2 Program and Class Construction Extending the Foundation

Classes, Iterators, and Patterns 6

Nature uses only the longest threads to weave her pattern, so each small piece of the fabric reveals the organization of the entire tapestry.
Richard Feynman
in Grady Booch, Object Solutions

The control structures we studied in Chaps. 4 and 5 permit program statements to be executed selectively or repeatedly according to values entered by the user or calculated by the program. In Section 5.4 we saw how using classes like `Dice` and `Date` extends the domain of problems we can solve by programming. In this chapter we'll extend the idea of repetition by processing various data in several applications. A common pattern emerges from all these applications: the pattern of iterating over a sequence of data. Examples of such iteration include processing each word in one of Shakespeare's plays, processing the elements of a set, and processing movement of a simulated molecule. We'll also explore classes in more depth, studying class implementation as well as class use. We'll explore design guidelines for constructing classes and programs.

In particular, we'll cover classes used to read information from files stored on a disk rather than entered from the keyboard. We'll use a standard stream class that behaves in the same way `cin` behaves but that allows data to flow from text files. We'll develop a new class for reading words from files and build on the pattern of iteration developed for the class to develop new classes. Writing programs and functions that use these classes requires a new kind of parameter, called a *reference parameter*, which we'll discuss in some detail.

6.1 Classes: From Use to Implementation

In Section 5.4 we studied code examples using classes `Dice` and `Date`. These classes complement the standard C++ class `string` and will be part of the toolkit of classes we'll extend and use throughout the book. In this section we'll examine classes more closely. This will help you get comfortable with the syntax of **class implementation** in addition to the syntax of using classes that you've already seen. You'll learn how to modify classes and how to implement your own.

6.1.1 Class Documentation: The Interface (.h File)

C++ classes encapsulate **state** and **behavior.** The behavior of a class is what the class does. Behavior is often described with verbs: cats eat, sleep, and play; dice are rolled. The state of a class depends on physical properties. For example, dice have a fixed number of sides.

Class behavior is defined by **public member functions**; these are the class functions that client programs can call. Public member functions of the class `Dice` are the `Dice` constructor and the functions `NumRolls()`, `NumSides()`, and `Roll()`. The **class declaration** for `Dice` is shown below; the entire header file *dice.h* is found in Howto G as Program G.3 (the header file includes many comments that aren't shown below.)

```
class Dice
{
  public:
    Dice(int sides);          // constructor
    int Roll();               // return the random roll
    int NumSides() const;     // how many sides this die has
    int NumRolls() const;     // # times this die rolled

  private:
    int myRollCount;          // # times die rolled
    int mySides;              // # sides on die
};
```

The state of an object is usually specified by class **private data** like `myRollCount` and `mySides` for a `Dice` object. Private state data are often called **member data**, **data members**, **instance variables** or **data fields**. As we'll see, the term instance variable is used because each `Dice` instance (or object) has its own data members.

When an object is **defined,** by a call to a constructor, memory is allocated for the object, and the object's state is initialized. When a built-in variable is defined, the variable's state may be uninitialized. For programmer-defined types such as `Dice`, initialization takes place when the `Dice` variable is defined. As a programmer using the `Dice` class, you do not need to be aware of how a `Dice` object is initialized and constructed or what is in the private section of the `Dice` class. You do need to know some properties, such as when a `Dice` object is constructed it has been rolled zero times. As you begin to design your own classes, you'll need to develop an understanding of how the state of an object is reflected by its private data and how member functions use private data. Class state as defined by private data is not directly accessible by **client** programs. A client program is a program like *roll.cpp*, Program 5.11, that uses a class. We'll soon see how a class like `Dice` is implemented so that client programs that use `Dice` objects will work.

6.1.2 Comments in .h Files

The documentation for a class, in the form of comments in the **header file** in which the class is declared, furnishes information about the constructor's parameters and about all public member functions.

The names of header files traditionally end with a `.h` suffix. When the C++ standard was finalized, the `.h` suffix was no longer used so that what used to be called `<iostream.h>` became `<iostream>`. I continue to use the `.h` suffix for classes supplied with this book, but use the standard C++ header file names.

In this book the name of a header file almost always begins with the name of the class that is declared in the header file. The header file provides the compiler with the information it needs about the form of class objects. For programmers using the header file, the header file may serve as a manual on how to use a class or some other set of routines (as `<cmath>` or `<math.h>` describes math functions such as `sqrt`). Not all header files are useful as programmer documentation, but the compiler uses the header files to determine if functions and classes are used correctly in client programs. The compiler *must* know, for example, that the member functions `NumSides` and `Roll` are legal `Dice` member functions and that each returns an `int` value. By reading the header file you can see that two **private data variables**, `myRollCount` and `mySides`, define the state of a `Dice` object. As the designer and writer of client programs, you do *not* need to look at the private section of a class declaration. Since client programs can access a class only by calling public member functions, you should take the view that class behavior is described only by public member functions and not by private state.

A header file is an **interface** to a class or to a group of functions. The interface is a description of what the behavior of a class is, but not of how the behavior is implemented. You probably know how to use a stereo—at least how to turn one on and adjust the volume. From a user's point of view, the stereo's interface consists of the knobs, displays, and buttons on the front of the receiver, CD player, tuner, and so on. Users don't need to know how many watts per channel an amplifier delivers or whether the tuner uses phase-lock looping. You may know how to drive a car. From a driver's point of view, a car's interface is made up of the gas and brake pedals, the steering wheel, and the dashboard dials and gauges. To drive a car you don't need to know whether a car engine is fuel-injected or whether it has four or six cylinders.

The `dice.h` header file is an interface to client programs that use `Dice` objects. Just as you use a stereo without (necessarily) understanding fully how it works, and just as you use a calculator by pressing the $\sqrt{\ }$ button without understanding what algorithm is used to find a square root, a `Dice` object can be used in a client program without knowledge of its private state. As the buttons and displays provide a means of accessing a stereo's features, the public member functions of a class provide a means of accessing (and sometimes modifying) the private fields in the class. The displays on an amp, tuner, or receiver are like functions that show values; the buttons that change a radio station actually change the state of a tuner, just as some member functions can change the state of a class object.

When a stereo is well-designed, one component can be replaced without replacing all components. Similarly, several models of personal computer offer the user the ability to upgrade the main chip in the computer (the central processing unit, or CPU) without buying a completely new computer. In these cases the implementation can be replaced, provided that the interface stays the same. The user won't notice any difference in how the buttons and dials on the box are arranged or in how they are perceived to work. Replacing the implementation of a class may make a user's program execute more quickly, or use less space, or execute more carefully (by checking for precondition violations) but should not affect whether the program works as intended. Since client programs depend only on the interface of a class and not on the implementation, we say that classes provide a method of **information hiding**—the state of a class is hidden from client programs.

William H. (Bill) Gates *(b. 1955)*

Bill Gates is the richest person in the United States and CEO of Microsoft. He began his career as a programmer writing the first BASIC compiler for early microcomputers while a student at Harvard.

When asked whether studying computer science is the best way to prepare to be a programmer, Gates responded: *No, the best way to prepare is to write programs, and to study great programs that other people have written. In my case, I went to the garbage cans at the Computer Science Center and I fished out listings of their operating system. You've got to be willing to read other people's code, then write your own, then have other people review your code.* Gates is a visionary in seeing how computers will be used both in business and in the home. Microsoft publishes best-selling word processors, programming languages, and operating systems as well as interactive encyclopedias for children. Some people question Microsoft's business tactics, but in late 1994 and again in 1999 antitrust proceedings did little to deter Microsoft's progress. There is no questioning Gates's and Microsoft's influence on how computers are used.

Although Gates doesn't program anymore, he remembers the satisfaction that comes from programming.

> *When I compile something and it starts computing the right results, I really feel great. I'm not kidding, there is some emotion in all great things, and this is no exception.*

For more information see [Sla87].

6.1.3 Class Documentation: the Implementation or .cpp File

The header file `<cmath>` (or `<math.h>`) contains function prototypes, or headers, for functions like `sqrt` and `sin`. The bodies of the functions are not part of the header file. A function prototype provides information that programmers need to know to call the function. A prototype also provides information that enables the compiler to determine if a function is called correctly. The prototype is an interface, just as the class declaration in `"dice.h"` is an interface for users of the `Dice` class.

The bodies of the `Dice` member functions are not part of the header file *dice.h*, Program G.3. These function bodies provide an implementation for each member function and are put in a separate file. As a general rule (certainly one we will follow in this book), the name of the **implementation file** will begin with the same prefix as the header file but will end with a `.cpp` suffix, indicating that it consists of C++ code[1].

Like all functions we've studied, a member function has a return type, a name, and a parameter list. However, there must be some way to distinguish member functions from nonmember functions when the function is defined. The double colon `::` **scope resolution operator** specifies that a member function is part of a given class. The prototype `int Dice::NumSides()` indicates that `NumSides()` is a member function of the `Dice` class. Constructors have no return type. The prototype `Dice::Dice(int sides)` is the `Dice` class constructor. The prototype for the constructor of the `Balloon` class described in *gballoon.h*, Program 3.7, is `Balloon::Balloon()`, since no parameters are required. As an analogy, when I'm with my family, I'm known simply as `Owen`, but to the world at large I'm `Astrachan::Owen`. This helps identify which of many possible Owens I am; I belong to the `Astrachan` "class." The implementation of each `Dice` member function is in *dice.cpp*, Program 6.1. Each `Dice` member function is implemented with only a few lines of code. The variable `mySides`, whose value is returned by `Dice::NumSides`, is not a parameter and is not defined within the function. Similarly, the variable `myRollCount`, incremented within the function `Dice::Roll`, is neither a parameter nor a variable locally defined in `Dice::Roll`.

Syntax: member function prototype

`ClassName::ClassName` (parameters)
`//`constructor (cannot have return type)

type `ClassName::FunctionName` (parameters)
`//`nonconstructor member function

Program 6.1 dice.cpp

```
#include "dice.h"
#include "randgen.h"

// implementation of dice class
// written Jan 31, 1994, modified 5/10/94 to use RandGen class
// modified 3/31/99 to move RandGen class here from .h file

Dice::Dice(int sides)
// postcondition: all private fields initialized
{
    myRollCount = 0;
    mySides = sides;
}

int Dice::Roll()
```

[1]A suffix of `.cc` is used in the code provided for use in Unix/Linux environments.

```
// postcondition: number of rolls updated
//                random 'die' roll returned
{
    RandGen gen;    // random number generator

    myRollCount= myRollCount + 1;         // update # of times die rolled
    return gen.RandInt(1,mySides);        // in range [1..mySides]
}

int Dice::NumSides() const
// postcondition: return # of sides of die
{
    return mySides;
}

int Dice::NumRolls() const
// postcondition: return # of times die has been rolled
{
    return myRollCount;
}
```

dice.cpp

The variables `myRollCount` and `mySides` are private variables that make up the state of a `Dice` object. As shown in Figure 6.1, each object or **instance** of the `Dice` class has its own state variables. Each object may have a different number of sides or be rolled a different number of times, so different variables are needed for each object's state. The convention of using the prefix `my` with each private data field emphasizes that the data belongs to a particular object. The variable `cube` in *roll.cpp*, Program 5.11, has a `mySides` field with value six, whereas the `mySides` that is part of the `dodeca` variable has value 12. This is why `dodeca.NumSides()` returns 12 but `cube.NumSides()` returns 6; the member function `NumSides` returns the value of `mySides` associated with the object to which it is applied with `.`, the dot operator.

