked list. When the clear method sets the std::shared_ptr head to null, no other smart pointers will be pointing to the first node. The smart pointer reassignment in this case calls a method behind the scenes to destroy the object to which it points (the object to which head points; that is, the first node in the list). This clean-up method does not return until the process is complete, and, for long lists, this process can be quite lengthy. The destruction of the first node requires the destruction of its next pointer, head->next. This head->next pointer points to the second node in the list. The destruction of this head->next pointer triggers the destruction of the second node because no other std::shared_ptr objects point to it. This means that before head's clean-up method returns, the chain of events it sets off must invoke the clean-up method of head->next which leads to the destruction of the second node. The destruction of the second node necessarily must destroy the next smart pointer of the second node (head->next->next). This in turn leads to the destruction of the object to which head->next->next points (that is, the third node in the list). By this same process, the destruction of the third node in the list leads to the destruction of the fourth node, and then the fifth node, etc. This cascade of method calls continue until it reaches the last node in the list. Two smart pointers point to the last node: the next pointer of the next-to-the-last node in the list and the tail pointer of the LinkedList object itself. The destruction of the next-to-the-last node removes one of the smart pointers but not the other. This means the clean-up method of the next-to-the-last node cannot destroy the last node. This turns out not to be a memory leak, however, as the clear method also sets tail to nullptr, so eventually the last node will be destroyed properly.

Everything appears to work perfectly—automatic resource management as advertised—so what is the problem for very long lists? We left the activity of the LinkedList::clear method as the destruction of the next-to-the-next-to-last smart pointer's clean-up method was destroying the next-to-the-last node in the list. The problem is this: The destruction of the first node of the list is not complete until the destruction of the second node is complete. The destruction of the second node is not complete until the destruction of the third node is complete, and so forth. This means the clean-up method destroying the first node does not do its job, return, and then clean up the node's next pointer; it does not return until the second completes its clean up, which, of course, does not complete until the third node's clean up is finished, etc. Theoretically, this chain of function calls eventually returns back to the action that initiated it—the reassignment of head to nullptr, and for lists that are not too long, this is the case. Each method invocation consumes some stack space, and for very long lists this call chain can be so deep so as to overflow the stack, causing the program to crash. The scenario is similar to a recursive function that recurses too deeply. On one system the program crashed when attempting to destroy a linked list containing 10,000 nodes.

The version of clear in Listing 18.25 (intlist6.cpp) iterates through the list setting the next pointers of the nodes it most recently visited to nullptr. On the very next iteration the temporary pointer temp destroys the node during its reassignment from cursor. This means it must destroy temp's next smart pointer, but the previous iteration set this to null. This eliminates the chaining problem and allows the previous node to be destroyed with a single function call that returns immediately. This version deallocates the list through a series of separate, isolated function calls that return immediately rather than a chain of function invocations that potentially can overflow the stack.

Listing 18.26 (smartlist.cpp) exercises our IntList6 class, verifying that the smart pointers properly manage the dynamic memory.

Listing 18.26: smartlist.cpp

```
// smartlist.cpp

#include <iostream>
#include "intlist6.h"

void test() {
    IntList6 list;
    for (int i = 0; i < 10; i++)
        list.insert(i);
    list.print();
}

int main() {
    for (int i = 0; i < 10; i++)
        test();
}
```

When you run Listing 18.26 (smartlist.cpp) you will see that all `Node` object creations have corresponding destructions. The `std::shared_ptr` objects take care of all the heap management.

Our linked list class has an `insert` method that allows clients to add elements to the list, but it has no method that enables a client to remove a single element. In our most recent non-smart-pointer class, `IntList5`, we could add the following method:

```
// Removes the first occurrence of n from the list.
// Returns true if successful (found n and removed it).
// Returns false if n is not originally present in the list.
bool IntList5::remove(int n) {
    Node *cursor = head,    // Start at head of list
         *prev = head;   // Keep track of previous node seen
    // Loop until we run off the end of the list or find n,
    // whichever comes first
    while (cursor && cursor->data != n) {
        prev = cursor;              // Remember previous node
        cursor = cursor->next;      // Move to next node
    }
    if (!cursor)                    // Did we run off the end of the list?
        return false;               // Indicate we did not find n

    // Found n; cursor is pointing at the node containing n
    if (head == tail)               // n was the only element in the list
        head = tail = nullptr;      // cursor still points to node with n
    else if (cursor == head)        // Is n the first element in the list?
        head = head->next;          // Redirect head around n
    else                            // n is not the first element
        prev->next = cursor->next;  // Redirect previous node around n
    if (cursor == tail)             // Was n the last element in the list?
        tail = prev;                // Update tail to new last element

    delete cursor;                  // Delete n's node
    len--;                          // List size decreases by 1
```

```
    return true;                    // We found n and deleted its node
}
```

The `remove` method is more complicated than the `insert` method because the `insert` method always adds new elements to the back end of the list. The `remove` method must be able to remove elements from anywhere in the list—front, back, and anywhere in between.

The `remove` method for our `IntList6` smart list class would look like the following:

```
// Removes the first occurrence of n from the list.
// Returns true if successful (found n and removed it).
// Returns false if n is not originally present in the list.
bool IntList6::remove(int n) {
    auto cursor = head,    // Start at head of list
         prev =  head;     // Keep track of previous node seen
    // Loop until we run off the end of the list or find n,
    // whichever comes first
    while (cursor && cursor->data != n) {
        prev = cursor;              // Remember previous node
        cursor = cursor->next;      // Move to next node
    }
    if (!cursor)                    // Did we run off the end of the list?
        return false;               // Indicate we did not find n

    // Found n; cursor is pointing at the node containing n
    if (head == tail)               // n was the only element in the list
        head = tail = nullptr;      // cursor still points to node with n
    else if (cursor == head)        // Is n the first element in the list?
        head = head->next;          // Redirect head around n
    else                            // n is not the first element
        prev->next = cursor->next;  // Redirect previous node around n
    if (cursor == tail)             // Was n the last element in the list?
        tail = prev;                // Update tail to new last element

    // No delete is necessary
    len--;                          // List size decreases by 1
    return true;                    // We found n and deleted its node
}
```

The `IntList6::remove` method still needs to perform all the checks and properly redirect the pointers within the list, but it no longer needs to call `delete` explicitly.

Chapter 19

Generic Programming

CAUTION! CHAPTER UNDER CONSTRUCTION

The flexible sorting examples in Listing 12.2 (flexibleintsort.cpp), Listing 16.5 (loggingflexiblesort.cpp), and Listing 17.35 (polymorphicsort.cpp) allow us to arrange the elements in integer vectors in creative ways and perform other interesting activities such as logging. Those examples demonstrate the power of function pointers, inheritance, and polymorphism. As flexible and powerful as these techniques are, they all contain one major limitation: they deal with vectors of *integers* only. Consider the `Comparer` class from Listing 17.33 (comparer.h) and Listing 17.34 (comparer.cpp), and the "flexible" `selection_sort` function from Listing 17.38 (loggingsort.cpp). The `selection_sort` function accepts a vector of integers and a `Comparer` reference. What if we need to sort a vector of double-precision floating-point numbers or a vector of `std::string` objects? Unfortunately, all of this "flexible" code cannot handle this seemingly minor variation.

In this chapter we look at C++'s template mechanism that enables programmers to develop truly generic algorithms and data structures. We will see how the standard C++ library has embraced the template technology to provide a wealth of generic algorithms and data structures that greatly assist developers in the construction of quality software.

19.1 Function Templates

Consider the following comparison function found in Listings Listing 12.2 (flexibleintsort.cpp), Listing 16.5 (loggingflexiblesort.cpp), and Listing 17.38 (loggingsort.cpp):

```
/*
 *  less_than(a, b)
 *     Returns true if a < b; otherwise, returns
 *     false.
 */
bool less_than(int a, int b) {
    return a < b;
}
```

Listing 19.1 (testlessthan.cpp) tests the `less_than` function with various arguments.

Listing 19.1: testlessthan.cpp

```
#include <iostream>

/*
 *  less_than(a, b)
 *     Returns true if a < b; otherwise, returns
 *     false.
 */
bool less_than(int a, int b) {
    return a < b;
}

int main() {
    std::cout << less_than(2, 3) << '\n';
    std::cout << less_than(2.2, 2.7) << '\n';
    std::cout << less_than(2.7, 2.2) << '\n';
}
```

The compiler generates warnings for the last two statements. The `less_than` function expects two integer arguments, but the calling code sends two double-precision floating-point values. The automatic conversion from `double` to `int` can lose information, hence the warnings. The output of Listing 19.1 (testlessthan.cpp) shows that we should take these warnings seriously:

```
1
0
0
```

Obviously $2 < 3$ and $2.7 \not< 2.2$, but $2.2 \not< 2.7$? The automatic `double` to `int` conversion truncates, so the `less_than` function treats both 2.2 and 2.7 as 2, and clearly $2 \not< 2$.

The situation is even worse for the following code:

```
std::string word1 = "ABC", word2 = "XYZ";
std::cout << (word1 < word2) << '\n';
std::cout << (word2 < word1) << '\n';
// The next statement will not compile
//std::cout << less_than(word1, word2) << '\n';
```

The C++ standard library provides a global `operator<` function that compares two `std::string` objects lexicographically. The first two `std::cout` statements produce the expected output:

```
1
0
```

but the last line that is commented out will not compile if we remove the commenting symbols. This is because the C++ library does not provide a standard conversion from `std::string` to `int`, and `less_than` requires two `int` parameters.

As another example, consider the following function that computes the sum of the elements in a vector of integers:

```
int sum(const std::vector<int>& v) {
    int result = 0;
    for (int elem : v)
```

```
        result += elem;
    return result;
}
```

The following client code works well:

```
std::vector<int> v {10, 20, 30};
std::cout << sum(v) << '\n';
```

It prints 60. The following code does not compile:

```
std::vector<double> v {10, 20, 30};
std::cout << sum(v) << '\n';
```

The second code fragment attempts to pass a vector of double-precision floating-point values to the `sum` function. Unfortunately, the `sum` function accepts only vectors that contain integers. More precisely, `sum` accepts only arguments of type `std::vector<int>`, and a `std::vector<double>` object is not a `std::vector<int>`. The solution is easy; just copy and paste the original `sum` function and change all the occurrences of "`int`" to "`double`," creating an overloaded `sum` function:

```
double sum(const std::vector<double>& v) {
    double result = 0;
    for (double elem : v)
        result += elem;
    return result;
}
```

This works, but the duplicated effort is unsatisfying. These two overloaded `sum` functions are identical except for the types involved. The actions of the two functions—the initialization, vector traversal and arithmetic—are essentially the same. In general, code duplication is undesirable. A programmer that discovers and repairs an error in one of the functions must remember to apply the corresponding correction to the function's overloaded counterpart.

Both the `less_than` and `sum` functions share a common shortcoming: they are tied in some way to a particular data type. It would be convenient to be able to specify the common pattern and let the compiler fill in the types as required. In the case of `less_than` we would like to be able to write a function that can use the `<` operator on parameters of any type for which `<` is compatible. For `sum`, we would like to create a generic function that works for vectors containing any numeric type.

C++ enables programmers to write such generic functions via *templates*. A function template specifies a pattern of code, and either the programmer or the compiler supplies the exact type as needed. Listing 19.2 (templatelessthan.cpp) uses C++'s template mechanism to create a generic `less_than` function.

Listing 19.2: `templatelessthan.cpp`

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

/*
 *  less_than(a, b)
 *     Returns true if a < b; otherwise, returns
 *     false.
 */
template <typename T>
bool less_than(T a, T b) {
```

```
    return a < b;
}

int main() {
    std::cout << less_than(2, 3) << '\n';
    std::cout << less_than(2.2, 2.7) << '\n';
    std::cout << less_than(2.7, 2.2) << '\n';

    std::string word1 = "ABC", word2 = "XYZ";
    std::cout << less_than(word1, word2) << '\n';
    std::cout << less_than(word2, word1) << '\n';
}
```

The output of Listing 19.2 (templatelessthan.cpp) is

```
1
1
0
1
0
```

In the `less_than` function definition:

```
template <typename T>
bool less_than(T a, T b) {
    return a < b;
}
```

`template` and `typename` are reserved words, and `T` is a *type parameter*. The `template` keyword indicates that the function definition that follows is not a normal function definition but rather is a pattern or template from which the compiler can attempt to produce the correct function definition. A function template is also known as a *generic function*. The `typename` keyword indicates that the identifier that follows (in this case `T`) is a placeholder for a C++ type name. The type parameter `T` stands for an actual type to be determined elsewhere. `T` is an identifier and so can have any name legal for a variable, function, or class; some programmers prefer more descriptive names such as `CompareType` or `TypeParam`. Single capital letters such as `T`, `S`, `U`, and `V` are popular type parameter names among C++ programmers. Just as a regular parameter of a function or method represents a value, a template parameter represents a type.

You may use the reserved word `class` interchangeably with the keyword `typename` in this context:

```
template <class T>
bool less_than(T a, T b) {
    return a < b;
}
```

In Listing 19.2 (templatelessthan.cpp) the compiler generates three different `less_than` functions: one that accepts two integers, one that accepts two double-precision floating-point values, and one the accepts two string objects. Although the programmer cannot see them in the source code, the compiler in effect creates the following literal function definitions:

```
bool less_than(int a, int b) {
    return a < b;
}
```

```
bool less_than(double a, double b) {
    return a < b;
}

bool less_than(std::string a, std::string b) {
    return a < b;
}
```

When the compiler generates an actual function definition from a template function, the process is called *template instantiation*, not to be confused with an object instantiation from a class.

The compiler instantiates functions only as needed. If Listing 19.2 (templatelessthan.cpp) did not contain the `std::string` objects, when processing the source code the compiler would not instantiate the version of `less_than` that accepts two `std::string` objects.

For efficiency purposes, the better way to write `less_than` is

```
template <typename T>
bool less_than(const T& a, const T& b) {
    return a < b;
}
```

Notice that parameters a and b are passed by reference, not by value. This means the caller passes only the addresses of objects a and b and does not need to make copies to send to the function. This can make a big difference if T represents large objects. Since the parameters are `const` references, the function cannot change the state of the caller's actual parameters; therefore, the function offers call-by-value safety without the overhead of copying the parameters.

If a caller attempts a statement such as

```
std::cout << less_than(2, 2.2) << '\n';
```

the compiler generates an error. This is because the template definition specifies that its two type parameters are identical. The literal 2 is an `int`, and the literal `2.2.` is a `double`. The compiler can automatically instantiate `less_than` functions that accept two `int`s or two `double`s but not mixed types. In the expression

```
less_than(2, 2.2)
```

the compiler cannot know which instantiated version of the template to call.

One solution requires the programmer to explicitly instantiate the function a shown here:

```
std::cout << less_than<int>(2, 2.2) << '\n';
```

This forces the compiler to instantiate and call the integer version. The compiler will issue a warning about truncating the 2.2. The statement

```
std::cout << less_than<double>(2, 2.2) << '\n';
```

would use the double-precision floating-point version of `less_than`, and it compiles cleanly with no warnings since widening the `int` 2 to a `double` is automatic.

We can rewrite the `less_than` template function to avoid the need for explicit instantiation. For the most flexibility, we can express the `less_than` template function as

```
template <typename T, typename V>
bool less_than(const T& a, const V& b) {
    return a < b;
}
```

This version uses two type parameters, `T` and `V`, and they can represent different types. With this new definition all of the following statements are acceptable:

```
std::cout << less_than(2, 3) << '\n';
std::cout << less_than(2.2, 2.7) << '\n';
std::cout << less_than(2, 2.2) << '\n';
std::cout << less_than(2.2, 2) << '\n';
```

No explicit template instantiation is necessary. Given this calling code, the compiler will automatically instantiate four overloaded `less_than` functions:

- `less_than(2, 3)` instantiates `bool less_than(const int& a, const int& b)`
- `less_than(2.2, 2.7)` instantiates `bool less_than(const double& a, const double& b)`
- `less_than(2, 2.2)` instantiates `bool less_than(const int& a, const double& b)`
- `less_than(2.2, 2)` instantiates `bool less_than(const double& a, const int& b)`

While this more flexible `less_than` template function allows us to compare mixed numeric expressions, in general it makes more sense to expect the two parameters to be the same type.

Returning to our vector summation function, we can express `sum` generically as

```
template <typename T>
T sum(const std::vector<T>& v) {
    T result = 0;
    for (T elem : v)
        result += elem;
    return result;
}
```

As in the case of the generic `less_than` function, the keyword `template` introduces a template definition. The angle brackets (`<>`) that follow the `template` keyword contain declarations of template parameters.

When the compiler sees calling code like

```
std::vector<int> v {10, 20, 30};
std::cout << sum(v) << '\n';
```

it knows `v`'s declared type is `std::vector<int>`; therefore, it uses the pattern in the template to instantiate an actual function that looks like the following:

```
// The compiler automatically generates this from the template:
int sum(const std::vector<int>& v) {
    int result = 0;
    for (int elem : v)
        result += elem;
    return result;
}
```

Notice how the compiler simply substitutes `int` for every occurrence of the type parameter `T`. If the compiler sees code like

```
std::vector<double> v {10, 20, 30};
std::cout << sum(v) << '\n';
```

it again uses the template to instantiate the following function:

```
// The compiler automatically generates this:
double sum(const std::vector<double>& v) {
    double result = 0;
    for (double elem : v)
        result += elem;
    return result;
}
```

In a program that has code like

```
std::vector<double> v {10, 20, 30};
std::vector<int> w {10, 20, 30};
std::cout << sum(v) << '\n';
std::cout << sum(w) << '\n';
```

the compiler will instantiate *both* versions of the `sum` function. The compiler generates these automatically and secretly. The programmer never sees these instantiated template functions because they do not appear in any source code. Unlike the preprocessor that sends a modified version of the source code to the compiler (see Section 1.2 to review the preprocessor's role in the program's build process), a modern C++ compiler generates the machine language versions of these instantiated functions directly from the programmer's source code.

Recall our `swap` function from Listing 10.19 (swapwithreferences.cpp). Callers can use it to interchange the values of two integer variables. With our knowledge of C++ templates, we now can write a generic version that interchanges the values of two variables of *any* type:

```
/*
 *  swap(a, b)
 *     Interchanges the values of memory
 *     referenced by its parameters a and b.
 *     It effectively interchanges the values
 *     of variables in the caller's context.
 */
template <typename ElemType>
void swap(ElemType& a, ElemType& b) {
   ElemType temp = a;
   a = b;
   b = temp;
}
```

The only restriction is the two parameters must be the same type. Here the type parameter's name is `ElemType`. This `swap` function works fine, but we do not need to write such a function ourselves; the C++ standard library includes a `std::swap` template function that does the same thing as our `swap` function shown above.

The following function will print the contents of a vector of any type:

```
/*
 *  print
 *     Prints the contents of a vector
 *     a is the vector to print.
 *     a is not modified.
 */
template <typename T>
void print(const std::vector<T>& a) {
    int n = a.size();
    std::cout << '{';
    if (n > 0) {
        std::cout << a[0];  // Print the first element
        for (int i = 1; i < n; i++)
            std::cout << ',' << a[i];  // Print the rest
    }
    std::cout << '}';
}
```

The restriction here is that the elements in the vector must be printable to the `std::cout` object using `<<`.

Listing 19.3 (templateflexsort.cpp) uses our new generic functions to increase the flexibility of our flexible sorting program from Listing 12.2 (flexibleintsort.cpp).

Listing 19.3: `templateflexsort.cpp`

```
#include <iostream>
#include <string>
#include <vector>
#include <utility>  // For generic swap function

/*
 *  less_than(a, b)
 *     Returns true if a < b; otherwise, returns
 *     false.
 */
template <typename T>
bool less_than(const T& a, const T& b) {
    return a < b;
}

/*
 *  greater_than(a, b)
 *     Returns true if a > b; otherwise, returns
 *     false.
 */
template <typename T>
bool greater_than(const T& a, const T& b) {
    return a > b;
}

/*
 *   selection_sort(a, n, compare)
 *       Arranges the elements of vector vec in an order determined
 *       by the compare function.
```

```
 *       vec is a vector.
 *       compare is a function that compares the ordering of
 *            two types that support the < operator.
 *       The contents of vec are physically rearranged.
 */
template <typename T>
void selection_sort(std::vector<T>& vec,
                    bool (*compare)(const T&, const T&)) {
    int n = vec.size();
    for (int i = 0;  i < n - 1;  i++) {
        // Note: i,small, and j represent positions within vec.
        // vec[i], vec[small], and a[j] represents the elements at
        // those positions.
        // small is the position of the smallest value we've seen
        // so far; we use it to find the smallest value less
        // than vec[i]
        int small = i;
        // See if a smaller value can be found later in the vector
        for (int j = i + 1;  j < n;  j++)
            if (compare(vec[j], vec[small]))
                small = j;  // Found a smaller value
        // Swap vec[i] and vec[small], if a smaller value was found
        if (i != small)
            std::swap(vec[i], vec[small]); // Uses swap from <utility>
    }
}


/*
 *  print
 *     Prints the contents of a vector
 *     vec is the vector to print.
 *     The function does not modify vec.
 */
template <typename T>
void print(const std::vector<T>& vec) {
    int n = vec.size();
    std::cout << '{';
    if (n > 0) {
        std::cout << vec[0];  // Print the first element
        for (int i = 1; i < n; i++)
            std::cout << ',' << vec[i];  // Print the rest
    }
    std::cout << '}';
}


int main() {
   std::vector<int> list{23, -3, 4, 215, 0, -3, 2, 23, 100, 88, -10};
   std::cout << "Original:   ";
   print(list);
   std::cout << '\n';
   selection_sort(list, less_than<int>);
   std::cout << "Ascending:  ";
   print(list);
```

```
    std::cout << '\n';
    selection_sort(list, greater_than<int>);
    std::cout << "Descending: ";
    print(list);
    std::cout << '\n';
    std::cout << "-------------------------------\n";
    std::vector<std::string> words { "tree", "girl", "boy", "apple",
                                     "dog", "cat", "bird" };
    std::cout << "Original:   ";
    print(words);
    std::cout << '\n';
    selection_sort(words, less_than<std::string>);
    std::cout << "Ascending:  ";
    print(words);
    std::cout << '\n';
    selection_sort(words, greater_than<std::string>);
    std::cout << "Descending: ";
    print(words);
    std::cout << '\n';
}
```

Notice that all the functions except for `main` are generic functions. The `selection_sort` function is the most interesting one. It accepts a generic array and a function pointer that accepts two generic parameters of the same type. In one of its invocations in `main`:

```
selection_sort(list, greater_than<int>);
```

the expression `greater_than<int>` is an explicit template instantiation, and it tells the compiler we want to send to the `selection_sort` function a pointer to the `greater_than` function instantiated with integer parameters. This explicit instantiation is required because the compiler cannot automatically instantiate the `greater_than` function without an actual call to the function passing real parameters, and a pointer to a function is not a function invocation. At this point in the program we are not calling the `greater_than` function but only passing a pointer to it. Calling it would provide the necessary type information to the compiler via the actual parameters used, but a pointer to a function does not contain parameter information. The compiler needs the assistance of this explicit template instantiation.

The call to `swap` within `selection_sort` is actually invoking the `std::swap` function from the standard library. The `#include <utility>` directive brings in the declaration for `std::swap`, which itself is a template function.

The compiler can instantiate an actual function from a template function only if the type substituted for the template parameter supports its use within the function. Consider the following code:

```
// Make two ofstream objects
std::ofstream fout1("file1.data"), fout2("file2.data");
// Compare them
if (less_than(fout1, fout2)   // <-- Error!
    std::cout << "fout1 is less than fout2\n";
```

The expression `less_than(fout1, fout2)` is illegal because no `operator<` exists to compare two `std::ofstream` objects; thus, the compiler cannot produce the machine language that represents the expression `a < b` when `a` and `b` are `std::ofstream`s. Ultimately, therefore, the compiler cannot instantiate an actual `less_than` function that accepts two `std::ofstream` objects.

When programmers use templates incorrectly or construct incorrect template functions or classes, the compiler can detect and report the problems. All current C++ compilers share one weakness with template processing: poor human-readable error messages. The error messages the compiler produces in these situations tend to be lengthy and difficult to decipher even for experienced C++ programmers. This is because the compiler bases its error messages on its processing of the *instantiated* code, not the original *un-instantiated* code. Since programmers see only the un-instantiated source, deciphering the compiler's template error messages tends to be more challenging than usual.

As Listing 19.4 (templatescale.cpp) shows, a template can accept *non-type parameters*.

Listing 19.4: `templatescale.cpp`

```
#include <iostream>

template <int N>
int scale(int value) {
    return value * N;
}

int main() {
    std::cout << scale<3>(5) << '\n';
    std::cout << scale<4>(10) << '\n';
}
```

Note that in Listing 19.4 (templatescale.cpp) the `typename` keyword is missing, and in its place is the actual type `int`. This means parameter N is a non-type template parameter. The program prints

```
15
40
```

Listing 19.5 (templatescale2.cpp) uses both a type-parameter and a non-type parameter.

Listing 19.5: `templatescale2.cpp`

```
#include <iostream>

template <typename T, int N>
T scale(const T& value) {
    return value * N;
}

int main() {
    std::cout << scale<double, 3>(5.3) << '\n';
    std::cout << scale<int, 4>(10) << '\n';
}
```

Listing 19.5 (templatescale2.cpp) displays the following during its execution:

```
15.9
40
```

The compiler will instantiate two `scale` functions:

```
// Instantiated by the compiler due to scale<double, 3>:
double scale(const double& value) {
```

```
    return value * 3;
}
```

and

```
// Instantiated by the compiler due to scale<int, 4>:
int scale(const int& value) {
    return value * 4;
}
```

The types of non-type template parameters permitted by C++ are very limited. Integral types (`int`, `unsigned`, `long`, etc.) are acceptable, as are pointers and references. Other types such as floating-point and programmer-defined types may not be non-type template parameters.

The C++ Standard Library contains many generic functions, some of which we have seen already:

- `std::max`—computes the maximum of two values
- `std::min`—computes the maximum of two values
- `std::swap`—interchanges the values of two variables

19.2 Class Templates

Function templates (Section 19.1) enable us to build generic functions. C++ supports *class templates* as well. With class templates we can specify the pattern or structure of a class of objects in a type-independent way. The class template mechanism is a key tool for creating generic types.

As a simple example, consider a basic `Point` class that represents two-dimensional point objects. Mathematical point objects should have real-valued coordinates that we can approximate with double-precision floating-point values. A point on a graphical display, on the other hand, better uses integer coordinates because screen pixels have discrete, whole number locations.

Rather than providing two separate classes, we can write one class template let the compiler instantiate the coordinates as the particular program requires. Listing 19.6 (genericpoint.h) contains the generic class.

Listing 19.6: `genericpoint.h`

```
#ifndef GENERICPOINT_H_
#define GENERICPOINT_H_

template <typename T>
class Point {
  public:
    T x;
    T y;
    Point(T x, T y): x(x), y(y) {}
};

#endif
```

If a client declares a `Point` object as

```
Point<int> pixel(10, 10);
```

the compiler will instantiate a `Point<int>` class that looks like

```
class Point {
  public:
    int x;
    int y;
    Point(int x, int y): x(x), y(y) {}
};
```

If a client instead declares a `Point` object as

```
Point<double> pixel(10.0, 20.0);
```

the compiler will instantiate a `Point<double>` class that looks like

```
class Point {
  public:
    double x;
    double y;
    Point(double x, double y): x(x), y(y) {}
};
```

Neither of these instantiations appear in any source code, so a programmer will not see them. The compiler substitutes actual types (`int` or `double`) for the parameterized type (`T`).

As a more significant example, Listing 19.7 (genericcomparer.h) contains the declaration of a `Comparer` class that is the generic equivalent to Listing 17.33 (comparer.h).

Listing 19.7: `genericcomparer.h`

```
#ifndef GENERICCOMPARER_H_
#define GENERICCOMPARER_H_
/*
 *  Comparer objects manage the comparisons and element
 *  interchanges on the selection sort function below.
 */
template <typename T>
class Comparer {
    // The object's data is private, so it is inaccessible to
    // clients and derived classes

    // Keeps track of the number of comparisons
    // performed
    int compare_count;

    // Keeps track of the number of swaps performed
    int swap_count;

    // Function pointer directed to the function to
    // perform the comparison
    bool (*comp)(const T&, const T&);

protected:
    // Method that actually performs the comparison
    // Derived classes may customize this method
    virtual bool compare_impl(const T& m, const T& n) {
```

```
        return comp(m, n);
    }

    // Method that actually performs the swap
    // Derived classes may customize this method
    virtual void swap_impl(T& m, T& n) {
        T temp = m;
        m = n;
        n = temp;
    }

public:
    // The client must initialize a Comparer object with a
    // suitable comparison function.
    Comparer(bool (*f)(const T&, const T&)):
                   compare_count(0), swap_count(0), comp(f) {}

    // Resets the counters to make ready for a new sort
    void reset() {
        compare_count = swap_count = 0;
    }

    // Method that performs the comparison.  It delegates
    // the actual work to the function pointed to by comp.
    // This method logs each invocation.
    bool compare(const T& m, const T& n) {
        compare_count++;
        return compare_impl(m, n);
    }

    // Method that performs the swap.
    // Interchange the values of
    // its parameters a and b which are
    // passed by reference.
    // This method logs each invocation.
    void swap(T& m, T& n) {
        swap_count++;
        swap_impl(m, n);
    }

    // Returns the number of comparisons this object has
    // performed since it was created.
    int comparisons() const {
        return compare_count;
    }

    // Returns the number of swaps this object has
    // performed since it was created.
    int swaps() const {
        return swap_count;
    }
};

#endif
```

As with Listing 17.33 (comparer.h), the `Comparer` objects specified in Listing 19.7 (genericcomparer.h) compare and swap any types of values and are meant to be used in a vector or array sorting program. Given the function

```
bool less_than(int m, int n) {
    return m < n;
}
```

the following code fragment instantiates a `Comparer` template that works with integers:

```
Comparer comp(less_than);  // Make an integer comparer object
```

Listing 19.8 (genericstatisticssort.cpp) provides the client code that tests the new `Comparer` class.

Listing 19.8: genericstatisticssort.cpp

```
#include <iostream>
#include <vector>
#include <string>
#include "genericcomparer.h"

/*
 *   selection_sort(a, compare)
 *       Arranges the elements of vector a in an order determined
 *       by the compare object.
 *       a is a vector.
 *       compare is a function that compares the ordering of
 *            two integers.
 *       The contents of a are physically rearranged.
 */
template <typename T>
void selection_sort(std::vector<T>& a, Comparer<T>& compare) {
    int n = a.size();
    for (int i = 0;  i < n - 1;  i++) {
        // Note: i,small, and j represent positions within a
        // a[i], a[small], and a[j] represents the elements at
        // those positions.
        // small is the position of the smallest value we've seen
        // so far; we use it to find the smallest value less
        // than a[i]
        int small = i;
        // See if a smaller value can be found later in the array
        for (int j = i + 1;  j < n;  j++)
            if (compare.compare(a[j], a[small]))
                small = j;  // Found a smaller value
        // Swap a[i] and a[small], if a smaller value was found
        if (i != small)
            compare.swap(a[i], a[small]);
    }
}

/*
 *     Prints the contents of a vector
 *     a is the vector to print.
 *     a is not modified.
 */
```

```
template <typename T>
std::ostream& operator<<(std::ostream& os, const std::vector<T>& a) {
    int n = a.size();
    os << '{';
    if (n > 0) {
        os << a[0];  // Print the first element
        for (int i = 1; i < n; i++)
            std::cout << ',' << a[i];  // Print the rest
    }
    os << '}';
    return os;
}

/*
 *  less_than(a, b)
 *     Returns true if a < b; otherwise, returns
 *     false.
 */
template <typename T>
bool less_than(const T& a, const T& b) {
    return a < b;
}

/*
 *  greater_than(a, b)
 *     Returns true if a > b; otherwise, returns
 *     false.
 */
template <typename T>
bool greater_than(const T& a, const T& b) {
    return a > b;
}

int main() {
   // Make a vector of integers
   std::vector<int> original { 23, -3, 4, 215, 0, -3, 2, 23, 100, 88, -10 };

   // Make a working copy of the original vector
   std::vector<int> working = original;
   std::cout << "Before:   ";
   std::cout << working << '\n';
   Comparer<int> lt(less_than<int>), gt(greater_than<int>);
   selection_sort(working, lt);
   std::cout << "Ascending:  ";
   std::cout << working << '\n';
   std::cout << "  (" << lt.comparisons() << " comparisons, "
             << lt.swaps() << " swaps)\n";
   std::cout << "----------------------------\n";
   // Make another copy of the original vector
   working = original;
   std::cout << "Before:   ";
   std::cout << working << '\n';
   selection_sort(working, gt);
   std::cout << "Descending: ";
   std::cout << working << '\n';
```

```
    std::cout << "  (" << gt.comparisons() << " comparisons, "
              << gt.swaps() << " swaps)\n";
    std::cout << "----------------------------\n";

    // Make a vector of strings
    std::vector<std::string> str_original { "tree", "boy", "apple",
                                            "girl", "dog", "cat" };