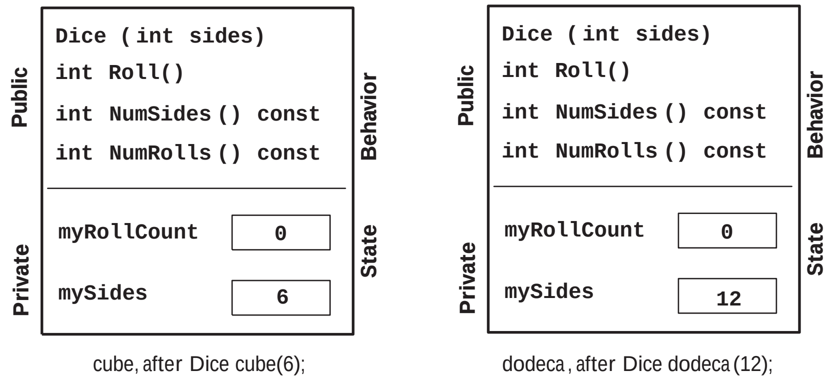


Figure 6.1 After `Dice` constructors have executed.

If the interface (header file) is well designed, you can change the implementation without changing or recompiling the client program.[2] Similarly, once the implementation is written and compiled, it does not need to be recompiled each time the client program changes. For large programs this can result in a significant savings in the overhead of designing and testing a program. With the advent of well-constructed class **libraries** that are available for a fee or for free, users can write programs much more easily and without the need for extensive changes when a new implementation is provided. This process of compiling different parts of a program separately is described in Section 3.5.

6.1.4 Member Function Implementation

We'll look briefly at the implementation of each member function of the `Dice` class as given in *dice.cpp*, Program 6.1.

The `Dice` Constructor. A class's constructor must initialize all private data (instance variables), so each data member should be given a value explicitly by the constructor. In the body of the constructor `Dice::Dice()` both instance variables `mySides` and `myRollCount` are initialized.

Program Tip 6.1: Assign a value to all instance variables in every class constructor. It's possible that you won't know what value to assign when an object is constructed, because the actual value will be determined by another member function. In this case, provide some known value, such as zero for an `int` instance variable. Known values will help as you debug your code.

The Member Functions `Dice::NumRolls` and `Dice::NumSides`. Class member functions are often divided into two categories:

- **Accessor functions** that access state, but do not alter the state.
- **Mutator functions** that alter the state.

The functions `Dice::NumRolls()` and `Dice::NumSides()` are **accessor functions** since they return information about a `Dice` object, but they do not change the object's state. Note that the implementation of these functions is a single line that returns the value of the appropriate instance variable. Accessor functions often have simple implementations like this. Nearly every programmer that designs classes adheres to the **design heuristic** of making all state data private. A heuristic is a rule of thumb or guideline. As a guideline, there may be exceptional situations in which the guideline is not followed, but in this book all class state will be private.

[2]You *will* need to relink the client program with the new implementation.

Program Tip 6.2: All state or instance variables in a class should be private. You can provide accessor functions for clients to get information about an object's state, but all access should be through public member functions; no instance variables should be public.

Accessor functions in C++ almost always have the keyword **const** following the parameter lists, both in the .h file and in the .cpp file. We discuss this use of `const` in detail in Howto D. Since accessor functions like `Dice::NumSides` do not change an object's state, the word `const` is used by the compiler to actually prohibit changes to state.

Program Tip 6.3: Make accessor functions `const`. You make a member function a `const` function by putting the keyword `const` after the parameter list.

The Member Function `Dice::Roll`. The function `Dice::Roll()` is a **mutator function** since it alters the state of a `Dice` object. State is altered since a `Dice` object keeps track of how many times it has been rolled. The private instance variable `myRollCount` is modified as follows.

```
myRollCount = myRollCount + 1;
```

Because the state changes, the function `Dice::Roll()` cannot be a `const` function.

The other lines in `Dice::Roll()` actually generate the random roll using another class `RandGen` that generates pseudo-random numbers.

6.1.5 Scope of Private Variables

The instance variables defined in the private section of a class declaration are accessible in all member functions of the class. Private variable names are **global** to all member functions, since they can be accessed in each member function. In the `Dice` class the instance variable `mySides` is initialized in the constructor and used in `Dice::Roll()` to generate a random roll. The instance variable `myRollCount` is initialized in the constructor, incremented in `Dice::Roll()` and used to return a value in `Dice::NumRolls()`.

Program Tip 6.4: If a variable is used in only one member function, it's possible that the variable should be defined locally within the function, and not as a private instance variable. There are occasions when this heuristic doesn't hold (e.g., when a variable must maintain its value over multiple calls of the same member function), but it's a good, general class design heuristic.

By defining a variable at the beginning of a program and outside of any function, you can make it global to all the functions in a program. A **global variable** is accessible everywhere in a program without being passed as a parameter. This is considered poor programming style, because the proliferation of global variables in large programs makes it difficult to modify one part of the program without affecting another part. Because global variables cannot be used in large programs without great care (and even then global variables can cause problems) we will not use any global variables even in small programs.

Program Tip 6.5: Avoid using global program variables. Global variables don't work in large programs, so practice good coding style by avoiding their use in small programs.

Pause to Reflect

6.1 How do the displays and buttons on a stereo receiver provide an interface to the receiver? If you purchase a component stereo system (e.g., a CD player, a tuner, a receiver, and a cassette deck), do you need to buy a new receiver if you upgrade the CD player? How is this similar to or different from a header file and its corresponding implementation?

6.2 Do you know how a soda-vending machine works (on the inside)? Can you "invent" a description of how one works that is consistent with your knowledge based on using such machines?

6.3 Why are there so many comments in the header file `dice.h`?

6.4 What is the purpose of the member functions `NumSides` and `NumRolls`? For example, why won't the lines

```
Dice tetra(4);
cout << "# of sides = " << tetra.mySides << endl;
```

compile, and what is an alternative that will compile?

6.5 In the member function `Dice::Roll()` the value returned is specified by the following:

```
gen.RandInt(1,mySides)
```

What type/class of variable is `gen` and where is the class declared?

6.6 What changes to *roll.cpp*, Program 5.11, permit the user to enter the number of sides in the simulated die?

6.7 Can the statement `myRollCount++` by used in place of `myRollCount = myRollCount + 1` in `Dice::Roll()`?

6.8 Suppose a member function `Dice::LastRoll()` is added to the class `Dice`. The function returns the value of the most recent roll. Should `Dice::LastRoll()` be a `const` function? What changes to private data and to other member functions are needed to implement the new member function?

```
int Dice::LastRoll()  // is const needed here?
// post: returns value of last time Roll() was called
```

6.2 Program Design with Functions

To see how useful classes are in comparison to using only free functions[3] in the design and implementation of programs we'll study a program that gives a simple quiz on arithmetic using addition. For example, you might be asked to write a program like this to help your younger sibling practice with math problems, or to help an elementary school teacher with a drill-and-practice program for the computer. We'll begin with a program that uses free functions to implement the quiz. In the next chapter we'll modify the quiz programs from this chapter so that several collaborating classes are used instead of free functions. The version developed in this chapter serves as a prototype of the final version. A **prototype** is not a finished product, but is useful in understanding design issues and in getting feedback from users.

Program Tip 6.6: A prototype is a good way to start the implementation phase of program development and to help in the design process. A prototype is a "realistic model of a system's key functions" [McC93]. Booch says that "prototypes are by their very nature incomplete and only marginally engineered." [Boo94] A prototype is an aid to help find some of the important issues before design and implementation are viewed as frozen, or unchanging. For those developing commercial software, prototypes can help clients articulate their needs better than a description in English.

Program 6.2 uses classes and functions we've used in programs before. The header file `randgen.h` for class `RandGen` is in Howto G, but we'll need only the function `RandGen::RandInt` that returns a random integer between (and including) the values of the two parameters as illustrated in Program 6.2

Program 6.2 simpquiz.cpp

```
#include <iostream>
#include <iomanip>      // for setw
#include <string>
using namespace std;
#include "randgen.h"    // for RandInt
#include "prompt.h"
```

[3]Recall that a free function is any function defined outside of a class.

```
// simple quiz program

int MakeQuestion()
// postcondition: creates a random question, returns the answer
{
    const WIDTH = 7;
    RandGen gen;
    int num1 = gen.RandInt(10,20);
    int num2 = gen.RandInt(10,20);

    cout << setw(WIDTH) << num1 << endl;
    cout << "+" << setw(WIDTH−1) << num2 << endl;
    cout << "——-" << endl;

    return num1 + num2;
}

int main()
{
    string name = PromptString("what is your name? ");
    int    correctCount = 0;
    int    total = PromptRange(name + ", how many questions, ",1,10);
    int    answer,response, k;

    for(k=0; k < total; k++)
    {   answer = MakeQuestion();
        cout << "answer here: ";
        cin >> response;
        if (response == answer)
        {   cout << "correct! " << endl;
            correctCount++;
        }
        else
        {   cout << "incorrect, answer = " << answer << endl;
        }
    }
    int percent = double(correctCount)/total ∗ 100;
    cout << name << ", your score is " << percent << "%" << endl;

    return 0;
}
```

simpquiz.cpp

OUTPUT

```
prompt> simpquiz
what is your name? Owen
Owen, how many questions, between 1 and 10: 3
     20
+    18
-------
answer here: 38
correct
     13
+    17
-------
answer here: 20
incorrect, answer = 30
     18
+    10
-------
answer here: 28
correct
Owen, your score is 66%
```

6.2.1 Evaluating Classes and Code: Coupling and Cohesion

This program works well for making simple quizzes about arithmetic, but it's hard to modify the program to make changes such as these:

1. Allow the student (taking the quiz) more than one chance to answer the question. A student might be allowed several chances depending on the difficulty of the question asked.
2. Allow more than one student to take a quiz at the same time, say two students sharing the same keyboard.
3. Record a student's results so that progress can be monitored over several quizzes.

As we noted in Program Tips 4.4 and 4.10, writing code that's simple to modify is an important goal in programming. You can't always anticipate what changes will be needed, and code that's easy to modify will save lots of time in the long run.

The modifications above are complicated for a few reasons.

1. There's no way to repeat the same question. If the student is prompted for an answer several times, the original question may scroll off the screen.
2. The body of the `for` loop could be moved into another function parameterized by

name. This might be the first step in permitting a quiz to be given to more than one student at the same time, but in the current program it's difficult to do this.

3. Once we learn about reading and writing information from and to files we'll be able to tackle this problem more easily, but it will still be difficult using the current program. It's difficult in part because the code for giving the quiz and the code for recording quiz scores will be mixed together, making it hard to keep the code dealing with each part separate. Keeping the code separate is a good idea because it will be easier to modify each part if it is independent of the other parts.

The last item is very important. It is echoed by two program and class design heuristics.

Program Tip 6.7: Code, classes, and functions should be as cohesive as possible. A **cohesive function** does one thing rather than several things. A **cohesive class** captures one idea or set of related behaviors rather than many more unrelated ideas and behaviors. When designing and implementing functions and classes you should make them highly cohesive.

The function `MakeQuestion` from Program 6.2 does two things: it makes a question and it returns the answer to the question. Doing two things at the same time makes it difficult to do just one of the two things, (e.g., ask the same question again). Functions that do one thing are more cohesive than functions that do two things.

Program Tip 6.8: Code, classes, and functions should not be coupled with each other. Each function and class should be as independent from others as possible, or **loosely coupled**. It's impossible to have no coupling or functions and classes wouldn't be able to call or use each other. But loose coupling is a goal in function and class design.

A function is tightly coupled with another function if the functions can't exist independently or if a change in one causes a change in the other. Ideally, changing a function's implementation without changing the interface or prototype should cause few changes in other functions. In Prog 6.2, *simpquiz.cpp* the function `MakeQuestion` which makes questions and `main` which gives a quiz are tightly coupled with each other and with the student taking the quiz. These three parts of the program should be less coupled than they are.

6.2.2 Toward a Class-based Quiz Program

We want to develop a quiz program that will permit different kinds of questions, that is not just different kinds of arithmetic problems, but questions about state capitals, English literature, rock and roll songs, or whatever you think would be fun or instructive. We'd like the program to be able to give a quiz to more than one student at the same time, so that two people sharing a keyboard at one computer could both participate. If possible, we'd like to allow a student to have more than one chance at a question.

In the next chapter we'll study one design of a program that will permit different kinds of quizzes for more than one student. That program will use three collaborating classes. However, we need to study a few more C++ language features and some new classes before we tackle the quiz program.

Before we develop the class design we must study another mode of parameter passing that we'll need in developing more complex classes, functions, and programs. We'll use a modified version of *simpquiz.cpp*, Program 6.2.

As we move toward a new quiz program, think about how the program changes. You'll find that there is no "best design" or "correct design" when it comes to writing programs. However, there are criteria by which classes and programs can be evaluated, such as **coupling** and **cohesion** as outlined in Program Tips 6.8 and 6.7.

6.2.3 Reference parameters

Program 6.3, *simpquiz2.cpp*, is a modified version of Program 6.2 that uses a function `GiveQuiz` as an encapsulation of the code in `main` of Program 6.2. This encapsulation makes it easier to give a quiz to more than one person in the same program and is a step toward developing a class-based program. The output of Program 6.2 and Program 6.3 are exactly the same (given that the random questions may be different). The function `GiveQuiz` passes two values back to `main` when `GiveQuiz` is called: the number of questions answered correctly and the total number of questions. Since two values are passed back, it's not possible to use a return type which passes only one value back.

In the header of `GiveQuiz` in Program 6.3, *simpquiz2.cpp*, note that the last two parameters are preceded by an ampersand, &. Using an ampersand permits values to be passed back from the function to the calling statement.

Program 6.3 simpquiz2.cpp

```
#include <iostream>
#include <iomanip>      // for setw
#include <string>
using namespace std;
#include "randgen.h"    // for RandInt
#include "prompt.h"

// simple quiz program

int MakeQuestion()
// postcondition: creates a random question, returns the answer
{
    const WIDTH = 7;
    RandGen gen;
    int num1 = gen.RandInt(10,20);
    int num2 = gen.RandInt(10,20);

    cout << setw(WIDTH) << num1 << endl;
    cout << "+" << setw(WIDTH−1) << num2 << endl;
    cout << "——-" << endl;

    return num1 + num2;
}

void GiveQuiz(string name, int & correct, int & total)
// precondition:  name = person taking the quiz
// postcondition: correct = # correct answers, total = # questions
{
    correct = 0;
    total = PromptRange(name + ", how many questions, ",1,10);
    int  answer,response, k;

    for(k=0; k < total; k++)
    {   answer = MakeQuestion();
        cout << "answer here: ";
        cin >> response;
        if (response == answer)
        {   cout << "correct! " << endl;
            correct++;
        }
        else
        {   cout << "incorrect, answer = " << answer << endl;
        }
    }
}

int main()
{
    int correctCount, total;
    string student = PromptString("what is your name? ");
    GiveQuiz(student, correctCount, total);
    int percent = double(correctCount)/total ∗ 100;
    cout << student << ", your score is " << percent << "%" << endl;

    return 0;
}
```

simpquiz2.cpp

The first parameter of the function `GiveQuiz` represents the name of the student taking the quiz. This value is passed into the function. The other parameters are used to pass values back from the function `GiveQuiz` to the statement calling the function. These last three parameters are **reference** parameters; the ampersand appearing between the type and name of the parameter indicates a reference parameter. The diagram in Figure 6.2 shows how information flows between `GiveQuiz` and the statement that calls `GiveQuiz` from `main`. The ampersand modifier used for the last three parameters in the prototype of `GiveQuiz` makes these references to integers rather than integers. We'll elaborate on this distinction, but a reference is used as an alias to refer to a variable that has already been defined. The memory for a reference parameter is defined somewhere else, whereas the memory for a nonreference parameter, also called a **value** parameter, is allocated in the function.

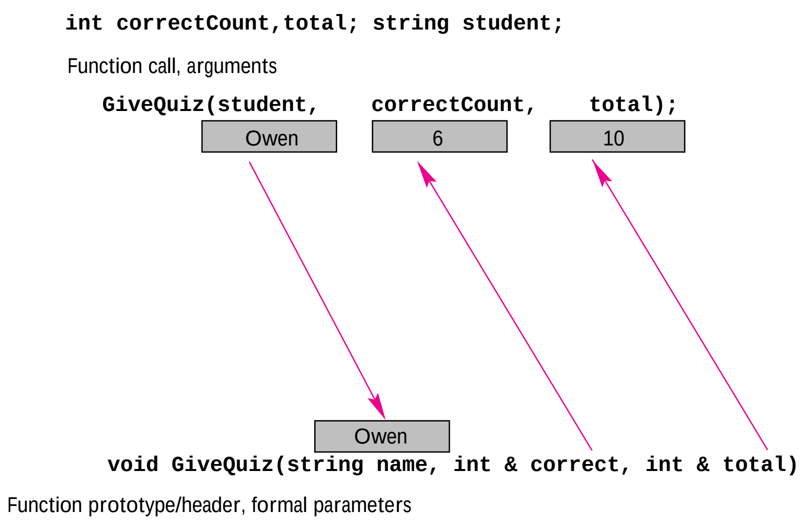

int correctCount,total; string student;
Function call, arguments
GiveQuiz(student, correctCount, total);
Owen
6
10
Owen
void GiveQuiz(string name, int & correct, int & total)
Function prototype/header, formal parameters

Figure 6.2 Passing parameters by value and by reference in `simpquiz2.cpp`

The value of `student` (*Owen*, in the figure) is copied from `main` into the memory location associated with the parameter `name` in `GiveQuiz`. Once the value is copied, the variable `student` defined in `main` and the parameter `name` in `GiveQuiz` are not connected or related in any way. For example, if the value of `name` in `GiveQuiz` is changed, the value of `name` in `main` is *not* affected. This is very different from how reference parameters work. As indicated in Figure 6.2, the storage for the last two arguments in the function call is referenced, or referred to, by the corresponding parameters in `GiveQuiz`. For example, the variable `correctCount` defined in `main` is referred to by the name `correct` within the function `GiveQuiz`. When one storage location (in this case, defined in `main`) has two different names, the term **aliasing** is sometimes used. Whatever happens to `correct` in `GiveQuiz` is really happening to the variable `correctCount` defined in `main` since `correct` refers to `correctCount`. This means that if the statement `correct++;` assigns 3 to `correct` in `GiveQuiz`, the value is actually stored in the memory location allocated in `main` and referred to by the name `correctCount` in `main`. Rich Pattis, author of *Get A-Life: Advice for the Beginning C++ Object-Oriented Programmer* [Pat96] calls reference parameters "voodoo doll" parameters: if you "stick" `correct` in `GiveQuiz`, the object `correctCount` in `main` yells "ouch."

One key to understanding the difference between the two kinds of parameters is to remember where the storage is allocated. For reference parameters, the storage is allocated somewhere else, and the name of the parameter refers back to this storage. For value parameters, the storage is allocated in the function, and a value is copied into this storage location. This is diagrammed by the leftmost arrow in Figure 6.2. When reference parameters are used, memory is allocated for the arguments, and the formal parameters are merely new names (used within the called function) for the memory locations associated with the arguments. This is shown in Figure 6.2 by the arrows that point "up" from the identifiers `correct` and `total` that serve as aliases for the memory locations allocated for the variables `correctCount` and `total` in `main`.

6.2.4 Pass by Value and Pass by Reference

Program 6.4, *pbyvalue.cpp* shows a contrived (but hopefully illustrative) example of parameter passing.

Program 6.4 pbyvalue.cpp

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

// illustrates pass-by-value/pass-by-reference semantics

void DoStuff(int number, string & word)
{
    cout << "DoStuff in:\t" << number << " " << word << endl;
    number *= 2;
    word = "What's up Doc?";
    cout << "DoStuff out:\t" << number << " " << word << endl;
}

void DoStuff2(int & one, int & two, string & word)
{
    cout << "DoStuff2 in:\t" << one << " " << two << " " << word << endl;
    one *= 2;
    cout << "DoStuff2 mid:\t" << one << " " << two << " " << word << endl;
    two += 1;
    word = "What's up Doc?";
    cout << "DoStuff2 out:\t" << one << " " << two << " " << word << endl;
}

int main()
{
    int num = 30;
    string name = "Bugs Bunny";

    DoStuff(num,name);
    cout << endl << "DoStuff main:\t" << num << " " << name << endl << endl;

    DoStuff2(num,num,name);
    cout << endl << "DoStuff2 main:\t" << num << " " << name << endl;
    return 0;
}
```

pbyvalue.cpp

The parameter `number` in the function `DoStuff` is passed by value, not by reference, so assignment to `number` does *not* affect the value of the argument `num`. The same does not hold for the reference parameter `word`; the changed value does change the value of the argument `name` in `main`.

In contrast, all parameters are reference parameters in `DoStuff2`. What's very tricky[4] about `DoStuff2` is that the reference parameters `one` and `two` both alias the same memory location `num` in `main`. Assignment to `one` is really assignment to `num` and thus also assignment to `two` since both `one` and `two` reference the same memory. It helps to draw a diagram like the one in Figure 6.2, but with arrows from `one` and `two` both pointing to the same memory location associated with `num` in `main`.

O U T P U T

```
prompt> pbyvalue
DoStuff in:     30 Bugs Bunny
DoStuff out:    60 What's up Doc?

DoStuff main:   30 What's up Doc?

DoStuff2 in:    30 30 What's up Doc?
DoStuff2 mid:   60 60 What's up Doc?
DoStuff2 out:   61 61 What's up Doc?