    // Make a working copy of the original vector
    std::vector<std::string> str_working = str_original;
    std::cout << "Before:   ";
    std::cout << str_working << '\n';
    Comparer<std::string> str_lt(less_than<std::string>),
                          str_gt(greater_than<std::string>);
    selection_sort(str_working, str_lt);
    std::cout << "Ascending:  ";
    std::cout << str_working << '\n';
    std::cout << "  (" << str_lt.comparisons() << " comparisons, "
              << str_lt.swaps() << " swaps)\n";
    std::cout << "----------------------------\n";
    // Make another copy of the original vector
    str_working = str_original;
    std::cout << "Before:   ";
    std::cout << str_working << '\n';
    selection_sort(str_working, str_gt);
    std::cout << "Descending: ";
    std::cout << str_working << '\n';
    std::cout << "  (" << str_gt.comparisons() << " comparisons, "
              << str_gt.swaps() << " swaps)\n";
    std::cout << "----------------------------\n";
}
```

Listing 19.8 (genericstatisticssort.cpp) demonstrates that the generic `Comparer` class works equally as well with integers and strings; the program's output is

```
Before:   {23,-3,4,215,0,-3,2,23,100,88,-10}
Ascending:  {-10,-3,-3,0,2,4,23,23,88,100,215}
  (55 comparisons, 7 swaps)
----------------------------
Before:   {23,-3,4,215,0,-3,2,23,100,88,-10}
Descending: {215,100,88,23,23,4,2,0,-3,-3,-10}
  (55 comparisons, 5 swaps)
----------------------------
Before:   {tree,boy,apple,girl,dog,cat}
Ascending:  {apple,boy,cat,dog,girl,tree}
  (15 comparisons, 3 swaps)
----------------------------
Before:   {tree,boy,apple,girl,dog,cat}
Descending: {tree,girl,dog,cat,boy,apple}
  (15 comparisons, 4 swaps)
----------------------------
```

Recall the `IntList5` linked list class from Listings Listing 18.18 (intlist5.h) and Listing 18.19 (intlist5.cpp). It implemented a linked list of integer elements. Suppose we instead want to make a linked list of `std::string`

objects. Without C++ templates, we would have to copy the `IntList` code and modify it so it works with `string` objects instead of integers. We could call this class `StringList`. Note that the code in the `IntList` and `StringList` classes would be almost identical; the only major difference would be the type of elements the list can hold. C++'s class template feature is ideal for designing generic container classes.

Listing 19.9 (linkedlist.h) defines a such a generic linked list in its `LinkedList` class.

Listing 19.9: `linkedlist.h`

```
// linkedlist.h

#include <iostream>
#include <utility> // For std::swap
#include <memory>  // For std::shared_ptr

template <typename T>
class LinkedList {
    // The nested private Node class from before
    struct Node {
        T data;                       // A data element of the list
        std::shared_ptr<Node> next;   // The node that follows this one
        // Constructor
        Node(const T& item): data(item), next(nullptr) {
            std::cout << "Creating node " << data
                      << "  (" << reinterpret_cast<uintptr_t>(this) << ")\n";
        }
        // Destructor
        ~Node() {
            std::cout << "Destroying node " << data
                      << "  (" << reinterpret_cast<uintptr_t>(this) << ")\n";
        }
        Node(const Node&) = default;            // Copy constructor
        Node(Node&&) = default;                 // Move constructor
        Node& operator=(const Node&) = default; // Copy assignment
        Node& operator=(Node&&) = default;      // Move assignment
    };


    std::shared_ptr<Node> head;  // Points to the first item in the list
    std::shared_ptr<Node> tail;  // Points to the last item in the list

    int len;                     // Number of elements in the list

public:
    // The constructor makes an initially empty list.
    // The list is empty when head and tail are null.
    LinkedList(): head(nullptr), tail(nullptr), len(0) {}

    // Copy constructor makes a copy of the other object's list
    LinkedList(const LinkedList& other): LinkedList() {
        // Walk through other's list inserting each of its elements
        // into this list
        for (auto cursor = other.head; cursor; cursor = cursor->next)
            insert(cursor->data);
    }
```

```
    // Move constructor takes possession of the temporary's list
    LinkedList(LinkedList&& temp): LinkedList() {
        // Swap contents with the temporary
        std::swap(head, temp.head);
        std::swap(tail, temp.tail);
        std::swap(len, temp.len);
    }

    // Assignment operator
    LinkedList& operator=(const LinkedList& other) {
        // Make a local, temporary copy of other
        LinkedList temp{other};
        // Exchange the head and tail pointers and len from this list
        // with those of the new, temporary list
        std::swap(head, temp.head);
        std::swap(tail, temp.tail);
        std::swap(len, temp.len);
        // The other list now points to this list's original contents,
        // and this list now points to the copy of other's list
        return *this;
    }

    // Move assignment operator
    LinkedList& operator=(LinkedList&& temp) {
        // Exchange the head and tail pointers and len from this list
        // with those of the new, temporary list
        std::swap(head, temp.head);
        std::swap(tail, temp.tail);
        std::swap(len, temp.len);
        // The temporary list now points to this list's original contents,
        // and this list now points to the temporary's list
        // The temporary list will be destroyed since it is a temporary
        return *this;
    }

    // The destructor deallocates the memory held by the list
    ~LinkedList() {
        clear();
    }

    // Inserts item onto the back of the list.
    // item is the element to insert.
    void insert(const T& item) {
        // Make a node for the new element n
        auto new_node = std::make_shared<LinkedList::Node>(item);
        if (tail) {   // Is tail non-null?
           tail->next = new_node;  // Link the new node onto the back
           tail = new_node;        // The new node is the new tail of the list
        }
        else // List is empty, so make head and tail point to new node
            head = tail = new_node;
        len++;
    }
```

```
    // Removes the first occurrence of item from the list.
    // Returns true if successful (found item and removed it).
    // Returns false if item is not originally present in the list.
    bool remove(const T& item) {
        auto cursor = head,    // Start at head of list
             prev =  head;     // Keep track of previous node seen
        // Loop until we run off the end of the list or find n,
        // whichever comes first
        while (cursor && cursor->data != item) {
            prev = cursor;              // Remember previous node
            cursor = cursor->next;      // Move to next node
        }
        if (!cursor)                    // Did we run off the end of the list?
            return false;               // Indicate we did not find n

        // Found n; cursor is pointing at the node containing n
        if (head == tail)               // n was the only element in the list
            head = tail = nullptr;      // cursor still points to node with n
        else if (cursor == head)        // Is n the first element in the list?
            head = head->next;          // Redirect head around n
        else                            // n is not the first element
            prev->next = cursor->next;  // Redirect previous node around n
        if (cursor == tail)             // Was n the last element in the list?
            tail = prev;                // Update tail to new last element

        // No need to delete node; shared_ptr takes care of it
        len--;                          // List size decreases by 1
        return true;                    // We found n and deleted its node
    }

    // Prints the contents of the linked list of integers.
    void print() const {
        for (auto cursor = head; cursor; cursor = cursor->next)
            std::cout << cursor->data << ' ';
        std::cout << '\n';
    }

    // Returns the length of the linked list.
    int length() const {
        return len;
    }

    // Removes all the elements in the linked list.
    void clear() {
        auto cursor = head;
        while (cursor) {
            auto temp = cursor;     // Remember where we are
            cursor = cursor->next;  // Move next node
            temp->next = nullptr;   // Sever link from previous node
        }
        head = tail = nullptr;  // Null head signifies list is empty
        len = 0;
    }

    // Provide a convenient way to print a linked list
```

```
    template <typename V>
    friend std::ostream& operator<<(std::ostream& os,
                                    const LinkedList<V>& list);
};

// Prints a linked list object to an output stream
template <typename T>
std::ostream& operator<<(std::ostream& os, const LinkedList<T>& list) {
    os << '{';
    if (list.length() > 0) {
        auto cursor = list.head;
        os << cursor->data;
        cursor = cursor->next;
        while (cursor) {
            os << ", " << cursor->data;
            cursor = cursor->next;
        }
    }
    os << '}';
    return os;
}
```

Notice that in Listing 19.9 (linkedlist.h) the linked list elements have the generic type T. Observe carefully where the T type parameter appears in the code.

Listing 19.10 (intlistmain.cpp) shows how a client can instantiate the `LinkedList` template for integer elements.

Listing 19.10: `intlistmain.cpp`

```
// intlistmain.cpp

#include <iostream>
#include "linkedlist.h"

int main() {
    bool done = false;
    char command;
    int value;
    LinkedList<int> list;     // Instantiated for integers

    while (!done) {
        std::cout << "I)nsert <item>  D)elete <item> P)rint  L)ength  E)rase Q)uit >>";
        std::cin >> command;
        switch (command) {
          case 'I':   // Insert a new element into the list
          case 'i':
            if (std::cin >> value)
                list.insert(value);
            else
                done = true;
            break;
          case 'D':   // Insert a new element into the list
          case 'd':
            if (std::cin >> value)
                if (list.remove(value))
```

```
                    std::cout << value << " removed\n";
                else
                    std::cout << value << " not found\n";
            else
                done = true;
            break;
          case 'P':  // Print the contents of the list
          case 'p':
            list.print();
            break;
          case 'L':  // Print the list's length
          case 'l':
            std::cout << "Number of elements: " << list.length() << '\n';
            break;
          case 'E':  // Erase the list
          case 'e':
            list.clear();
            break;
          case 'Q':  // Exit the loop (and the program)
          case 'q':
            done = true;
            break;
        }
    }
}
```

The statement

```
LinkedList<int> list;
```

instantiates an actual class where all the occurrences of `T` become `int`. It also declares `list` to be an object of this instantiated class.

The following shows a sample run of Listing 19.10 (intlistmain.cpp).

```
I)nsert <item>  P)rint  L)ength D)elete <item>  E)rase Q)uit >>p

I)nsert <item>  P)rint  L)ength D)elete <item>  E)rase Q)uit >>i 44
Creating node 44  (16085084)
I)nsert <item>  P)rint  L)ength D)elete <item>  E)rase Q)uit >>i 23
Creating node 23  (16084476)
I)nsert <item>  P)rint  L)ength D)elete <item>  E)rase Q)uit >>i 88
Creating node 88  (16084380)
I)nsert <item>  P)rint  L)ength D)elete <item>  E)rase Q)uit >>i 99
Creating node 99  (16084668)
I)nsert <item>  P)rint  L)ength D)elete <item>  E)rase Q)uit >>p
44 23 88 99
I)nsert <item>  P)rint  L)ength D)elete <item>  E)rase Q)uit >>l
Number of elements: 4
I)nsert <item>  P)rint  L)ength D)elete <item>  E)rase Q)uit >>d 44
Destroying node 44  (16085084)
44 removed
I)nsert <item>  P)rint  L)ength D)elete <item>  E)rase Q)uit >>l
Number of elements: 3
I)nsert <item>  P)rint  L)ength D)elete <item>  E)rase Q)uit >>p
23 88 99
```

```
I)nsert <item>  P)rint  L)ength D)elete <item>  E)rase Q)uit >>d 99
Destroying node 99  (16084668)
99 removed
I)nsert <item>  P)rint  L)ength D)elete <item>  E)rase Q)uit >>p
23 88
I)nsert <item>  P)rint  L)ength D)elete <item>  E)rase Q)uit >>l
Number of elements: 2
I)nsert <item>  P)rint  L)ength D)elete <item>  E)rase Q)uit >>i 9
Creating node 9  (16085084)
I)nsert <item>  P)rint  L)ength D)elete <item>  E)rase Q)uit >>p
23 88 9
I)nsert <item>  P)rint  L)ength D)elete <item>  E)rase Q)uit >>q
Destroying node 23  (16084476)
Destroying node 88  (16084380)
Destroying node 9  (16085084)
```

Listing 19.11 (stringlistmain.cpp) shows how a client can instantiate the `LinkedList` template for `std::string` elements.

Listing 19.11: `stringlistmain.cpp`

```
// stringlistmain.cpp

#include <iostream>
#include <string>
#include "linkedlist.h"

int main() {
    bool done = false;
    char command;
    std::string value;
    LinkedList<std::string> list;  // Instantiated for strings

    while (!done) {
        std::cout << "I)nsert <item>  P)rint  L)ength D)elete <item>  E)rase Q)uit >>";
        std::cin >> command;
        switch (command) {
          case 'I':   // Insert a new element into the list
          case 'i':
            if (std::cin >> value)
                list.insert(value);
            else
                done = true;
            break;
          case 'D':   // Insert a new element into the list
          case 'd':
            if (std::cin >> value)
                if (list.remove(value))
                    std::cout << value << " removed\n";
                else
                    std::cout << value << " not found\n";
            else
                done = true;
            break;
          case 'P':  // Print the contents of the list
```

```
            case 'p':
              list.print();
              break;
            case 'L':  // Print the list's length
            case 'l':
              std::cout << "Number of elements: " << list.length() << '\n';
              break;
            case 'E':  // Erase the list
            case 'e':
              list.clear();
              break;
            case 'Q':  // Exit the loop (and the program)
            case 'q':
              done = true;
              break;
          }
      }
}
```

Listing 19.11 (stringlistmain.cpp) adds the extra `#include` directive for the string library, but observe that the only executable statements that differ from Listing 19.10 (intlistmain.cpp) are

```
std::string value;
LinkedList<std::string> list;  // Instantiated for strings
```

vs.

```
int value;
LinkedList<int> list;     // Instantiated for integers
```

All the other code in both programs is identical.

Just as we may instantiate function templates with multiple different types within the same program, we can do the same for class templates. The class template `LinkedList` in Listing 19.9 (linkedlist.h) makes the following client code possible:

```
LinkedList<int> intlist;
LinkedList<std::string> strlist;
intlist.insert(5);
strlist.insert("Wow");
intlist.print();
strlist.print();
```

This code fragment simultaneously manages lists of integers and lists of string objects.

Listing 19.12 (listoflists.cpp) shows how we can use our generic list to build a list of lists.

Listing 19.12: `listoflists.cpp`

```
#include <iostream>
#include "linkedlist.h"

int main() {
    // First build some lists that contain integers
    LinkedList<int> intlist1, intlist2, intlist3;
    for (int i = 10; i < 50; i += 10)
```

```
        intlist1.insert(i);
    for (int i = 100; i < 500; i += 100)
        intlist2.insert(i);
    for (int i = 5; i < 25; i += 5)
        intlist3.insert(i);

    // Next build a list that contains lists of integers
    LinkedList<LinkedList<int>> listoflists;
    listoflists.insert(intlist1);
    listoflists.insert(intlist2);
    listoflists.insert(intlist3);
    std::cout << "--------------------\n";
    std::cout << listoflists << '\n';
    std::cout << "--------------------\n";

}
```

By now you probably have a deduced the fact that `std::vector` is a class template. The `std::vector` class template is much more sophisticated than our simple `LinkedList` template, but the basic mechanism for its ability to store different types of elements, a class template, is the same.

19.3 Exercises

CAUTION! SECTION UNDER CONSTRUCTION

1. Write a generic function named `is_ascending` that accepts a `std::vector` containing any elements comparable by `operator<`. The function should return `true` if the elements in the vector appear in non-decreasing (or ascending) order; otherwise, the function should return `false`.

2. Write a generic function named `is_member` that accepts a argument of any type comparable by `operator==` and a second argument consisting of a `std::vector` containing elements of same type as the first argument. The function should return `true` if the first argument is a member of the second argument; otherwise, the function should return `false`.

3. Augment Listing 19.9 (linkedlist.h) by adding a method named `prepend` that adds an element to the front of a generic linked list.

4. Augment Listing 19.9 (linkedlist.h) by adding an `operator==` method that returns `true` if two generic linked list contain exactly the same elements in exactly the same order; otherwise, the method returns `false`. Also provide a one-line `operator!=` method that utilizes your `operator==` method.

5. Augment Listing 19.9 (linkedlist.h) by adding an

Chapter 20

The Standard Template Library

CAUTION! CHAPTER UNDER CONSTRUCTION

Chapter 19 introduced the mechanics of generic programming with templates. The C++ standard library leverages templates to provide a rich collection of standard generic containers and algorithms to manipulate the containers and process the elements they contain. This part of the standard library commonly is known as the *standard template library*, or *STL* for short. As its name implies, the STL contains a number of generic functions and classes built with templates. We have seen the `std::vector` class (Chapter 11) and `std::swap` function Section 12.1.

The STL consists of four majors components:

- generic (that is, templated) containers for storing data, such as `std::vector`,
- iterators for traversing containers,
- algorithms for processing the data within containers, and
- miscellaneous utility classes and functions, such as `std::swap` and `std::max`.

The design of the containers and their iterators make the STL's generic algorithms possible. The algorithms provided by the STL are powerful and flexible, but they also are somewhat arcane and can be difficult to use for casual C++ programmers.

In this chapter we explore some of the features of the STL, providing examples that illustrate its capabilities.

20.1 Containers

We are familiar with one STL container: `std::vector`. A vector object manages a primitive C array. C arrays come in two varieties: static and dynamic (see Section 11.2). Vectors manage dynamically-allocated arrays. A vector object can expand as needed the storage space for its low-level array. The STL provides an equivalent to a static array in its `std::array` class. A programmer must specify the size of a `std::array` object when declaring it. The following code creates a `std::array` that can hold 10 integer values:

```
std::array<int, 10> arr;
```

Note that the second template parameter is a non-type parameter (see Section 19.1) specifying the size of the array. Besides not being able to modify the array's size through methods such as `push_back`, `std::array` objects work very much like `std::vector`s. Array objects keep track of their size, unlike primitive C arrays.

In Listing 19.9 (linkedlist.h) we developed a generic linked list class named `LinkedList`. That was good practice dealing with generic types, but our work really was re-inventing the wheel because the STL provides the generic `std::list` class. The STL implements a linked list with pointers connecting a node to both its successor node and predecessor node. A list with both forward and backward pointers is known as *doubly linked list*. Listing 20.1 (stdlistmain.cpp) is the third and final variation of our list programs. Listing 18.5 (listmain.cpp) used a custom class to implement a singly linked list of integers (forward pointers only). We used C++ś template feature to build a type generic list class in Listing 19.10 (intlistmain.cpp). Listing 20.1 (stdlistmain.cpp) uses the `std::list` class provided by the STL.

Listing 20.1: `stdlistmain.cpp`

```
// stdlistmain.cpp

#include <iostream>
#include <list>   // Use the standard doubly linked list class

int main() {
    bool done = false;
    char command;
    int value;
    std::list<int> mylist;    // Initially empty

    while (!done) {
        std::cout << "I)nsert <item>  P)rint  L)ength  E)rase Q)uit >>";
        std::cin >> command;
        switch (command) {
          case 'I':   // Insert a new element into the list
          case 'i':
            if (std::cin >> value)
                mylist.push_back(value);
            else
                done = true;
            break;
          case 'P':  // Print the contents of the list
          case 'p':
            for (const auto& elem : mylist)
                std::cout << elem << ' ';
            std::cout << '\n';
            break;
          case 'L':  // Print the list's length
          case 'l':
            std::cout << "Number of elements: " << mylist.size() << '\n';
            break;
          case 'E':  // Erase the list
          case 'e':
            mylist.clear();
            break;
          case 'Q':  // Exit the loop (and the program)
```

```
            case 'q':
              done = true;
              break;
          }
      }
  }
```

Listing 20.1 (stdlistmain.cpp) contains all the code we need to write. The STL provides all of the linked list implementation code. Note that the differences between Listing 20.1 (stdlistmain.cpp) and Listing 19.10 (intlistmain.cpp) are minimal.

A *priority queue* is a data structure that allows clients to store objects with various priorities or rankings. Clients can insert the objects in any order, and the priority queue always will serve (remove) higher priority objects before lower priority objects. A properly designed priority queue supports very fast insertion and removal operations.

The C++ standard library provides the `std::priority_queue` class. Listing 20.2 (priorityqueuesimple.cpp) provides a simple example using a `std::priority_queue`.

Listing 20.2: `priorityqueuesimple.cpp`

```
#include <iostream>
#include <queue>

int main() {
    std::priority_queue<int> queue;
    // Insert some integers into the priority queue
    queue.push(11);
    queue.push(2);
    queue.push(4);
    queue.push(15);
    queue.push(15);
    queue.push(4);
    queue.push(7);
    // Serve items from the queue until it is empty
    while (!queue.empty()) {
        std::cout << queue.top() << ' ';  // See what's on top
        queue.pop();          // Remove highest priority element
    }
    std::cout << '\n';
}
```

Listing 20.2 (priorityqueuesimple.cpp) displays the following:

```
15 15 11 7 4 4 2
```

Note that the priority queue serves larger values before smaller values regardless of their insertion order. It equates larger-valued numbers with a higher priority.

If we want to change the notion of priority so that lower-valued numbers are considered higher priority than larger numbers, the `std::priority_queue` class provides the flexibility for us to do so. We must define a special class or `struct` that provides a public method named `operator()`. This method must return a Boolean result and accept two arguments of the same type as the elements held by the priority queue.

Consider the following `Cmp` type:

```
struct Cmp {
    bool operator()(int i1, int i2) {
        return i2 < i1;
    }
};
```

This special method is the *function invocation operator*, or *function call operator*. It enables any instance of the `Cmp` class to be used like a function. To see how this works, consider Listing 20.3 (calloperator.cpp).

Listing 20.3: `calloperator.cpp`

```
#include <iostream>

struct Cmp {
    bool operator()(int i1, int i2) {
        return i2 < i1;
    }
};

int main() {
    Cmp op;
    std::cout << std::boolalpha << op(2, 3) << '\n';
    std::cout << std::boolalpha << op(3, 2) << '\n';
    std::cout << std::boolalpha << op(3, 3) << '\n';
}
```

Listing 20.3 (calloperator.cpp) prints the following:

```
false
true
false
```

The first output line indicates it is false that $3 < 2$. The second line reveals that it is true that $2 < 3$, and in the last line we see that it is false that $3 < 3$.

The expression `op(2, 3)` looks like a call to the function named `op`, but it really is just syntactic sugar for `op.operator()(2, 3)`. We see that we really are invoking the method nameed `operator()` on behalf of the `op` object.

Listing 20.4 (reversepriority.cpp) provides the complete code for a priority queue that holds integers and recognizes lower-valued integers as higher priority than high-valued integers.

Listing 20.4: `reversepriority.cpp`

```
#include <iostream>
#include <queue>
#include <vector>

struct Cmp {
    bool operator()(int i1, int i2) {
        return i2 < i1;
    }
};

int main() {
```

```
    std::priority_queue<int, std::vector<int>, Cmp> queue;
    //std::priority_queue<int> queue;
    // Insert some integers into the priority queue
    queue.push(3);
    queue.push(12);
    queue.push(55);
    queue.push(11);
    queue.push(2);
    queue.push(4);
    // Serve items from the queue until it is empty
    while (!queue.empty()) {
        std::cout << queue.top() << ' ';  // See what's on top
        queue.pop();         // Remove highest priority element
    }
    std::cout << '\n';
}
```

Listing 20.4 (reversepriority.cpp) prints

```
2 3 4 11 12 55
```

Since we are providing a custom comparer class to modify the default priority queue ordering, the declaration of the `std::priority_queue` object is a little more complicated:

```
std::priority_queue<int, std::vector<int>, Cmp> queue;
```

The template instantiation requires three type parameters:

- `int` is the type of items the priority queue holds.
- `std::vector<int>` is the type of container that holds the items in the priority queue; programmers can specify a different type of container. The `std::priority_queue` class serves as a wrapper around an exiting container type.
- `Cmp` is the type of comparer object that the priority queue will use to order its items.

Armed with the ability to precisely specify how the priority queue defines priorities, we can store any type of objects in a priority queue. By default, the `std::priority_queue` class can store any type of objects that can be compared using the `<` operator. This includes the basic numeric types (`int`, `double`, `unsigned`, etc.) and `std::string` objects. The `std::complex` class (see Section 13.4) models mathematical complex numbers. The relational operators $<$, $\leq$, $>$, and $\geq$ have no practical mathematical meaning in the context of complex numbers; therefore, C++ does not automatically support operator `<` comparison between two `std::complex` objects. We can define our own ordering, however. Let $a+bi$ and $c+di$ be two complex numbers. We will define $a+bi$ to have higher priority that $c+di$ if the distance between $a+bi$ and 0 on the complex plane is less than the distance between $c+di$ and 0. The distance between $a+bi$ and 0 is $\sqrt{a^2+b^2}$, and the distance between $c+di$ and 0 is $\sqrt{c^2+d^2}$. The following custom comparer class implements this notion of complex number priorities:

```
// Define priorities for complex numbers.
// If c1's distance to zero on the complex plain is less
// than c2's distance to zero, c1 has higher priority than c2.
struct ComplexCompare {
    bool operator()(Complex c1, Complex c2) {
```

```
        double a = c1.real(), b = c1.imag(),
               c = c2.real(), d = c2.imag();
        return sqrt((a*a + b*b) < (c*c + d*d));
    }
};
```

With this our definition of complex number less than we overload `operator<` for complex numbers, as shown in Listing 20.5 (complexdiff.cpp).

Listing 20.5: `complexdiff.cpp`

```
#include <iostream>
#include <queue>
#include <complex>

using Complex = std::complex<double>;

// Define priorities for complex numbers.
// If c1's distance to zero on the complex plain is less
// than c2's distance to zero, c1 has higher priority than c2.
struct ComplexCompare {
    bool operator()(Complex c1, Complex c2) {
        double a = c1.real(), b = c1.imag(),
               c = c2.real(), d = c2.imag();
        return sqrt((a*a + b*b) < (c*c + d*d));
    }
};

int main() {
    std::priority_queue<Complex, std::vector<Complex>,
                        ComplexCompare> queue;
    // Insert some complex numbers into the priority queue
    queue.push({3, 2});
    queue.push({12, 3});
    queue.push({5, 45});
    queue.push({11, 5});
    queue.push({2, 1});
    queue.push({2.5, 5.4});
    // Serve items from the queue until it is empty
    while (!queue.empty()) {
        std::cout << queue.top() << ' ';  // See what's on top
        queue.pop();         // Remove highest priority element
    }
    std::cout << '\n';
}
```

If we want to prioritize complex numbers in a different way, we need only provide a different custom comparer type.

Custom comparer objects allow us to place items in a priority queue that inherently are not numerical themselves. Suppose we are writing part of a messaging system that processes text messages. In addition to the text in the message, each message has a integer priority in the range 1 . . . 10. Lower numbered messages have priority over messages with higher numbers. Instances of the following `struct` model our simple messages:

```
struct Message {
    int rank; // Lower number = higher priority
    std::string text;
    // Constructor
    Message(const std::string& t, int rank): text(t), priority(rank) {}
};
```

The following shows how we can compare two message objects to give lower-numbered messages higher priority:

```
// Custom comparer object to properly prioritize Message objects
struct Cmp {
    bool operator()(Message m1, Message m2) {
        return m2.rank < m1.rank;
    }
};
```

Listing 20.6 (messagepriority.cpp) provides the complete simple messaging example.

Listing 20.6: messagepriority.cpp

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

struct Message {
    int rank; // Lower number = higher priority
    std::string text;
    // Constructor
    Message(const std::string& t, int rank): text(t), rank(rank) {}
};

// Custom comparer object to properly prioritize Message objects
struct Cmp {
    bool operator()(Message m1, Message m2) {
        return m2.rank < m1.rank;
    }
};

int main() {
    std::priority_queue<Message, std::vector<Message>, Cmp> queue;
    // Insert some messages into the priority queue
    queue.push({"Be there soon", 2});
    queue.push({"Gimme a few minutes", 3});
    queue.push({"Will be late!", 1});
    queue.push({"Have to work next week", 7});
    queue.push({"I bought a new computer", 5});
    queue.push({"The project deadline is the first", 3});
    // Serve items from the queue until it is empty
    while (!queue.empty()) {
        // See what's on top
        Message msg = queue.top();
        std::cout << msg.text << ": " << msg.rank << '\n';
        queue.pop();         // Remove highest priority element
    }
```

```
    std::cout << '\n';
}
```

Listing 20.6 (messagepriority.cpp) prints

```
Will be late!: 1
Be there soon: 2
The project deadline is the first: 3
Gimme a few minutes: 3
I bought a new computer: 5
Have to work next week: 7
```

Vectors, arrays, lists, and priority queues are all sequence containers. This means their elements appear in a linear ordering from the beginning of the data structure to its end. The STL provides other kinds of containers that we will explore in Chapter 21.

20.2 Iterators

Section 11.2 demonstrates how we can use a pointer to access the elements within an array. A vector is an object that manages a primitive array, and the developers of the vector class designed vectors to look and feel as much as possible as arrays. They adopted the square bracket (`[]`) operator to access elements. They also devised a way for programmers to use pointer-like objects to access the elements within a vector. These pointer-like objects are called *iterators*.

An iterator is an object that allows a client to traverse and access elements of a data structure in an implementation independent way. C++ defines two global functions, `std::begin` and `std::end`, that produce iterators to the front and back, respectively, of a data structure like a vector or static array. Containers defined int the STL provide `begin` and `end` methods that serve the same purpose; for example, if `v` is a `std::vector`, `std::begin(v)` returns the same iterator as the call `v.begin()`. Functions in the standard library that accept iterators as arguments rather than arrays or vectors work equally well with both vectors and arrays. Since they accept iterator arguments, these standard functions additionally are able to work with other, more sophisticated data structures. We will examine some of these standard functions later in this chapter.

In order to behave like a pointer, an iterator object provides the following methods:

- `operator*`: used to access the element at the itertator's current position. The syntax is exactly like pointer dereferencing (see Section 10.7).
- `operator++`: used to move the iterator to the next element within the data structure. The syntax is exactly like pointer arithmetic (see Section 11.2).
- `operator!=`: used to determine whether two iterator objects currently refer to different elements within the data structure.

Listing 20.7 (simpleiterator.cpp) shows how to use these methods to manipulate an iterator object.

Listing 20.7: `simpleiterator.cpp`

```
#include <iostream>
#include <vector>
```

```
int main() {
    // Make a simple integer vector
    std::vector<int> vec {10, 20, 30, 40, 50};
    // Direct an iterator to the vector's first element
    std::vector<int>::iterator iter = std::begin(vec);
    // Print the element referenced by the iterator
    std::cout << *iter << '\n';
    // Advance the iterator
    iter++;
    // See where the iterator is now
    std::cout << *iter << '\n';
}
```

Listing 20.7 (simpleiterator.cpp) prints

```
10
20
```

The statement

```
std::vector<int>::iterator iter = std::begin(vec);
```

declares and initializes an iterator object, `iter`. The type of `iter` is `std::vector<int>::iterator`. This complicated expression indicates that `iterator` is a type defined within the `std::vector<int>` type. A shorter way to express this statement takes advantage of the compiler's ability to infer the variable's type from its context:

```
auto iter = std::begin(vec);
```

(Section 3.10 introduced C++'s type inference capabilities.) The compiler can deduce the type of `iter` based on the return type of the `std::begin` function.

While the `std::begin` function returns an iterator pointing to the first element in a data structure, the `std::end` function returns an iterator pointing to the element *just past the last element* in the data structure. In the code fragment

```
std::vector<int> vec {10, 20, 30, 40, 50};
auto iter = std::end(vec);
```

the iterator object `iter` does not reference a viable element in the vector. As such, we never attempt to dereference the iterator; the expression `*iter` in this case is undefined. We normally use the iterator returned by the `std::end` function to test to see if we are past the end of our data structure, Listing 20.8 (iteratorloop.cpp) demonstrates.

Listing 20.8: `iteratorloop.cpp`

```
#include <iostream>
#include <vector>

int main() {
    // Make a simple integer vector
    std::vector<int> vec {10, 20, 30, 40, 50};
    // Print the contents of the vector
    for (auto iter = std::begin(vec); iter != std::end(vec); iter++)
        std::cout << *iter << ' ';
```

```
    std::cout << '\n';
}
```

Listing 20.8 (iteratorloop.cpp) prints

```
10 20 30 40 50
```

The expression

```
iter != std::end(vec)
```

checks each time through the loop to ensure the iterator object has not run off the back end of the vector.

We can use the vector methods `begin` and `end` instead of the `std::begin` and `std::end` functions. Listing 20.9 (iteratorloopmethods.cpp) is a modified version of Listing 20.8 (iteratorloop.cpp) that uses the iterator methods.

Listing 20.9: `iteratorloopmethods.cpp`

```
#include <iostream>
#include <vector>

int main() {
    // Make a simple integer vector
    std::vector<int> vec {10, 20, 30, 40, 50};
    // Print the contents of the vector
    for (auto iter = vec.begin(); iter != vec.end(); iter++)
        std::cout << *iter << ' ';
    std::cout << '\n';
}
```

The `std::begin` and `std::end` functions are overloaded to work with vector objects, arrays, and other container classes found in the standard library. Listing 20.10 (iteratorlooparray.cpp) is the array version of Listing 20.8 (iteratorloop.cpp).

Listing 20.10: `iteratorlooparray.cpp`

```
#include <iostream>
#include <vector>

int main() {
    // Make a static integer array
    int arr[] =  {10, 20, 30, 40, 50};
    // Print the contents of the array
    for (auto iter = std::begin(arr); iter != std::end(arr); iter++)
        std::cout << *iter << ' ';
    std::cout << '\n';
}
```

The output of Listing 20.10 (iteratorlooparray.cpp) is identical to Listing 20.8 (iteratorloop.cpp):

```
10 20 30 40 50
```

Any type for which the `std::begin` and `std::end` functions have been overloaded may participate in a range-based `for` statement. In fact, the compiler internally transforms a range-based `for` construct into its equivalent `std::begin`/`std::end` form during the compilation process. The following code

```
for (auto i : vec)
    std::cout << i << ' ';
```

is short for (one possible internal compiler representation)

```
for (auto _i = std::begin(vec); _i != std::end(vec); _i++)
    std::cout << *_i << ' ';
```

The STL supports various kinds of iterators. The kind of iterator available to `std::vector` objects is known as a *random access itertator*. Random access iterators behave exactly like a pointers. Recall from Section 11.2 that we can manipulate pointers with pointer arithmetic. If `p` points to the first element of an array, `p + 5` points to the sixth element in the array (because `p + 0` is the same as `p`). The expression `p++` makes the `p` point to the next element in the array, while `p--` moves `p` backwards one position. We can use similar arithmetic with iterators as Listing 20.11 (iterarith.cpp) illustrates.