DoStuff2 main:  61 What's up Doc?
```

The first line of output prints the values that are passed to `DoStuff`. The value of the parameter `number` in `DoStuff` is the same as the value of `num` in `main` since this value is copied when the argument is passed to `DoStuff`. After the value is copied, there is no relationship between `number` and `num`. This can be seen in the first line of output generated in `main`: `num` is still 30. However, the change to parameter `word` does change `name` in `main`. Values are *not* copied when passed by reference. The identifiers `word` and `name` are aliases for the same memory location.

When a function is called and an argument passed to a reference parameter, we use the term **call by reference**. When an argument is copied into a function's parameter, we use the term **call by value**. Value parameters require time to copy the value and require memory to store the copied value; it's possible for this time and space to have an impact on a program's performance. Sometimes reference parameters are used to save time and space. Unfortunately, this permits the called function to change the value of the argument—the very reason we used reference parameters in Program 6.3. You can, however, protect against unwanted change and still have the efficiency of reference parameters when needed.

[4] I could have written, "what's verwy twicky," but I didn't.

6.2.5 `const` Reference Parameters

Value parameters are copied from the corresponding argument, as shown in *pbyvalue.cpp*, Program 6.4. For parameters that require a large amount of memory, making the copy takes time in addition to the memory used for the copy. In contrast, reference parameters are not copied, and thus no extra memory is required and less time is used.

Some programs must make efficient use of time and memory space. Value parameters for large objects are problematic in such programs. Using **const reference** or **constant reference** parameters yields the efficiency of reference parameters and the safety of value parameters. "Safety" means that it's not possible to change a value parameter so that the argument is also changed. The argument is protected from accidental or malicious change. Like value parameters, const reference parameters cannot be changed by a function so that the argument changes (as we'll see, assignments to const reference parameters are prohibited by the compiler.) A `const` reference parameter is defined using the `const` modifier in conjunction with an ampersand as shown in *constref.cpp*, Program 6.5. Const reference parameters are also called **read-only** parameters.

Program 6.5 constref.cpp

```
#include <iostream>
#include <string>
using namespace std;
#include "prompt.h"

// illustrates const reference parameters
// Owen Astrachan, 7/11/96, 4/19/99

void Print(const string & word);

int main()
{
    string word = PromptString("enter a word: ");

    Print("hello world");
    Print(word);
    Print(word + " " + word);

    return 0;
}

void Print(const string & word)
{
    cout << "printing: " << word << endl;
}
```

constref.cpp

OUTPUT

```
prompt> constref
enter a word: rabbit
printing: hello world
printing: rabbit
printing: rabbit rabbit
```

The parameter `word` in `Print` is a `const` reference parameter. The use of `const` prevents the code in `Print` from "accidentally" modifying the value of the argument corresponding to `word`. For example, adding the statement `word = "hello"` just before the output statement generates the following error message with one compiler:

```
Error   : cannot pass const/volatile data object to
          non-const/volatile member function
constref.cpp line 23  {word = "hello";
```

In addition, const reference parameters allow literals and expressions to be passed as arguments. In *constref.cpp*, the first call of `Print` passes the literal `"hello world"`, and the third call passes the expression `word + " " + word`. Literals and expressions can be arguments passed to value parameters since the value parameter provides the memory. However, literals and expressions cannot be passed to reference parameters since there is no memory associated with either a literal or an expression. Fortunately, the C++ compiler will generate a temporary variable for literals and expressions when a const reference parameter is used. If the `const` modifier is removed from `Print` in *constref.cpp*, the program will fail to compile with most compilers.

Program Tip 6.9: Parameters of programmer-defined classes like `string` should be `const` reference parameters rather than value parameters. (Occasionally a copy is needed rather than a const reference parameter, but such situations are rare). There is no reason to worry about this kind of efficiency for built-in types like `int` and `double`; these use relatively little memory, so that a copy takes no more time to create than a reference does and no temporary variables are needed when literals and expressions are passed as arguments.

For some classes a specific function is needed to create a copy. If a class does not supply such a "copy-making" function—actually a special kind of constructor called a **copy constructor**—one will be generated by the compiler. This default copy constructor may not behave properly in certain situations that we'll discuss at length later. A brief discussion of copy constructors can be found in Section 12.10.

The compiler will allow only accessor functions (see Section 6.1) labelled as `const` member functions to be applied to a const reference parameter. If you try to invoke a mutator (non-const) member function on a const reference parameter the compilation will catch this error and fail to compile the program.[5]

Pause to Reflect

6.9 What is the function header and body of a function `GetName` that prompts for a first and last name and returns two strings, one representing each name?

6.10 Write a function `Roots` having the following function header:

```
void Roots(double a, double b, double c,
           double & root1, double & root2)
// precondition:   a,b,c coefficients of
//                 ax^2 + bx + c
// postcondition:   sets root1 and root2 to roots
//                  of quadratic
```

that uses the quadratic formula:

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

to find the roots of a quadratic. The call `Roots(1,5,6,r1,r2)` would result in `r1` and `r2` being set to -2 and -3. You'll have to decide what to do if there are no real roots.

6.11 Suppose that a function `Mystery` has only value parameters. What is printed by the following statements? Why?

```
int num = 3;
double top = 4.5;
Mystery(num,top);
cout << num << " " << top << endl;
```

6.12 Write the header for a function that returns the number of weekdays (Monday through Friday) and weekend days (Saturday and Sunday) in a month and year that are input to the function as integer values using 1 for January and 12 for December. Don't write the function, just a header with pre- and post-conditions.

6.13 Two formal parameters can alias the same argument as shown in `Change`:

```
void Change(int & first, int & second)
{
    first += 2;
    second *= 2;
}
```

[5]Some older compilers may issue a warning rather than an error, but 32-bit compilers will catch const errors and fail.

Using the function `Change` above, explain why 20 is printed by the code fragment below and determine what is printed if `num` is initialized to 3 rather than 8.

```
int main()
{
    int num = 8;
    Change(num,num);
    cout << num << endl;
    return 0;
}
```

6.14 It is often necessary to interchange, or swap, the values of two variables. For example, if $a = 5$ and $b = 7$, then swapping values would result in $a = 7$ and $b = 5$. Write the body of the function `Swap` (*Hint:* You'll need to define a variable of type `int`).

```
void Swap(int & a, int & b)
// postcondition: interchanges values of a and b
```

We want to develop question classes for different kinds of quizzes, but we need some more programming tools. In the next sections we'll see how to read from files instead of just from the keyboard. We'll see how to write to files too.

6.3 Reading Words: Stream Iteration

If you steal from one author, it's plagiarism;
if you steal from many, it's research.
Wilson Mizner

Word processing programs merely manipulate words and characters, but scholars sometimes use programs that process character data to determine authorship. For example, literary investigators have sought to determine the authorship of Shakespeare's plays and sonnets. Some have argued that they were written by philosopher Francis Bacon or dramatist Christopher Marlowe, but most Shakespearean authorities doubt these claims. To amass evidence regarding the authorship of a literary work, it is possible to gather statistics to create a "literary fingerprint." Such a fingerprint can be based on frequently used words, phrases, or allusions. It can also include a count of uncommon words. Computer programs facilitate the gathering of these data.

In this section, we demonstrate the pattern of iterating over words and characters by simpler, but similar, kinds of programs. These programs will count words and letters—the kind of task that is built into many word processing programs and used when a limit on the number of words in an essay is set, (e.g., by newspaper columnists and students writing English papers). We'll first write a program that counts words entered by the user or stored in a text file. A text file is the kind of file in which C++ programs are stored or word processing documents are saved when the latter are saved as plain text[6].

I'll adopt a four-step process in explaining how to develop the program. As you write and develop programs, you should think about these steps and use them if they make sense. These steps are meant as hints or guidelines, not rules that should be slavishly followed.

6.3.1 Recommended Problem-solving and Programming Steps

1. Think about how to solve the problem with paper, pencil, and brain (but no computer). Consider how to extend the human solution to a computer-based solution. You may find it useful to sketch a solution using **pseudocode,** a mixture of English and C++.
2. If, after thinking about how to solve the problem with a computer (and perhaps writing out a solution), you are not sure how to proceed, pause. Try thinking about solving a related problem whose solution is similar to a program previously studied.
3. Develop a working program or class in an iterative manner, implementing one part at a time before implementing the entire program. This can help localize problems and errors because you'll be able to focus on small pieces of the program rather than the entirety.
4. When you've finished, pause to reflect on what you've learned about C++, programming, and program design and implementation. You may be able to develop guidelines useful in your own programming, and perhaps useful to others as well.

We will use these steps to solve the word count problem. First we'll specify the problem in more detail and develop a pseudocode solution. This step will show that we're missing some knowledge of how to read from files, so we'll solve a related problem on the way to counting the words in a text file. After writing a complete program we'll develop a class-based alternative that will provide code that's easier to reuse in other contexts.

6.3.2 A Pseudocode Solution

Counting the words in this chapter or in Shakespeare's play *Hamlet* by hand would be a boring and arduous task. However, it's an easy task for a computer program—simply scan the text and count each word. It would be a good idea to specify more precisely what a "word" is. The first part of any programming task is often a careful **specification** of exactly what the program should do. This may require defining terms such as *word.* In this case, we'll assume that a word is a sequence of characters separated from other words by white space. White space characters are included in the escape characters in Table A.5 in Howto A; for our purposes, white space is `' '`, `'\t'`, and `'\n'`: the space, tab, and newline characters[7]. Escape sequences represent certain characters such

[6]The adjective *plain* is used to differentiate text files from files in word processors that show font, page layout, and formatting commands. Most word processors have an option to save files as plain text.

[7]Other white space characters are formfeed, return, and vertical tab.

as the tab and newline in C++. To print a backslash requires an escape sequence; \\ prints as a single backslash[8].

For this problem, we'll write a pseudocode description of a loop to count words. Pseudocode is a language that has characteristics of C++ (or Java, or some other language), but liberties are taken with syntax. Sketching such a description can help focus your attention on the important parts of a program.

```
numWords = 0;
while (words left to read)
{   read a word;
    numWords++;
}
print number of words read
```

White Space Delimited Input for Strings. These pseudocode instructions are very close to C++, except for the test of the `while` loop and the statement `read a word`. In fact, we've seen code that reads a word using the extraction operator >> (e.g., Program 3.1, *macinput.cpp*). White space separates one string from another when the extraction operator >> is used to process input. This is just what we want to read words. As an example, what happens if you type `steel-gray tool-box` when the code below is executed?

```
string first, second;
cout << "enter two words:";
cin >> first >> second;
cout << first << " : " << second << endl;
```

Since the space between the *y* of *steel-gray* and the *t* of *tool-box* is used to delimit the words, the output is the following:

OUTPUT

```
steel-gray : tool-box
```

As another example, consider this loop, which will let you enter six words:

```
string word;
int numWords;
for(numWords=0; numWords < 6; numWords++)
{   cin >> word;
    cout << numWords << " " << word << endl;
}
```

[8]Consider buying groceries. Often a plastic bar is used to separate your groceries from the next person's. What happens if you go to a store to buy one of the plastic bars? If the person behind you is buying one too, what can you use to separate your purchases?

Suppose you type the words below with a tab character between `it` and `ain't`, the return key pressed after `broke`, and two spaces between `don't` and `fix`.

```
If it       ain't broke,
don't   fix it.
```

OUTPUT

```
0  If
1  it
2  ain't
3  broke,
4  don't
5  fix
```

Although the input typed by the user appears as two lines, the input stream `cin` processes a sequence of characters, not a sequence of words or lines. The characters on the input stream appear as literally a stream of characters (the symbol ⊔ is used to represent a space).

```
If⊔it\tain\'t⊔broke,\ndon\'t⊔⊔fix it.
```

There are three different escape characters in this stream: the tab character, `\t`, the newline character, `\n`, and the apostrophe character, `\'`. We don't need to be aware of these escape characters, or any other individual character, to read a sequence of words using the loop shown above. At a low level a stream is a sequence of characters, but at a higher level we can use the extraction operator, `>>`, to view a stream as a sequence of words.

The extraction operator, `>>`—when used with `string` variables—groups adjacent, nonwhite space characters on the stream to form words as shown by the output of the `while` loop above. Note that punctuation is included as part of the word `broke,` because all nonwhite space characters, including punctuation, are the same from the point of view of the input stream `cin`. Since the operator `>>` treats all white space the same, the newline is treated the same as the spaces or tabs between adjacent words. Any sequence of white space characters is treated as white space, as can be seen in the example above, where a tab character space separates `it` from `ain't` and two spaces separate `don't` from `fix`.

Now that we have a better understanding of how the extraction operator works with input streams, characters, and words, we need to return to the original problem of counting words in a text file. We address two problems: reading an arbitrary number of words and reading from a file. We cannot use a definite loop because we don't know in advance how many words are in a file—that's what we're trying to determine.

6.3.3 Solving a Related Problem

How can a loop be programmed to stop when there are no more words in the input? Step two of our method requires solving a familiar but related problem when confronted with a task whose solution isn't immediately apparent. In this case, suppose that words are to be entered and counted until you enter some specific word signaling that no more words will be entered. The test of a while loop used to solve this task can consist of `while (word != LAST_WORD)`, where `LAST_WORD` is the special word indicating the end of the input and `word` holds the value of the string that you enter.

This is an example of a **sentinel** loop—the sentinel is the special value that indicates the end of input. Such loops are classic fence post problems: you must enter a word before the test is made and, if the test indicates there are more words, you must enter another word. Program 6.6 shows such a sentinel loop accepting entries until the user enters the word *end*. The special sentinel value is *not* considered part of the data being processed. Sometimes it's difficult to designate a sentinel value since no value can be singled out as invalid data. In the second run the number of words does not appear immediately after the word *end* is entered since more typing takes place afterward. The number of words is not output until after the return key is pressed, and this occurs several words after the word *end* is entered.

Program 6.6 sentinel.cpp

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

   // count words in the standard input stream, cin

int main()
{
    const string LAST_WORD = "end";

    string word;
    int numWords = 0;                  // initially, no words

    cout << "type '" << LAST_WORD << "' to terminate input" << endl;

    cin >> word;
    while (word != LAST_WORD)          // read succeeded
    {   numWords++;
        cin >> word;
    }
    cout << "number of words read = " << numWords << endl;

    return 0;
}
```

sentinel.cpp

OUTPUT

```
prompt> sentinel
type 'end' to terminate input
One fish, two
fish, red fish, blue fish
end
number of words read = 8

prompt> sentinel
type 'end' to terminate input
How will the world end — with a bang or a whimper?
number of words read = 4
```

This apparent delay is a side effect of **buffered input,** which allows the user to make corrections as input is entered. When input is buffered, the program doesn't actually receive the input and doesn't do any processing until the return key is pressed. The input is stored in a memory area called a *buffer* and then passed to the program when the line is finished and the return key pressed. Most systems use buffered input, although sometimes it is possible to turn this buffering off.

Although we still haven't solved the problem of developing a loop that reads all words (until none are left), the sentinel loop is a start in the right direction and will lead to a solution in the next section.

Pause to Reflect

6.15 The sentinel loop shown here reads integers until the user enters a zero. Modify the loop to keep two separate counts: the number of positive integers entered and the number of negative integers entered. Use appropriate identifiers for each counter.

```
const int SENTINEL = 0;
int count = 0;

int num;
cin >> num;
while (num != SENTINEL)
{   count++;
    cin >> num;
}
```

6.16 Does your system buffer input in the manner described in this section? What happens if Program 6.6 is run and the user enters the text below? Why?

```
This is the start, this is the end --- nothing
is in between.
```

6.17 Another technique used with sentinel loops is to force the loop to iterate once. This is called **priming** the loop[9]. If the statement `cin >> word` before the `while` loop in Program 6.6 is replaced with the statement `word = "dummy";`, how should the body of the while loop be modified so that the program counts words in the same way?

6.18 Suppose that you want to write a loop that stops after either of two sentinel values is read. Using the technique of the previous problem in which the loop is forced to iterate once by giving a dummy value to the string variable used for input, write a loop that counts words entered by the user until the user enters either the word `end` or the word `finish`. Be sure to use appropriate `const` definitions for both sentinels.

6.3.4 The Final Program: Counting Words

We are finally ready to finish a program that counts all the words in a text file or all the words a user enters. We would like to refine the loop in Program 6.6, *sentinel.cpp*, so that it reads all input but does not require a sentinel value to identify the last word in the input stream. This will let us calculate the number of words (or characters, or occurrences of the word *the*) in any text file since we won't need to rely on a specific word to be the sentinel last word. This is possible in C++ because the extraction operator not only extracts strings (or numbers) from an input stream, but returns a result indicating whether the extraction succeeds. For example, a code fragment used earlier read the words `steel-gray` and `tool-box` using the statement

```
cin >> first >> second;
```

This statement is read, or parsed, by the C++ compiler as though it were written as

```
(cin >> first) >> second;
```

because >> is left-associative (see Table A.4 in Howto A.) Think of the input stream, `cin`, as flowing through the extraction operators, >>. The first word on the stream is extracted and stored in `first`, and the stream continues to flow so that the second word on the stream can be extracted and stored in `second`. The result of the first extraction, the value of the expression `(cin >> first)`, is the input stream, `cin`, without the word that has been stored in the variable `first`.

The Return Value of `operator >>`. The most important point of this explanation is that the expression `(cin >> first)` not only reads a string from `cin` but returns the stream so that it can be used again, (e.g., for another extraction operation). Although it may seem strange at first, the stream itself can be tested to see if the extraction succeeded. The following code fragment shows how this is done.

[9]The derivation of priming probably comes from old water-pumps that had to be *primed* or filled with water before they started.

```
int num;
cout << "enter a number: ";
if (cin >> num)
{   cout << "valid integer: "   << num << endl;
}
else
{   cout << "invalid integer: " << num << endl;
}
```

OUTPUT

```
enter a number: 23
valid integer: 23
enter a number: skidoo23
invalid integer: 292232
enter a number: 23skidoo
valid integer: 23
```

The expression (cin » num) evaluates to true when the extraction of an integer from cin has succeeded. The characters skidoo23 do not represent a valid integer, so the message invalid integer is printed. The integer printed here is a garbage value. Since no value is stored in the variable num when num is first defined, whatever value is in the memory associated with num is printed. Other runs of the program may print different values. Note that when 23skidoo is entered, the extraction succeeds and 23 is stored in the variable num. In this case, the characters skidoo remain on the input stream and can be extracted by a statement such as cin >> word, where word is a string variable. The use of the extraction operator to both extract input and return a value used in a boolean test can be confusing since the extraction operation does two things.

Some people prefer to write the if statement using the fail member function of the stream cin.

```
cin >> num;
if (! cin.fail())
{   cout << "valid integer: "   << num << endl;
}
```

The member function fail returns true when an extraction operation has failed and returns false otherwise. You do not need to use fail explicitly since the extraction operator returns the same value as fail, but some programmers find it clearer to use fail. The stream member function fail returns true whenever a stream operation has failed, but the only operations we've seen so far are I/O operations. Details of all the stream member functions can be found in Howto B.

Program 6.7 correctly counts the number of words in the input stream `cin` by testing the value returned by the extraction operator in a `while` loop.

Program 6.7 countw.cpp

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

   // count words in the standard input stream, cin

int main()
{
    string word;
    int numWords = 0;                  // initially, no words

    while (cin >> word)                // read succeeded
    {   numWords++;
    }
    cout << "number of words read = " << numWords << endl;
    return 0;
}
```

countw.cpp

The test of the `while` loop is false when the extraction operation fails. When reading strings, extraction fails only when there is no more input. As shown above, input with integers (and `double`s) can fail if a noninteger value is entered. Since any sequence of characters is a string, extraction fails for strings only when there is no more input. If you're using the program interactively, you indicate no more input by typing a special character called the **end-of-file** character. This character should be typed as the first and only character on a line, followed by pressing the return key. When UNIX or Macintosh computers are used, this character is Ctrl-D, and on MS-DOS/Windows machines this character is Ctrl-Z. To type this character the control key must be held down at the same time as the D (or Z) key is pressed. Such control characters are sometimes not shown on the screen but are used to indicate to the system running the program that input is finished (end of file is reached).

O U T P U T

```
prompt> countw
How shall I love thee? Let
me count
the ways.
^D
number of words read = 10
```

The end-of-file character was *not* typed as the string `^D` but by holding down the Control key and pressing the D key simultaneously.

We'll modify *countw.cpp* so that it will count words stored in a text file; then we'll see how to turn this program into a class that makes it a general-purpose programming tool.

6.3.5 Streams Associated with Files

In Program 6.6, *sentinel.cpp*, and Program 6.7, *countw.cpp*, the standard input stream, `cin`, was used as the source of words. Clearly you can't be expected to type in all of *Hamlet* to count the words in that play. Instead, you need some way to create a stream associated with a text file (rather than with the keyboard and the standard input stream, `cin`). A class `ifstream`, accessible by including the file `<fstream>` (or `<fstream.h>` on some systems), is used for streams that are associated with text files. Program 6.8, *countw2.cpp*, is a modification of Program 6.7 but uses an `ifstream` variable.

Program 6.8 countw2.cpp

```
#include <iostream>
#include <fstream>              // for ifstream
#include <string>
#include "prompt.h"

   // count words in a file specified by the user

int main()
{
    string word;
    int numWords = 0;                        // initially no words
    int sum = 0;                             // sum of all word lengths
    ifstream input;

    string filename = PromptString("enter name of file: ");

    input.open(filename.c_str());           // bind input to named file

    while (input >> word)                   // read succeeded
    {   numWords++;
        sum += word.length();
    }
    cout << "number of words read = " << numWords << endl;
    cout << "average word length = " << sum/numWords << endl;

    return 0;
}
```

countw2.cpp

In the following runs, the file `melville.txt` is the text of Herman Melville's *Bartleby, The Scrivener: A Story of Wall-Street.* The file `hamlet.txt` is the complete text of William Shakespeare's *Hamlet*. These, as well as other works by Shakespeare, Edgar Allen Poe, Mark Twain, and others, are accessible as text files.[10]

OUTPUT

```
prompt> countw2
enter name of file: melville.txt
number of words read = 14353
average word length = 4
prompt> countw2
enter name of file: hamlet.txt
number of words read = 31956
average word length = 4

prompt>  countw2
enter name of file: macbet.txt
number of words read = 0
Floating exception
```

The variable `input` is an instance of the class `ifstream`—an **input file stream**—and supports extraction using `>>` just as `cin` does. The variable `input` is associated, or **bound,** to a particular user-specified text file with the member function `ifstream::open()`.

```
input.open(filename.c_str());    // bind input to named file
```

The string `filename` that holds the name of the user-specified file is an argument to the member function `ifstream::open()`. The standard string member function `c_str()` returns a C-style string required by the prototype for the function `open()`. The `open()` function may be modified to accept standard strings, but the conversion function `c_str()` will always work. Once `input` is bound to a text file, the extraction operator `>>` can be used to extract items from the file (instead of from the user typing from the keyboard as is the case with `cin`).

There is a similar class **ofstream** (for *output file stream*) also accessible by including the header file `<fstream>`. This class supports the use of the insertion operator, `<<`, just as `ifstream` supports extraction, using the `>>` operator. The code fragment below writes the numbers 1 to 1,024 to a file named `"nums.dat"`, one number per line.

```
ofstream output;
output.open("nums.dat");
int k;
for(k=0; k < 1024; k++)
```

[10]The files containing these literary works are available with the material that supports this book. These texts are in the public domain, which makes on-line versions of them free.

```
{  output << k << endl;
}
```

In the example of using Program 6.8, file `macbet.txt` has no words. I made a mistake when entering the name of the file to read (I meant to type `macbeth.txt`), which caused the extraction operation to fail because the file does not exist. Because no words were read, the average calculation resulted in a division-by-zero error. On some systems, division by zero can cause the machine to crash. A **robust** program protects against errors. Program 6.8 could be made robust by guarding the average calculation with an `if` statement to check whether `numWords == 0`. It's also possible to check the result of the function `ifstream::open` as shown.