Listing 20.11: `iterarith.cpp`

```
#include <iostream>
#include <vector>

// Print the contents of vector v, traversing with a
// caller supplied increment value (inc)
void print(const std::vector<int>& v, int inc) {
    for (auto p = std::begin(v); p != std::end(v); p += inc)
        std::cout << *p << ' ';
    std::cout << '\n';
}

// Print the contents of the vector v backwards,
// traversing with a caller supplied decrement value (dec)
void print_reverse(const std::vector<int>& v, int dec) {
    auto p = std::end(v);
    while (p != std::begin(v)) {
        p -= dec;
        std::cout << *p << ' ';
    }
    std::cout << '\n';
}

int main() {
   std::vector<int> vec(20);
   for (int i = 0; i < 20; i++)
       vec[i] = i;

   print(vec, 1);
   print(vec, 2);
   print(vec, 4);
   print(vec, 5);
   print(vec, 10);

   std::cout << '\n';

   print_reverse(vec, 1);
   print_reverse(vec, 2);
   print_reverse(vec, 4);
```

```
    print_reverse(vec, 5);
    print_reverse(vec, 10);
}
```

Listing 20.11 (iterarith.cpp) prints

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
0 2 4 6 8 10 12 14 16 18
0 4 8 12 16
0 5 10 15
0 10

19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
18 16 14 12 10 8 6 4 2 0
16 12 8 4 0
15 10 5 0
10 0
```

Some specialized data structures in the C++ standard library support specialized iterators that have some limits on the kinds of arithmetic they support; for example, some container types support forward but not backward traversal. Subtraction is not an option for these kinds of iterators.

20.3 Iterator Ranges

Iterators are valuable for specifying *ranges* over sequences (vectors and arrays). We can specify a range with a pair of iterators: one iterator points to the first element in the sequence, and the other iterator points to the position just past the end of the sequence. The global functions `std::begin` and `std::end` fit well into this definition of range.

Suppose `vec` is a vector. The two iterators `std::begin(vec)` and `std::end(vec)` span the complete range of values in `vec`. In the following statement:

```
auto b = std::begin(vec), e = b + 1;
```

the range of two iterators `b` and `e` includes just the first element in `vec`. Compare this range to the one represented by the following statement:

```
auto b = std::begin(vec), e = b + 10;
```

Here the range of two iterators `b` and `e` spans the first 10 elements of `vec`. The range of the iterators `b` and `e` defined by the statement

```
auto e = std::end(vec), b = e - 5;
```

encompasses the last five elements of `vec`.

Section 11.2 introduced a technique for traversing an array via a pointer range. Rather than passing an array (literally the address of the first element) and its size (number of elements) to a traversal function, the range-based approach passes the array (again, literally a pointer to the first element) and a pointer to the position just past the end of the array. Notice how the pointers in range-based array technique correspond directly to the iterator objects returned by the `std::begin` and `std::end` functions. Since iterator objects behave like pointers, we can write truly generic code that can process both arrays and vectors.

The following function uses iterators to count the number of times a value appears within a vector:

```
int count_value(std::vector<int>::iterator iter_begin,
                std::vector<int>::iterator iter_end, int seek) {
    int cnt = 0;
    for (std::vector<int>::iterator cursor = iter_begin;
         cursor != iter_end; iter++)
        if (*cursor == seek)
            cnt++;
    return cnt;
}
```

We can streamline its appearance using type aliasing:

```
using Iter = std::vector<int>::iterator;

int count_value(Iter iter_begin, Iter iter_end, int seek) {
    int cnt = 0;
    for (Iter iter_begin; cursor != iter_end; iter++)
        if (*cursor == seek)
            cnt++;
    return cnt;
}
```

Listing 20.12 (countitems.cpp) exercises the `count` function.

Listing 20.12: `countitems.cpp`

```
#include <iostream>
#include <vector>

using Iter = std::vector<int>::iterator;

// Count the elements in a vector of integers that match seek
int count_value(Iter iter_begin, Iter iter_end, int seek) {
    int cnt = 0;
    for (Iter cursor = iter_begin; cursor != iter_end; cursor++)
        if (*cursor == seek)
            cnt++;
    return cnt;
}

int main() {
    std::vector<int> a {34, 5, 12, 5, 8, 5, 11, 2};
    // Count multiple elements
    std::cout << count_value(std::begin(a), std::end(a), 5) << '\n';
    // Count single element
    std::cout << count_value(std::begin(a), std::end(a), 12) << '\n';
    // Count missing element
    std::cout << count_value(std::begin(a), std::end(a), 13) << '\n';
    a = {};  // Try an empty vector
    std::cout << count_value(std::begin(a), std::end(a), 5) << '\n';
}
```

Listing 20.12 (countitems.cpp) prints

```
3
1
0
0
```

This is its expected output.

The program works as expected, but we can expand its capability greatly by making it generic, as Listing 20.13 (genericcount.cpp) shows.

Listing 20.13: genericcount.cpp

```
#include <iostream>
#include <vector>
#include <array>
#include <list>
#include <string>

// Count the elements in a range that match seek.
// Type Iter is an iterator type working with a container that
// contains elements of type T.  Type T elements must be
// comparable with operator==.
template <typename Iter, typename T>
int count_value(Iter iter_begin, Iter iter_end, const T& seek) {
    int cnt = 0;
    for (auto cursor = iter_begin; cursor != iter_end; cursor++)
        if (*cursor == seek)
            cnt++;
    return cnt;
}

int main() {
    // Test with a vector of integers
    std::cout << "---Vector of integers--------------\n";
    std::vector<int> a {34, 5, 12, 5, 8, 5, 11, 2};
    std::cout << count_value(std::begin(a), std::end(a), 5) << '\n';
    a = {};  // Try an empty vector
    std::cout << count_value(std::begin(a), std::end(a), 5) << '\n';
    std::cout << count_value(std::begin(a), std::end(a), 8) << '\n';

    std::cout << "---STL array of integers-----------\n";
    // Test with a std::array of integers
    std::array<int, 8> arr {34, 5, 12, 5, 8, 5, 11, 2};
    std::cout << count_value(std::begin(arr), std::end(arr), 5) << '\n';
    arr = {};  // Try an empty array
    std::cout << count_value(std::begin(arr), std::end(arr), 5) << '\n';
    std::cout << count_value(std::begin(arr), std::end(arr), 8) << '\n';

    std::cout << "---Primitive C array of integers-----\n";
    // Test with a primitive C array of integers
    int carr[] = {34, 5, 12, 5, 8, 5, 11, 2};
    std::cout << count_value(std::begin(carr), std::end(carr), 5) << '\n';
    std::cout << count_value(std::begin(carr), std::end(carr), 8) << '\n';

    std::cout << "---Vector of strings---------------\n";
    // Test with a vector of strings
```

```
    std::vector<std::string> b {"mae", "al", "pat", "mel", "al",
                                "ray", "al"};
    std::cout << count_value(std::begin(b), std::end(b), "al") << '\n';
    b = {};
    std::cout << count_value(std::begin(b), std::end(b), "al") << '\n';
    std::cout << count_value(std::begin(b), std::end(b), "pat") << '\n';

    std::cout << "---Linked list of strings----------\n";
    // Test with a linked list of strings
    std::list<std::string> lst {"mae", "al", "pat", "mel", "al",
                                "ray", "al"};
    std::cout << count_value(std::begin(lst), std::end(lst), "al") << '\n';
    lst = {};
    std::cout << count_value(std::begin(lst), std::end(lst), "al") << '\n';
    std::cout << count_value(std::begin(lst), std::end(lst), "pat") << '\n';


    std::cout << "---Primitive C array of Points-----\n";
    struct Point {
        int x;
        int y;
        bool operator==(const Point& other) {
            return x == other.x && y == other.y;
        }
    };
    // Test with a primitive array of Point objects
    Point pts[] =  {{5, 3}, {0, 0}, {5, 3}, {3, 5}, {2, 1}};
    std::cout << count_value(std::begin(pts), std::end(pts), Point{5, 3})
              << '\n';
    std::cout << count_value(std::begin(pts), std::end(pts), Point{3, 5})
              << '\n';
    std::cout << count_value(std::begin(pts), std::end(pts), Point{2, 3})
              << '\n';
}
```

Listing 20.13 (genericcount.cpp) prints

```
---Vector of integers--------------
3
0
0
---STL array of integers-----------
3
0
0
---Primitive C array of integers-----
3
1
---Vector of strings---------------
3
0
0
---Linked list of strings----------
3
0
```

```
0
---Primitive C array of Points-----
2
1
0
```

As we can see, Listing 20.13 (genericcount.cpp) can count the number of occurrences of any type that supports testing for equality using the == operator in a container that supports iterators.

Listing 20.14 (genericloggingflexiblesort.cpp) uses template functions and template classes to make our logging flexible sort code truly generic. It also uses inheritance to derive a class from a template class.

Listing 20.14: genericloggingflexiblesort.cpp

```
#include <iostream>
#include <vector>
#include <cstdlib>
#include <fstream>
#include <string>

/*
 *  Comparer objects manage the comparisons and element
 *  interchanges on the selection sort function below.
 */
template <typename T>
class Comparer {
    // The object's data is private, so it is inaccessible to
    // clients and derived classes

    // Keeps track of the number of comparisons
    // performed
    int compare_count;

    // Keeps track of the number of swaps performed
    int swap_count;

    // Function pointer directed to the function to
    // perform the comparison
    bool (*comp)(const T&, const T&);

protected:
    // Method that actually performs the comparison
    // Derived classes may override this method
    virtual bool compare_impl(const T& m, const T& n);

    // Method that actually performs the swap
    // Derived classes may override this method
    virtual void swap_impl(T& m, T& n);

public:
    // The client must initialize a Comparer object with a
    // suitable comparison function.
    Comparer(bool (*f)(const T&, const T&));

    // Resets the counters to make ready for a new sort
```

```
    void reset();

    // Method that performs the comparison.  It delegates
    // the actual work to the function pointed to by comp.
    // This method logs each invocation.
    bool compare(const T& m, const T& n);

    // Method that performs the swap.
    // Interchange the values of
    // its parameters a and b which are
    // passed by reference.
    // This method logs each invocation.
    void swap(T& m, T& n);

    // Returns the number of comparisons this object has
    // performed since it was created.
    int comparisons() const;

    // Returns the number of swaps this object has
    // performed since it was created.
    int swaps() const;
};


// Method that actually performs the comparison
// Derived classes may override this method
template <typename T>
bool Comparer<T>::compare_impl(const T& m, const T& n) {
    return comp(m, n);
}

// Method that actually performs the swap
// Derived classes may override this method
template <typename T>
void Comparer<T>::swap_impl(T& m, T& n) {
    T temp = m;
    m = n;
    n = temp;
}

// The client must initialize a Comparer object with a
// suitable comparison function.
template <typename T>
Comparer<T>::Comparer(bool (*f)(const T&, const T&)):
      compare_count(0), swap_count(0), comp(f) {}

// Resets the counters to make ready for a new sort
template <typename T>
void Comparer<T>::reset() {
    compare_count = swap_count = 0;
}

// Method that performs the comparison.  It delegates
// the actual work to the function pointed to by comp.
// This method logs each invocation.
```

```
template <typename T>
bool Comparer<T>::compare(const T& m, const T& n) {
    compare_count++;
    return compare_impl(m, n);
}

// Method that performs the swap.
// Interchange the values of
// its parameters a and b which are
// passed by reference.
// This method logs each invocation, so
// we do not use std::swap here.
template <typename T>
void Comparer<T>::swap(T& m, T& n) {
    swap_count++;
    swap_impl(m, n);
}

// Returns the number of comparisons this object has
// performed since it was created.
template <typename T>
int Comparer<T>::comparisons() const {
    return compare_count;
}

// Returns the number of swaps this object has
// performed since it was created.
template <typename T>
int Comparer<T>::swaps() const {
    return swap_count;
}


/*
 *   selection_sort(a, compare)
 *       Arranges the elements of a sequence in an order determined
 *       by the compare object.
 *       begin points to the beginning of the sequence.
 *       end points to the imaginary element just past the last
 *       element of the sequence.
 *       compare is an object that compares the ordering of two
 *       elements and records the actions it performs.
 *       The function physically rearranges the contents of the
 *       sequence.
 */
template <typename T, typename V>
void selection_sort(const T begin, const T end, Comparer<V>& compare) {
    for (auto i = begin; i != end - 1; i++) {
        // Note: i, small, and j represent positions within the
        // sequence.
        // a[i], a[small], and a[j] represents the elements at
        // those positions.
        // small is the position of the smallest value we've seen
        // so far; we use it to find the smallest value less
        // than a[i]
```

```
        auto small = i;
        // See if a smaller value can be found later in the sequence
        for (auto j = i + 1; j != end; j++)
            if (compare.compare(*j, *small))
                small = j;  // Found a smaller value
        // Swap a[i] and a[small], if a smaller value was found
        if (i != small)
            compare.swap(*i, *small);
    }
}

/*
 *  print
 *     Prints the contents of a sequence
 *     begin points to the beginning of the sequence.
 *     end points to the imaginary element just past the last.
 *     The function does not modify the sequence.
 */
template <typename T>
void print(const T begin, const T end) {
    std::cout << '{';
    if (begin != end) {
        T iter = begin;
        std::cout << *iter;  // Print the first element
        iter++;         // Move to next element
        while (iter != end) {  // Print the rest
            std::cout << ',' << *iter;
            iter++;     // Move to next element
        }
    }
    std::cout << '}';
}

/*
 *  less_than(a, b)
 *     Returns true if a < b; otherwise, returns
 *     false.
 */
template <typename T>
bool less_than(const T& a, const T& b) {
    return a < b;
}

/*
 *  greater_than(a, b)
 *     Returns true if a > b; otherwise, returns
 *     false.
 */
template <typename T>
bool greater_than(const T& a, const T& b) {
    return a > b;
}


/*
```

```
 *  Comparer objects manage the comparisons and element
 *  interchanges on the selection sort function below.
 */
template <typename T>
class LogComparer: public Comparer<T> {
    // Output stream to which logging messages are directed
    std::ofstream fout;

protected:
    // Method that actually performs the comparison
    bool compare_impl(const T& m, const T& n) override;

    // Method that actually performs the swap
    void swap_impl(T& m, T& n) override;

public:
    // The client must initialize a LogComparer object with a
    // suitable comparison function and the file name of a text
    // file to which the object will direct logging messages
    LogComparer(bool (*f)(const T&, const T&),
                        const std::string& filename);

    // The destructor must close the log file
    ~LogComparer();
};

// Method that actually performs the comparison
// Derived classes may override this method
template <typename T>
bool LogComparer<T>::compare_impl(const T& m, const T& n) {
    fout << "Comparing " << m << " to " << n << '\n';
    // Base class method does the comparision
    return Comparer<T>::compare_impl(m, n);
}

// Method that actually performs the swap
// Derived classes may override this method
template <typename T>
void LogComparer<T>::swap_impl(T& m, T& n) {
    fout << "Swapping " << m << " and " << n << '\n';
    //T temp = m;
    //m = n;
    //n = temp;
    // Base class method does the swap
    Comparer<T>::swap_impl(m, n);
}

// The client must initialize a LogComparer object with a
// suitable comparison function and the file name of the
// text file to receive logging messages.
template <typename T>
LogComparer<T>::LogComparer(bool (*f)(const T&, const T&),
                        const std::string& filename): Comparer<T>(f) {
    fout.open(filename);
    if (!fout.good()) {
```

```
        std::cout << "Could not open log file " << filename
                  << " for writing\n";
        exit(1);  // Terminate the program
    }
    // fout is an instance variable, not a local variable,
    // so the file stays open when the constructor finishes
}


int main() {
   // Make a vector of integers
   std::vector<int> vec = { 23, -3, 10, 4, 215, 0, -3, 2 };

   std::cout << "Before:   ";
   print(std::begin(vec), std::end(vec));
   std::cout << '\n';
   LogComparer<int> lt(less_than<int>, "upsort.log");
   selection_sort(std::begin(vec), std::end(vec), lt);
   std::cout << "Ascending:  ";
   print(std::begin(vec), std::end(vec));
   std::cout << "  (" << lt.comparisons() << " comparisons, "
             << lt.swaps() << " swaps)\n";
   LogComparer<int> gt(greater_than<int>, "downsort.log");
   selection_sort(std::begin(vec), std::end(vec), gt);
   std::cout << "Descending:  ";
   print(std::begin(vec), std::end(vec));
   std::cout << "  (" << gt.comparisons() << " comparisons, "
             << gt.swaps() << " swaps)\n";

   std::cout << "-------------------------------------\n";

   // Make a vector of string objects
   std::vector<std::string> words{"tree", "girl", "boy", "dog",
                        "cat", "girl", "bird"};

   // Make a working copy of the original vector
   std::cout << "Before:   ";
   print(std::begin(words), std::end(words));
   std::cout << '\n';
   LogComparer<std::string> wlt(less_than<std::string>, "upwords.log");
   selection_sort(std::begin(words), std::end(words), wlt);
   std::cout << "Ascending:  ";
   print(std::begin(words), std::end(words));
   std::cout << "  (" << wlt.comparisons() << " comparisons, "
             << wlt.swaps() << " swaps)\n";
   LogComparer<std::string> wgt(greater_than<std::string>, "downwords.log");
   selection_sort(std::begin(words), std::end(words), wgt);
   std::cout << "Descending:  ";
   print(std::begin(words), std::end(words));
   std::cout << "  (" << wgt.comparisons() << " comparisons, "
             << wgt.swaps() << " swaps)\n";

   std::cout << "-------------------------------------\n";

   // Make an array of integers
```

```
    int arr[] = { 23, -3, 10, 4, 215, 0, -3, 2 };

    std::cout << "Before:   ";
    print(arr, arr + 8);
    std::cout << '\n';
    LogComparer<int> lt2(less_than<int>, "upsort2.log");
    selection_sort(arr, arr + 8, lt2);
    std::cout << "Ascending:  ";
    print(arr, arr + 8);
    std::cout << "  (" << lt2.comparisons() << " comparisons, "
              << lt.swaps() << " swaps)\n";
    LogComparer<int> gt2(greater_than<int>, "downsort2.log");
    selection_sort(arr, arr + 8, gt2);
    std::cout << "Descending:  ";
    print(arr, arr + 8);
    std::cout << "  (" << gt2.comparisons() << " comparisons, "
              << gt.swaps() << " swaps)\n";
}
```

Observe that the `selection_sort` function in Listing 20.14 (genericloggingflexiblesort.cpp) does not restrict the sequence it can process to be a vector or an array. The first two parameters of `selection_sort` are generic, and the code within the function treats these two parameters as if they are pointers. Since iterator objects behave exactly like pointers to the extent exercised within `selection_sort`, the compiler can instantiate the template to accept pointer or iterator arguments. Similarly, the `print` function works equally well with both pointer ranges for arrays and iterators for vectors. The `main` function demonstrates the flexibility of `print` and `selection_sort` functions by sending them both array pointer ranges and vector iterators. Behind the scenes the compiler will automatically instantiate two overloaded `print` functions and two overloaded `selection_sort` functions.

During its execution Listing 20.14 (genericloggingflexiblesort.cpp) produces the following output:

```
Before:   {23,-3,10,4,215,0,-3,2}
Ascending:  {-3,-3,0,2,4,10,23,215}  (28 comparisons, 5 swaps)
Descending:  {215,23,10,4,2,0,-3,-3}  (28 comparisons, 4 swaps)
-------------------------------------
Before:   {tree,girl,boy,dog,cat,girl,bird}
Ascending:  {bird,boy,cat,dog,girl,girl,tree}  (21 comparisons, 3 swaps)
Descending:  {tree,girl,girl,dog,cat,boy,bird}  (21 comparisons, 4 swaps)
-------------------------------------
Before:   {23,-3,10,4,215,0,-3,2}
Ascending:  {-3,-3,0,2,4,10,23,215}  (28 comparisons, 5 swaps)
Descending:  {215,23,10,4,2,0,-3,-3}  (28 comparisons, 4 swaps)
```

The code in Listing 20.14 (genericloggingflexiblesort.cpp) truly is generic:

- Clients may use vectors, arrays, and any sequence type that overloads the `begin` and `end` functions.
- Clients may work with sequences that contain elements of any type that supports the < and = operators.
- Clients may customize the element ordering.
- Clients may customize the behavior of the comparison and swapping procedures.

20.4 Lambda Functions

CAUTION! SECTION UNDER CONSTRUCTION

One of the primary benefits of functions is that we can write a function's code once and invoke it from many different places within the program (and even invoke it from other programs). Ordinarily, in order to call a function, we must know its name. Almost all the examples we have seen have invoked a function via its name. Listing 10.20 (arithmeticeval.cpp) in Section 10.10 provided examples of invoking functions without using their names directly. There we saw a function named `evaluate` that accepts a function as a parameter:

```
int evaluate(int (*f)(int, int), int x, int y) {
    return f(x, y);
}
```

The `evaluate` function calls `f`. The question is, what function does `evaluate` call? The name `f` refers to one of `evaluate`'s parameters; there is no separate function named `f` defined within Listing 10.20 (arithmeticeval.cpp). The answer, of course, is that `evaluate` invokes the function passed in from the caller. The function `main` in Listing 10.20 (arithmeticeval.cpp) calls `evaluate` passing the `add` function on one occasion and the `multiply` function on another.

The code in the `evaluate` function demands that callers send a function as the first parameter. Does this mean we have to write a separate function with a name in order to call `evaluate`? Once we create a function with a name it is available to be called from anywhere within the program after its definition or declaration. What if we want to ensure that our function will execute exactly one time and only when invoked by `evaluate`?

C++ supports the definition of anonymous functions via *lambda expressions*. The general form of a lambda expression is

[*capture list*] (*parameter list*) -> *return type* { *statements* }

where:

- *capture list* specifies the calling context to which the function has access (more on this follows)
- *parameter list* is a comma-separated list of parameters as you would find in any function definition
- *return type* is the type of the result the function returns
- *statements* are the statements as you would find in any function definition.

The term *lambda* comes from *lambda calculus* (see `http://en.wikipedia.org/wiki/Lambda_calculus`), a function-based mathematical system developed by Alonzo Church in the 1930s. Concepts from lambda calculus led to the development of the modern computer. The lambda calculus is the basis for modern functional languages like Haskell and F#.

We can use a lambda function in a call to the `evaluate` function:

```
int val = evaluate([](int x, int y)->int { return  x * y; }, 2, 3);
```

The `[...](...){...}` construct identifies a lambda function; thus, the first argument being passed to `evaluate` is indeed a function that takes two integer parameters. Notice that result of passing the lambda function here is the same as passing the `multiply` function from Listing 10.20 (arithmeticeval.cpp)—both compute the product of the two parameters.

When the compiler can deduce the type of the result of a lambda function, we can omit the return type:

```
int val = evaluate([](int x, int y) { return  x * y; }, 2, 3);
```

Here the compiler can deduce that an integer times an integer is an integer.

Given the `evaluate` function as before, the expression

```
evaluate([](int x, int y) { return  3*x + y; }, 10, 2);
```

evaluates to 32. The statement

```
[](int x, int y) { std::cout << x << " " << y << '\n'; } (10, 20);
```

prints

```
10 20
```

Observe that this statement defines the lambda function and explicitly invokes it.

The following two statements:

```
auto f = [](int x) { return  5*x; };
std::cout << f(10) << '\n';
```

show how we can assign a lambda function to a variable and invoke it through that variable at a later time.

One interesting aspect of lambda functions is that they can be used to create *closures*. A closure is a unit of code (in this case a function-like object) that can capture variables from its surrounding context. These captured variables then can be used outside of their original context. In order to demonstrate closure, we first must explore *function objects*.

A `std::function` object works like a function pointer (see Section 10.10). The `std::function` class is a generic class parameterized by a function's type specifications. A function object representing a function that accepts two integer parameters and returns an integer result can be declared as

```
std::function<int(int, int)> f;
```

Note the type parameter `int(int, int)` inside the angle brackets. The first `int` represents the function's return type. The two `int`s inside the parentheses specify the function's parameters. Function objects take the place of function pointers. Function pointers are available in C as well as C++, but C does not provide function objects. Function objects provide capabilities above simple function pointers, including the ability to manage closures.

Listing 20.15 (closurein.cpp) demonstrates a simple closure.

Listing 20.15: `closurein.cpp`

```
#include <iostream>
#include <functional>

int evaluate2(std::function<int(int, int)> f, int x, int y) {
    return f(x, y);
```

```
}

int main() {
    int a;
    std::cout << "Enter an integer: ";
    std::cin >> a;
    std::cout << evaluate2([a](int x, int y) {
                                if (x == a)
                                    x = 0;
                                else
                                    y++;
                                return x + y;
                            }, 2, 3) << '\n';
}
```

Note that `main` creates a function via the lambda expression that it passes to `evaluate2`. The variable `a` appears within the capture square brackets. This makes `a`'s value available to whatever code invokes the lambda function. In this case the `evaluate2` function can see `a`'s value even though `a` is local to `main` and is not passed to `evaluate2` as an explicit parameter.

We say the lambda function *captures* the variable `a`. When `evaluate2` invokes the function sent by the caller, `evaluate2` has no access to a variable named `a`. The `a` involved in the conditional expression is captured from `main`. This is an example of a closure transporting a captured variable into a function call.

For an example of a closure transporting a captured local variable out of a function, consider Listing 20.16 (makeadder.cpp) which includes a function that returns a lambda function to its caller.

Listing 20.16: `makeadder.cpp`

```
#include <iostream>
#include <functional>

std::function<int(int)> make_adder() {
    int loc_val = 2;   // Local variable definition
    return [loc_val](int x){ return x + loc_val; };  // Returns a function
}

int main() {
    auto f = make_adder();
    std::cout << f(10) << '\n';
    std::cout << f(2) << '\n';
}
```

Ordinarily when a function returns, all of its local variables disappear. This means that after the following statement in the `main` function executes:

```
auto f = make_adder();
```

`make_adder`'s `loc_var` local variable should no longer exist. The function that `make_adder` returns, however, uses `loc_var` in its computation. This means the function that `make_adder` returns forms a closure that captures `make_adder`'s local variable `loc_var`. In the output of Listing 20.16 (makeadder.cpp) we can see that function `f` still has knowledge of `loc_val`'s value:

```
12
4
```

C++'s lambda capture mechanism provides capabilities that go beyond the examples shown here; for example, the expression `&a` within the square brackets would capture variable `a` by reference, allowing an external context to modify the variable. Note that when a closure captures a variable by reference, that variable must exist in its original context as long as the closure is viable. This means you should not attempt to capture a local non-`static` variable of a function by reference by a closure to be returned by that function. The local variables disappear when the function returns, and so the references will refer to garbage values.

We can assign a variable to a lambda function; the following code fragment:

```
auto f = [](int x) { return 2*x; };
std::cout << f(10) << '\n';
```

will print 20. This gives the function a name, so the statement

```
auto f = [](int x) { return 2*x; };
```

is roughly equivalent to the function definition

```
int f(int x) {
    return 2*x;
}
```

While not a particularly useful application of lambda function, to demonstrate the regularity of the C++ language, we can define an anonymous function and invoke it immediately; for example, the statement

```
std::cout << [](int x, int y) {  return x * y; }(2, 3);
```

prints 6. In this case the statement

```
std::cout << 2*3 << '\n';
```

or, even better,

```
std::cout << 6 << '\n';
```

produces the same result much more simply.

As a more practical example using a lambda function and a closure, consider the calculation of a *derivative*. Those familiar with basic calculus will recall the derivative of a function f at a is defined to be

$$f'(a) = \lim_{h \to 0} \frac{f(a+h) - f(a)}{h}$$

If you are unfamiliar with calculus, all you need to know is that the derivative of a function is itself a function; the above formula shows how to transform a function into its derivative. The process of computing a derivative is known as *differentiation*. The $\lim_{h \to 0}$ notation indicates that the answer becomes more precise as the value h gets closer to zero. Letting h be exactly zero would result in division by zero, which is undefined. The trick is to make h as small as possible, keeping in mind that the computer's floating-point values have limitations.

Based on the mathematical definition we can define a C++ function that computes the derivative of another function, as shown here:

```
std::function<double(double)> derivative(std::function<double(double)> f,
                                         double h) {
```

```
    // Capture function f and h value and return the function's derivative
    return [f, h] (double x) { return (f(x + h) - f(x)) / h; };
}
```

Note that the `derivative` function returns a function, as a `lambda` expression is a simple function definition. The function that `derivative` returns is a closure because it captures the function parameters `f` and `h`.

Our `derivative` function allows us to compute the derivative of a function at a given value. This is known as *numerical differentiation*. Another approach (the one emphasized in calculus courses) uses *symbolic differentiation*. Symbolic differentiation transforms the formula for a function into a different formula. The details of symbolic differentiation are beyond the scope of this text, but we will use one of its results for a particular function to check our computed numerical results.

A particular function f is defined as follows:

$$f(x) = 3x^2 + 5$$

If you have studied calculus, you can confirm that f's derivative, f', is:

$$f'(x) \;=\; 6x$$

Without a knowledge of calculus, simply just accept this as the correct answer so we can test our `derivative` function.

Listing 20.17 (derivative.cpp) uses the `derivative` function on $f(x) = 3x^2 + 5$ and compares its results with the known solution, $f'(x) = 6x$.

Listing 20.17: `derivative.cpp`

```
#include <iostream>
#include <iomanip>
#include <functional>

// Approximates the derivative of function f given an h value.
// The closer h is to zero, the better the estimate.
std::function<double(double)> derivative(std::function<double(double)> f,
                                         double h) {
    // Capture function f and h value
    return [f, h] (double x) { return (f(x + h) - f(x)) / h; };
}

double fun(double x) {   // The function we wish to differentiate
    return 3*x*x + 5;
}

double ans(double x) {   // The known derivative of function fun
    return 6*x;
}

int main() {
    // Difference: Approximation better as h -> 0
    double h = 0.0000001;

    // Compute the function representing an approximation
    // of the derivative of function fun
```

```
    auto der = derivative(fun, h);

    // Compare the computed derivative to the exact derivative
    // derived symbolically
    double x = 5.0;
    std::cout << "-------------------------------------------------------\n";
    std::cout << "                                    Approx.    Actual \n";
    std::cout << "   x        f(x)          h         f\'(x)      f\'(x)\n";
    std::cout << "-------------------------------------------------------\n";
    while (x < 5.1) {
        std::cout << std::fixed << std::showpoint << std::setprecision(5);
        std::cout << x << "   " << fun(x) << "     " << h << "    " << der(x)
                  << "   " << ans(x) << '\n';
        x += 0.01;
    }
}
```

With $h = 0.0000001$, Listing 20.17 (derivative.cpp) produces good results to the fifth decimal place:

```
-------------------------------------------------------
                                    Approx.    Actual
   x        f(x)          h         f'(x)      f'(x)
-------------------------------------------------------
5.00000   80.00000     0.00000    30.00000   30.00000
5.01000   80.30030     0.00000    30.06000   30.06000
5.02000   80.60120     0.00000    30.12000   30.12000
5.03000   80.90270     0.00000    30.18000   30.18000
5.04000   81.20480     0.00000    30.24000   30.24000
5.05000   81.50750     0.00000    30.30000   30.30000
5.06000   81.81080     0.00000    30.36000   30.36000
5.07000   82.11470     0.00000    30.42000   30.42000
5.08000   82.41920     0.00000    30.48000   30.48000
5.09000   82.72430     0.00000    30.54000   30.54000
5.10000   83.03000     0.00000    30.60000   30.60000
```

Even with h as large as 0.01, the results are not too bad:

```
-------------------------------------------------------
                                    Approx.    Actual
   x        f(x)          h         f'(x)      f'(x)
-------------------------------------------------------
5.00000   80.00000     0.01000    30.03000   30.00000
5.01000   80.30030     0.01000    30.09000   30.06000
5.02000   80.60120     0.01000    30.15000   30.12000
5.03000   80.90270     0.01000    30.21000   30.18000
5.04000   81.20480     0.01000    30.27000   30.24000
5.05000   81.50750     0.01000    30.33000   30.30000
5.06000   81.81080     0.01000    30.39000   30.36000
5.07000   82.11470     0.01000    30.45000   30.42000
5.08000   82.41920     0.01000    30.51000   30.48000
5.09000   82.72430     0.01000    30.57000   30.54000
5.10000   83.03000     0.01000    30.63000   30.60000
```

In Listing 20.17 (derivative.cpp), the statement

```
auto der = derivative(fun, h);
```

assigns `der` to the function returned by `derivative`; thus, `der(x)` returns the value of the derivative of `fun` at `x`. In order for `der` to compute its answer, it must have access to function `fun`. It has this access because `der` is a lambda function that captured `fun` and `h` during the call to `derivative`.

20.5 Algorithms in the Standard Library

CAUTION! SECTION UNDER CONSTRUCTION

The flexible design of the STL containers and their iterators make possible the STL's generic algorithms. The algorithms provided by the STL are powerful and flexible, but they also are somewhat arcane and can be difficult to use for casual C++ programmers. The STL algorithms are best viewed as the building blocks for more specific tasks required by applications.

In order to use the STL algorithms you must include the following preprocessor directive:

```
#include <algorithm>
```

One relatively simple algorithm is `std::for_each`. The `for_each` function applies a unary function to each element in the container. Like the other algorithms we will see, `for_each` uses an iterator to drive a loop behind the scenes. The following statement prints out each element in a `std::list` object called `seq` holding integers:

```
std::for_each(std::begin(seq), std::end(seq),
              [](int x) { std::cout << x << ' '; });
```

If `sec` were instead a `std::vector` of integers, this statement will work equally well with no changes. That is the beauty of the STL algorithms: many of them work seamlessly on a variety of the different containers provided by the STL.

Listing 20.18 (incelements.cpp) increases by one the value of every element in a list.

Listing 20.18: `incelements.cpp`

```
#include <iostream>
#include <list>
#include <algorithm>

int main() {
    // Make a list of integers
    std::list<int> seq{5, 22, 6, -3, 8, 4};
    // Display the vector
    std::for_each(std::begin(seq), std::end(seq),
                  [](int x) { std::cout << x << ' '; });
    std::cout << '\n';
    // Increase each element in the vector by 1
    std::for_each(std::begin(seq), std::end(seq),
                  [](int& x) { x++; });
    // Redisplay the vector
    std::for_each(std::begin(seq), std::end(seq),
                  [](int x) { std::cout << x << ' '; });
    std::cout << '\n';
```

```
}
```

Listing 20.18 (incelements.cpp) prints the following:

```
5 22 6 -3 8 4
6 23 7 -2 9 5
```

In the statement

```
std::for_each(std::begin(seq), std::end(seq), [](int& x) { x++; });
```

note that the lambda function passes the parameter `x` by reference; this allows the function to modify each element in the sequence.

We can use lambda capture to compute the sum of the elements in a sequence, as shown in Listing 20.19 (sumvec.cpp).

Listing 20.19: `sumvec.cpp`

```
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    int sum = 0;
    std::vector<int> vec{5, 22, 6, -3, 8, 4};
    for_each(std::begin(vec), std::end(vec), [&sum](int x) {  sum += x; });
    std::cout << "The sum is " << sum << '\n';
}
```

Listing 20.19 (sumvec.cpp) prints

```
The sum is 42
```

The lambda capture of `sum` by reference allows the `for_each` function to modify `sum` as it iterates over the vector `vec`.

The `std::iota` function, delared in the `<numerics>` header, is a simple but handy function that fills a container with ascending numbers. The `std::iota` function allows us to replace the following code:

```
// Make a vector to hold 1,000 elements
std::vector<int> seq(1000);
// Populate the vector with 0, 1, 2, 3, ..., 999
int count = 0;
for (auto& elem : seq)
    elem = count++;
```

with

```
// Make a vector to hold 1,000 elements
std::vector<int> seq(1000);
// Populate the vector with 0, 1, 2, 3, ..., 999
std::iota(std::begin(seq), std::end(seq), 0);
```

(The function's non-descriptive name comes from the Greek letter ι, a function from the APL programming language that fills a vector with ascending numbers.)

The `std::find` algorithm locates an element within a container. Given a range within a container specified with iterators, the find function returns an iterator to the sought object. If the object is not present in the container, the function returns the iterator object just past the end of the container. Listing 20.20 (testfind.cpp) tests the find function.

Listing 20.20: `testfind.cpp`

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>

int main() {
    // Make a vector of 1,000 integers
    std::vector<int> seq(1000, 0);
    // Populate the vector with 0, 1, 2, 3, ..., 999
    std::iota(std::begin(seq), std::end(seq), 0);

    // Look for 678
    auto iter = std::find(std::begin(seq), std::end(seq), 678);
    // Do we find it?
    if (iter != std::end(seq))
        std::cout << *iter << " is present" << '\n';
    else
        std::cout << "678 is NOT present" << '\n';

    // Look for -200
    iter = std::find(std::begin(seq), std::end(seq), -200);
    // Do we find it?
    if (iter != std::end(seq))
        std::cout << *iter << " is present" << '\n';
    else
        std::cout << "-200 is NOT present" << '\n';
}
```

Listing 20.20 (testfind.cpp) prints

```
678 is present
-200 is NOT present
```

The `std::copy` function copies the elements from one container to another. The `std::transform` function works like `std::copy` except it expects a function that that can modify the copied elements. Both functions require the begin and end iterators from the source container and the begin iterator of the destination container. Listing 20.21 (testtransform.cpp) illustrates the use of `std:copy` and `std::transform`.