```
input.open(filename.c_str());
if (input.fail())
{   cout << "open for " << filename << " failed " << endl;
}
```

You should always look carefully at program output to determine if it meets your expectations. The average printed for both *Hamlet* and *Melville* is four. This is surprising; you probably do not expect the averages to be exactly the same. To fix this problem we'll need to change how the average is calculated; we need to use `double` values.

6.3.6 Type Casting

Since both `numWords` and `sum` are `int` variables, the result of the division operator, `/`, is an `int`. How can the correct average be calculated? One method is to define `sum` to be a `double` variable. Since the statement

```
sum += word.length();
```

will correctly accumulate a sum of integers even when `sum` is a `double` variable, this method will work reasonably well. However, it may not be the best method, since the wrong type (`double`) is being used to accumulate the sum of integers. Instead, we can use a **type cast.** This is a method that allows one type to be converted (sometimes called **coerced**) into another type. For example, the statement:

```
cout << "average length = " << double(sum)/numWords << endl;
```

yields the correct average of 4.705 for *Melville* and 4.362 for *Hamlet.* The expression `double(sum)` shows that the type `double` is used like a function name with an argument `sum`. The result is a `double` that is as close to the integer value of `sum` as possible. Since the result of a mixed-type arithmetic expression is of the highest type (in this case, `double`), 3.5 will be printed. You can also write a cast as `((double) sum)/numWords`.[11] A cast has higher precedence than arithmetic operators (see Table A.4 in Howto A), so `(double) sum/numWords` will also work because `sum` is cast to a `double` value before the division occurs.

[11]This is the C-style of casting but can be used in C++ and is useful if the cast is to a type whose name is more than one word, such as `long int`.

Alternatively, the statement

```
cout << "average length = " << sum/double(numWords) << endl;
```

also gives a correct result since the mixed-type expression yields a `double` result.

Program Tip 6.10: Be careful when casting a value of one type to another. It is possible that a type cast will result in a value changing. Casting is sometimes necessary, but you should be cautious in converting values of one type to another type.

For example, using Turbo C++ the output of the three statements

```
cout << int(32800.2) << endl;
cout << double(333333333333333) << endl;
cout << int(3.6) << endl;
```

follows.

OUTPUT

```
-32736
9.214908e+08
3
```

The third number printed is easy to explain—casting a double to an `int` **truncates,** or chops off, the decimal places. The first two numbers exceed the range of valid values for `int` and `double`, respectively using Turbo C++.

In general, casting is sometimes necessary, but you must be careful that the values being cast are valid values in the type being cast to.

Casting with `static_cast`. Four cast operators are part of standard C++. In this book the operator `static_cast` will be used.[12] As an example, the statement

```
cout << double(sum)/numWords << endl;
```

is written as shown in the following to use the `static_cast` operator.

```
cout << static_cast<double>(sum)/numWords << endl;
```

Your C++ compiler may not support `static_cast`, but this will change soon as the C++ standard is adopted. Using `static_cast` makes casts easier to spot in code. Also, since casting a value of one type to another is prone to error, some people prefer to use `static_cast` because it leads to ugly code and will be less tempting to use.

[12]The other cast operators are `const_cast`, `dynamic_cast`, and `reinterpret_cast`; we'll have occasion to use these operators, but rarely.

6.3.7 A Word-Reading Class Using `ifstream`

The third of the program development guidelines given in Section 6.3.1 calls for programs to be developed using an iterative process. Sometimes this means redesigning an already working program so that it will be useful in different settings. In the case of Program 6.7, *countw.cpp*, we want to reimplement the program in a new way to study a programming pattern that we will see on many occasions throughout this book. The resulting program will be longer, but it will yield a C++ class that is easier to modify for new situations than the original program. It will also help us focus on a pattern you can use in other classes and programs: the idea of processing "entries." In this case we'll process all the words in a text file. The same design pattern can be used to process all the prime numbers between 1000 and 9999, all the files in a computer disc directory, and all the tracks on a compact disc.

The pattern of iteration over entries is expressed in pseudocode as

```
find the first entry;
while (the current entry is valid)
{
   process the current entry;
   advance to the next entry;
}
```

We'll use a `WordStreamIterator` class to get words one at a time from the text file.

As an example of how to use the class, the function `main` below is black-box equivalent to Program 6.7, *countw.cpp*. For any input, the output of these two programs is the same.

```
int main()
{
   string word;
   int numWords = 0;           // initially, no words
   WordStreamIterator iter;

   iter.Open(PromptString("enter name of file: "));

   for(iter.Init(); iter.HasMore(); iter.Next())
   {   numWords++;
   }
   cout << "number of words read = " << numWords << endl;

   return 0;
}
```

This program fragment may seem more complex than the code in *countw2.cpp*, Program 6.8. This is often the case; using a class can yield code that is lengthier and more verbose than non class-based code. However, class-based code is often easier to adapt to different situations. Using classes also makes programs easier to develop on more than one computing platform. For example, if there are differences in how text files are read using C++ on different computers, these differences can be encapsulated in classes and made invisible to programmers who can use the classes without knowing the implementation details. This makes the code more portable. The process of developing code in one computing environment and moving it to another is called **porting** the code. The member functions `Init`, `HasMore`, `Next`, and `Current` together form a programming pattern called an **iterator**. This iterator pattern is used to loop over values stored somewhere, such as in an `ifstream` variable. By using the same names in other iterating contexts we may be able to develop correct code more quickly. Using the same names also lets us use programming tools developed for iterators.

We have focused on how to use classes rather than on how to design classes. In general, designing classes and programs is a difficult task. One design rule that helps is based on building new designs on proven designs. This is especially true when a design pattern can be reused.

Program Tip 6.11: A pattern is a solution to a problem in a context. In the case of the `WordStreamIterator` class, the problem is accessing the strings in a stream many times within the same program. The class hides the details of the stream functions and lets us concentrate on accessing a sequence of strings rather than on details of how to re-read a stream.

The class declaration for`WordStreamIterator` is found in *worditer.h*, Program G.6 in Howto G. We won't look at the implementation in *worditer.cpp*, but the code is provided for use with this book. You need to understand how to use the class, but to use the class you don't need to understand the implementation.

In the case of a `WordStreamIterator` object, you should know that the member function `WordStreamIterator::HasMore` will return true if there is more input to be read. When `HasMore` returns false, accessing the current word using the `Current` member function is not a valid operation.

The constructor `WordStreamIterator::WordStreamIterator` leaves the object `iter` in a state where accessing the current word is *not* valid. In this state the function `HasMore` will return false. You must call the function `Init` to access words. The call `iter.Init` reads the first word from the input stream and updates the internal state accordingly.

Pause to Reflect

6.19 What statements can be added to *countw2.cpp*, Program 6.8 so that three values are tracked: the number of small (1–3 letter) words, the number of medium (4–7 letter) words, and the number of large (8 or more letter) words.

6.20 What is the function header for a function that accepts a file name and returns the number of small, medium, and large words as defined in the previous exercise (the function has four parameters, the file name is passed into the function, the other values are returned from the function via parameters.)

6.21 What is the value of `1/2` and why is it different from `1/2.0`? What is the value of `20/static_cast<double>(6)`?

6.22 Write code that prompts for two file names, one for input and one for output. Every word in the input file should be written to the output file, one word per line.

6.23 The statement below reads one string and two ints.

```
string s; int m,n;
cin >> s >> m >>n;
```

The statement succeeds in reading three values if the user types `"hello 12 3"` (without the quotes.) What is the value of `n` in this case? If the user types `"hello 1 2 3 4 5"` the statement succeeds (what is the value of `n`?), but if it is executed immediately again, the value of `n` will be 5. Why, and what is the value of `s` after the statement executes again.

6.24 Suppose a text file named `"quiz.dat"` stores student information, one student per line. Each student's first name, last name, and five test scores are on one line (there are no spaces other than between names and scores.)

```
owen astrachan 70 85 80 70 60
josh astrachan 100 100 95 97 93
gail chapman 88 90 92 94 96
susan rodger 91 91 91 55 91
```

Write a loop to read information for all students and to print the average for each student.

6.25 Why can't the `WordStreamIterator` class be used to solve the problem in the previous exercise (knowing what you've learned so far, there is a way to solve the problem using the function `atoi` from `"strutils.h"`, see Howto G.)

6.4 Finding Extreme Values

We ascribe beauty to that which is simple,
which has no superfluous parts;
which exactly answers its end,
which stands related to all things,
which is the mean of many extremes.
Ralph Waldo Emerson
The Conduct of Life

The maximum and minimum values in a set of data are sometimes called **extreme** values. In this section we'll examine code to find the maximum (or minimum) values in a set of data. For example, instead of just counting the number of words in Shakespeare's *Hamlet* we might like to know what word occurs most often. Using the `WordStreamIterator` class we can do so, although the program is very slow. Later in the chapter I will introduce a mechanism for speeding up the program. As a preliminary step, we'll look at *mindata.cpp*, Program 6.9, designed to find the minimum of all numbers in the standard input stream.

The `if` statement compares the value of the number just read with the current minimum value. A new value is assigned to `minimum` only when the newly read number is smaller. However, Program 6.9 does not always work as intended, as you may be able to see from the second run of the program. Using the second run, you may reason about a mistake in the program: the variable `minimum` is initialized incorrectly. You may wonder about what happens when the string `"apple"` is entered when a number is expected. As you can see from the output, the program only counts four numbers as read in the second run.

The operator `>>` fails when you attempt to extract an integer but enter a noninteger value such as `"apple"`. The operator `>>` fails in the following situations:

1. There are no more data to be read (extracted) from the input stream; (i.e., all input has been processed).
2. There was never any data because the input stream was not bound to any file. This can happen when an `ifstream` object is constructed and initialized with the name of a file that doesn't exist or isn't accessible.
3. The data to be read are not of the correct type, (e.g., attempting to read the string `"apple"` into an integer variable).

Program 6.9 mindata.cpp

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

   // determine minimum of all numbers in input stream

int main()
{
    int numNums = 0;                  // initially, no numbers
    int minimum = 0;                  // tentative minimal value is 0

    int number;
    while (cin >> number)
    {   numNums++;
        if (number < minimum)
        {   minimum = number;
        }
    }
    cout << "number of numbers = " << numNums << endl;
    cout << "minimal number is " << minimum << endl;

    return 0;
}
```

mindata.cpp

OUTPUT

```
prompt> mindata
−3 5 2 135 −33 14 3
199 257 −582 9392 78
number of numbers = 19
minimal number is −582

prompt> mindata
20 30 40 50 apple 60 70
number of numbers = 4
minimal number is 0
```

There are two methods for fixing the program so that it will work regardless of what integer values are entered; currently the test in the `if` statement of *mindata.cpp* will never be true if the user enters only positive numbers.

- Initialize `minimum` to "infinity" so the first time the `if` statement is executed the entered value will be less than `minimum`.
- Initialize `minimum` to the first value entered on the input stream.

We'll elaborate on each of these approaches in turn.

6.4.1 Largest/Smallest Values

To implement the first approach we'll take advantage of the existence of a largest integer in C++. Since integers (and other types such as `double`) are stored in a computer using a fixed amount of memory, there cannot be arbitrarily large or small values. In the standard system file `<climits>` (or `limits.h`), several useful constants are defined:

```
  INT_MAX    INT_MIN     LONG_MAX     LONG_MIN
```

These constants represent, respectively, the largest and smallest `int` values and the largest and smallest `long` values. We can now initialize `minimum` from Program 6.9 as follows (assuming `<climits>` is included.)

```
int main()
{
    int numNums = 0;       // initially, no numbers
    int minimum = INT_MAX; // all values less than this
}
```

The program finds the correct minimum because the `if` test evaluates to true the first time, since any integer value is less than or equal to `INT_MAX`. However, if only values of INT_MAX are encountered, the test of the if statement will never be true. In this case the program still finds the correct minimum of INT_MAX.