Listing 20.21: `testtransform.cpp`

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>
```

```
int main() {
    const int SIZE = 20;

    // Make a vector of SIZE integers
    std::vector<int> seq(SIZE);
    // Populate the vector with 0, 1, 2, 3, ..., SIZE - 1
    std::iota(std::begin(seq), std::end(seq), 0);

    // Display the vector
    std::for_each(std::begin(seq), std::end(seq),
                  [](int x) { std::cout << x << ' '; });
    std::cout << '\n';

    // Make a vector large enough to hold the copied values
    std::vector<int> seq2(SIZE);

    // Copy the seq to seq2
    std::copy(std::begin(seq), std::end(seq), std::begin(seq2));

    // Display seq2
    std::for_each(std::begin(seq2), std::end(seq2),
                  [](int x) { std::cout << x << ' '; });
    std::cout << '\n';

    // Make a vector large enough to hold the transformed values
    std::vector<int> seq3(SIZE);

    // Copy the seq to seq3
    std::transform(std::begin(seq), std::end(seq), std::begin(seq3),
                   [](int n) { return 2*n; });

    // Display seq3
    std::for_each(std::begin(seq3), std::end(seq3),
                  [](int x) { std::cout << x << ' '; });
    std::cout << '\n';
}
```

Listing 20.21 (testtransform.cpp) prints

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38
```

The `std::copy` function may seem superfluous since `operator=` for vectors also performs a vector copy. With direct assignment there is no need to ensure the receiving vector has enough space; the assignment operator takes care of this detail. The `std::copy` function, however, is much more flexible than `operator=`, as Listing 20.22 (trimvector.cpp) illustrates.

Listing 20.22: `trimvector.cpp`

```
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
```

```
    // Make a vector of SIZE integers
    std::vector<int> seq { 2, 3, 4, 5, 6 };

    // Display seq
    std::for_each(std::begin(seq), std::end(seq),
                  [](int x) { std::cout << x << ' '; });
    std::cout << '\n';

    // Make a copy of vec with the first and last element trimmed off
    if (seq.size() >= 2) {
        // Make a vector large enough to hold trimmed values
        std::vector<int> seq2(seq.size() - 2);
        std::copy(std::begin(seq) + 1, std::end(seq) - 1,
                  std::begin(seq2));
        // Display seq2
        std::for_each(std::begin(seq2), std::end(seq2),
                      [](int x) { std::cout << x << ' '; });
        std::cout << '\n';
    }
}
```

Listing 20.22 (trimvector.cpp) makes a copy of a vector with the first and last elements trimmed off, as shown in its output:

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

Note how Listing 20.22 (trimvector.cpp) advances the begin iterator of the source vector and decreases the end iterator to restrict the range of values it copies.

The `std::transform` function is handy for capitalizing all the letters in a `std::string`. Listing 20.23 (uppercasestring.cpp) applies the C function `toupper` to all the characters in a string.

Listing 20.23: `uppercasestring.cpp`

```
#include <iostream>
#include <string>
#include <algorithm>
#include <cctype>

int main() {
    std::string name = "Fred",
                str = "abcDEF-GHIjkl345qw";

    std::cout << "Before: " << name << "   " << str << '\n';
    // Uppercase the strings
    std::transform(std::begin(name), std::end(name),
                   std::begin(name), std::toupper);
    std::transform(std::begin(str), std::end(str),
                   std::begin(str), std::toupper);
    std::cout << "After : " << name << "   " << str << '\n';
}
```

Listing 20.23 (uppercasestring.cpp) prints

```
Before: Fred   abcDEF-GHIjkl345qw
After : FRED   ABCDEF-GHIJKL345QW
```

The standard library provides an special iterator, `std::ostream_iterator`, that enables the `std::copy` function to copy the contents of a container to an output stream instead of another container. Listing 20.24 (copytostream.cpp) shows how this special iterator works.

Listing 20.24: `copytostream.cpp`

```
#include <iostream>
#include <vector>
#include <iterator>

int main() {
    std::vector<int> vec { 10, 20, 30, 35, 40, 45, 50, 55 };

    // Copy the contents of the container to std::cout, separating
    // elements with a single space
    auto strm = std::ostream_iterator<int>(std::cout, " ");

    std::copy(std::begin(vec), std::end(vec), strm);
    std::cout << '\n';
}
```

Listing 20.24 (copytostream.cpp) prints the following to the console:

```
10 20 30 35 40 45 50 55
```

Note that `std::ostream_iterator` is a generic class parameterized by the type of object the output stream is to receive.

Listing 20.25 (copytofile.cpp) shows how we can use `std::copy` to just as easily save the contents of a container to a text file.

Listing 20.25: `copytofile.cpp`

```
#include <fstream>
#include <vector>
#include <iterator>

int main() {
    std::vector<int> vec { 10, 20, 30, 35, 40, 45, 50, 55 };

    // Open a text file for writing
    std::ofstream fout("output.txt");

    if (fout.good()) {  // Confirm the file is ready to receive output
        // Copy the contents of the container to a text file,
        // separating elements with a single space
        auto strm = std::ostream_iterator<int>(fout, " ");
        std::copy(std::begin(vec), std::end(vec), strm);
        fout << '\n';
    }
}
```

Note that the two statements

```
auto strm = std::ostream_iterator<int>(fout, " ");
std::copy(std::begin(vec), std::end(vec), strm);
```

may be colapsed into a single statement, as

```
std::copy(std::begin(vec), std::end(vec),
          std::ostream_iterator<int>(fout, " "));
```

thus avoiding the extra variable.

The `std::count` serves the role of our `count_value` function from Listing 20.13 (genericcount.cpp). A similar function, `std::count_if`, counts the number of elements in a container that possess a certain property; for example, the following code will print the number of values in `vec` that are even:

```
// vec is a vector of integers
std::cout << std::count_if(std::begin(vec), std::end(vec),
                           [](int n) { return n % 2 == 0; });
```

The last argument to `std::count_if` is a function that accepts a single value of the type the container is declared to hold. The function returns `true` or `false`. A function that returns a Boolean result is known as a *predicate*. Some STL algorithms, like `std::count_if`, expect a predicate to allow them to process only elements that satisfy the predicate. As another example, the `std::copy_if` function copies only elements from one container to another that satisfy a given predicate. Listing 20.26 (copyevens.cpp) uses `std::count_if` and `std::copy_if` to copy relevant portions of vectors.

Listing 20.26: `copyevens.cpp`

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>

int main() {
    // Make a vector of SIZE integers
    std::vector<int> seq { 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };

    auto output = std::ostream_iterator<int>(std::cout, " ");

    // Display seq
    std::copy(std::begin(seq), std::end(seq), output);
    std::cout << '\n';

    // A function to test for evenness
    auto is_even = [](int n) { return n % 2 == 0; };

    // Count the number of even integers in seq
    int even_count = count_if(std::begin(seq), std::end(seq), is_even);

    // Make a copy of vec omitting all the odd numbers
    std::vector<int> seq2(even_count);
    std::copy_if(std::begin(seq), std::end(seq), std::begin(seq2), is_even);
    // Display seq2
    copy(std::begin(seq2), std::end(seq2), output);
    std::cout << '\n';
```

```
}
```

Listing 20.26 (copyevens.cpp) prints

```
2 3 4 5 6 7 8 9 10 11 12
2 4 6 8 10 12
```

Observe that in Listing 20.26 (copyevens.cpp) we assigned the lambda functions to the variables `print` and `is_even`. This is so we did not have to write the same lambda expressions twice, for printing `seq` and `seq2` and for testing evenness in for `count_if` and `copy_if`. Predicates do not have to be lambda functions—they can be global named functions as well, but lambda function often are more convenient.

In Section 12.4 we wrote a function to randomly permute a vector. The STL provides a function that does the work for us. Listing 20.27 (testshuffle.cpp) shows how to permute a vector using `std::shuffle`.

Listing 20.27: `testshuffle.cpp`

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>
#include <iterator>
#include <random>

int main() {
    std::vector<int> vec(20);

    auto output = std::ostream_iterator<int>(std::cout, " ");

    // Sequence is 0, 1, 2, ..., 19
    std::iota(std::begin(vec), std::end(vec), 0);

    // Confirm original order
    std::copy(std::begin(vec), std::end(vec), output);
    std::cout << '\n';

    std::random_device dev;
    std::mt19937 generator(dev());

    // Permute the vector
    std::shuffle(std::begin(vec), std::end(vec), generator);

    // Confirm the permutation
    std::copy(std::begin(vec), std::end(vec), output);
    std::cout << '\n';
}
```

Listing 20.27 (testshuffle.cpp) prints the vector before and after its permutation. The following shows the output of one run of Listing 20.27 (testshuffle.cpp):

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
5 18 8 13 10 16 14 0 15 2 6 11 19 9 12 3 1 17 4 7
```

Note that the `std::shuffle` function applies only to containers that support random access. The `std::list`

class, for example, has no `operator[]` method and does not support random access. This means you cannot permute the elements in a `std::list` object using `std::shuffle`.

We can use the `std::generate` function to populate a container based on a *generating function*. A generating function typically returns a different value each time it is called. Listing 20.28 (testgenerate.cpp) uses `std::generate` to

1. fill a container with identical values,
2. fill a container with a sequence produced from a formula, and
3. fill a container with pseudorandom values.

Listing 20.28: `testgenerate.cpp`

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>
#include <random>

// Uniform pseudorandom number generator from an earlier chapter
#include "uniformrandom.h"

int main() {
    std::vector<int> nums(20);

    UniformRandomGenerator gen(0, 100);

    auto output = std::ostream_iterator<int>(std::cout, " ");

    // First, populate the vector with the value 2
    std::generate(std::begin(nums), std::end(nums), [](){ return 2;});
    std::copy(std::begin(nums), std::end(nums), output);
    std::cout << '\n';

    // Next, populate the vector with a formulaic sequence
    int i = 10;
    std::generate(std::begin(nums), std::end(nums),
                  [&i](){
                      int result = i;
                      if (i % 10 == 0)
                          i += 5;
                      else
                          i++;
                      return result;
                  });
    std::copy(std::begin(nums), std::end(nums), output);
    std::cout << '\n';

    // Finally, populate the vector with pseudorandom
    // integers in the range 0, 1, ..., 100
    std::generate(std::begin(nums), std::end(nums), gen);
    std::copy(std::begin(nums), std::end(nums), output);
    std::cout << '\n';
```

```
}
```

One run of Listing 20.28 (testgenerate.cpp) produces the following output:

```
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
10 15 16 17 18 19 20 25 26 27 28 29 30 35 36 37 38 39 40 45
59 55 55 40 53 1 48 91 10 13 65 78 52 28 17 23 30 12 0 20
```

The first two lines of the output will always be the same, but the last line will vary from one run to the next.

The `std::accumulate` function applies a binary operation to all the elements of a container to produce a single value. (A binary operation is an operator or function that expects two arguments.) One example of its use is adding up the elements in a numeric list. Listing 20.29 (testaccumulate.cpp) shows the `std::accumulate` function in action.

Listing 20.29: `testaccumulate.cpp`

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <numeric>
#include <iterator>
#include <random>

// Uniform pseudorandom number generator from an earlier chapter
#include "uniformrandom.h"

int main() {
    std::vector<int> nums(6);

    UniformRandomGenerator gen(1, 5);

    auto output = std::ostream_iterator<int>(std::cout, " ");

    // Populate the vector with pseudorandom
    // integers in the range 0, 1, ..., 10
    std::generate(std::begin(nums), std::end(nums), gen);
    std::copy(std::begin(nums), std::end(nums), output);
    std::cout << '\n';

    // Add up the elements
    std::cout << std::accumulate(std::begin(nums), std::end(nums), 0,
                        [](int a, int b) { return a + b; })
              << '\n';

    // Multiply the elements
    std::cout << std::accumulate(std::begin(nums), std::end(nums), 1,
                        [](int a, int b) { return a * b; })
              << '\n';
}
```

One run of Listing 20.29 (testaccumulate.cpp) produces the following output:

```
2 1 3 4 1 2
```

```
13
48
```

This is the expected output, since $2+1+3+4+1+2=13$, and $2\times1\times3\times4\times1\times2=48$.

It is possible to use `std::accumulate` on containers holding nonnumeric values; for example, the following code

```
std::vector<std::string> words { "fred", "ella", "jo", "sam", "pat" };
std::copy(std::begin(words), std::end(words),
          std::ostream_iterator<std::string>(std::cout, " "));
std::cout << '\n';
// Concatenate the elements
std::cout << std::accumulate(std::begin(words), std::end(words),
                   std::string(""),
                   [](std::string a, std::string b) { return a + b; })
          << '\n';
```

will concatenate all the strings in the vector to produce the following output:

```
fred ella jo sam pat
fredellajosampat
```

While it may seem good in theory, this approach is not efficient and should be avoided. Each application of the `operator+` concatenation operator creates a new `std::string` object from two existing `std::string` objects. Especially for longer lists, this repeated string creation process unnecessarily wastes time and space. For a container that contains n strings, the `std::accumulate` function will create $n-1$ new string objects on its way to produce the final accumulated string result. Numeric arithmetic does not create new objects, and so `std::accumulate` is best used on containers holding numbers. (That is why its declaration appears in the `<numeric>` header.)

It you need to concatenate the strings in a container to make one big string, you should use `operator+=`. This operator appends characters onto the end of an existing string object, thus avoiding the creation of new string objects. In this case a simple familiar `for` loop is preferable to a sophisticated STL algorithm. The following code illustrates a more efficient solution:

```
std::vector<std::string> words { "fred", "ella", "jo", "sam", "pat" };
std::copy(std::begin(words), std::end(words),
          std::ostream_iterator<std::string>(std::cout, " "));
std::cout << '\n';
// Concatenate the elements
std::string joined = "";
for (const std::string& word : words)
    joined += word;
std::cout << joined << '\n';
```

This version produces the same results, but it executes faster and uses less memory. If the number of strings is small, it does not make much difference. It is only when the collection of strings to concatenate becomes large that we will detect a performance difference. Listing 20.30 (teststrconcat.cpp) tests the two algorithms.

Listing 20.30: `teststrconcat.cpp`

```
#include <iostream>
```

```
#include <vector>
#include <string>
#include <algorithm>
#include <numeric>
#include <iterator>
#include <random>
#include <ctime>

// Uniform pseudorandom number generator from an earlier chapter
#include "uniformrandom.h"

// Some global resources

// Valid characters in our words
const std::string character_set = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

// Our words will contain 1-8 letters
UniformRandomGenerator word_len_gen(1, 8);
// We will select from a random index in character_set
UniformRandomGenerator char_chooser_gen(0, 25);

std::string make_random_string() {
    std::string result = "";
    int size = word_len_gen();  // Length of this word
    for (int i = 0; i < size; i++)
        result += character_set[char_chooser_gen()];
    return result;
}

std::vector<std::string> make_random_string_vector(int size) {
    std::vector<std::string> result(size);
    for (auto& elem : result)
        elem = make_random_string();
    return result;
}

std::string concat1(const std::vector<std::string>& words) {
    return std::accumulate(std::begin(words), std::end(words),
              std::string(""),
              [](std::string a, std::string b) { return a + b; });
}

std::string concat2(const std::vector<std::string>& words) {
    std::string joined = "";
    for (const std::string& word : words)
        joined += word;
    return joined;
}

int main() {
    auto words = make_random_string_vector(25);

    copy(std::begin(words), std::end(words),
               std::ostream_iterator<std::string>(std::cout, " "));
    std::cout << '\n';
```

```
    std::cout << "========================\n";
    clock_t start_time, stop_time;
    start_time = clock();
    auto s = concat1(words);
    std::cout << s << '\n';
    stop_time = clock();
    std::cout << "Accumulate: " << stop_time - start_time << " msec\n";
    std::cout << "------------------------\n";
    start_time = clock();
    s = concat2(words);
    std::cout << s << '\n';
    stop_time = clock();
    std::cout << "for loop:   " << stop_time - start_time << " msec\n";
}
```

Listing 20.30 (teststrconcat.cpp) compares the two algorithms on a vector containing 25 strings generated at random with lengths ranging from 1 to 8 letters. Its output on one run is

```
IQW WOP V WVAQZGNG M BJMBBIM T GS HBIPVLUU ZAIN NODPNQDR TO SUHV QY K M O LAY
========================
IQWWOPVWVAQZGNGMBJMBBIMTGSHBIPVLUUZAINNODPNQDRTOSUHVQYKMOLAYDOPKQHDWOAEVFZGBG
Accumulate: 2 msec
------------------------
IQWWOPVWVAQZGNGMBJMBBIMTGSHBIPVLUUZAINNODPNQDRTOSUHVQYKMOLAYDOPKQHDWOAEVFZGBG
for loop:   1 msec
```

Some runs show `std::accumulate` to be faster, but the differences are usually single-digit milliseconds. If crank up the size of the vector to 200,000 and comment out the statements that print the strings, we see a more significant difference:

```
========================
Accumulate: 55805 msec
------------------------
for loop:   34 msec
```

Here `std::accumulate` requires almost one minute to build the string, while the `for` loop with `operator+=` takes less than one-tenth of a second.

The `std::partition` function reorders the elements in a container so that all the elements that satisfy a predicate will appear before the elements that do not meet satisfy the predicate. Listing 20.31 (testpartition.cpp) rearranges the elements in a vector of integers so that all the even numbers appear before any odd number. It also partitions a list of strings so that words that contain more than three letters appear before any word with three or fewer letters.

Listing 20.31: `testpartition.cpp`

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <iterator>
#include <random>

// Uniform pseudorandom number generator from an earlier chapter
```

```
#include "uniformrandom.h"

int main() {
    std::vector<int> nums(25);

    UniformRandomGenerator gen(0, 100);

    auto int_output = std::ostream_iterator<int>(std::cout, " ");

    // Populate the vector with pseudorandom
    // integers in the range 0, 1, ..., 100
    std::generate(std::begin(nums), std::end(nums), gen);

    std::cout << "Original sequence\n";
    std::copy(std::begin(nums), std::end(nums), int_output);
    std::cout << '\n';
    std::cout << "-----------------\n";
    std::cout << "Partitioned sequence\n";
    std::partition(std::begin(nums), std::end(nums),
                   [](int n) { return n % 2 == 0; });
    std::copy(std::begin(nums), std::end(nums), int_output);
    std::cout << '\n';
    std::cout << "=================\n\n";

    auto string_output = std::ostream_iterator<std::string>(std::cout, " ");
    std::vector<std::string> words { "fred", "ella", "adam", "jo",
                                     "pat", "mel", "anna", "ed",
                                     "oscar", "will", "tom", "ingrid" };
    std::cout << "Original sequence\n";
    std::copy(std::begin(words), std::end(words), string_output);
    std::cout << '\n';
    std::cout << "-----------------\n";
    std::cout << "Partitioned sequence" << '\n';
    std::partition(std::begin(words), std::end(words),
                   [](const std::string& w ) { return w.length() > 3; });
    std::copy(std::begin(words), std::end(words), string_output);
    std::cout << '\n';
}
```

Listing 20.31 (testpartition.cpp) prints

```
Original sequence
81 80 2 63 13 25 12 7 8 16 84 91 56 91 56 7 59 94 18 68 68 71 78 66 60
-----------------
Partitioned sequence
60 80 2 66 78 68 12 68 8 16 84 18 56 94 56 7 59 91 91 7 25 71 13 63 81
=================

Original sequence
fred ella adam jo pat mel anna ed oscar will tom ingrid
-----------------
Partitioned sequence
fred ella adam ingrid will oscar anna ed mel pat tom jo
```

Note that `std::partition` does not attempt to preserve the original relative order of elements within a

partition. Also, `std::partition` does not make any new string objects. It simply moves existing string around within the container. This makes it as efficient for strings as for simple numbers.

The `std::merge` function merges two ordered containers into a third container. The resulting container will be ordered. Listing 20.32 (testmerge.cpp) shows an example of `testmerge.cpp`.

Listing 20.32: `testmerge.cpp`

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>
#include <random>

// Uniform pseudorandom number generator from an earlier chapter
#include "uniformrandom.h"

int main() {
    std::vector<int> nums1(15),
                     nums2(13);

    UniformRandomGenerator gen1(0, 40),
                           gen2(0, 40);

    auto output = std::ostream_iterator<int>(std::cout, " ");

    // Populate the vectors with pseudorandom
    // integers in the range 0, 1, ..., 50
    std::generate(std::begin(nums1), std::end(nums1), gen1);
    std::generate(std::begin(nums2), std::end(nums2), gen2);

    std::cout << "Original sequences\n";
    std::copy(std::begin(nums1), std::end(nums1), output);
    std::cout << '\n';
    std::cout << "-----------------\n";
    std::copy(std::begin(nums2), std::end(nums2), output);
    std::cout << '\n';
    std::cout << "=================\n\n";

    std::sort(std::begin(nums1), std::end(nums1));
    std::sort(std::begin(nums2), std::end(nums2));
    std::cout << "Sorted sequences\n";
    std::copy(std::begin(nums1), std::end(nums1), output);
    std::cout << '\n';
    std::cout << "-----------------\n";
    std::copy(std::begin(nums2), std::end(nums2), output);
    std::cout << '\n';
    std::cout << "=================\n\n";

    // Merge the sequences
    std::vector<int> merged(nums1.size() + nums2.size());
    std::merge(std::begin(nums1), std::end(nums1),
               std::begin(nums2), std::end(nums2),
               std::begin(merged));
    std::cout << "Merged sequence\n";
    std::copy(std::begin(merged), std::end(merged), output);
```

```
    std::cout << '\n';
}
```

One run of Listing 20.32 (testmerge.cpp) prints the following:

```
Original sequences
8 26 8 17 26 23 31 7 16 30 5 21 37 20 4
-----------------
29 0 22 32 18 4 36 26 38 21 14 34 32
=================

Sorted sequences
4 5 7 8 8 16 17 20 21 23 26 26 30 31 37
-----------------
0 4 14 18 21 22 26 29 32 32 34 36 38
=================

Merged sequence
0 4 4 5 7 8 8 14 16 17 18 20 21 21 22 23 26 26 26 29 30 31 32 32 34 36 37 38
```

We can use the `std::remove` and `std::remove_if` functions to remove elements from a container. Unlike the other algorithms we have seen so far, these functions work in an unexpected way given their names. Neither `remove` nor `remove_if` by themselves achieve the results we generally desire.

Listing 20.33 (removeonly.cpp) shows what happens when we attempt to use `remove` or `remove_if` in isolation.

Listing 20.33: `removeonly.cpp`

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>
#include <iterator>

bool is_even(int n) {
    return n % 2 == 0;
}

int main() {
    std::vector<int> nums(20);
    auto output = std::ostream_iterator<int>(std::cout, " ");

    // Fill the vector with 0, 1, ..., 19
    std::iota(std::begin(nums), std::end(nums), 0);
    std::copy(std::begin(nums), std::end(nums), output);
    std::cout << '\n';

    // Remove 10
    std::remove(std::begin(nums), std::end(nums), 10);
    std::copy(std::begin(nums), std::end(nums), output);
    std::cout << '\n';

    // Remove remaining even numbers
    std::remove_if(std::begin(nums), std::end(nums), is_even);
```

```
    copy(std::begin(nums), std::end(nums), output);
    std::cout << '\n';
}
```

The output of Listing 20.33 (removeonly.cpp) is surprising to those unfamiliar with `std::remove` and `std::remove_if`:

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
0 1 2 3 4 5 6 7 8 9 11 12 13 14 15 16 17 18 19 19
1 3 5 7 9 11 13 15 17 19 19 12 13 14 15 16 17 18 19 19
```

The first line shows the contents of the original vector. The second line shows the modified vector with 10 removed. Element 10 definitely is gone, but the vector contains the same number of elements as before! A closer inspection reveals that all the elements that followed 10 in the original vector have been shifted forward one position. It is as if the removal of 10 left a hole in the vector that the following elements had to shift forward to fill. That would be fine, but the last element, 19, was copied forward and remains the last element in the vector.

In the last line we see that all the evens are missing in the front of the vector, but, again, some unusual shifting and copying occurred to keep the vector the same size.

In order to actually remove the elements and make the resulting container smaller, we must couple `remove` or `remove_if` with the container's `erase` method. The `erase` method accepts two iterators establishing a range of elements to delete from the container. Listing 20.34 (vectorerase.cpp) shows how to remove a section of a vector using the `std::vector<T>::erase` method.

Listing 20.34: `vectorerase.cpp`

```
#include <iostream>
#include <vector>
#include <iterator>

int main() {
    std::vector<int> nums {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
    auto output = std::ostream_iterator<int>(std::cout, " ");

    // Print the vector
    copy(std::begin(nums), std::end(nums), output);
    std::cout << '\n';

    // Delete elements from index 3 to index 7
    nums.erase(std::begin(nums) + 3, std::begin(nums) + 8);
    std::copy(std::begin(nums), std::end(nums), output);
    std::cout << '\n';
}
```

Listing 20.34 (vectorerase.cpp) prints

```
10 20 30 40 50 60 70 80 90 100
10 20 30 90 100
```

The `remove` and `remove_if` functions return an iterator that points to the first element in the container that *was not* removed. Recall the output of Listing 20.33 (removeonly.cpp), reproduced here:

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
0 1 2 3 4 5 6 7 8 9 11 12 13 14 15 16 17 18 19 19
1 3 5 7 9 11 13 15 17 19 19 12 13 14 15 16 17 18 19 19
```

The return value of the call to `remove` returns an iterator pointing to the second occurrence of 19 in the second line of output. In the third line of output the call to `remove_if` returns a pointer to second occurrence of 19 as well. Note that in both cases the iterators point to the first element in the remaining values that ought to be erased from the vector.

Listing 20.35 (testremove.cpp) combines the vector's `erase` method and the `remove` function to achieve the desired results.

Listing 20.35: `testremove.cpp`

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <numeric>
#include <iterator>

bool is_even(int n) {
    return n % 2 == 0;
}

int main() {
    std::vector<int> nums(20);
    std::string letters = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

    auto output = std::ostream_iterator<int>(std::cout, " ");

    // Fill the vector with 0, 1, ..., 19
    std::iota(std::begin(nums), std::end(nums), 0);
    std::copy(std::begin(nums), std::end(nums), output);
    std::cout << '\n';

    // Remove 10
    nums.erase(std::remove(std::begin(nums), std::end(nums), 10),
               std::end(nums));
    std::copy(std::begin(nums), std::end(nums), output);
    std::cout << '\n';

    // Remove remaining even numbers
    nums.erase(std::remove_if(std::begin(nums), std::end(nums), is_even),
               std::end(nums));
    std::copy(std::begin(nums), std::end(nums), output);
    std::cout << '\n';
}
```

Listing 20.35 (testremove.cpp) prints the expected results:

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
0 1 2 3 4 5 6 7 8 9 11 12 13 14 15 16 17 18 19
1 3 5 7 9 11 13 15 17 19
```

The combination of `std::vector<T>::erase` and `std::remove` is known in the C++ community as the *erase-remove idiom*.

The STL contains a number of other algorithms to process generic containers. Some examples include

- `std::reverse` reverses the elements of a container,
- `std::all_of` returns true if all the elements in a container satisfy a predicate,
- `std::any_of` returns true if at least one of the elements in a container satisfy a predicate,
- `std::none_of` returns true if none of the elements in a container satisfy a predicate,
- `std::find_if` returns an iterator to the first element in a container that satisfies a predicate,

20.6 Namespaces

In Section 2.3 we reviewed the various approaches for using names from the `std` namespace. We have been avoiding the blanket `using` directive:

```
using namespace std;
```

claiming that its disadvantages outweight its occasional convenience.

We now are creating our own names for functions and types within the programs we write. Using the blanket `using` directive can present problems at times because the `std` namespace contains hundreds of type and function names, and it is too easy for a programmer to accidentally reuse for another purpose a name already claimed by the `std` namespace. Such name conflicts lead to problems during software development, and since the development process generally has enough other challenges to address, it is best practice to avoid the blanket `using` directive.

Listing 20.36 (maxvector.cpp) defines a `max` function that computes the number of times the maximum value appears in a vector of integers. The `is_sorted` function determines if all the elements in a vector appear in nondecreasing order. The `main` function tries out the function on a sample vector.

Listing 20.36: `maxvector.cpp`

```
#include <iostream>
#include <vector>

using namespace std;

// Counts the number of occurrences of the maximum value in
// a nonempty vector vec
int max(const vector<int>& vec) {
    auto p = std::begin(vec);  // p points to the first element
    // Determine maximum value
    int m = *p++;          // Set m to the first element and move p
    while (p != std::end(vec)) {
        if (*p > m)
            m = *p;        // Found a reason to update m
        p++;
    }
    // Count the number of times the maximum appears
```

```
    int count = 0;
    for (auto elem : vec)
        if (elem == m)
            count++;
    return count;
}

// Returns true if all the elements in vector vec appear
// in nondecreasing order; otherwise, returns false.
bool is_sorted(const vector<int>& vec) {
    for (unsigned i = 0; i < vec.size() - 1; i++)
        if (vec[i] > vec[i + 1])
            return false;  // Found elements out of order
    return true;
}

int main() {
    cout << boolalpha << is_sorted({1, 2, 3, 4, 5}) << '\n';
    cout << boolalpha << is_sorted({5, 4, 3, 2, 1}) << '\n';
    cout << boolalpha << is_sorted({5, 5, 5, 5, 5}) << '\n';
    cout << "-----------------------\n";
    cout << max({1, 2, 3, 4, 5}) << '\n';
    cout << max({5, 4, 3, 2, 1}) << '\n';
    cout << max({3, 4, 5, 2, 1}) << '\n';
    cout << max({5, 2, 3, 4, 5}) << '\n';
    cout << max({5, 4, 5, 2, 5}) << '\n';
    cout << max({5}) << '\n';
    cout << max({5, 5, 5, 5, 5}) << '\n';
}
```

Notice the blanket `using namespace std` directive that makes all the function and type names in the `std` namespace available to this program. This has not been our common practice so far, but you frequently will encounter this approach in published C++ code. This program works correctly, however, as it prints the following:

```
true
false
true
-----------------------
1
1
1
2
3
1
5
```

The C++ standard library contains a function named `is_sorted`, declared in the `<algorithm>` header. The standard `is_sorted` performs the same task as this `is_sorted`, except that the standard function, striving to be as generic as possible, accepts two iterators instead of the container itself. In preparation to replace our `is_sorted` function with the standard `is_sorted` function, we add the following directive to the top of the code with the other `#include`s:

```
#include <algorithm>
```

If we compile and run our program with this addition, we get no errors or warnings, and the program prints the following:

```
true
false
true
----------------------
5
5
5
5
5
5
5
```

Observe that our `max` function behaves very differently after this slight change. In fact, the code that executes for the `max` invocations is not the code in Listing 20.36 (maxvector.cpp) but rather the `max` function in the standard library! This is because a call such as

```
max({1, 2, 3, 4, 5})
```

passes an *initializer list* to `max`, not an actual vector object. Before the inclusion of `<algorithm>`, the only `max` the compiler knew of was the one defined in Listing 20.36 (maxvector.cpp). This `max` function expects a reference to a `std::vector` object. The `std::vector` class contains a constructor that accepts an initializer list. The compiler, therefore, automatically can create a vector from an initializer list. The compiler automatically uses the initializer list to create a vector object and sends it off the `max` function for processing.

The inclusion of the `<algorithm>` header provides the declaration of `std::max` that accepts, among other arguments, an initializer list. When the compiler matches a function call to its corresponding definition, it always seeks the best match for the arguments passed. Since the `std::max` accepts an initializer list, it better matches the calls in the `main` function.

Remember, our purpose for including the `<algorithm>` header was to gain access to `is_sorted`, and we had no intention of changing our `max` implementation. Unfortunately, the compiler, following a well-established and sensible algorithm, silently (no warnings or errors) substituted a different function to change the behavior of our program. This is prime example of why the blanket `using namespace std` directive can be dangerous.

To avoid the danger, avoid the `using` directive, as shown in Listing 20.37 (bettermaxvector.cpp)

Listing 20.37: `bettermaxvector.cpp`

```
#include <iostream>
#include <algorithm>
#include <vector>

// Counts the number of occurrences of the maximum value in
// a nonempty vector vec
int max(const std::vector<int>& vec) {
    auto p = std::begin(vec);  // p points to the first element
    // Determine maximum value
    int m = *p++;          // Set m to the first element and move p
    while (p != std::end(vec)) {
        if (*p > m)
            m = *p;        // Found a reason to update m
```

```
        p++;
    }
    // Count the number of times the maximum appears
    int count = 0;
    for (auto elem : vec)
        if (elem == m)
            count++;
    return count;
}

// Returns true if all the elements in vector vec appear
// in nondecreasing order; otherwise, returns false.
bool is_sorted(const std::vector<int>& vec) {
    for (unsigned i = 0; i < vec.size() - 1; i++)
        if (vec[i] > vec[i + 1])
            return false;  // Found elements out of order
    return true;
}

int main() {
    std::cout << std::boolalpha << is_sorted({1, 2, 3, 4, 5}) << '\n';
    std::cout << std::boolalpha << is_sorted({5, 4, 3, 2, 1}) << '\n';
    std::cout << std::boolalpha << is_sorted({5, 5, 5, 5, 5}) << '\n';
    std::cout << "-----------------------\n";
    std::cout << max({1, 2, 3, 4, 5}) << '\n';
    std::cout << max({5, 4, 3, 2, 1}) << '\n';
    std::cout << max({3, 4, 5, 2, 1}) << '\n';
    std::cout << max({5, 2, 3, 4, 5}) << '\n';
    std::cout << max({5, 4, 5, 2, 5}) << '\n';
    std::cout << max({5}) << '\n';
    std::cout << max({5, 5, 5, 5, 5}) << '\n';
}
```

In Listing 20.37 (bettermaxvector.cpp) because of the missing `std::` prefixes the compiler knows that the calls to `max` cannot be the calls to `std::max`.

Identifiers (variable names, function names, and type names) should be meaningful, clearly communicating their intent. Well-chosen English words or simple variations of common words are ideal. This means it is quite likely that a library developed by a team of developers would contain function names and type names that also appear in a different library developed by a different team of programmers. Each library used separately is not a problem, but issues can arise when building a software system atop multiple libraries; for example, consider the task of building a program that models and visualizes the spread of an infectious agent. Such a system could make good use of the following libraries:

- a 3-D graphics library that defines a mathematical vector class, used for a number computer graphics rendering processes, such as scene orientation and light intensities,
- an epidemiology modeling library with a vector type that represents the means of transmission of a disease, such as a mosquito or tick, and
- the C++ standard library which includes a vector generic class used to store sequences of any type.

The name *vector* has a different meaning in each of these contexts. C++ namespaces allow us to manage simultaneously and effectively all of these vector types within a single software system.

Ideally, any code meant for widespread use should be part of a namespace. A company developing libraries for internal use could place them in a namespace based on the company name. Components intended for public consumption could be placed in a namespace with a different name.

In C++ it is easy to put the functions and types we develop into custom namespaces. Suppose we wish to make the functions from Listing 20.37 (bettermaxvector.cpp) available to a wider audience. Listing 20.38 (vectutils.h) and Listing 20.39 (vectutils.cpp) package the two functions into a namespace named `vectutils`. Listing 20.38 (vectutils.h) provides the function declarations, and Listing 20.39 (vectutils.cpp) contains the function definitions.

Listing 20.38: `vectutils.h`

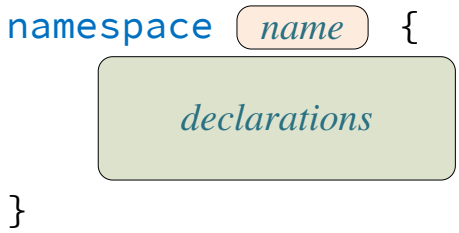
```
#include <vector>

namespace vecutils {

// Counts the number of occurrences of the maximum value in
// a nonempty vector vec
int max(const std::vector<int>& vec);

// Returns true if all the elements in vector vec appear
// in nondecreasing order; otherwise, returns false.
bool is_sorted(const std::vector<int>& vec);

} // End vecutils namespace
```

Listing 20.39: `vectutils.cpp`

```
#include "vectutils.h"

namespace vecutils {
// Counts the number of occurrences of the maximum value in
// a nonempty vector vec
int max(const std::vector<int>& vec) {
    auto p = std::begin(vec);  // p points to the first element
    // Determine maximum value
    int m = *p++;          // Set m to the first element and move p
    while (p != std::end(vec)) {
        if (*p > m)
            m = *p;        // Found a reason to update m
        p++;
    }
    // Count the number of times the maximum appears
    int count = 0;
    for (auto elem : vec)
        if (elem == m)
            count++;
    return count;
}

// Returns true if all the elements in vector vec appear
// in nondecreasing order; otherwise, returns false.
bool is_sorted(const std::vector<int>& vec) {
    for (unsigned i = 0; i < vec.size() - 1; i++)
        if (vec[i] > vec[i + 1])
```

```
            return false;  // Found elements out of order
    return true;
}

} // End vecutils namespace
```

The general form of a namespace declaration is



The name of a namespace is an identifier; therefore, the same rules governing variable names, function names, and type names apply equally to namespace names.

Listing 20.38 (vectutils.h) and Listing 20.39 (vectutils.cpp) demonstrate that namespace declarations can span multiple files. The components of the `std` namespace are scattered over many different files.

Listing 20.40 (testns.cpp) tests the functions in our new namespace.

Listing 20.40: `testns.cpp`

```
#include <iostream>
#include "vectutils.h"

int main() {
    std::cout << std::boolalpha << vecutils::is_sorted({1, 2, 3, 4, 5}) << '\n';
    std::cout << std::boolalpha << vecutils::is_sorted({5, 4, 3, 2, 1}) << '\n';
    std::cout << std::boolalpha << vecutils::is_sorted({5, 5, 5, 5, 5}) << '\n';
    std::cout << "-----------------------\n";
    std::cout << vecutils::max({1, 2, 3, 4, 5}) << '\n';
    std::cout << vecutils::max({5, 4, 3, 2, 1}) << '\n';
    std::cout << vecutils::max({3, 4, 5, 2, 1}) << '\n';
    std::cout << vecutils::max({5, 2, 3, 4, 5}) << '\n';
    std::cout << vecutils::max({5, 4, 5, 2, 5}) << '\n';
    std::cout << vecutils::max({5}) << '\n';
    std::cout << vecutils::max({5, 5, 5, 5, 5}) << '\n';
}
```

Note that instead of using the long names for `vecutils::is_sorted` and `vecutils::max`, we could have used the focused `using` declarations:

```
using vecutils::is_sorted;
using vecutils::max;
```

We also could us the blanket `using` directive

```
using namespace vecutils;
```

but this introduces the possibility of name clashes, especially with the similar-named functions from the `std` namespace declared in the `<algorithm>` header. Just as with the `std` namespace, it is best to avoid blanket `using` directives for custom namespaces.

C++ permits namespaces to be nested, as in

```
namespace utils {
    namespace graphics {
        namespace math {
            double f(double v) { /* Details omitted . . . */ }
        }
    }
}
```

Absent of any `using` directives, we would call function `f` as

```
std::cout << utils::graphics::math::f(9.4) << '\n';
```

Since nested namespace names can become quite lengthy, C++ supports namespace aliases, as in

```
namespace ugm = utils::graphics::math;
```

This namespace alias makes it possible for us to call `f` with the more compact statement

```
std::cout << ugm::f(9.4) << '\n';
```

Nesting allows developers to organize better the components within libraries; for example, a company may have an outer namespace derived from the company's name and nested namespaces that correspond to departments or divisions within the company.

Chapter 21

Associative Containers

21.1 Associative Containers

Chapter 11 introduced the vector and array data types. Both of these data structures represent linear sequences of elements. Vectors and arrays are convenient for storing collections of data, but they have some limitations. For one, we locate an element within a vector or array based on its position (index). To retrieve a specific element we must supply its index. While this approach is fine for many applications, in other situations this access-by-index approach is awkward or inefficient.

In contrast, an *associative container* permits access based on a *key*, rather than an index. Unlike an index, a key is not restricted to an integer expression. The C++ standard library supports three kinds of associative containers: `set`, `map`, and `unordered_map`. Here we will examine each of these types in some detail.

21.2 The `std::set` Data Type

C++ provides a data structure that models to some extent a mathematical set. Unlike C++ vectors, arrays, and lists, mathematical sets are unordered and may contain no duplicate elements. The C++ `std::set` container, like mathematical sets, ignores any attempts to include duplicate elements; however, `std::set` objects do order their elements internally in a particular way. This ordering permits very fast access to elements. Listing 21.1 (settest.cpp) demonstrates these properties.

Listing 21.1: `settest.cpp`

```
#include <iostream>
#include <set>

// A custom type
struct MyClass {
    int data;
    MyClass(int d): data(d) {}
};

// We need this operator so we can insert our MyClass into a set
bool operator<(const MyClass& a, const MyClass& b) {
```

```
    return a.data < b.data;
}

// Define this operator so output streams can print
// instances of our custom type.
std::ostream& operator<<(std::ostream& os, const MyClass& obj) {
    os << obj.data;
    return os;
}

int main() {
    // Attempt to add duplicates
    std::set<int> S {10, 3, 7, 2, 7, 11, 3};
    for (auto elem : S)
        std::cout << elem << ' ';
    std::cout << '\n';

    std::set<int> T {5, 4, 5, 2, 4, 9};
    for (auto elem : T)
        std::cout << elem << ' ';
    std::cout << '\n';

    // Make a set of our custom types
    std::set<MyClass> U {MyClass(10), MyClass(3), MyClass(7), MyClass(2)};
    for (auto& elem : U)
        std::cout << elem << ' ';
    std::cout << '\n';
}
```

The output of Listing 21.1 (settest.cpp) is

```
2 3 7 10 11
2 4 5 9
2 3 7 10
```

Note the element ordering in the initializer list for the set constructor is different from the order of the printed elements when traversed via a `for` loop. The order actually is ascending order in all of our tests in Listing 21.1 (settest.cpp), and this is not a coincidence. Also observe that sets do not admit duplicate elements.

The implementation of the `std::set` class on all the major C++ libraries use a *red-black tree* (see `https://en.wikipedia.org/wiki/RedŰblack_tree`) as the internal data structure. The ordering imposed by this internal data structure enables very fast access to the set's elements. A red-black tree permits binary search (see Section 12.3.2). A set determines the order of its elements via `operator<`. The `<` operator works with all the primitive numeric types (`int`, `double`, `float`, `unsigned`, etc.). The `operator<` is defined for `std::string` class instances to compare two string objects lexicographically. Instances of any type we wish to store in a `std::set` must support the `<` operator.

In Listing 21.1 (settest.cpp) we define a new custom type, `MyClass`. We must ensure that it is possible to compare two `MyClass` instances using the `<` operator in order to put `MyClass` objects in a set. To achieve this, we define `operator<` as a global function that compares two `MyClass` objects. This allows us to create a set of `MyClass` objects.

We can make a set out of a vector using the set constructor that accepts a pair of iterators to another container:

```
std::vector<int> vec {40, 10, 20, 50, 10, 30};
set<int> my_set(std::begin(vec), std::end(vec));
for (auto elem : my_set)
    std::cout << elem << ' ';
std::cout << '\n';
```

As with an initializer list, this construction from a vector will not preserve the original ordering found in the vector. Also, any duplicate elements in the vector will appear only once in the set object.

Unlike in mathematics, all sets in C++ must be finite.

Listing 21.2 (setoutput.h) provides a convenient function that prints a `std::set` in familiar human-readable form.

Listing 21.2: `setoutput.h`

```
#ifndef SETOUTPUT_H_DEFINED
#define SETOUTPUT_H_DEFINED

 #include <iostream>
 #include <set>

 // Print out a set in a familiar form
 template <typename T>
 std::ostream& operator<<(std::ostream& os, const std::set<T>& s) {
     os << '{';
     auto iter = std::begin(s);
     auto done = std::end(s);
     if (iter != done) {
         os << *iter++;
         while (iter != done)
             os << ", " << *iter++;
     }
     os << '}';
     return os;
 }

 #endif
```

Listing 21.2 (setoutput.h) enables us to write code such as

```
std::set<int> S = {20, 40, 60, 80};
std::cout << S << '\n';
```

which prints

```
{20, 40, 60, 80}
```

just like the set would appear in a mathematics book. We will include this header file in subsequent programs that use `std::set` so we easily can display a set's contents.

C++ supports the standard mathematical set operations of intersection, union, set difference, and symmetric difference. These functions are part of the algorithms library, and thus they are designed to work with as wide an array of containers as possible. The downside of this flexibility is they are somewhat arcane and awkward to use, as Listing 21.3 (setopsbuiltin.cpp) shows.

Listing 21.3: setopsbuiltin.cpp

```
#include <iostream>
#include <set>
#include <vector>
#include <algorithm>
#include <iterator>

#include "setoutput.h"


int main() {
    // Compute the intersection and union of two sets
    std::set<int> s1{1, 2, 3, 4, 5, 6};
    std::set<int> s2{2, 5, 7, 9, 10};
    std::set<int> s1_inter_s2;

    std::set_intersection(std::begin(s1), std::end(s1),
                          std::begin(s2), std::end(s2),
                          std::inserter(s1_inter_s2, std::end(s1_inter_s2)));
    std::cout << "Intersection of " << s1 << " and " << s2
              << " = " << s1_inter_s2 << '\n';

    std::cout << "\n-------------\n";

    std::set<int> s1_union_s2;
    std::set_union(std::begin(s1), std::end(s1),
                   std::begin(s2), std::end(s2),
                   std::inserter(s1_union_s2, std::end(s1_union_s2)));
    std::cout << "Union of " << s1 << " and " << s2
              << " = " << s1_union_s2 << '\n';

    std::cout << "\n-------------\n";

    std::vector<int> v1{1, 2, 3, 4, 5, 6};
    std::vector<int> v2{2, 5, 7, 9, 10};
    // Note: v1 and v2 both must be sorted for this to work
    // Since we created them sorted the next two statements are not
    // technically necessary, but we include them here to emphasize
    // the fact that the vectors must be sorted to compute their
    // intersection and union in this manner.  Sets are inherently
    // sorted, and so need no such preprocessing.
    sort(std::begin(v1), std::end(v1));
    sort(std::begin(v2), std::end(v2));
    std::vector<int> v1_intersection_v2;
    std::vector<int> v1_union_v2;
    std::set_intersection(std::begin(v1), std::end(v1),
                          std::begin(v2), std::end(v2),
                          std::back_inserter(v1_intersection_v2));
    std::set_union(std::begin(v1), std::end(v1),
                   std::begin(v2), std::end(v2),
                   std::back_inserter(v1_union_v2));
    for (auto elem : v1_intersection_v2)
        std::cout << elem << ' ';
    std::cout << '\n';
```

```
    for (auto elem : v1_union_v2)
        std::cout << elem << ' ';
    std::cout << '\n';
}
```

Listing 21.3 (setopsbuiltin.cpp) shows how the intersection and union functions work for vectors as well as sets. The catch is that the elements in the containers must be ordered for these functions to work properly. We saw that `std::set` objects are ordered automatically. Since `std::vector` objects do not impose an order on their elements, we must first sort any vectors we wish to send to the standard intersection and union functions.

Listing 21.4 (setops.cpp) implements `operator&` and `operator|` to work with set objects to provide a more convenient interface to programmers. The & operator represents set intersection, and | represents set union.

Listing 21.4: `setops.cpp`

```
#include <iostream>
#include <set>
#include <algorithm>
#include <iterator>

#include "setoutput.h"

// Computes the intersection of sets s1 and s2
template<typename T>
std::set<T> operator&(const std::set<T>& s1, const std::set<T>& s2) {
    std::set<T> result;
    std::set_intersection(std::begin(s1), std::end(s1),
                          std::begin(s2), std::end(s2),
                          std::inserter(result, std::end(result)));
    return result;
}

// Computes the union of sets s1 and s2
template<typename T>
std::set<T> operator|(const std::set<T>& s1, const std::set<T>& s2) {
    std::set<T> result;
    std::set_union(std::begin(s1), std::end(s1), std::begin(s2), std::end(s2),
                   std::inserter(result, std::end(result)));
    return result;
}


int main() {
    std::set<int> s1 {1, 2, 3, 4, 5, 6, 7, 8};
    std::set<int> s2 {2, 5, 7, 9, 10};

    std::cout << s1 << " & " << s2 << " = " << (s1 & s2) << '\n';
    std::cout << s1 << " | " << s2 << " = " << (s1 | s2) << '\n';
}
```

If we did a lot of programming with mathematical sets, we probably would add these two functions to our header file that overloads `operator<<` to print set objects.

The real value of the `set` class comes from the speed of access it provides to its elements. As an associative container, we do not access an element via an index as we do with a vector or array. In fact, `std::set` does not even provide `operator[]`. We want to place an item into a set and retrieve it later without regard to its location within its internal data structure.

Recall the `std::find` algorithm provided by the STL (see Section 20.5). Given an iterator to the beginning of the container, an iterator just past the end of the container, and an item to find, `std::find` returns an iterator that points to the sought element within the container. If the container does not contain the sought element, `std::find` returns the iterator that equals the just-past-the-end iterator.

The `std::find` function works with a `std::set` object just as it does with other containers, but the `std::set` class provides its own find method that exploits the structure of the data within the set object to locate elements very quickly. Listing 21.5 (setvsvector.cpp) illustrates the use the `std::set::find` method, comparing the time to locate an element in a vector versus the time to locate the same element in a set containing the exact elements contained in the vector.

Listing 21.5: `setvsvector.cpp`

```
#include <iostream>
#include <set>
#include <vector>
#include <algorithm>
#include <iterator>
#include <ctime>

#include "uniformrandom.h"

int main() {
    const int SIZE = 250000;
    // Make a vector and set
    std::vector<int> v;
    std::set<int> s;

    // Populate the vector and set with the first SIZE integers
    for (int i = 0; i < SIZE; i++)
        v.push_back(i);
    for (int i = 0; i < SIZE; i++)
        s.insert(i);

    // Make a vector that contains random numbers in the range
    // stored within the vector and set
    UniformRandomGenerator gen(0, SIZE - 1);
    std::vector<int> search_values;
    for (int i = 0; i < SIZE; i++)
        search_values.push_back(gen());

    clock_t start_time, stop_time;

    // Search each data structure for the integers 0 to 1,000,000
    start_time = clock();
    for (int i = 0; i < SIZE; i++) {
        int seek = search_values[i];
        std::find(std::begin(v), std::end(v), seek);
    }
    stop_time = clock();
```

```
    std::cout << "Vector time: " << stop_time - start_time << '\n';

    start_time = clock();
    for (int i = 0; i < SIZE; i++) {
        int seek = search_values[i];
        s.find(seek);
    }
    stop_time = clock();
    std::cout << "Set time: " << stop_time - start_time << '\n';
}
```

The vector and set objects within Listing 21.5 (setvsvector.cpp) each contain the integers $0, 1, \ldots, 249,000$. Note that the vector contains the elements in ascending order. We then create a vector containing pseudo-random values in the range $0, 1, \ldots, 249,000$. The program will search for these random values within each container. The program measures the time it takes to complete the searches.

The output Listing 21.5 (setvsvector.cpp) reveals the dramatic difference in performance between the `std::find` function on a vector and the `std::set::find` method with a set:

```
Vector time: 84044 msec
Set time: 234 msec
```

We see on this particular run that the vector search required almost one and a half minutes to complete, while the set search took only about a quarter of a second. While the exact numbers will vary from run to run and be higher or lower depending on the host machine, the disparity of times will be consistent. The two data structure contain exactly the same elements, and program uses the exact same search values. The `std::set`'s search is so much quicker because `std::set::find` performs binary search on its internal data structure, while `std::find` uses linear search (see Section 12.3.2).

21.3 Tuples

In C++ a function can return only one thing. That one thing might be an integer or a single object like a vector or an instance of some other class, but it must be just one thing. What if we need a function to return more than one thing? We could put the desired return values into a vector and return the vector. The caller then would extract the components from the vector upon the function's return. Unfortunately, this will not work if the multiple elements to return have different types. The container types we have examined thus far—vectors, arrays, lists, and sets—have been homogeneous; that is, their elements all have the same type.

We could define a `struct` or `class` that has fields of different types, but this creates a new, named, custom type. In a large program we may need many different `struct`s to cover all the combinations of multiple-valued return types we need. Providing a named `struct` or `class` introduces a new type into the system, thereby increasing the system's complexity.

All we want is to be able to package multiple elements together for some simple purpose, without needing to create a new custom type. Fortunately, C++ provides the `std::tuple` generic class in its standard library. Listing 21.6 (simpletuples.cpp) shows how to use tuples in a C++ program.

Listing 21.6: `simpletuples.cpp`

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

```
#include <cmath>  // For sqrt

int main() {
    // Declare some local variables
    std::string word;
    int number;
    double quantity;

    // Construct a tuple directly with some literal values
    std::tuple<std::string, int, double> t1 {"Bob", 4, 9.5};
    // Unpack the tuple's components
    std::tie(word, number, quantity) = t1;
    // Print the results
    std::cout << "word = " << word << ", number = " << number
              << ", quantity = " << quantity << '\n';

    // Use std::make_tuple convenience function
    auto t2 = std::make_tuple("Eve", 22, 8.3);
    // Unpack the tuple's components
    std::tie(word, number, quantity) = t2;
    // Print the results
    std::cout << "word = " << word << ", number = " << number
              << ", quantity = " << quantity << '\n';

    // Declare some variables
    std::string name = "Jan";
    int age = 12;
    double amount = 50.2;
    // Build a tuple from more general expressions
    auto t3 = std::make_tuple(name, age * 2, std::sqrt(amount));
    // Unpack the tuple's components one at a time
    word     = std::get<0>(t3);  // 1st component is at index 0
    number   = std::get<1>(t3);  // 2nd component is at index 1
    quantity = std::get<2>(t3);  // 3rd component is at index 2
    // Print the results
    std::cout << "word = " << word << ", number = " << number
              << ", quantity = " << quantity << '\n';
}
```

When compiled and executed, Listing 21.6 (simpletuples.cpp) prints

```
word = Bob, number = 4, quantity = 9.5
word = Eve, number = 22, quantity = 8.3
word = Jan, number = 24, quantity = 7.0852
```

As the program shows, we can construct a `std::tuple` directly, as in the case of `t1`, but the `std::make_tuple` function results in simpler code. The arguments used when making a tuple can consist of literals, variables, and other more complex expressions.

Extracting the components of a tuple is known as *unpacking*. Listing 21.6 (simpletuples.cpp) shows how to use the `std::tie` function to assign the components of a tuple to individual variables in one statement. The `std::tie` function is a generic function that works with any types storable within a tuple. If we wish to extract just one element from a tuple, we can use the `std::get` generic function. The generic `std::get` function is parameterized by an integer that serves as an index into the tuple. As with vectors,

the first element of the tuple is found at index zero.

C++ supports another way of unpacking tuples. Structured binding is similar to `std::tie` but can be used only when declaring the variables used to represent the components of a tuple. With structured binding we can rewrite the following code:

```
std::string word, name = "Jan";
int number, age = 25;
double quantity, amount = 50.2;
auto t = std::make_tuple(name, age, amount);
// Unpack the tuple's components
std::tie(word, number, quantity) = t1;
```

as

```
std::string name = "Jan";
int age = 25;
double amount = 50.2;
auto t = std::make_tuple(name, age, amount);
// Unpack the tuple's components
auto [word, number, quantity] = t;
```

The statement

```
auto [word, number, quantity] = t;
```

constitutes the structured binding. Note that we can use this special syntax only when declaring the variables to be bound to the components of the tuple. This copies the components of the tuple into the newly declared variables. Specifically, the following statement:

```
[word, number, quantity] = t;    // Illegal, not a declaration
```

is illegal, even if the variables `word`, `number`, and `quantity` are properly declared before the this statement.

We can specify that the structured binding be done by reference, as in the following:

```
auto& [word, number, quantity] = t;
```

In this case each name within the binding becomes an alias to a component of the tuple. In the previous code fragment above, reassigning `number` will similarly modify the second component of tuple `t`.

Structured binding is especially convenient when processing tuples within loops. Listing 21.7 (tupleloop.cpp) uses structured bindings by reference to reassign the elements in a vector of tuples. The program then uses structured bindings to extract the components of each tuple to print them out.

Listing 21.7: `tupleloop.cpp`

```
#include <iostream>
#include <string>
#include <vector>
#include <tuple>

int main() {
    // Create a vector than can hold ten tuples
    std::vector<std::tuple<std::string, int, double>> vec(10);
```

```
    int count = 1;
    double quant = 10.0;
    // Populate the vector
    for (auto& [s, i, d] : vec) {
        s = "Item #" + std::to_string(count);
        i = count++;
        d = quant;
        quant += 10;
    }
    // Print the contents of the vector
    for (auto [s, i, d] : vec)
        std::cout << "[" << s << ", " << i << ", " << d << "]\n";
}
```

Listing 21.7 (tupleloop.cpp) prints the following:

```
[Item #1, 1, 10]
[Item #2, 2, 20]
[Item #3, 3, 30]
[Item #4, 4, 40]
[Item #5, 5, 50]
[Item #6, 6, 60]
[Item #7, 7, 70]
[Item #8, 8, 80]
[Item #9, 9, 90]
[Item #10, 10, 100]
```

The C++ standard library supports a special case of `std::tuple` called `std::pair`. While a `std::tuple` supports zero or more components, each `std::pair` object must contain exactly two components. Listing 21.8 (pairtest.cpp) shows `std::pair` in action.

Listing 21.8: `pairtest.cpp`

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

int main() {
    // Declare some local variables
    std::string word;
    int number;
    double quantity;

    // Construct a pair directly with some literal values
    std::pair<std::string, double> t1 {"Bob", 9.5};
    // Unpack the pair using std::tie
    std::tie(word, quantity) = t1;
    // Print the results
    std::cout << "word = " << word << ", quantity = " << quantity << '\n';

    // Use std::make_pair convenience function
    auto t2 = std::make_pair("Eve", 22);
    // Unpack the pair using std::get
    word   = std::get<0>(t2);  // 1st component
    number = std::get<1>(t2);  // 2nd component
```

```
    // Print the results
    std::cout << "word = " << word << ", number = " << number << '\n';

    // Declare some variables
    std::string name = "Jo";
    int age = 12;
    // Build a pair from more general expressions
    auto t3 = std::make_pair(name + "-ann", 2* age);
    // Unpack the tuple's using first and second fields
    word   = t3.first;   // 1st component
    number = t3.second;  // 2nd component
    // Print the results
    std::cout << "word = " << word << ", number = " << number << '\n';
    // Use structured binding to extract compoents
    auto [wrd, nbr] = t3;
    // Print the results
    std::cout << "word = " << wrd << ", number = " << nbr << '\n';
}
```

As Listing 21.8 (pairtest.cpp) shows, we can use `std::tie`, `std::get`, and structured binding with a `std::pair` object, and they work exactly as they do with `std::tuple`. Since pair objects always contain exactly two elements the `std::pair` class provides the public fields `first` and `second` through which we can unpack a pair using simpler syntax.

Neither `std::tuple` nor `std::pair` are associative containers. The `std::pair` class is used by the two associative containers we will consider next: `std::map` and `std::unordered_map`.

21.4 The `std::map` Data Type

The `std::map` data structure is another example of a C++ associative container. In order to use a `std::map` object within a program we must use the appropriate `#include` directive:

```
#include <map>
```

The `std::map` class is a template class, so when declaring an instance we must supply type parameters within angle brackets as shown in the following example:

```
std::map<std::string, int> my_map;
```

Here the `my_map` variable refers to an associative container that holds `std::pair` objects, each of which consist of a string *key* and an integer *value*. We say each integer in the map object has an associated `string` key. In the expression `std::map<std::string, int>` the first template argument (here `std::string`) indicates the map's key type, and the second template argument indicates the type of values stored in the map (here `int`). Listing 21.9 (simplemap.cpp) provides a simple example that uses a C++ map object.

Listing 21.9: `simplemap.cpp`

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

int main() {
```

```
    std::map<std::string, int> container;

    container["Fred"]   = 22;
    container["Ella"]  = 21;
    container["Owen"] = 34;
    container["Zoe"]  = 29;

    std::cout << container["Ella"] << '\n';
    std::cout << container["Zoe"] << '\n';
}
```

Note that in this case we access an element in a map object via `operator[]` with a string "index" rather than a nonnegative integer index as we would in a vector. In Listing 21.9 (simplemap.cpp) the statement

```
container["Fred"] = 22;
```

associates the value 22 with the key `"Fred"`. Keys must be unique; that is, no two values in an unordered map can have the same key. Duplicate values are allowed; that is, two different keys can have the same value, but two different values cannot have the same key. This is similar to a vector or array, in that the same value may be stored at different indices, but only one value may be associated with a particular index.

One big difference between a map and vector is a vector of size n allows any index in the range $0 \ldots n - 1$, while a map contains a key only when the programmer specifically uses it with the unordered map. Consider the statement

```
container["Fred"] = 22;
```

if the key within the square brackets (`"Fred"`) does not exist in the map before this statement executes, the statement adds the key to the map and pairs it with the value on the right of the assignment operator. If the key already exists in the map, the statement replaces the value previously associated with the key with the new value on the right of the assignment operator.

Consider accessing a value with a given key rather than assigning the value; the statement

```
std::cout << container["Fred"] << '\n';
```

prints the value associated with the key `"Fred"`. If `"Fred"` is not currently a key in the map, this statement will add the key `"Fred"` with the default value for the type of value declared for the map. For integers, the default value is 0. As other examples, the default floating-point value is 0.0, the default string value is the empty string, `""`, and the default `bool` value is `false`.

You should use a map object when you need fast and convenient access to an element of a collection based on a search key rather than an index. Consider the problem of implementing a simple telephone contact list. Most people are very familiar with the names of their friends, family, and business contacts but can remember only a handful of telephone numbers. A contact list associates a name with a telephone number.