Similar constants exist for double values; these are accessed by including <cfloat> (or <float.h>). The largest and smallest double values are represented by the constants DBL_MIN and DBL_MAX, respectively.

6.4.2 Initialization: Another Fence Post Problem

Implementing the second approach to the extreme value problem—using the first item read as the initial value for minimum—is a typical fence post problem. An item must be read before the loop to initialize minimum. Items must continue to be read within the loop. In developing code for this approach, we must decide what to do if no items are entered. What is the minimum of no values? Perhaps the safest approach is to print an error message as shown in Program 6.10.

Program 6.10 mindata2.cpp

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

   // determine minimum of all numbers in input stream
   // illustrates fencepost problem: first item is minimum initialization

int main()
{
    int numNums = 0;                  // initially, no numbers
    int minimum;                      // smallest number entered
    int number;                       // user entered number

    if (cin >> number)                // read in first value
    {   minimum = number;             // to initialize minimum
        numNums++;
    }
    while (cin >> number)             // read in any remaining values
    {   numNums++;
        if (number < minimum)
        {   minimum = number;
        }
    }
    if (numNums > 0)
    {   cout << "number of numbers = " << numNums << endl;
        cout << "minimal number is " << minimum << endl;
    }
    else
    {   cout << "no numbers entered, no minimum found" << endl;
    }
    return 0;
}
```

mindata2.cpp

The input statement `cin >> number` is the test of the `if` statement. It ensures that a number was read. Another approach to using the first number read as the initial value of `minimum` uses an `if` statement in the body of the `while` loop to differentiate between the first number and all other numbers. The value of `numNums` can be used for this purpose.

```
while (cin >> number)
{   numNums++;
    if (numNums == 1 || number < minimum)
    {   minimum = number;
    }
}
```

Many people prefer the first approach because it avoids an extra check in the body of the `while` loop. The check `numNums == 1` is true only once, but it is checked every time through the loop. In general, you should prefer an approach that does not check a special case over and over when the special case can only occur once. On the other hand, the check in the loop body results in shorter code because there is no need to read an initial value for `minimum`. Since code isn't duplicated (before the loop and in the loop), there is less of a maintenance problem because code won't have to be changed in two places. The extra check in the loop body may result in slightly slower code, but unless you have determined that this is a time-critical part of a program, ease of code maintenance should probably be of greater concern than a very small gain in efficiency. There is no single rule you can use to determine which is the best method. As with many problems the best method depends on the exact nature of the problem.

Program Tip 6.12: The safest approach to solving extreme problems is to use the first value for initialization of all variables that track the extreme (minimum or maximum). If you're finding the minimum or maximum of numeric values represented by `int` or `double`, then constants like `INT_MIN` can be used, but using the first value is always safe.

Pause to Reflect

6.26 If *mindata.cpp*, Program 6.9, is modified so that it reads floating-point numbers (of type `double`) instead of integers, which variables' types change? What other changes are necessary?

6.27 If the largest and smallest in a sequence of `BigInt` values are being determined, what is the appropriate method for initializing the variables tracking the extreme values? (The type `BigInt` was introduced in Section 5.1.3.)

6.28 What happens if each of the following statements is used to calculate the average of the values entered in Program 6.8 Why?

```
cout << "average word length = "
     << (double) sum/numWords << endl;
```

Does the statement below produce different output?

```
cout << "average word length = "
     << (double sum/numWords) << endl;
```

6.29 Write and run a small program to output the largest and smallest integer values on your system.

6.30 Modify *mindata.cpp*, Program 6.9, and *mindata2.cpp*, Program 6.10, to calculate the maximum of all values read.

6.31 Strings can be compared alphabetically (also called *lexicographically*) using the operators < and > so that `"apple" < "bat"` and `"cabinet" > "cabbage"`. What is the function header and body of a function that exhaustively reads input and returns the alphabetically first and last word read?

6.4.3 Word Frequencies

We can use the method of finding extreme values from *mindata.cpp*, Program 6.9, and the `WordStreamIterator` class to find the word that occurs most often in a text file. The idea is to read one word at a time using a `WordStreamIterator` object and to use another iterator to read the entire text file from beginning to end counting how many times the given word occurs. This is shown in Program 6.11. Using nested iterators in this way results in a very slow program, because if there are 2000 words in a file, the file will be read 2000 times. Redundancy occurs because we don't have the programming tools to track whether a word is already counted; thus we may count the number of times *the* occurs more than 100 times.

Program 6.11 maxword.cpp

```
#include <iostream>
#include <string>
using namespace std;
#include "worditer.h"

#include "prompt.h"

// illustrates nested loops using WordStreamIterator class
// to find the word that occurs most often in a file
// Owen Astrachan, 2/13/96, 4/10/99

int main()
{
    int maxOccurs = 0;
    int wordCount = 0;
    string word,maxWord;
    string filename = PromptString("enter file name: ");
    WordStreamIterator outer,inner;
    outer.Open(filename);         // open two iterators
    inner.Open(filename);

    for(outer.Init(); outer.HasMore(); outer.Next())
    {   wordCount++;
        word = outer.Current();       // current word for comparison
        int count = 0;                // count # occurrences

        for(inner.Init(); inner.HasMore(); inner.Next())
        {   if (inner.Current() == word)   // found another occurrence
            {   count++;
            }
        }
        if (count > maxOccurs)             // maximal so far
        {   maxOccurs = count;
            maxWord = word;
        }
        if (wordCount % 100 == 0)          // update "progress bar"
        {   cout << "..";
            if (wordCount % 1000 == 0) cout << endl;
        }
    }
    cout << endl << "word \"" << maxWord << "\" occurs "
         << maxOccurs << " times" << endl;
    return 0;
}
```

maxword.cpp

OUTPUT

```
prompt> maxword
enter file name: poe.txt
....................
....................
......
word "the" occurs 149 times
```

The outer loop, using the iterator `outer`, processes each word from a text file one at a time. The inner loop reads the entire file, counting how many times `word` occurs in the file. Since each `WordStreamIterator` object has its own state, the iterator `outer` keeps track of where it is in the input stream, even as the iterator `inner` reads the entire stream from beginning to end.

Pause to Reflect

6.32 According to *countw2.cpp*, Program 6.8, *Hamlet* has 31,956 words and an average word length of 4.362 characters. If a computer can read 200,000 characters per second, provide a rough but reasoned estimate of how long it will take *maxword.cpp* to find the word in *Hamlet* that occurs most often.

6.33 Suppose that the code in `main` from *mindata.cpp*, Program 6.9, is moved to a function named `ReadNums` so that the new body of `main` is

```
{
  ....
  ReadNums(numNums,minimum);
  cout << "number of numbers = " << numNums << endl;
  cout << "minimal number is " << minimum << endl;
}
```

What is the function header and body of `ReadNums`? How would the function header and body change if only the average of the numbers read is to be returned?

6.34 How can you modify *maxword.cpp* so that instead of printing two dots every 100 words as it does currently, it prints a percentage of how much it has processed, like this:

```
10%...20%...30%...40%...50%...60%...70%...80%...90%...
  word "the" occurs 149 times
```

(Hint: count the total words first.)

6.4.4 Using the `CTimer` class

Program 6.11 is slow, but how slow is it? User-interface studies show that people are more willing to put up with slow programs, or slow internet connections, if feedback is provided about how much time a program or download is expected to take[13]. Using the class `CTimer`, whose interface is given in *ctimer.h* as Program G.5 in Howto G, allows us to provide a user with feedback. The `CTimer` class also allows us to time how long code fragments or functions take to execute which, in turn, allows us to evaluate algorithm and program efficiency.

Program 6.12 shows how `CTimer` can be used to time loop-execution time. The program shows all but one of the `CTimer` member functions. The function `CTimer::Reset` resets a `CTimer`'s internal stopwatch to zero. The precision or **granularity** of the timing done by `CTimer` may depend on the machine on which it's run. On many machines, the class "ticks" in increments of one-sixtieth of a second[14].

[13]Have you ever watched the progress bar in an internet browser as it updates the time to complete a download?

[14]The tick-value is found as the constant `CLOCKS_PER_SEC` in the header file `<ctime>` or `time.h`.

Program 6.12 usetimer.cpp

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

#include "ctimer.h"
#include "prompt.h"

// illustrate CTimer class and loop timings

int main()
{
    int inner = PromptRange("# inner iterations x 10,000 ",1,10000);
    int outer = PromptRange("# outer iterations",1,20);

    long j,k;
    CTimer timer;

    for(j=0; j < outer; j++)
    {   timer.Start();
        for(k=0; k < inner*10000L; k++)
        {
            // nothing done here
        }
        timer.Stop();
        cout << j << "\t" << timer.ElapsedTime() << endl;
    }
    cout << "——-" << endl;
    cout << "total = " << timer.CumulativeTime() << "\t"
         << inner*outer*10000L << " iterations "<< endl;

    return 0;
}
```

usetimer.cpp

OUTPUT

```
run on a PII, 300 Mhz machine running Windows NT
prompt> usetimer
#inner iterations x 10,000  between 1 and 10000: 10000
# outer iterations between 1 and 20: 3
0       2.364
1       2.353
2       2.373
-------
total = 7.090  300000000 iterations

run on a P100 machine running Linux
prompt> usetimer
#inner iterations x 10,000  between 1 and 10000: 10000
# outer iterations between 1 and 20: 3
0       17.11
1       17.11
2       17.12
-------
total = 51.34  300000000 iterations
```

Using the `CTimer` class we can add code to Program 6.11 to give the user an estimate of how long the program will take to run. The modified program is `maxword2.cpp`. The entire program is accessible online, or with the code that comes with this book. The timing portions of the code are shown as Program 6.13 after the output.

OUTPUT

```
prompt> maxword2
enter file name: poe.txt
2.314   of 46.5

timing data removed

46.197  of 46.5
48.5    of 46.5
50.804  of 46.5
53.107  of 46.5
word "the" occurs 149 times
```

Program 6.13 maxword2time.cpp

```
CTimer timer;
timer.Start();
for(outer.Init(); outer.HasMore(); outer.Next())
{  wordCount++;
}
timer.Stop();
double totalTime = timer.ElapsedTime()*wordCount;
wordCount = 0;
timer.Reset();

for(outer.Init(); outer.HasMore(); outer.Next())
{   word = outer.Current();       // current word for comparison
    wordCount++;

    int count = 0;                // count # occurrences
    timer.Start();
    for(inner.Init(); inner.HasMore(); inner.Next())
    {   if (inner.Current() == word)   // found another occurrence
        {   count++;
        }
    }
    if (count > maxOccurs)             // maximal so far
    {   maxOccurs = count;
        maxWord = word;
    }
    if (count > maxOccurs)             // maximal so far
    {   maxOccurs = count;
        maxWord = word;
    }
    timer.Stop();
    if (wordCount % 100 == 0)
    {   cout  << timer.CumulativeTime() << "\tof " << totalTime << endl;
    }
}
```

maxword2time.cpp

As you can see in the output, the time-to-completion is underestimated by the program. The loop that calibrates the time-to-completion reads all the words, but does not compare words. The string comparisons in the inner nested loop take time that's not accounted for in the time-to-completion calibrating loop.

6.5 Case Study: Iteration and String Sets

We'll take one step toward speeding up *maxword.cpp*, Program 6.11 by studying the class `StringSet` and its associated iterator class `StringSetIterator`.