It would be inappropriate to place the names in a vector, for example, and locate a name using the associated phone number as an index into the vector. This look-up method is backwards—we do not want to find a name given a phone number; we want to look up a number based on a name. Besides, each phone number contains many digits, and we would not need or want to have a vector with indices that large—most of the space in the data structure would be unused.

In our situation a person or company's name is a unique identifier for that contact. In this case the name is a *key* to that contact. A map is an ideal data structure for mapping keys to values. It allows for the fast

retrieval of a value given its associated key. Listing 21.10 (phonelist.cpp) uses a C++ map to implement a simple telephone contact database with a rudimentary command line interface.

Listing 21.10: phonelist.cpp

```
#include <iostream>
#include <string>
#include <map>
#include <algorithm>

int main() {
    std::map<std::string, int> contacts; // Telephone contact list
    bool running = true;

    while (running) {
        std::string name;
        int number;
        char command;
        std::cout << "A)dd   L)ook up   Q)uit: ";
        std::cin >> command;
        switch (command) {
            case 'A':
            case 'a':
                //std::cout << "Enter new name: " << '\n';
                std::cin >> name;
                std::transform(std::begin(name), std::end(name),
                               std::begin(name), toupper);
                //std::cout << "Enter phone number for " << name << ": ";
                std::cin >> number;
                contacts[name] = number;
                break;
            case 'L':
            case 'l':
                std::cin >> name;
                transform(std::begin(name), std::end(name), std::begin(name),
                          ::toupper);
                std::cout << name << " " << contacts[name] << '\n';
                break;
            case 'Q':
            case 'q':
                running = false;
                break;
            case 'D':  // Secret command
            case 'd':
                for (auto& elem : contacts)
                    std::cout << elem.first << " " << elem.second << '\n';
                break;
            default:
                std::cout << command << "is not a valid command" << '\n';
        }
    }
}
```

The following shows a sample run of Listing 21.10 (phonelist.cpp):

```
A)dd   L)ook up   Q)uit: a Fred 5550134
A)dd   L)ook up   Q)uit: d
FRED 5550134
A)dd   L)ook up   Q)uit: a Ella 5559921
A)dd   L)ook up   Q)uit: l Ella
ELLA 5559921
A)dd   L)ook up   Q)uit: l Fred
FRED 5550134
A)dd   L)ook up   Q)uit: d
ELLA 5559921
FRED 5550134
A)dd   L)ook up   Q)uit: q
```

Listing 21.10 (phonelist.cpp) contains a secret "dump" command that prints out the contents of the map object. This command executes the following loop:

```
for (auto& elem : contacts)
    std::cout << elem.first << " " << elem.second << '\n';
```

This code reveals the fact that `std::map` objects store their elements as `std::pair` objects.

Note that we are storing a phone number as an integer, but values within a map object are not limited to integers.

To further motivate maps, consider Listing 21.11 (translateif.cpp) which uses conditional logic to translate some Spanish words into English.

Listing 21.11: `translateif.cpp`

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

int main() {
    std::string word = "";         // Initial word to ensure loop entry
    while (word != "quit") {  // Loop until user presses return by itself
        // Obtain word from the user
        std::cout << "Enter Spanish word: ";
        std::cin >> word;
        if (word == "uno")
            std::cout << "one\n";
        else if (word == "dos")
            std::cout << "two\n";
        else if (word == "tres")
            std::cout << "three\n";
        else if (word == "cuatro")
            std::cout << "four\n";
        else if (word == "cinco")
            std::cout << "five\n";
        else if (word == "seis")
            std::cout << "six\n";
        else if (word == "siete")
            std::cout << "seven\n";
        else if (word == "ocho")
            std::cout << "eight\n";
        else    // Unknown word
            std::cout << "???\n";
```

```
    }
}
```

Listing 21.11 (translateif.cpp) can successfully translate eight Spanish words into English. If we wish to increase the program's vocabulary, we must modify the program's logic by adding another `else if` block for each new word. Listing 21.12 (translatemap.cpp) provides a better approach; it uses a map to assist the translation.

Listing 21.12: `translatemap.cpp`

```
#include <iostream>
#include <fstream>
#include <string>
#include <map>
#include <algorithm>

// Convenient type name alias
using Dictionary = std::map<std::string, std::string>;

// English-Spanish word list
Dictionary word_map;

// Load the list of words from a file
void load_words(std::string filename, Dictionary& words) {
    std::ifstream in(filename);
    if (in.good()) {  // Make sure the file was opened properly
        std::string english_word, spanish_word;
        while (in >> spanish_word >> english_word) // Read until end of file
            words[spanish_word] = english_word;
    }
    else
        std::cout << "Unable to load in the file\n";
}

int main() {
    // Load words from file
    load_words("engspanwords.txt", word_map);

    std::string word = "";
    while (word != "quit") {  // Loop until user presses return by itself
        // Obtain word from the user
        std::cout << "Enter Spanish word: ";
        std::cin >> word;
        std::cout << word << ": " << word_map[word] << '\n';
    }
}
```

Note that Listing 21.12 (translatemap.cpp) uses a map in which both the keys and values are strings. Listing 21.12 (translatemap.cpp) in conjunction with the text file engspanwords.txt containing the following data:

```
uno one
dos two
tres three
```

```
cuatro four
cinco five
seis six
siete seven
ocho eight
```

translates the same words as Listing 21.11 (translateif.cpp), but we do not need to touch the program's logic at all to expand the program's vocabulary; all we need do is add the appropriate words to our text file. The words will become the *key* and *value* items in our map when the program runs. This is a significant difference if wish to include enough words to make the program practical.

Like `std::set`, the major C++ platforms use a red-black tree (see `https://en.wikipedia.org/wiki/Red-black_tree`) as the underlying data structure for the `std::map` class. The nature of this data structure makes key lookup within a map object very fast.

21.5 The `std::unordered_map` Data Type

The `std::unordered_map` type is another example of a C++ associative container. A `std::unordered_map` object works like a `std::map` object, but they each use a different underlying data structure. Most `std::map` implementations use a *red-black tree* (see `https://en.wikipedia.org/wiki/Red-black_tree`) to store the map's elements, while an unordered map uses a *hash table* (see `https://en.wikipedia.org/wiki/Hash_table`). This difference makes an unordered map faster in general than a map for accessing an element via its key. Map objects trade raw speed for the ability to access efficiently elements in order based on their keys. Unordered map objects truly are unordered; they provide no efficient way for clients to traverse their key-value pairs by order of their keys.

In order to use a `std::unordered_map` object within a program we must use the appropriate `#include` directive:

```
#include <unordered_map>
```

Like the `std::map` class, the `std::unordered_map` class is a template class, so when declaring an instance we must supply type parameters within angle brackets as shown in the following example:

```
std::unordered_map<std::string, int> my_map;
```

Here each integer in the map object has an associated string key.

Listing 21.13 (ordvsunord.cpp) adds in the same order equal strings to both a map object and an unordered map object. It then iterates over each container, printing each element as visited by the iterator.

Listing 21.13: `ordvsunord.cpp`

```
#include <iostream>
#include <string>
#include <map>
#include <unordered_map>

int main() {
    std::map<std::string, int> m;
    std::unordered_map<std::string, int> um;
```

```
    // Add some names in random order
    m["fred"] = um["fred"] = 23;
    m["adam"] = um["adam"] = 99;
    m["cathy"] = um["cathy"] = 2;
    m["wilma"] = um["wilma"] = 50;
    m["betty"] = um["betty"] = 19;
    m["roger"] = um["roger"] = 44;
    m["kim"] = um["kim"] = 7;
    m["doug"] = um["doug"] = 10;
    m["zach"] = um["zach"] = 5;
    m["tom"] = um["tom"] = 34;

    // Print the names in each container
    for (auto& p : m)
        std::cout << p.first << ":" << p.second << "  ";
    std::cout << '\n';
    std::cout << "-------------------\n";
    for (auto& p : um)
        std::cout << p.first << ":" << p.second << "  ";
    std::cout << '\n';
    std::cout << "===================\n";

    // Add some more names to each container
    m["tammy"] = um["tammy"] = 80;
    m["joey"] = um["joey"] = 80;
    m["abby"] = um["abby"] = 80;

    // Again, print the names in each container
    for (auto& p : m)
        std::cout << p.first << ":" << p.second << "  ";
    std::cout << '\n';
    std::cout << "-------------------\n";
    for (auto& p : um)
        std::cout << p.first << ":" << p.second << "  ";
    std::cout << '\n';
}
```

A run of Listing 21.13 (ordvsunord.cpp) on one system prints

```
adam:99  betty:19  cathy:2  doug:10  fred:23  kim:7  roger:44
tom:34  wilma:50  zach:5
-------------------
fred:23  roger:44  adam:99  kim:7  doug:10  cathy:2  wilma:50
betty:19  zach:5  tom:34
===================
abby:80  adam:99  betty:19  cathy:2  doug:10  fred:23  joey:80
kim:7  roger:44  tammy:80  tom:34  wilma:50  zach:5
-------------------
fred:23  roger:44  adam:99  kim:7  doug:10  cathy:2  wilma:50
abby:80  betty:19  zach:5  tom:34  tammy:80  joey:80
```

The output shows that we can iterate over the elements of a map object in the order of their keys. String keys are ordered lexicographically. The ordering of the unordered map appears random.

Listing 21.13 (ordvsunord.cpp) adds a few additional elements to both map objects. The insertions do

not disturb the lexicographical key order of the map object, but note that the insertions appear in seemingly random locations in the unordered map.

The ordering of the elements in an unordered map is not really random; a special function known as a *hash function* determines their position within the unordered map at the time of their insertion. This hashing process makes unordered maps slightly faster than regular map objects. If the speed of access is critical and accessing the elements in order is not necessary, a C++ `std::unordered_map` is the better choice.

21.6 Counting with Associative Containers

Associative containers are useful for counting things. We have experience using variables to count; recall Listing 6.3 (countup.cpp) and Listing 6.21 (startree.cpp). These programs counted one thing at a time, so they each use just one counter variable. In general, we need to use a separate variable for each count we manage. Listing 21.14 (countnegnonneg.cpp) uses a function to count the number of negative and nonnegative numbers in a vector of integers and returns a `std::pair` with the results:

Listing 21.14: `countnegnonneg.cpp`

```
#include <iostream>
#include <vector>
#include <tuple>

std::pair<int, int> count_neg_nonneg(const std::vector<int>& nums) {
    // Initialize counters
    int neg_count = 0, nonneg_count = 0;
    for (int num : nums)
        if (num < 0)
            neg_count++;
        else
            nonneg_count++;
    return std::make_pair(neg_count, nonneg_count);
}

int main() {
    auto counts = count_neg_nonneg({8, -3, -1, 7, 7, -2, 0, 3});
    std::cout << counts.first << " " << counts.second << '\n';

}
```

Since we needed to count two different kinds of things, we had to use two separate counters. In this case we used `neg_count` and `nonneg_count` to serve the role of counter variables.

As an aside, consider a variation of Listing 21.14 (countnegnonneg.cpp), shown in Listing 21.15 (countnegnonneg2.cpp).

Listing 21.15: `countnegnonneg2.cpp`

```
#include <iostream>
#include <vector>
#include <tuple>

// Make an alias for a more complicated type name
using IntPair = std::pair<int, int>;
```

```
IntPair count_neg_nonneg(const std::vector<int>& nums) {
    // Initialize counters
    int neg_count = 0, nonneg_count = 0;
    for (int num : nums)
        if (num < 0)
            neg_count++;
        else
            nonneg_count++;
    return {neg_count, nonneg_count}; //std::make_pair(neg_count, nonneg_count);
}

int main() {
    auto counts = count_neg_nonneg({8, -3, -1, 7, 7, -2, 0, 3});
    std::cout << counts.first << " " << counts.second << '\n';

}
```

Note that we created a type alias (`IntPair`) for the more complicated type (`std::pair<int, int>`). This is not particularly helpful in this simple program (we only use the type in one place), but it illustrates that we can use the `using` statement for type aliasing `std::pair` as easily as with `std::vector` or any other more complicated looking type name. We also simplified the `return` statement at the end of the `count_neg_nonneg` function. The compiler knows the return type of the function (declared as `IntPair`—a pair of integers), so it must return a pair object. The curly brace notation without any other verbiage indicates the arguments to the `std::pair<int, int>` class constructor that it should use when creating the object to return. The compiler automatically generates the machine code to call the pair constructor with the arguments within the curly braces.

Suppose we wish to keep track of the quantity of each letter that appears in a text file. There are 26 letters in the alphabet, so we would need 26 counter variables. Managing 26 different counter variables is inconvenient, so we could use a vector containing 26 integers. The element at index 0 could store the number of *A*s, the element at index 1 could keep track of the number of *B*s, etc. An associative container may be useful but is not necessary to solve our problem. Listing 21.16 (lettercount.cpp) provides one implementation.

Listing 21.16: `lettercount.cpp`

```
// Counts the number of occurrences of each
// letter in a text file.

#include <iostream>
#include <fstream>
#include <string>
#include <vector>     // To store the counters
#include <cctype>     // For toupper and isalpha

int main(int argc, char *argv[]) {
    if (argc < 2) {
        std::cout << "Usage: lettercount <filename>\n";
        std::cout << " where <filename> is the name of a text file.\n";
    }
    else {  // User provided file name
        std::string filename = argv[1];
        // Make counter vector 26 big, all filled with zeros
```

```
        std::vector<int> counters(26, 0);
        std::ifstream fin(filename);
        if (fin.good()) {     // Open the file for reading
            char ch;
            while (fin >> ch) {
                // Capitalize the letter
                ch = static_cast<char>(toupper(ch));
                if (isalpha(ch)) // Only count alphabetic characters
                    // Compute offset into counter vector
                    counters[ch - 'A']++;
            }
            // Report the counts for each letter
            ch = 'A';
            for (auto count : counters)
                std::cout << ch++ << ": " << count << '\n';
        }
        else
            std::cout << "Cannot open file for reading\n";
    }
}
```

The following paragraph appears in the *Declaration of Independence of the United States* (for simplicity all punctuation has been removed):

> When in the Course of human events it becomes necessary for one people to dissolve the political bands which have connected them with another and to assume among the powers of the earth the separate and equal station to which the Laws of Nature and of Nature's God entitle them a decent respect to the opinions of mankind requires that they should declare the causes which impel them to the separation

When Listing 21.16 (lettercount.cpp) processes this file it reports:

```
A: 26
B: 2
C: 13
D: 11
E: 50
F: 6
G: 2
H: 27
I: 17
J: 0
K: 1
L: 10
M: 9
N: 24
O: 27
P: 9
Q: 2
R: 14
S: 22
T: 37
U: 9
V: 3
W: 7
X: 0
Y: 2
Z: 0
```

In this case we knew we needed to count 26 different things (letters), so we were prepared with 26 counters (26 vector elements). The number of letters in the English alphabet does not change, so this approach works well.

So far so good, but what if we face a situation in which we must count multiple kinds of things, and we cannot know ahead of time how many kinds of things there will be to count? How can we determine how many counter variables to use in a program that attempts to solve such a problem?

The answer is this: We cannot know how many counter variables we will need, so we must use a different approach. With an associative container we can use the items we wish to count as the keys. The value associated with a key will be the count of that item. As a concrete example, Listing 21.17 (wordcount.cpp) reads the content of a text file containing words. After reading the file the program prints a count of each word. To simplify things, the text file contains only words with no punctuation. The user supplies the file name on the command line when launching the program (see Section 11.2.7).

Listing 21.17: `wordcount.cpp`

```
// Uses a map to count the number of occurrences of each
// word in a text file.

#include <iostream>
#include <fstream>
#include <string>
#include <map>
#include <algorithm>
#include <cctype>

int main(int argc, char *argv[]) {
    if (argc < 2) {
        std::cout << "Usage: wordcount <filename>\n";
        std::cout << " where <filename> is the name of a text file.\n";
    }
    else {  // User provided file name
        std::string filename = argv[1];
        std::map<std::string, int> counters; // Make a counting dictionary
        std::ifstream fin(filename);
        if (fin.good()) {     // Open the file for reading
            std::string word;
            while (fin >> word) {
                // Capitalize all the letters in the word
                transform(std::begin(word), std::end(word),
                          std::begin(word), toupper);
                counters[word]++;   // Increment counter for word
            }
            // Report the counts for each word
            for (auto item : counters)
                std::cout << item.first << ":" << item.second << '\n';
        }
        else
            std::cout << "Cannot open file for reading\n";
    }
}
```

If we send to Listing 21.17 (wordcount.cpp) the same excerpt as before from the *Declaration of Independence of the United States* it prints the following:

```
A:1
AMONG:1
AND:3
ANOTHER:1
ASSUME:1
BANDS:1
BECOMES:1
CAUSES:1
CONNECTED:1
COURSE:1
DECENT:1
DECLARE:1
DISSOLVE:1
EARTH:1
ENTITLE:1
EQUAL:1
EVENTS:1
FOR:1
GOD:1
HAVE:1
HUMAN:1
IMPEL:1
IN:1
IT:1
LAWS:1
MANKIND:1
NATURE:1
NATURE'S:1
NECESSARY:1
OF:5
ONE:1
OPINIONS:1
PEOPLE:1
POLITICAL:1
POWERS:1
REQUIRES:1
RESPECT:1
SEPARATE:1
SEPARATION:1
SHOULD:1
STATION:1
THAT:1
THE:9
THEM:3
THEY:1
TO:5
WHEN:1
WHICH:3
WITH:1
```

In Listing 21.17 (wordcount.cpp), since we cannot predict what words will appear in a document, we cannot use a separate variable for each counter. Instead, we use the document's words as keys in a map object. For each key in the map we associate an integer value that keeps track of the number of times the word appears in the file. We use a `map` instead of an `unordered_map` so that we can list the words and their counts in

lexicographical order.

In Listing 21.17 (wordcount.cpp) the statement

```
counters[word]++;
```

increments the integer mapped to the string `word`. If the map currently holds the string `word`, the expression simply adds one to the integer value; it the string `word` does not appear in the `counters` map object, the statement first inserts the string `word` with an associated value of 0 and then immediately increments it to 1. This is the exact behavior we need for our word counting program.

21.7 Grouping with Associative Containers

Associative containers are useful for grouping items. Like Listing 21.17 (wordcount.cpp), Listing 21.18 (groupwords.cpp) reads in the contents of a text file. Instead of counting the words, Listing 21.18 (groupwords.cpp) groups the words into sets based on the number of letters in the word. All the words containing only one letter are in one set, all the words containing two letters are in another set, etc.

Listing 21.18: `groupwords.cpp`

```
// Uses a map to group words in a document by their length.

#include <iostream>
#include <fstream>
#include <string>
#include <map>
#include <set>
#include <algorithm>
#include <cctype>   // For toupper

// Conveniently send to an output stream a set of any printable type
template<typename T>
std::ostream& operator<<(std::ostream& os, const std::set<T>& s) {
    os << '{';
    auto iter = std::begin(s);      // iter points to first element initially
    const auto iend  = std::end(s); // iend always points just past the end
    if (iter != iend)          // Set empty?
        os << *iter++;         // No?  Print first element
    while (iter != iend)       // More left?
        os << ", " << *iter++; // Print element, comma, move to next element
    os << '}';
    return os;
}

int main(int argc, char *argv[]) {
    if (argc < 2) {
        std::cout << "Usage: wordgroup <filename>\n";
        std::cout << " where <filename> is the name of a text file.\n";
    }
    else {  // User provided file name
        std::string filename = argv[1];
        // Make a grouping map of sets
        std::map<int, std::set<std::string>> groups;
        std::ifstream fin(filename);
```

```
        if (fin.good()) {     // Open the file for reading
            std::string word;
            while (fin >> word) {
                // Capitalize all the letters in the word
                transform(std::begin(word), std::end(word),
                          std::begin(word), toupper);
                // Add word to appropriate set based on the word's length
                groups[word.length()].insert(word);
            }
            // Report the counts for each word
            for (auto item : groups)
                std::cout << item.first << ":" << item.second << '\n';
        }
        else
            std::cout << "Cannot open file for reading\n";
    }
}
```

The following shows a sample run of Listing 21.18 (groupwords.cpp) on our snippet from the *Declaration of Independence*:

```
1:{A}
2:{IN, IT, OF, TO}
3:{AND, FOR, GOD, ONE, THE}
4:{HAVE, LAWS, THAT, THEM, THEY, WHEN, WITH}
5:{AMONG, BANDS, EARTH, EQUAL, HUMAN, IMPEL, WHICH}
6:{ASSUME, CAUSES, COURSE, DECENT, EVENTS, NATURE, PEOPLE, POWERS, SHOULD}
7:{ANOTHER, BECOMES, DECLARE, ENTITLE, MANKIND, RESPECT, STATION}
8:{DISSOLVE, NATURE'S, OPINIONS, REQUIRES, SEPARATE}
9:{CONNECTED, NECESSARY, POLITICAL}
10:{SEPARATION}
```

Each integer key represents the length of all the strings in the set it oversees.

In this example the need for a map object is not as compelling as in Listing 21.17 (wordcount.cpp), because in practice the length of English words is limited. We could instead use a vector of 50 sets. Only rarely used, highly technical words exceed 50 letters. We could handle these "too-long" words in a special way, perhaps storing them in the set in the last position in the vector with other words of more than 49 letters. Our program could use special processing for this set of long words if it ever becomes nonempty. This approach under typical circumstances would result in a number of empty sets at higher indices because most English text contains words of at most about 20 letters. The advantage of a map is that it stores only what it needs. Listing 21.19 (groupwordsvector.cpp) is close transliteration of Listing 21.18 (groupwords.cpp) that uses a vector in place of a map.

Listing 21.19: `groupwordsvector.cpp`

```
// Uses a vector to group words in a document by their length.

#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <set>
#include <algorithm>
#include <cctype>
```

```
// Conveniently send to an output stream a set of any printable type
template<typename T>
std::ostream& operator<<(std::ostream& os, const std::set<T>& s) {
    os << '{';
    auto iter = std::begin(s);      // iter points to first element initially
    const auto iend  = std::end(s); // iend always points just past the end
    if (iter != iend)          // Set empty?
        os << *iter++;         // No?  Print first element
    while (iter != iend)       // More left?
        os << ", " << *iter++; // Print element, comma, move to next element
    os << '}';
    return os;
}

const size_t MAX_WORD_LENGTH = 20;

int main(int argc, char *argv[]) {
    if (argc < 2) {
        std::cout << "Usage: wordgroup <filename>\n";
        std::cout << " where <filename> is the name of a text file.\n";
    }
    else {  // User provided file name
        std::string filename = argv[1];
        // Make a grouping map of sets
        std::vector<std::set<std::string>> groups(MAX_WORD_LENGTH);
        std::ifstream fin(filename);
        if (fin.good()) {     // Open the file for reading
            std::string word;
            while (fin >> word) {
                // Capitalize all the letters in the word
                std::transform(std::begin(word), std::end(word),
                               std::begin(word), toupper);
                // Add word to appropriate et based on the word's length
                if (word.length() > MAX_WORD_LENGTH)
                    std::cout << "Ignoring word, too long\n";
                groups[word.length()].insert(word);
            }
            // Report the counts for each word
            for (size_t i = 1; i < groups.size(); i++)
                std::cout << i << ":" << groups[i] << '\n';
        }
        else
            std::cout << "Cannot open file for reading\n";
    }
}
```

We can see the unused space in the output of Listing 21.19 (groupwordsvector.cpp):

```
1:{A}
2:{IN, IT, OF, TO}
3:{AND, FOR, GOD, ONE, THE}
4:{HAVE, LAWS, THAT, THEM, THEY, WHEN, WITH}
5:{AMONG, BANDS, EARTH, EQUAL, HUMAN, IMPEL, WHICH}
```

```
6:{ASSUME, CAUSES, COURSE, DECENT, EVENTS, NATURE, PEOPLE, POWERS, SHOULD}
7:{ANOTHER, BECOMES, DECLARE, ENTITLE, MANKIND, RESPECT, STATION}
8:{DISSOLVE, NATURE'S, OPINIONS, REQUIRES, SEPARATE}
9:{CONNECTED, NECESSARY, POLITICAL}
10:{SEPARATION}
11:{}
12:{}
13:{}
14:{}
15:{}
16:{}
17:{}
18:{}
19:{}
```

In a different document there could be empty sets in the groups representing the words with fewer than 11 letters.

21.8 Memoization

We know a program can use variables to remember values as it executes. A programmer must be able to predict the number of values the program must manage in order to write enough variables in the code. An associative container provides an opportunity to create an arbitrary amount of new storage during a program's execution. We will consider a simple problem that demonstrates the value of the dynamic storage provided by `unordered_map` objects.

Section 10.5 introduced the Fibonacci number sequence. That section provided a function to compute the nth Fibonacci number, reproduced here:

```
// Returns the nth Fibonacci number
int fibonacci(int n) {
    if (n <= 0)
        return 0;
    else if (n == 1)
        return 1;
    else
        return fibonacci(n - 2) + fibonacci(n - 1);
}
```

This `fibonacci` function is correct, but it does not scale well—its execution time grows significantly as its parameter, `n`, increases. The problem is this: when computing a solution for a particular Fibonacci number the function can repeat exactly the same work multiple times. Figure 10.2 illustrates the repetitive work performed by the call `fibonacci(5)`. As we can see from the figure, the function's recursive execution calls `fibonacci(1)` five times during the computation of `fibonacci(5)`. We can verify the results shown in Figure 10.2 by augmenting our `fibonacci` function with a global map that counts function calls.

Listing 21.20 (fibonacciinstrumented.cpp).

Listing 21.20: `fibonacciinstrumented.cpp`

```
#include <iostream>
#include <map>
```

```
// This associative container will keep track of the number of
// calls to the fibonacci function.
std::map<int, int> call_counter;

// Returns the nth Fibonacci number
int fibonacci(int n) {
    // Count the call
    call_counter[n]++;

    if (n <= 0)
        return 0;
    else if (n == 1)
        return 1;
    else
        return fibonacci(n - 2) + fibonacci(n - 1);
}

int main() {
    // Call fibonacci(5)
    std::cout << "fibonacci(5) = " << fibonacci(5) << "\n\n";

    // Report the total number of calls to the fibonacci function
    std::cout << "Argument    Calls" << '\n';
    std::cout << "-----------------\n";
    for (auto& counts: call_counter)
        std::cout << "   " << counts.first << "         "
                  << counts.second << '\n';
}
```

Listing 21.20 (fibonacciinstrumented.cpp) prints

```
fibonacci(5) = 5

Argument    Calls
-----------------
   0         3
   1         5
   2         3
   3         2
   4         1
   5         1
```

Note the results that Listing 21.20 (fibonacciinstrumented.cpp) prints agree exactly with the call count shown in Figure 10.2. As we compute larger Fibonacci numbers, the amount of repeated work worsens quickly; for example, the call `fibonacci(20)` recursively calls `fibonacci(1)` 6,765 times! For additional emphasis, the call `fibonacci(35)` recursively calls `fibonacci(1)`, `fibonacci(2)`, `fibonacci(3)`, `fibonacci(4)`, and `fibonacci(5)` over one million times *each*! We may be tempted to care less about the program's repeated work—after all, it is the computer doing the work, not us. Unfortunately, the computer, even though it is very fast, requires some amount of time to perform any task. As we multiply the number of tasks a program must do to solve a problem, the time to compute the solution increases, and, in the case of the `fibonacci` function, the time increases dramatically.

We can improve the performance of our `fibonacci` function using a technique known as *memoization*

(not to be confused with the word *memorization* which means to commit something to memory). Memoization is an algorithm design technique that records the result of a specific computation so that result can be used as needed at a later time during the algorithm's execution. It is as if the executing program "makes a note to itself" or "stores the result in a memo." When the program needs the result of an identical computation in the future, if simply reads the memo with the answer it stored earlier. In this way the program avoids repeating the work. Memoization is especially useful for problems that consist of subproblems that overlap and appear to require multiple computations with identical input.

The following function uses an unordered map object to cache previously computed Fibonacci numbers:

```
// Returns the nth Fibonacci number.  Caches a
// recursively computed result to be used when needed
// in the future.  Provides a huge performance improvement
// over the recursive version.
int fibonacci2(int n) {
    // Map for caching the results of the fib function
    // ans is declared static so it retains its value between
    // calls to fibonacci2.
    // Precomputes the results for 0 and 1.
    static unordered_map<int, int> ans {{0, 0}, {1, 1}};
    if (n > 1 && ans[n] == 0)  // Need to compute?
        ans[n] = fibonacci2(n - 2) + fibonacci2(n - 1);
    return ans[n];
}
```

The `fibonacci2` function uses an unordered map as a cache of stored values that persist for the duration of the program's execution. Since the `ans` local variable is `static`, the run-time environment creates and initializes it with keys 0 and 1 before executing the `main` function. When a caller invokes `fibonacci2`, if the answer is not in `ans`, it computes the result and stores the result in `ans` for future use. This not only helps future calls to the function from the outside, but it also speeds up recursive invocations that must perform the same work.

Listing 21.21 (fibonacci.h) declares the interfaces for `fibonacci` and `fibonacci2`, two functions that compute the nth Fibonacci number.

Listing 21.21: `fibonacci.h`

```
#ifndef FIBONACCI_H_DEFINED
#define FIBONACCI_H_DEFINED

// Values in the range 0...18,446,744,073,709,551,615 on Windows
using Integer = unsigned long long;

// Returns the nth Fibonacci number using
// recursion without memoization
Integer fibonacci(unsigned n);


// Returns the nth Fibonacci number using recursion
// with memoization.  Provides a huge performance improvement
// over the pure recursive version.
Integer fibonacci2(unsigned n);

#endif
```

The Fibonacci numbers grow very quickly, so the functions in Listing 21.21 (fibonacci.h) return `unsigned long long` values. For convenience it introduces the type alias `Integer` for `unsigned long long` to simplify the code. Under Visual C++, for example, this type supports values in the range $0 \ldots 18,446,744,073,709,551,615$, inclusive. Listing 21.22 (fibonacci.cpp) provides the implementations for the two functions.

Listing 21.22: fibonacci.cpp

```
#include <unordered_map>
#include "fibonacci.h"

// Returns the nth Fibonacci number
Integer fibonacci(unsigned n) {
    if (n <= 0)
        return 0;
    else if (n == 1)
        return 1;
    else
        return fibonacci(n - 2) + fibonacci(n - 1);
}


// Returns the nth Fibonacci number.  Caches a
// recursively computed result to be used when needed
// in the future.  Provides a huge performance improvement
// over the recursive version.
Integer fibonacci2(unsigned n) {
    // ans is a map for caching the results of the fibonacci2 function.
    // Declared static so it persists between function calls.
    // Precomputes the results for 0 and 1.
    static std::unordered_map<unsigned, Integer> ans {{0, 0}, {1, 1}};
    if (n > 1 && ans[n] == 0)
        ans[n] = fibonacci2(n - 2) + fibonacci2(n - 1);
    return ans[n];
}
```

When compiled and linked with the code in Listing 21.22 (fibonacci.cpp), Listing 21.23 (testfib.cpp) verifies that the `fibonacci2` function produces the same results as the `fibonacci` function when given the same arguments. It tests `unsigned` values up to 50. You will observe that as the arguments become larger, the program takes longer to print the results.

Listing 21.23: testfib.cpp

```
#include <iostream>
#include "fibonacci.h"

// Compares the behaviors of the fibonacci and fibonacci2 functions.
int main() {
    // Print the first 50 Fibonacci numbers
    for (unsigned i = 0; i <= 50; i++)
        std::cout << i << ": " << fibonacci(i) << " "
                  << fibonacci2(i) << '\n';
}
```

The following shows the first 20 lines of the output of Listing 21.23 (testfib.cpp):

```
0: 0 0
1: 1 1
2: 1 1
3: 2 2
4: 3 3
5: 5 5
6: 8 8
7: 13 13
8: 21 21
9: 34 34
10: 55 55
11: 89 89
12: 144 144
13: 233 233
14: 377 377
15: 610 610
16: 987 987
17: 1597 1597
18: 2584 2584
19: 4181 4181
20: 6765 6765
```

In total, all 50 lines match, and that gives us confidence that our memoized version is correct.

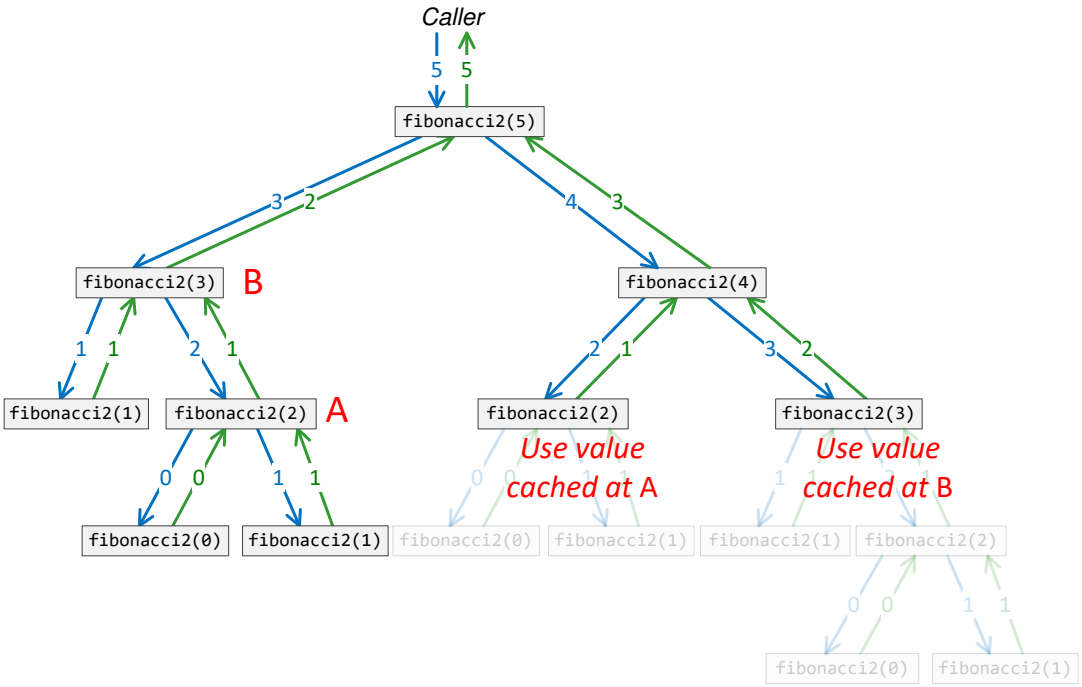
Listing 21.24 (timefib.cpp) reveals just much faster the memoized function executes compared to the non-memoized version.

Listing 21.24: `timefib.cpp`

```
#include <iostream>
#include <ctime>
#include <functional>

#include "fibonacci.h"

// f is a function that accepts a single parameter, n is a number
// to pass to the function.
// Measures the time for function f to execute given parameter n.
// Returns the cumulative elapsed time in milliseconds.
unsigned time_it(const std::function<Integer(unsigned)>& f, unsigned n) {
    clock_t start_time = clock();
    f(n);
    clock_t end_time = clock();
    return static_cast<unsigned>(end_time - start_time); // Return elapsed time
}

// Tests the performance of the fibonacci and fibonacci2 functions.
int main() {
    auto t1 = time_it(fibonacci, 50);
    auto t2 = time_it(fibonacci2, 50);
    std::cout << "Time: " << "fibonacci = "<< t1 << " msec, fibonacci2 = "
              << t2 << " msec\n";
}
```

One run of Listing 21.24 (timefib.cpp) prints

Figure 21.1 The hierarchy of recursive function calls that result from the call `fibonacci2(5)`. Note that the calls of `fibonacci(2)` and `fibonacci(3)` on the right side of the tree need no recursive calls; this is because their earlier calls on the left side of the tree stored their results in the map for instant retrieval.




```
Time: fibonacci = 111430 msec, fibonacci2 = 1 msec
```

The original recursive version requires over 111 seconds (almost two minutes), while the memoized recursive version takes only 1 millisecond. During this run the memoized version was over 100,000 times faster. The `fibonacci2` function is so fast because it avoids all the redundant recursive calls the `fibonacci` function must compute.

Listing 21.24 (timefib.cpp) provides an honest test, as it forces `fibonacci2` to compute the 50th Fibonacci number right away with no previous `fibonacci2` invocations. Once completing a call of `fibonacci2(`n`)`, all of the Fibonacci numbers from $0 \ldots n$ will be present in the function's `unordered_map` cache of precomputed values. This means future invocations involving any values in that range will be very fast. Any program that must compute Fibonacci numbers multiple times during their execution will especially benefit from the memoized version.

Figure 21.1 shows the recursion tree for our memoized Fibonacci function computing the fifth Fibonacci number. The figure shows only nine invocations of `fibonacci2`, compared to 15 invocations of the non-memoized `fibonacci` function. If you add the function call counting instrumentation used in Listing 21.20 (fibonacciinstrumented.cpp) to `fibonacci2`, you will find the numbers it reports agrees with Figure 21.1.

Chapter 22

Handling Exceptions

CAUTION! CHAPTER UNDER CONSTRUCTION

In our programming experience so far we have encountered several kinds of run-time errors. To this point, all of our run-time errors have resulted in the executing program's termination. C++ provides an *exception handling* framework that allows programmers to deal with certain kinds of run-time errors. Rather than always terminating the program's execution, the exception handling infrastructure enables programmers to detect a problem and execute code to correct the issue or manage it in other ways. This chapter explores C++'s exception handling mechanism.

22.1 Motivation

Algorithm design can be tricky because the details are crucial. It may be straightforward to write an algorithm to solve a problem in the general case, but there may be a number of special cases that must all be addressed within the algorithm for the algorithm to be correct. Some of these special cases might occur rarely under the most extraordinary circumstances. For the code implementing the algorithm to be robust, these exceptional cases must be handled properly; however, adding the necessary details to the algorithm may render it overly complex and difficult to construct correctly. Such an overly complex algorithm would be difficult for others to read and understand, and it would be harder to debug and extend.

Ideally, a developer would express the algorithm in its general form including any common special cases. Exceptional situations that should arise rarely, along with a strategy to handle them, could appear elsewhere, perhaps as an annotation to the algorithm. Thus, the algorithm is kept focused on solving the problem at hand, and measures to deal with exceptional cases are handled elsewhere.

C++'s exception handling infrastructure allows programmers to cleanly separate the code that implements the focused algorithm from the code that deals with exceptional situations that the algorithm may face. This approach is more modular and encourages the development of code that is cleaner and easier to maintain and debug.

An *exception* is an exceptional event that occurs during a program's execution. An exception always is possible, but it should be a relatively rare event. If it were not rare, it would be a customary or expected event, and the program should handle it as part of its normal processing.

An exception almost always represents a problem, usually some sort of run-time error. Suppose we have

©2018 Richard L. Halterman

a vector named `v` and an integer `i`. Unless $0 \leq$ `i` $<$ `v.size()`, the expression `v[i]` attempts to access an element outside the vector's bounds. The `operator[]` method performs no bounds checking; thus, an executing program using the expression `v[i]` would represent undefined behavior. Consider instead the expression `v.at(i)`. The `at` method works just like `operator[]`, except `at` does check the vector's bounds. If `i` $\geq$ `v.size()` or `i` $<$ 0, the expression `v.at(i)` represents an exceptional situation, and we say the vector's `at` method *throws*, or *raises*, an exception.

Without taking advantage of C++'s exception handling infrastructure, the following statement:

```
v.at(i) = 4;
```

could terminate the program with an error message. While this is better than undefined behavior, it is not ideal. The program crashes. When a real application crashes the user may lose unsaved data. Badly behaving programs terminate unexpectedly. It would be better to defend against the out-of-bounds index and keep the program alive.

An algorithm could handle many potential problems itself. For example, a programmer can use an `if` statement to test to see if a vector's index is within the proper bounds:

```
if (0 <= i && i < v.size())  // Ensure i is in range
    v.at(i) = 4;
```

However, if the programmer accesses a vector in several different locations within a given function or method and the index variable can vary from place to place, the number of `if` statements necessary to ensure safe vector access might obscure the overall logic of the function or method.

22.2 Exception Examples

Consider Listing 22.1 (vectorboundscrash.cpp). The program crashes whenever the user enters a value equal to an index outside the range of vector `nums`.

Listing 22.1: `vectorboundscrash.cpp`

```
#include <iostream>
#include <vector>

int main() {
    std::vector<double> nums { 1.0, 2.0, 3.0 };
    int input;
    std::cout << "Enter an index: ";
    std::cin >> input;
    std::cout << nums.at(input) << '\n';
}
```

The `at` method guarantees that an illegal access will not go undetected. To intercept the problem at run time and prevent the program from terminating due to an error, we use a `try`/`catch` block, which consists of two parts: a `try` block and a `catch` block. To form a `try`/`catch` block we

1. wrap the code that has the potential to throw an exception in a `try` block, and
2. provide code to execute only in the event of an exception in a `catch` block.

Listing 22.2 (vectorboundsexcept.cpp) adds the exception handling code to Listing 22.1 (vectorboundscrash.cpp).

Listing 22.2: vectorboundsexcept.cpp

```
#include <iostream>
#include <vector>

int main() {
    std::vector<double> nums { 1.0, 2.0, 3.0 };
    int input;
    std::cout << "Enter an index: ";
    std::cin >> input;
    try {
        std::cout << nums.at(input) << '\n';
    }
    catch (std::exception& e) {
        std::cout << e.what() << '\n';
    }
}
```

If a user enters a value less than zero or greater than two when executing Listing 22.2 (vectorboundsexcept.cpp), the program will print an error message and technically terminate successfully instead of crashing.

Both `try` and `catch` are keywords in C++. Unlike the bodies of the structured statements such as `if` and `while`, the statements within a `try` block and statements within a `catch` block must appear within curly braces, even if only one statement appears in the section.

Every `try` block or `catch` block must surround its statements with curly braces—even if it contains just one statement.

The variable `e` within the `catch` block of Listing 22.2 (vectorboundsexcept.cpp) is a reference to an `exception` object. The `exception` class is part of the standard C++ library and is the base class for all the standard exceptions. The `exception` class provides a `what` method that returns a string. The `exception::what` method's string is a message containing information about the exception. The C++ standard does not specify the exact message, but at the very least the `exception::what` method indicates the kind of exception caught. The following shows what the program reports under Visual C++ when the user enters the value 3:

```
Enter an index: 3
invalid std::vector<T> subscript
```

While Listing 22.2 (vectorboundsexcept.cpp) does not technically crash, its behavior is not much different from an actual abnormal program termination. Listing 22.3 (betterboundsexcept.cpp) provides a more practical example. This improved version uses a loop to continuously request integer values until the user supplies one that does not throw an exception.

Listing 22.3: betterboundsexcept.cpp

```
#include <iostream>
#include <vector>
```

```
int main() {
    std::vector<double> nums { 1.0, 2.0, 3.0 };
    int input;
    while (true) {
        std::cout << "Enter an index: ";
        std::cin >> input;
        try {
            std::cout << nums.at(input) << '\n';
            break;  // Printed successfully, so break out of loop
        }
        catch (std::exception&) {
            std::cout << "Index is out of range.  Please try again.\n";
        }
    }
}
```

What makes the exception code

```
try {
    std::cout << nums.at(input) << '\n';
}
catch (std::exception& e) {
    std::cout << e.what() << '\n';
}
```

better than the conditional code

```
if (0 <= input && input < nums.size()) {
    std::cout << nums.at(input) << '\n';
}
else {
    std::cout << "Vector out of bounds exception\n";
}
```

is not immediately obvious from this simple example. To see the real capability of exceptions, consider Listing 22.4 (callchainexception.cpp).

Listing 22.4: callchainexception.cpp

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

// Used to keep track of object creation and destruction
class Tracker {
    std::string name;
  public:
    Tracker(const std::string& s): name(s) {
        std::cout << "Creating Tracker (" << name << ")\n";
    }
    ~Tracker() {
        std::cout << "Destroying Tracker (" << name << ")\n";
    }
};
```

```
bool find(const std::vector<int>& v, int lower, int upper) {
    std::cout << "------[Entering find]------\n";
    Tracker obj("find");  // Local tracking object
    bool result = false;  // Not there by default
    int seek;             // What to look for
    std::cout << "Enter item to locate:  ";
    std::cin >> seek;
    for (int i = lower; i < upper; i++)
        if (v.at(i) == seek) {
            result = true;   // Found it
            break;
        }
    std::cout << "------[Leaving find]------\n";
    return result;
}

void process(const std::vector<int>& v) {
    std::cout << "------[Entering process]------\n";
    Tracker obj("process");  // Local tracking object
    int low, high;
    std::cout << "Enter a range: ";
    std::cin >> low >> high;
    if (find(v, low, high))
        std::cout << "Found it\n";
    else
        std::cout << "Not there\n";
    std::cout << "------[Leaving process]------\n";

}

// Global tracking object
Tracker obj("global");

int main() {
    std::cout << "------[Entering main]------\n";
    Tracker obj("main");  // Local tracking object
    std::vector<int> nums { 11, 42, 23 };
    try {
        process(nums);
    }
    catch (std::exception& e) {
        std::cout << "nums vector bounds exceeded\n";
    }
    std::cout << "------[Leaving main]------\n";
}
```

Listing 22.4 (callchainexception.cpp) is specially crafted to provide insight into the program's exact execution path. Each function prints a message at its beginning and end so we can see when the code within a function is active. Each function also declares a local `Tracker` object which is identified by the name of that function. An executing function constructs a local object at its point of declaration and automatically destroys the object when the function returns to its caller. These local objects, therefore, have a lifetime that mirrors the lifetime of the function that manages them. Finally, Listing 22.4 (callchainexception.cpp) declares a global `Tracker` object. An executing program creates global objects before it calls `main` and

automatically destroys them after main completes.

In Listing 22.4 (callchainexception.cpp) the vector nums, declared in main, holds three elements. The main function passes its nums vector to process. During its call to process, the main function has no control over the process function's attempts to access an element outside the range of nums. The process function, however, does not itself attempt to access any elements within the vector that main supplies. Instead, process passes the vector on to the find function. It is in find where problems may arise because find calls the vector's at method. If the find function calls the vector's at method with an out-of-bounds index, at will throw an exception. Observe that unlike in Listing 22.2 (vectorboundsexcept.cpp), the method that calls std::vector<int>::at does *not* do so within a try/catch block. This means find cannot catch any exceptions that std::vector<int>::at might throw.

The following shows a sample run of Listing 22.4 (callchainexception.cpp):

```
Creating Tracker (global)
------[Entering main]------
Creating Tracker (main)
------[Entering process]------
Creating Tracker (process)
Enter a range: 0 2
------[Entering find]------
Creating Tracker (find)
Enter item to locate:  4
------[Leaving find]------
Destroying Tracker (find)
Not there
------[Leaving process]------
Destroying Tracker (process)
------[Leaving main]------
Destroying Tracker (main)
Destroying Tracker (global)
```

In this output of Listing 22.4 (callchainexception.cpp) we see the following sequence of events:

1. The program creates the global Tracker object.
2. The program calls main.
3. The main function creates its Tracker object.
4. main calls process.
5. The process function creates its Tracker object.
6. The process function obtains the user's input for range values. Here the users entered the range 0...2.
7. The process function calls find.
8. The find function creates its Tracker object.
9. The find function obtains the user's input for the element to seek. Here the user enters 4 (which is not present in the vector).
10. The find function executes its last statement.
11. When find returns to process, the program destroys find's local Tracker object.
12. process uses find's result to print the message that the element sought is not present in the vector.
13. The process function executes its last statement.
14. When process returns to main, the program destroys process's local Tracker object.
15. The main function executes its last statement.
16. When main returns, the program destroys main's local Tracker object.
17. When the program terminates it destroys the global Tracker object.

Figure 22.1 Function call chain without an exception. The `std::vector<int>::at` method, the find function, and the `process` function all complete their work and return as usual to their callers.

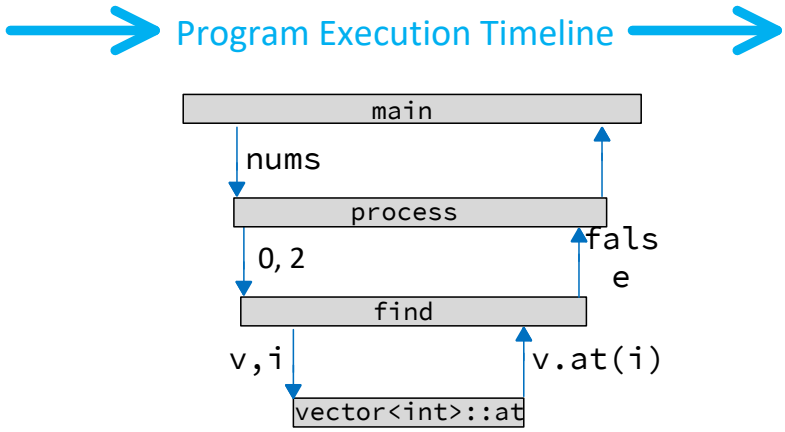


Figure 22.1 provides a more graphical representation of this process.

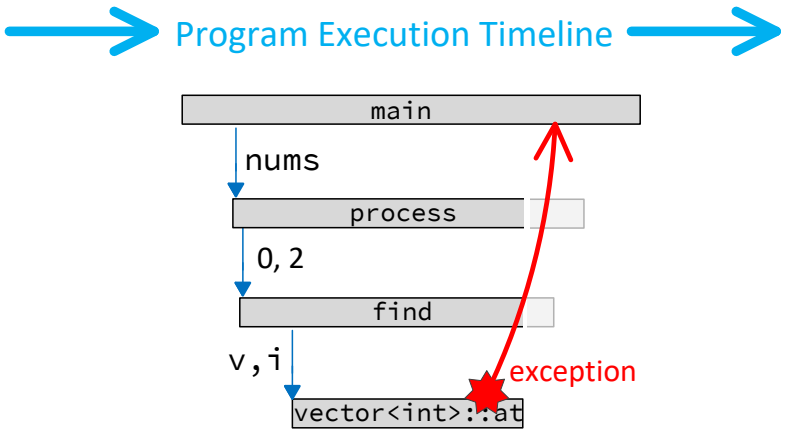
In this case the user did not enter a range in the `process` function that would cause a problem in find, so no exceptions arise. Since the program's execution produced no exceptions, it did not execute the printing statement in the `catch` block.

To see what happens during an exception consider the following sample run of Listing 22.4 (callchainexception.cpp):

```
Creating Tracker (global)
------[Entering main]------
Creating Tracker (main)
------[Entering process]------
Creating Tracker (process)
Enter a range: 10 20
------[Entering find]------
Creating Tracker (find)
Enter item to locate:  4
Destroying Tracker (find)
Destroying Tracker (process)
nums vector bounds exceeded
------[Leaving main]------
Destroying Tracker (main)
Destroying Tracker (global)
```

The user supplied range of 10. . . 20 definitely is outside the bounds of `main`'s `nums` vector. Because the program does not print find's exit message we know the find function does not complete its execution as usual. This is because the `std::vector<int>::at` method called by find throws an exception. Furthermore, it does not appear that find returns normally to `process`, since `process` does not print its exit message either. We do see, though, that the program does properly destroy the the local `Tracker`

Figure 22.2 Function call chain interrupted by an exception. Neither the `std::vector<int>::at` method, the `find` function, nor the `process` function completes its work and returns as usual to its caller. Control passes back to the `catch` block associated with the nearest `try` block in the call chain.



objects owned by `find` and `process`. The program's execution path lands immediately back in `main`, which we see does exit normally after executing the statement in its `catch` block. The program finishes by destroying `main`'s `Tracker` object followed by destroying the global `Tracker` object. Figure 22.2 better illustrates the program's execution.

Listing 22.4 (callchainexception.cpp) demonstrates that exceptions have the potential to radically alter the normal function call and return pathways. In one sense an exception works like a "super `goto`" statement. The program's execution jumps back to a previous point in its execution sequence, bypassing all functions in the call chain in between. Recall Section 6.3.2 discouraged the use of the `goto` statement except for it use to exit from within a nested loop. This is because the logic in code that uses indiscriminate `goto`s instead of the structured branching and looping constructs such as `if/else`, `switch`, `while`, and `for` tends to be obscure and difficult to extend and maintain. C++'s exception mechanism is different than a "super `goto`" in several important ways:

- A `goto` statement cannot jump to code outside of the function in which it appears. Exceptions have no such limitation.
- The programmer must specify a destination (label) for a `goto` statement. In the case of Listing 22.4 (callchainexception.cpp), the `std::vector<int>::at` method throws the exception. We can call `at` from many different functions within a single program and use `at` in many different programs. The code within the `at` method cannot possibly know where to go if it encounters an exceptional situation. An exception simply redirects the program's execution to the most recent `try/catch` block in the call chain that can handle the exception.

Listing 22.5 (badinput.cpp) follows.

Listing 22.5: badinput.cpp

```
#include <iostream>
#include <fstream>

// Sum the values the user enters
int main() {
    int input = 0, sum = 0;
    // Enable exceptions in the cin object
    std::cin.exceptions(std::ifstream::badbit | std::ifstream::failbit);
    std::cout << "Please enter integers to sum, 999 ends list: ";
    while (input != 999) {
        try {
            std::cin >> input;   // Watch for faulty (non-integer) input
            if (input != 999)
                sum += input; // Do not not include the terminating 999
        }
        catch (std::exception& e) {
            std::cout << "****Non-integer input detected\n";
            //cin.exceptions(std::ifstream::badbit | std::ifstream::failbit);
            std::cin.clear();  // Clear I/O error
            std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
            //std::cout << e.what() << '\n';
        }
    }
    std::cout << "Sum = " << sum << '\n';
}
```

Listing 22.6 (strboundsexcept.cpp) follows.

Listing 22.6: strboundsexcept.cpp

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

int main() {
    std::string s = "Wow";
    std::cout << s[3] << '\n';
    try {
        std::cout << s.at(3) << '\n';
    }
    catch (std::exception& e) {
        std::cout << e.what() << '\n';
    }
}
```

Listing 22.7 (filereadexcept.cpp) follows.

Listing 22.7: filereadexcept.cpp

```
#include <iostream>
#include <fstream>
#include <vector>

int main() {
    try {
```

```
        std::ifstream fin("data.dat");
        if (fin.good()) {
            int n;
            fin >> n;   // Size of dataset
            std::vector<int> data(n);  // Allocate vector
            for (int i = 0; i <= n; i++ ) { // Error: should be <
                int value;
                fin >> value;
                data.at(i) = value;
            }
            // Print the values
            for (auto value : data)
                std::cout << value << ' ';
            std::cout << '\n';

        }
        else
            std::cout << "File does not exist\n";
    }
    catch (std::exception& e) {
        std::cout << e.what() << '\n';
    }
}
```

22.3 Custom Exceptions

The standard C++ library has a limited number of standard exceptions. We can create our own custom exceptions for specialized error handling that our applications may require. Listing 22.8 (customfilereadexcept.cpp) defines a custom exception class, `FileNotFoundException`, derived from the standard `std::exception` class.

Listing 22.8: `customfilereadexcept.cpp`

```
#include <iostream>
#include <fstream>
#include <vector>
#include <string>

// Exception object to thrown when a client attempts to
// open a text file via a name that does correspond to a
// file in the current working directory.
class FileNotFoundException : public std::exception {
    std::string message;  // Identifies the exception and filename
  public:
    // Constructor establishes the exception object's message
    FileNotFoundException(const std::string& fname):
        message("File \"" + fname + "\" not found") {}

    // Reveal message to clients
    const char *what() const {
        return message.c_str();
    }
};
```

```
// Creates and returns a vector of integers from data stored
// in a text file.
// filename: the name of the text file containing the data
// Returns a vector containing the data in the file, if possible
std::vector<int> load_vector(const std::string& filename) {
    std::ifstream fin(filename);     // Open the text file for reading
    if (fin.good()) {                // Did the file open successfully?
        std::vector<int> result;     // Initially empty vector
        int n;
        fin >> n;                    // Size of data set
        for (int i = 0; i < n; i++ ) {
            int value;
            fin >> value;            // Read in a data value
            result.push_back(value); // Append it to the vector
        }
        return result;               // Return the populated vector
    }
    else   // Could not open the text file
        throw FileNotFoundException(filename);

}

int main() {
    try {
        std::vector<int> numbers = load_vector("values.data");
        for (int value : numbers)
            std::cout << value << ' ';
        std::cout << '\n';
    }
    catch (std::exception& e) {
        std::cout << e.what() << '\n';
    }
}
```

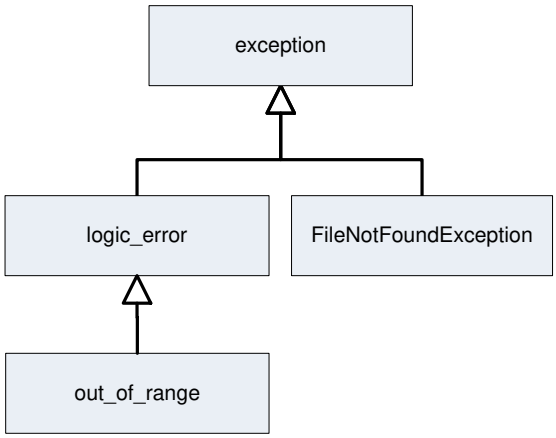
If we execute Listing 22.8 (customfilereadexcept.cpp) in a directory that does not contain a text file named values.data, it displays

```
File "values.data" not found
```

Since we derived the `FileNotFoundException` class from `std::exception`, any `FileNotFoundException` object also *is a* `std::exception` object (see Section 17.1). This enables the `main` function to catch a `FileNotFoundException` object in its `catch` body even though it is declared to catch `std::exception` objects.

The C++ standard library provides a number of exception classes, all derived from `std::exception`. The vector out-of-bounds exception objects thrown by the `at` method are instances of the standard `std::out_of_range` class. The `std::out_of_range` class is derived from the standard `logic_error` class which itself is derived from the standard `std::exception` class. Figure 22.3 shows the relationship among these exception classes.

Figure 22.3 The relationship among the exception classes involved in Listing 22.8 (customfilereadexcept.cpp). The classes `exception`, `logic_error`, and `out_of_range` are standard exception classes, `FileNotFoundException` is our custom exception class.



22.4 Catching Multiple Exceptions

Listing 22.9 (filereadmultiexcept1.cpp) has the potential to throw more than one type of exception.

Listing 22.9: filereadmultiexcept1.cpp

```
#include <iostream>
#include <fstream>
#include <vector>
#include <string>

// Exception object to thrown when a client attempts to
// open a text file via a name that does correspond to a
// file in the current working directory.
class FileNotFoundException : public std::exception {
    std::string message;  // Identifies the exception and filename
  public:
    // Constructor establishes the exception object's message
    FileNotFoundException(const std::string& fname):
        message("File \"" + fname + "\" not found") {}

    // Reveal message to clients
    const char *what() const {
        return message.c_str();
    }
};

// Creates and returns a vector of integers from data stored
// in a text file.
// filename: the name of the text file containing the data
```

```
// Returns a vector containing the data in the file, if possible
std::vector<int> load_vector(const std::string& filename) {
    std::ifstream fin(filename);         // Open the text file for reading
    if (fin.good()) {               // Did the file open successfully?
        int n;
        fin >> n;                   // Size of data set
        std::vector<int> result(n);      // Allocate space for the vector
        int value, i = 0;
        while (fin >> value)
            result.at(i++) = value; // Insert it into the vector
        return result;              // Return the populated vector
    }
    else   // Could not open the text file
        throw FileNotFoundException(filename);

}

int main() {
    try {
        std::vector<int> numbers = load_vector("values.data");
        for (int value : numbers)
            std::cout << value << ' ';
        std::cout << '\n';
    }
    catch (std::exception& e) {
        std::cout << e.what() << '\n';
    }
}
```

The text file containing the integers to load into the vector has a specific format. The first integer read from the file is not a data element; instead, it specifies the number of integers that follow. This gives us two options for creating and populating the vector:

1. Create the vector with sufficient size up front and then use a loop to place to individual elements in their proper places.
2. Create an empty vector and then use a loop to `push_back` the individual elements.

Option 1 is more efficient than Option 2. Successive calls to `std::vector<int>::push_back` must reallocate and copy elements multiple times. Option 2 is safer than Option 1 because the number of elements specified at the start may not agree with the actual number of data elements in the file. If the specified number is too small, the function will attempt to overrun the bounds of the vector.

Listing 22.9 (filereadmultiexcept1.cpp) opts for efficiency at the expense of safety. This means it depends on a correctly created data file. If the file does not exist, `load_vector` will throw our `FileNotFoundException` custom exception:

```
File "values.data" not found
```

If the file exists and contains

```
3
10
```

```
20
30
40
50
```

the `load_vector` function will throw the standard `std::out_of_range` exception.

Listing 22.10 (filereadmultiexcept2.cpp) uses two `catch` blocks to print different messages depending on the exact type of exception object thrown by a `load_vector`.

Listing 22.10: filereadmultiexcept2.cpp

```
#include <iostream>
#include <fstream>
#include <vector>
#include <string>

// Exception object to thrown when a client attempts to
// open a text file via a name that does correspond to a
// file in the current working directory.
class FileNotFoundException : public std::exception {
    std::string message;  // Identifies the exception and filename
  public:
    // Constructor establishes the exception object's message
    FileNotFoundException(const std::string& fname):
        message("File \"" + fname + "\" not found") {}

    // Reveal message to clients
    const char *what() const {
        return message.c_str();
    }
};

// Creates and returns a vector of integers from data stored
// in a text file.
// filename: the name of the text file containing the data
// Returns a vector containing the data in the file, if possible
std::vector<int> load_vector(const std::string& filename) {
    std::ifstream fin(filename);         // Open the text file for reading
    if (fin.good()) {               // Did the file open successfully?
        int n;
        fin >> n;                   // Size of data set
        std::vector<int> result(n);      // Allocate space for the vector
        int value, i = 0;
        while (fin >> value)
            result.at(i++) = value;   // Append it to the vector
        return result;              // Return the populated vector
    }
    else   // Could not open the text file
        throw FileNotFoundException(filename);

}

int main() {
    try {
        std::vector<int> numbers = load_vector("values.data");
```

```
        for (int value : numbers)
            std::cout << value << ' ';
        std::cout << '\n';
    }
    catch (std::out_of_range& e) {
        std::cout << "Error: vector bounds exceeded\n";
        std::cout << e.what() << '\n';
    }
    catch (FileNotFoundException& e) {
        std::cout << "Error: cannot open file\n";
        std::cout << e.what() << '\n';
    }
}
```

22.5 Exception Mechanics

An exception handler within a `catch` block may take a few steps to handle the exception and then *re-throw* the exception or throw a completely different exception.

Listing 22.11: `rethrow.cpp`

```
#include <iostream>
#include <fstream>
#include <vector>

void filter(std::vector<int>& v, int i) {
    v.at(i)++;
}

void compute(std::vector<int>& a) {
    for (int i = 0; i < 6; i++) {
        try {
            filter(a, i);
        }
        catch (std::exception& ex) {
            std::cout << "*********************************\n";
            std::cout << "* For loop terminated prematurely\n";
            std::cout << "* when i = " << i << '\n';
            std::cout << "*********************************\n";
            throw ex;  // Rethrow the same exception
        }
    }
}

int main() {
       std::vector<int> list { 10, 20, 30, 40, 50 };
       try {
           compute(list);
       }
       catch (std::exception& e) {
           std::cout << "Caught an exception: " << e.what() << '\n';
       }
       std::cout << "Program finished\n";
```

```
}
```

Listing 22.11 (rethrow.cpp) produces the following output:

```
********************************
* For loop terminated prematurely
* when i = 5
********************************
Caught an exception: invalid std::vector<T> subscript
Program finished
```

Notice that in Listing 22.11 (rethrow.cpp) the `compute` function does not create a new exception object; it simply re-throws the same exception object that it caught. This concept of intermediate exception handlers is important since a function invocation has complete knowledge of it own local context such as its local variables. This means that an intermediate function (or method) in the call chain has access to information that may be unavailable to all other functions (or methods) within that call chain. The `catch` block in the `compute` function of Listing 22.11 (rethrow.cpp) involves only local variables. The `catch` block within the `main` function cannot know the subscript that caused `compute`'s failure since `i` is local to `compute`. Some general exception guidelines include:

- Handle exceptions as close (up the call chain) as possible to the code that throws the exception.
- Use local information where applicable to correct as much of the problem as possible within that local context:
  - Local information which may be important for recovering from the exceptional situation is available; this local information is unavailable to functions or methods up the call chain.
  - The lower-level functions/methods that are prone to exceptions may be called by many different higher-level functions/methods. If the lower-level functions/methods are responsible for their own exception handling, then the higher-level functions/methods may be able to dispense with exception handling altogether making the lower-level methods more robust units of reuse.
  - The lower-level functions/methods may handle many different kinds of exceptions, do any local clean up they can based on the type of exception thrown, and then throw a different type of exception up to the higher-level methods. The higher-level method may just need to know there was a problem without needing to know exactly what the problem was. For example, in Listing 22.11 (rethrow.cpp), the `compute` function could throw a custom `ComputeException` object whenever `filter` throws its `std::out_of_range` instead of just passing up the call chain the `std::out_of_range` object it receives.

  If total recovery is not possible or desirable at the lower levels of the call chain, re-throw the exception so an exception handler up the call chain can take steps to recover from the problem in a manner that makes sense in its context.

When a section of code throws an exception, the `catch` blocks within the nearest exception handler are checked in the order they appear in the C++ source code. The first `catch` block that matches the type of the exception thrown is the one executed. For example, Listing 22.12 (multipleexceptions.cpp) expects two types of exceptions, and it uses the catch-all exception in case any unexpected exceptions arise.

Listing 22.12: `multipleexceptions.cpp`

```
#include <iostream>
#include <fstream>
```

```
#include <vector>
#include <string>

// Get an integer from the user. Note that std::stoi can
// throw a std::invalid_argument exception if the user's
// input is not an integer, and it can throw std::out_of_range
// if the string is a valid integer but outside the range of
// a C++ int on this platform.
int get_int() {
    std::string input;
    std::cin >> input;
    int result = stoi(input);
    return result;
}

int main() {
    std::vector<int> a { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
    try {  // What possibly could go wrong?
        int n = a.size();
        std::cout << "Enter index in the range 0..." << n - 1 << ": ";
        // Will user enter a valid integer value?
        int i = get_int();
        std::cout << "Index entered: " << i << '\n';
        // Will the user's index be in the range of valid indices?
        std::cout << "a[" << i << "] = " << a.at(i) << '\n';
    }
    catch (std::out_of_range&)  {
        std::cout << "Index provided is out of range\n";
    }
    catch (std::invalid_argument&) {
        std::cout << "Index provided is not an integer\n";
    }
    catch (...) {  // What have we forgotten?
        std::cout << "Unknown error\n";
    }
}
```

When an exception is caught, its type is first compared to `std::out_of_range`. If it matches, the `std::out_of_range` `catch` block is executed. If not, the exception's type is then compared to `invalid_argument`. The `invalid_argument` `catch` block is executed in the case of a match. If neither of the two types match, the catch-all code is executed.

When inheritance is involved, the situation becomes more interesting. The comparison performed in each `catch` block is a test for *assignment compatibility*. A reference or a pointer to an instance of a derived class can be assigned to a variable declared to be a reference a pointer to a base class. This up-casting operation is always legal because of the *is a* relationship between a derived class and its base class. This means an exception object of type `std::out_of_range` is assignment compatible with both `std::out_of_range` variables and `std::exception` variables. Since the `catch` blocks are checked in the order they appear in the source code, the following code would have a problem:

```
try {
    // Execute some code that can throw an
    // out_of_range exception
    . . .
```

```
}
catch (std::exception& e) {
    // Do something here
}
catch (std::out_of_range& e) {  // Error, this can never be reached
    // Do something else here
}
```

The `std::out_of_range` catch block will never execute since a `std::out_of_range` object is an `exception` instance also. For this reason it is illegal for a more specific type to follow a more general type in a sequence of `catch` blocks. The above code fragment will not compile in any context. Thus, in a sequence of `catch` blocks, list the more specific exceptions first followed by the more general exceptions.

All C++ standard exception classes have `std::exception` as their direct or indirect base class. C++ does not require exception objects to be derived from `exception`; in fact, primitive types such as `int` and `char` can be thrown and caught as well. Because of this, the "catch all" exception clause is

```
catch (...) {
    // Catches any exceptions not caught by more specific
    // catch blocks in this try/catch block
}
```

The ellipses (`...`) stand for an exception of any type.

Any function or method can throw an exception object. Given an exception of type `E`, the statement to do so is:

```
throw E();
```

22.6 Using Exceptions

Exceptions should be reserved for uncommon errors. For example, the following code adds up all the elements in an integer vector named `vec`:

```
int sum = 0, n = vec.size();
for (int i = 0; i < n; i++) {
    sum += vec[i];
}
std::cout << "Sum = " << sum << '\n';
```

This loop is fairly typical. Another approach uses exceptions:

```
sum = 0;
int i = 0;
try {
    while (true)
        sum += vec.at(i++);
}
catch (std::out_of_range&) {}
std::cout << "Sum = " << sum << '\n';
```

Both approaches compute the same result. The second approach terminates the loop when the array access is out of bounds. It interrupts the statement

```
sum += vec.at(i++);
```

in midstream, so it does not incorrectly modify `sum`'s value. The second approach, however, *always* throws and catches an exception. The exception definitely is **not** an uncommon occurrence.

You should not use exceptions to dictate normal logical flow. While very useful for its intended purpose, the exception mechanism adds some overhead to program execution, especially when an exception is thrown. This overhead is reasonable when exceptions are rare but not when exceptions are part of the program's normal execution.

Sometimes it is not clear when an exception is appropriate. Consider a function that returns the position of an element within a vector. The straightforward approach that does not use exceptions could be written:

```
int find(const std::vector<int>& a, int elem) {
    int n = a.size();
    for (int i = 0; i < n; i++)
        if (list[i] == elem)
            return i;  // Found it at position i
    return -1;  // Element not present
}
```

Here a return value of -1 indicates that the element sought is not present in the array. Should an exception be thrown if the element is not present? The following code illustrates:

```
int find(const std::vector<int>& a, int elem) {
    int n = a.size();
    for (int i = 0; i < n; i++)
        if (list[i] == elem)
            return i;  // Found it at position i
    // Element not there; throw an exception
    throw ElementNotPresentException();
}
```

In the first approach, an unwary programmer may not check the result and blindly use -1 as a valid position. The exception code would not allow this to happen. However, the first approach is useful for determining *if* an element is present in the vector. If `find(x)` returns -1, then `x` is not in the vector; otherwise, it is in the vector. If the exception approach is used, a client programmer cannot determine if an element is present without the risk of throwing an exception. Since exceptions should be rare, the second approach appears to be less than ideal. In sum,

- The first approach is more useful, but clients need to remember to properly check the result.
- The second approach provides an exception safety net, but an exception always will be thrown when searching for missing elements.

Which approach is ultimately better? The first version uses a common programming idiom and is the better approach for most programming situations. The exception version is a poorer choice since it is not uncommon to attempt to look for an element missing from a vector; exceptions should be reserved for uncommon error situations.

Appendices

©2018 Richard L. Halterman

Appendix A

Using Visual Studio 2015 to Develop C++ Programs

This appendix describes the task of C++ software development under Visual Studio:

- To begin creating a C++ program, you must first launch Visual Studio 2015 from the Windows start screen or other relevant shortcut. Figure A.1 shows the appropriate application tile to activate.

  You soon should see a splash screen similar to the one shown in Figure A.2.

  If you have never before used the Visual Studio application, you must wait a few moments while it configures the development environment for you. At this point you will indicate that Visual C++ is your preferred development language. You also may select a color scheme. The figures shown here reflect the *blue* color scheme.

  Figure A.3 shows what Visual Studio looks like when it is fully loaded and ready to use.

- After Visual Studio has started, you begin the process of developing a C++ program by creating a new project. As Figure A.4 shows, you can create a new project by following the menu sequence: *File→New→Project*

- In the dialog that results, shown on the right of Figure A.4, you should choose the project type to be *Visual C++* in the left pane, and use the *Win32 Console Application* option in the center pane. In the name field near the bottom of the dialog, enter a name for the project; we will use the name *simple*. You may change the location to a different folder if you like, or even a different drive (such as a USB pen drive). In this example, we chose to not change the default location provided by Visual Studio.

- When you select *OK* on the project creation dialog, a *Win32 Application Wizard* as shown on the left of Figure A.5 appears. At this point, the instructions in the dialog say "Click **Finish** from any window to accept the current settings." **Do not select Finish**; instead, select *Next* to continue. We have one more key step to complete so our project is set up correctly.

Figure A.1 Visual Studio application tile

Figure A.2 Visual Studio Splashscreen

Figure A.3 Visual Studio Ready for Use

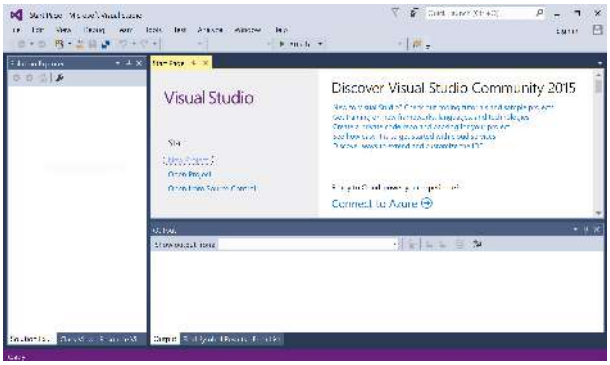
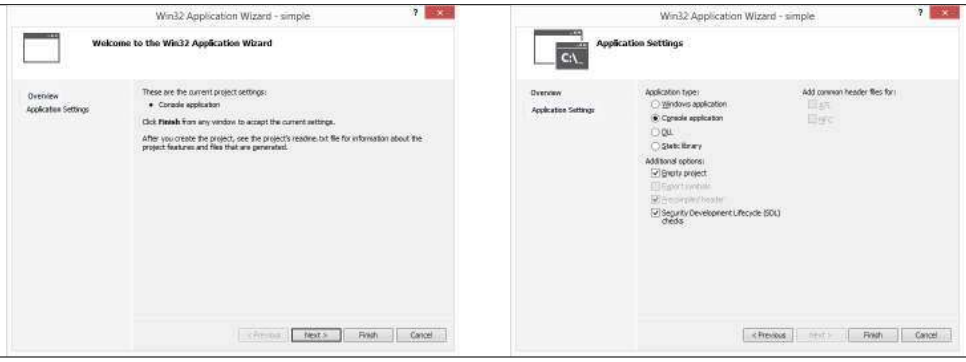


Figure A.4 Creating a New Project

Figure A.5 Configuring the New Project



- In the subsequent *Applications Settings* dialog (see the right image in Figure A.5), select the *Empty project* checkbox. The dialog should look like the right image in Figure A.5 before you proceed. Choose *Finish* when you are ready to continue.

- At this point, the *Solution Explorer* panel shows the structure of the newly created, albeit empty, project. The left image in Figure A.6 shows the newly populated *Solution Explorer pane*.

- Right click on the *simple* element in the *Solution Explorer*. As shown on the right of in Figure A.6, select *Add* and *New Item* from the resulting pop-up menu.

- In the *Add New Item* dialog box, shown on the left in Figure A.7, select *C++File (.cpp)* and enter a name for the C++ file in the text field at the bottom. You need not add `.cpp` to the end of the name, as Visual Studio will do that for you. The file here will be named simple.cpp. Press *Add* when done.

- As shown on the right in Figure A.7, the *Solution Explorer* pane now shows the file `simple.cpp`, and the large editor pane is ready for you to type in the source code for the program. The new source file is initially empty.

- In the editor pane with the *simple.cpp* tab at the top, type in the source code for our simple C++ program. Figure A.8 shows the completed code.

- You may save the source file at this point by selecting *Save* from the *File* menu or by pressing pressing at the same time the Ctrl S keys. If you do not save your program, Visual Studio will prompt you to do so before it builds and runs your program.

  To run the program, select *Debug→Start Without Debugging*, as shown on the left of Figure A.9. (You also may use the Ctrl F5 key sequence to build and run the program.) Visual Studio will attempt to build the executable program. It will prompt you to save your file if you have not saved it

Figure A.6 Adding a C++ Source File

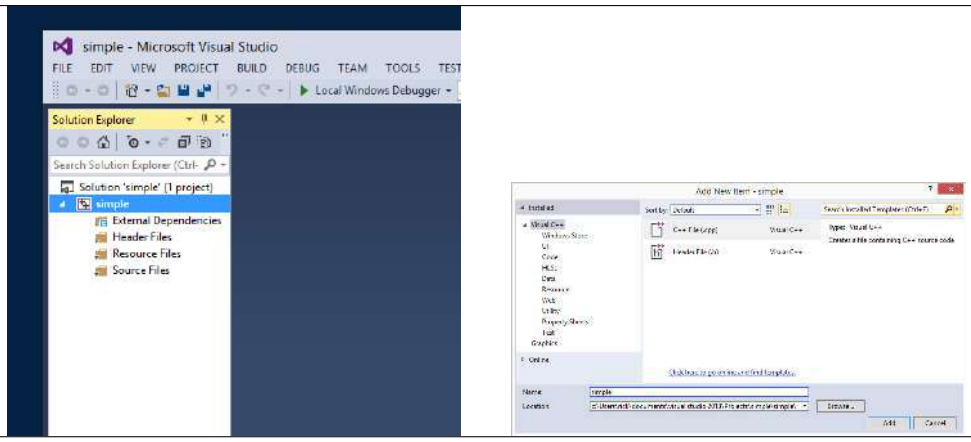
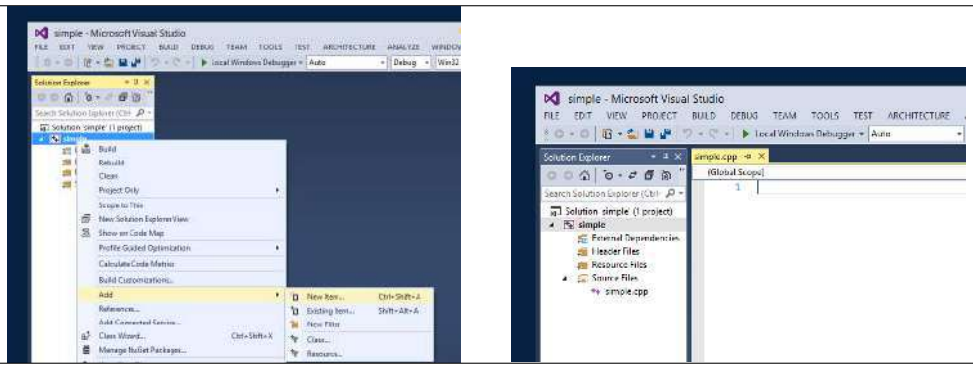


Figure A.7 Ready to Edit the C++ Source File



or have made changes since the last time you saved it. The progress of the build process is displayed is in the *Output* panel in the bottom window. One of the lines has

```
1>Compiling. . .
```

and another later has

```
1>Linking. . .
```

The last few lines are very important:

```
1>Build succeeded.
1>
1>Time Elapsed 00:00:02.98
========= Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
```

These lines indicate the build was successful.

- A console window appears with the output of your program. The right image in Figure A.9 shows this text window. You can press any key on the keyboard to close the window. If the console window does not appear, you have typographical errors in your program; return to the editor, fix the errors, and try to run the program again.

Figure A.8 Editor Pane

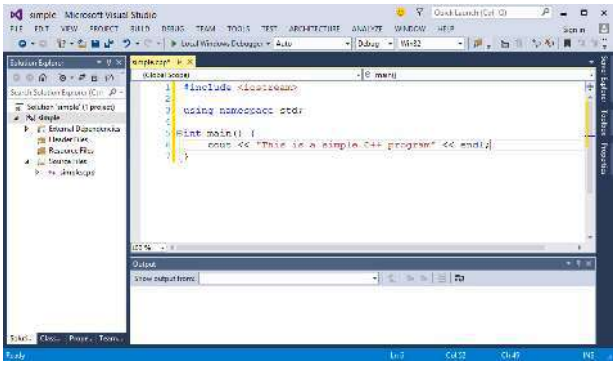
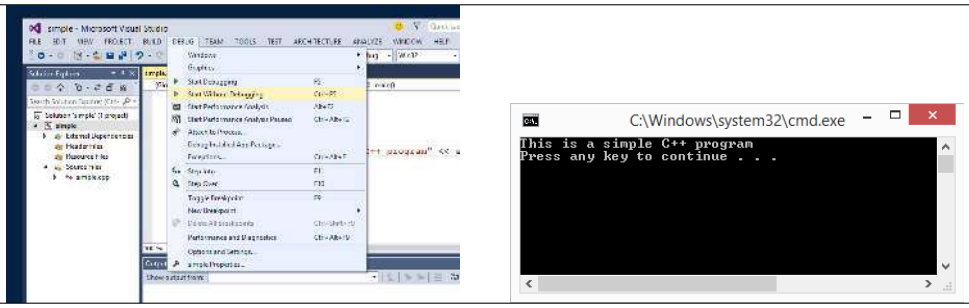


Figure A.9 Building and Running the Program



- When you are finished with Visual Studio, select the *File→Exit* menu items as shown in Figure A.10.

These are the steps for writing a basic C++ program in Visual Studio 2015. While the steps initially may seem complex and tedious, they will become natural after you have written a few C++ programs.

When the program was run, the Visual Studio environment created a console window in which to run the program.

```
This is a simple C++ program!
Press any key to continue . . .
```

The first line of the output was printed by the program. The second line prompting the user to press any key to continue is added by the Visual C++ run-time environment so the console window stays visible long enough for the user to see the output. If you run the program from a the standard Windows command shell (CMD.EXE, usually found in the *Start* menu under *Accessories* and named *Command Prompt*), only the program's output will appear and the "Press any key to continue . . ." message will not appear.

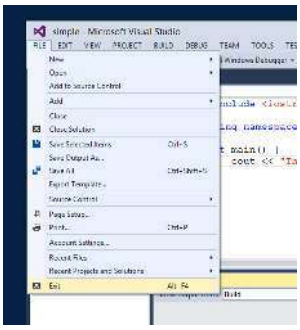
The following summarizes the steps you should follow when writing a C++ program using the Visual Studio IDE:

1. In the main menu:

   File→New→Project (or Ctrl Shift N)

2. Select "Win32 Console Project" and click "Next" to set the "Empty Project" option

Figure A.10 Exiting Visual Studio



3. In the Solution Explorer pane right click on "Source Files" and select
   Add→New Item...
4. Select C++ File (.cpp) and enter the name of your file
5. Type in the source code for your program in the editor panel
6. Save your file:
   File→Save *filename* (or Ctrl S)
7. In the main menu:
   Debug→Start Without Debugging (or Ctrl F5)

It is possible to develop C++ programs without using Visual Studio's integrated development environment. Visual Studio comes with some additional tools for command-line development. Appendix B describes how you can edit the C++ source code with a standalone text editor and compile and run the program from the Windows command prompt (CMD.EXE). Command-line development under Linux and macOS is covered in B. Some programmers prefer the freedom of using their favorite editor, a standalone compiler, and scripts to automate the build process.

Appendix B

Command Line Development

An integrated development environment (IDE) like Visual Studio combines all the tools a developer needs into one comprehensive application. While this approach works well for most programmers, some developers prefer a less centralized approach, using instead command line tools to manage the development process. Command line tools are focused and fast, and each tool itself is fairly simple compared to a full-featured IDE. A developers may prefer a different editor to one provided by the IDE. Developers can build scripts to around command line tools to automate the building and testing process.

There are a few commands that are essential for working within the command-line environment. These center around manipulating and using files and folders. Our discussion assumes the Windows CMD.EXE shell, and we will note differences with macOS and Linux as needed. Linux and macOS are both *Unix-like operating systems*.

When you run CMD.EXE in Windows, the Terminal application on a Mac, or a bash terminal in Linux, the operating system presents a text-based console window. You type commands into this window and receive text feedback. Most of the commands are programs that you launch by typing their name. The following lists common commands useful for using the command-line;

- `dir`

  When the command line interface starts it locates you in a particular folder, or directory, in the file system managed by the operating system. Your location is known as the *current working directory*. The command `dir` prints a list of the files in your current working directory. On Unix-like machine the `ls -l` command performs similar work. The command `dir /w` displays a wide listing of the files with fewer details; the `ls` command is the equivalent on Unix-like systems.

- `mkdir` *name*

  Creates a new subdirectory (subfolder) named *name* in the current directory. Subdirectories allow you to better organize your files.

- `cd` *name*

  Changes you current directory to be that specified by *name*. Some examples of its use include

  - `cd Code`

    Changes the current working directory to be the subdirectory named Code. Here the name is relative to the current directory. It is an error to provide a name that does not correspond to a subdirectory of the current directory.

- `cd ..`

  Changes the current working directory to its parent directory. Said another way, the current working directory is itself a subdirectory of some other directory. This command makes this other (parent) directory the current working directory. The `..` means "parent directory" or "parent folder."

- `cd C:\Users\rick\Documents\Code`

  Changes the current working directory to the completely specified one. The name here is known as a *full-path name*. The name is an absolute location; it is not relative to the current working directory. Unix-like systems do not have a drive letter (C:), and the backslashes (\) would be forward slashes (/).

- `cd .`

  Changes the current working directory to itself, which effectively does nothing! This command does no useful work. The dot (`.`) in this context means "current working directory."

• `del` *name*

  Deletes the file named *name*. Use this command carefully. Unix-like systems use the `rm` command in its place.

• `ren` *name1 name2*

  Renames the file *name1* to *name2*. The `move` command does the same thing, and Unix-like systems use the `mv` command to rename files.

In preparation for our programming, we can create a new directory for our code and make that new directory our current working directory with the commands:

```
mkdir Code
cd Code
```

We can edit our source code with a simple text editor such as Notepad or Wordpad for Windows, TextEdit on the Mac, or gEdit on Linux systems. More powerful programming editors such as Notepad++ (`http://notepad-plus.sourceforge.net`) (Windows) and Vim (`http://www.vim.org`) (Windows, Mac, and Linux) make editing source code more convenient.

B.0.1 Visual Studio Command Line Tools

Visual Studio provides a number of command line tools as an alternative to its integrated development environment.

To use the command line tools, select the *Visual Studio Command Prompt* from the *Start* menu. A command console appears similar to the console that the IDE provides when executing a program (see Section 2.2). Instead of selecting menu items and interacting with dialog boxes, you type commands into the console window to build and run your program. To edit your program you can use the editor from the IDE, or you can use a standalone editor such as Notepad++ (`http://www.notepad-plus.sourceforge.net`) or Vim (`http://www.vim.org`). Both Notepad++ and Vim provide color syntax highlighting and folding like the built-in Visual Studio editor.

In order to use the command prompt version of Visual Studio, it is convenient to first create a folder in which you will place all your development files; for example, you could create a folder named Code under your Documents folder. Suppose the full path to this folder is

```
C:\Users\rick\Documents\Code
```

Once your development folder is created, you can launch the Visual Studio Command Prompt.

The first command you should issue in the console window is

```
C:
```

to ensure you are working on the correct drive. Next issue

```
cd \Users\rick\Documents\Code
```

The `cd` command means "change directory." This command sets the console's working directory to the development folder you previously created. If you are working in another folder, you can adjust the path given in the `cd` command to your work area.

You are ready to edit your code. If your environment is set up so that your editor program is in the system path, you can type

```
notepad++ simple.cpp
```

If you are using Vim, type

```
gvim simple.cpp
```

If the file simple.cpp already exists, you can type

```
devenv simple.cpp
```

to launch the Visual Studio editor.

Within the editor type in your program and then save it to a file.

When you have finished creating the source file, we will need to build the executable program. Switch back to the *Visual Studio Command Prompt* window. and issue the command

```
cl /W3 /EHsc simple.cpp
```

The `cl` ("cee elle", not "cee one") command, which stands for *compile* and *link*, preprocesses, compiles, and links the program. The `/W3` switch uses the Level 3 warning level. Level 3 warnings provide more thorough analysis of your source code and can detect more programming mistakes. If you omit the `/W3` switch, by default the compiler operates at warning level 1. Warning level 1 is insufficient for catching many mistakes made by both beginning and experienced C++ programmers. By default the IDE compiler uses warning level 3, and the warning level can be changed by adjusting the project's properties. On the command line, specifying `/W4` (the highest warning level) is just as easy as specifying `/W3`. It is better to use `/W4` so the compiler can do a more thorough check for potential programmer mistakes. The `/EHsc` is required for all the C++ programs we write.

If want to reduce the level of detail in the output produced by the compiler and linker, you can add the additional switch `/nologo`:

```
cl /W4 /EHsc /nologo simple.cpp
```

Notice that we used the enhanced warning level (`/W4`) here. To see the complete list of compiler options, many more than you probably will ever need, type

```
cl /?
```

In C++ programs consisting of a single source file, the compiler produces an executable file with the same base name as the source file (simple, in our example), with an exe extension.

Once the program has been compiled successfully, you can run the program by entering the name of the executable produced. In our example, we would issue the command

```
simple
```

The program runs in the same window; no new console window is produced. When the program is finished executing, no "Press any key to continue" message is printed; you get simply a command prompt ready for you to enter your next command.

If you modify your source code in your editor, you must remember to recompile your code before you run it again. In the IDE, if you modify your code, the environment will suggest that you rebuild the project before running the program.

If the program consists of multiple source files, list all the required files on the command line. The exe file will be named after the first file listed; for example, the command

```
cl /W4 /EHsc /nologo app.cpp list.cpp util.cpp
```

compiles the three C++ sources files and links the compiled code together to produce app.exe.

In the Visual Studio command prompt program you can enter the Visual Studio IDE at any time with the command

```
devenv
```

(`devenv` stands for "development environment.") If you started Visual Studio in the IDE, not the command prompt, you can open a command prompt console window at any time by selecting the *Command Prompt* item under the *Tools* menu.

B.0.2 Developing C++ Programs with the GNU Tools

The tools provided in the GNU Compiler Collection (`http://gcc.gnu.org`), or GCC for short, include one of the most up-to-date standards compliant C++ compilers available. The GNU tools are free and are available for the major computing platforms:

- **Microsoft Windows XP–Windows 10**

  Windows-based GCC tools can be found at the MinGW website `http://www.mingw.org`. The Nuwen site (`www.nuwen.org`) offers an easy to install custom distribution of MinGW with a number of additional useful C++ libraries.

- **Apple macOS**

  The Homebrew (`http://brew.sh`) and MacPorts (`http://www.macports.org/`) projects bring the GCC toolset to Mac developers. Mac users also may use the clang++ compiler that comes with Xcode.

- **Linux**

Linux-based GCC tools are readily available for all the major Linux distributions. To get them, simply install the g++ development packages.

The name of the GNU C++ compiler is g++. It is a command-line tool, which means it is launched from a command shell—CMD.EXE in Windows or bash in the Mac and Linux environments.

To see how the GNU C++ buld system works, we will consider Listing 2.1 (simple.cpp) from Chapter 2. Use an editor to create a text file containing the code for Listing 2.1 (simple.cpp), and save the file with the name simple.cpp. In Windows, if *notepad++* is installed and set up properly, in the command shell you can type

```
notepad++ simple.cpp
```

This command brings up a separate window with an editor. We can type the source code for Listing 2.1 (simple.cpp) and save the file.

To create the executable program with GNU C++, back in our command-line window we issue the command

```
g++ -Wall -std=c++14 -o simple simple.cpp
```

This command has five parts:

- `g++ -Wall -std=c++11 -o simple simple.cpp`

  g++ invokes the GCC C++ build process. This one command runs the preprocessor, followed by the compiler, and finally the linker to produce the executable program. Even though the preprocessor and linker are also involved here, we generally simply say we are "compiling the program."

- `g++ -Wall -std=c++11 -o simple simple.cpp`

  -Wall stands for “warn all” and directs the compiler to be very strict when it is compiling the source code. It goes beyond checking that the program is well-formed C++ it also warns about code constructs that many programmers consider questionable because code of that nature often contains errors. The -Wall component can be interpreted to mean "warn about as many possible problems as possible." You should compile all C++ programs with this directive because it enables the compiler to catch many careless programmer mistakes that otherwise might go unnoticed. Use of the -Wall directive is desirable for all C++ programmers from novice to expert.

The C++ standards committee is responsible for defining the C++ language. The committee refines the language specification over time. In addition to adding new features to the language, the committee also addresses shortcomings in existing language features. As a result of the committee’s work newer compilers can check more thoroughly the validity of C++ programs. Modern compilers do a better job of catching common programming errors.

The -Wall directive is optional because sometimes developers must work with older C++ source code. The code was written originally under more relaxed C++ language rules, but it may not be worthwhile to rewrite the code to bring it up to modern standards. If this older code has proven to be reliable, it may be compiled without the more stringent checks. All new code, however, should take advantage of the newer, better checking capabilities of the most up-to-date C++ compiler; therefore, always use -Wall on the compiler command line for code that you write.

- `g++ -Wall -std=c++14 -o simple simple.cpp`

  The `-std=c++14` option directs the compiler to process the C++ source code according to the rules of the C++14 standard which the International Standards Organization (ISO) adopted in 2014. Since this standard is relatively new, if you omit this part of the command, `g++` assumes the programmer is writing the source code under the rules of the older, 1998 ISO C++ standard, usually referred to as C++98.

  The code in this book is based on the C++11 standard.

- `g++ -Wall -std=c++14 -o simple simple.cpp`

  The `-o simple` part of the command specifies the name of the executable program. The desired name appears after the `-o` part, so our executable program will be named simple.exe on Microsoft Windows systems. On Mac and Linux platforms the name of the executable file produced will be just simple (no exe name extension). If the C++ source code code contains no errors, the compiler will produce the executable file named simple.exe (or simple). On a Windows Machine we can run the compiled program from the command line by typing the command

  ```
  simple
  ```

  (We also can use the full name, simple.exe.) On a Unix-ike system (macOS or Linux), we instead would type

  ```
  ./simple
  ```

  to execute the program.

- `g++ -Wall -std=c++14 -o simple simple.cpp`

  `simple.cpp` specifies the name of the C++ source file we wish to compile. Here the file simple.cpp must reside in our current working directory for the compiler to process it.

The general form of the command to compile a C++ source file, therefore, is

`g++ -Wall -std=c++14 -o` *executable source*

where *source* is the name of the file containing the C++ source code, and *executable* is the name of the executable program the build process generates.

When you are finished with the executable file simple.exe, you can delete it with the following command:

```
del simple.exe
```

We can reconstruct the executable file whenever we like by simply recompiling the source file. Be careful not to delete accidentally the source file, or you have no choice but to type it in again.

Index

abstract class, 524
actual parameter, 208
adapter design pattern, 522
alias, 326
aliasing, pointer, 546
and, bitwise, 72
and, logical, 97
argument, 182
array, 311
array slice, 322
assignment, 19
associative container, 687
associativity, operator, 44
attributes, 409
auxiliary methods, 428

bias in pseudorandom number generation, 401
binary search, 359, 688
bitwise and, 72
bitwise exclusive or, 73
bitwise left shift, 74
bitwise negation, 73
bitwise operators, 72
bitwise or, 73
bitwise right shift, 74
bitwise shift operators, 74
bitwise xor, 73
block, 92
bottom-checking loop, 165
break statement, 140
buffer overrun, 334
buffering, 395

C string, 333
call by reference, 274
call by value, 215
caller, 182
carriage return, 17
case label in a switch statement, 162
catch block, 720
class derivation, 496
class specialization, 496
class templates, 618
client, 384
closure, 656
coincidental correctness, 491
command line, 9
command-line arguments, 336
compiler, 2
concrete class, 524
constructor, 422
constructor delegation, 572
constructor initialization list, 423
copy assignment, 583
copy constructor, 570
copy-and-swap, 572
CR, 17
current working directory, 747

data members, 409
default argument, 252
default constructor, 424
default parameter, 252
delegation, method, 502
delimit, 11
derivation, class, 496
derivative, 658
destructor, 559
differentiation, 658
dot operator (.), 411
doubly linked list, 634
dynamic binding, 509

early binding, 508
encapsulation, 430
enumerated type, 31
enumeration classes, 31
enumeration type, 31
erase-remove idiom, 678
escape sequence, 10
evaluation, 38
exception, 719
exception handling, 719
exclusive or, bitwise, 73

©2018 Richard L. Halterman

explicit template instantiation, 611, 616
expressions, 37

Fibonacci sequence, 260
fields, 409
for statment, 167
foreach, 298
fragmentation, memory, 544
`friend`, 466
full-path name, 748
function call, 182
function call operator, 636
function declaration, 231
function definition, 203
function invocation, 182
function invocation operator, 636
function objects, 656
function overloading, 251
function pointers, 278

garbage, 597
garbage collection, 590
generating function, 669
generic function, 610
goto statement, 141

hash function, 704
hash table, 702
heap fragmentation, 544
helper methods, 428
hiding variables, 420
higher-order function, 278
homogeneous, 291

if statement, 88
inheritance, 496
initialization, 20, 570
initializer list, 681
instance, 409
instance variables, 409
is-a relationship, 497
iteration, 123
iterators, 640

key, 687

lambda calculus, 655
lambda expressions, 655
late binding, 509
left shift, 74
LF, 17
line feed, 17
linear ordering, 291
linear search, 359
linker, 5
Liskov Substitution Principle, 497
logical and, 97
logical negation, 97
logical not, 97
logical or, 97
loop, 123
loop unrolling, 372
lvalue, 578

mask, 75
matrix, 307
member data, 409
member function, 419
member functions, 299, 417
members, 409
memoization, 713
memory fragmentation, 544
memory leak, 326, 441
Mersenne Twister pseudorandom number generator, 403
method delegation, 502
methods, 299, 417
move assignment, 583
move constructor, 583
multiple inheritance, 499

negation, bitwise, 73
negation, logical, 97
nested loop, 133
newline, 10
non-type parameters, template, 617
non-type template parameters, 617
not, logical, 97
numerical differentiation, 659

object slicing, 505
operator associativity, 44
operator precedence, 44
or, logical, 97
overflow, 64
overloaded function, 251
override, 502

parameter, 182
parameter, actual, 208
pass by reference, 274
pass by value, 215
period of a pseudorandom number generator, 401

pointer, 267
pointer aliasing, 546
polymorphism, 511
precedence, operator, 44
predicate, 667
pretty printer, 50
priority queue, 635
private inheritance, 498
private members, 418
profiler, 5
protected member, 523
pure virtual methods, 524

random access itertator, 643
range-based for, 298
ranges, 644
rational number, 425
re-throw, 733
re-throwing an exception, 734
real number, 64
red-black tree, 688, 702
reference counting, 597, 598
reference semantics, 547
reference variable, 271
relational operators, 86
return statement, 207
return value optimization, 306, 588
right shift, 74
Rule of Five, 588
Rule of Three, 577
rvalue reference, 580

scoped enumerations, 31
sequence, 290
shift, bitwise, 74
short-circuit evaluation, 99
slice, array, 322
smart pointer, 590
source code formatting, 47
specialization, class, 496
stack overflow, 544
standard namespace, 9
standard template library, 633
static binding, 508
steam manipulator, 129
STL, 633
string concatenation, 387
string objects, 384
struct, 465
structured binding, 695
subclassing, 496
switch statement, 159
symbolic differentiation, 659
symmetric, 468

template instantiation, 611
template type parameter, 610
template value parameters, 617
templates, 609
temporary, 579
this, 442
throws, 720
transitive, 468
try block, 720
try/catch block, 720
tuple unpacking, 694
two's complement, 74
two-dimensional array, 330
two-dimensional vector, 307
type alias, 308
type parameter, 610
`typedef`, 309

undefined behavior, 297
underflow, 64
`unordered_map`, 702
unpacking tuples, 694
unscoped enumeration, 31
unsigned, 24
`using` type alias, 308

value parameters, template, 617
value semantics, 547
variable aliasing with references, 271
variable hiding, 420
vector, 291
vector initializer list, 292
virtual memory, 544
vtable, 509

while statement, 124

xor, bitwise, 73