Sets used in programming are based on the mathematical notion of set: a collection of elements with no duplicates. Examples include sets of integers: $\{1, 3, 2, 4\}$, sets of shapes: $\{\triangle, \triangledown, \bowtie, \bigcirc\}$, and sets of spicy spices `{"paprika", "cayenne",` "chili"}. The collection {1, 3, 2, 3, 4, 3, 1} is not a set because it contains duplicate elements.

Program 6.14 illustrates how to program using the class `StringSet` and the associated class `StringSetIterator`. The member functions of `StringSetIterator` have the same names as those of the class `WordStreamIterator`.

Program 6.14 setdemo.cpp

```
#include <iostream>
using namespace std;
#include "stringset.h"

// demonstrate string set use

int main()
{
    StringSet sset;
    sset.insert("watermelon");
    sset.insert("apple");
    sset.insert("banana");
    sset.insert("orange");
    sset.insert("banana");
    sset.insert("cherry");
    sset.insert("guava");
    sset.insert("banana");
    sset.insert("cherry");

    cout << "set size = " << sset.size() << endl;

    StringSetIterator it(sset);
    for(it.Init(); it.HasMore(); it.Next())
    {   cout << it.Current() << endl;
    }
    return 0;
}
```

setdemo.cpp

OUTPUT

```
prompt> setdemo
set size = 6
apple
banana
cherry
guava
orange
watermelon
```

Client programs can call `StringSet::insert` hundreds of times with the same argument, but only the first call succeeds in inserting a new element into the set. Other `StringSet` member functions include `StringSet::clear` which removes all elements from a set and `StringSet::erase` which removes one element, if it is present; that is, `sset.erase("apple")` decreases the size of the set used in *setdemo.cpp*, Program 6.14 by removing ("apple"). The header file *stringset.h* is Program G.7 in Howto G.

6.5.1 Iterators and the `strutils.h` Library

Program 6.15, *setdemo2.cpp* shows two different kinds of iterators used in the same program. The program reads a file and stores all the words in a `StringSet` object. The words are first converted to lowercase and all leading and trailing punctuation is removed using functions `ToLower` and `StripPunc` from `"strutils.h"` (for details see Program G.8 in Howto G.) This reduces the number of different words in many of the English text files used in this book. For example, the line below occurs in *The Cask of Amontillado*, used in this book as the file *poe.txt*.

```
''Yes, yes,'' I said; ''yes, yes.''
```

If we don't strip punctuation and convert to lowercase this line contains four occurrences of the word "yes" (each word is shown surrounded by double quotes "" that aren't part of the word as read by the program): "`''Yes,`", "`yes,''`", "`''yes,`", and "`yes.''`".

6.5.2 The Type `ofstream`

Program 6.15, *setdemo2.cpp* shows how to print to a text file of type `ofstream`. Opening an `ofstream` variable uses the same syntax as opening an `ifstream` variable. Writing to an `ifstream` uses the same syntax as writing to `cout` as shown by the function `Print` which accepts either `cout` or the `ifstream` variable `output` as arguments. The reason that both streams can be arguments is that the parameter has type **ostream**. We'll explore why both `cout` and an `ifstream` object have the type `ostream` in a later chapter.[15]

Program Tip 6.13: When passing streams as parameters, use `ostream` for output streams and `istream` for input streams. Using the most general kind of stream as a parameter ensures that you'll be able to pass many different kinds of streams as arguments.

Although we've only studied `cin` and `ifstream` for input, and `cout` and `ofstream` for output, you'll encounter other kinds of streams later in this book and your study of C++.

[15]This works because of inheritance, but you do not need to understand inheritance conceptually, or how it is implemented in C++, to use streams.

Program Tip 6.14: Streams must be passed by reference. The compiler may not complain if you pass a stream by value, but your program will not work properly. Streams are almost never const-reference parameters since stream functions invariably change the state of the stream.

Program 6.15 setdemo2.cpp

```
#include <iostream>
#include <fstream>   // for ifstream and ofstream
#include <string>
using namespace std;

#include "worditer.h"
#include "stringset.h"
#include "strutils.h"
#include "prompt.h"

Print(StringSetIterator& ssi, ostream& output)
{
    for(ssi.Init(); ssi.HasMore(); ssi.Next())
    {    output << ssi.Current() << endl;
    }
}

int main()
{
    string filename = PromptString("enter file name: ");
    WordStreamIterator wstream;
    wstream.Open(filename);
    string word;

    StringSet wordset;
    for(wstream.Init(); wstream.HasMore(); wstream.Next())
    {   word = wstream.Current();
        ToLower(word);
        StripPunc(word);
        wordset.insert(word);
    }
    StringSetIterator ssi(wordset);
    Print(ssi,cout);
    cout << "# different words = " << wordset.size() <<  endl;

    filename = PromptString("file for output: ");
    ofstream output(filename.c_str());
    Print(ssi,output);

    return 0;
}
```

setdemo2.cpp

OUTPUT

```
prompt> hamlet.txt
1
1604
a
a'mercy

output words removed

yourself
yourselves
youth
zone
# different words = 4832
file for output: hamwords.dat
```

When the program is run on Shakespeare's *Hamlet* as shown, the file `hamwords.dat` is created and contains the 4,832 different words occurring in *Hamlet*. The words are printed in alphabetical order because of how the `StringSet` class is implemented. Note that words include "1" and "1604" and that these appear before words beginning with "a" because of the character system used in computers in which digits come before letters.

6.5.3 Sets and Word Counting

Using a `StringSet` object greatly speeds up the execution time for *maxword.cpp*, Program 6.11. The original program used nested iterators to find the most frequently occurring word in a file. The modified version below, *maxword3.cpp*, Program 6.16, puts the words in a set, then the outer iterator goes over the set while the inner iterator reads the file each time. The program uses an object of type `CircleStatusBar` to monitor how much time remains as it's reading a file and finding the most frequently occurring word.[16] Three snapshots of the `CircleStatusBar` timing a run using Poe's *The Cask of Amontillado* are shown in Figure 6.3. The program does not use the `"strutils.h"` functions `StripPunc` and `ToLower` that were used in *setdemo2.cpp*, Program 6.15. This is why the number of different words is shown as 1,040 for *maxword3.cpp*, but as 810 for *setdemo2.cpp*.

[16]The `CircleStatusBar` class in `tstatusbar.h` requires the use of the graphics library discussed in Howto H.

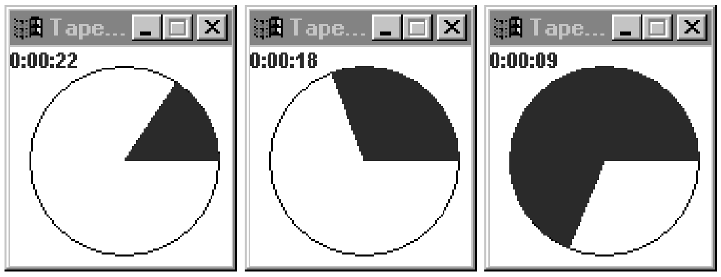


Figure 6.3 Timed output from *maxword3.cpp* using `StatusCircle`, `WordIter`, and `StringSet` classes.

Program 6.16 maxword3.cpp

```
#include <iostream>
#include <string>
using namespace std;
#include "worditer.h"
#include "stringset.h"
#include "prompt.h"
#include "statusbar.h"

// 4/23/99, find most frequently occurring word using stringsets/iterators

int main()
{
    int maxOccurs = 0;
    int wordsRead = 0;
    string word,maxWord;
    StringSet wordSet;
    StatusCircle circle(50);

    string filename = PromptString("enter file name: ");
    WordStreamIterator ws;
    ws.Open(filename);

    for(ws.Init(); ws.HasMore(); ws.Next())
    {    wordSet.insert(ws.Current());
    }
    cout << "read " << wordSet.size() << " different words" << endl;

    StringSetIterator ssi(wordSet);
    for(ssi.Init(); ssi.HasMore(); ssi.Next())
    {   circle.update(wordsRead/double(wordSet.size())*100);
        int count = 0;
        wordsRead++;
        word = ssi.Current();
        for(ws.Init(); ws.HasMore(); ws.Next())
        {   if (ws.Current() == word)
            {   count++;
            }
        }
        if (count > maxOccurs)
        {   maxOccurs = count;
            maxWord = word;
        }
    }
    cout << endl << "word \"" << maxWord << "\" occurs "
         << maxOccurs << " times" << endl;

    return 0;
}
```

maxword3.cpp

O U T P U T

```
enter file name: poe.txt
read 1040 different words
word "the" occurs 149 times
```

If the functions `StripPunc` and `ToLower` are used, the word "the" will occur more than 149 times.

Pause to Reflect

6.35 Write the body of the function below that creates the union of two string sets.

```
void union(const StringSet& lhs, const StringSet& rhs,
           StringSet& result)
// post: result contains elements in either lhs or rhs
```

6.36 Write the body of the function below that creates the intersection of two string sets.

```
void intersect(const StringSet& lhs, const StringSet& rhs,
               StringSet& result)
// post: result contains elements in both lhs and rhs
```

(If you compare the sizes of `lhs` and `rhs` you can make the function more efficient by looping over the smallest set).

6.37 Write a loop that prints all the strings in a set that are still elements of the set if the first character is removed, (e.g., like `"eat"` and `"at"` if both were in the set).

6.38 Write a loop to print all the strings in a set that are "pseudo-palindromes" — different words when written backwards, such as `"stressed"` and `"desserts"` (if both are in the set.)

6.6 Chapter Review

In this chapter we studied how classes were implemented, with an in-depth look at the class `Dice`. Member functions of classes are categorized as constructors, accessor, and mutator functions; private data makes up the state of a class. We studied different modes of parameter/argument passing. We saw how (relatively) simple it is to read and write text files in C++ because of the similarity of file streams to `cin` and `cout`. We saw a pattern of iteration using functions `Init`, `HasMore`, and `Next` used with both streams and with sets of strings. The pattern was used to permit programs to access the elements of a collection without real knowledge of how the collection is implemented. By using the same names for iterator functions, we'll make it easier to understand new iterators when we encounter them. We also studied how to solve extreme problems; such as finding the maximum and minimum in a collection.

Important topics covered include the following:

- Accessor and mutator functions allow a class' state to be examined and changed, respectively.
- Private instance variables are accessible only in member functions, not in client programs.
- Coupling and cohesion are important criteria for evaluating functions, classes, and programs.
- Reference parameters permit values to be returned from functions via parameters. This allows more than one value to be returned. Const reference parameters are used for efficiency and safety.
- Parameters are passed by value (a copy is made) unless an ampersand, `&`, is used for pass by reference. In this case the formal parameter identifier is an alias for the memory associated with the associated function argument.
- A variable is **defined** when storage is allocated. A variable is **declared** if no storage is allocated, but the variable's type is associated with the variable's identifier.
- Parameters for programmer-defined classes are often declared as `const` reference parameters to save time and space while ensuring safety.
- Programs are best designed in an iterative manner, ideally by developing a working program and adding pieces to it so that the program is always functional to some degree. Writing pseudocode first is often a good way of starting the process of program development.
- The extraction operator, `>>`, uses white space to delimit, or separate, one string from another.
- In sentinel loops, the sentinel value is *not* considered part of the data.
- The extraction operator returns a value that can be tested in a loop to see whether the extraction succeeds, so `while (cin » word)` is a standard idiom for reading streams until there is no more data (or until the extraction fails). The stream member function `fail` can be used too.
- Files can be associated with streams using `ifstream` variables. The extraction operator works with these streams. The `ifstream` member function `open` is

used to bind a named disk file to a file stream. An `ofstream` variable is used to associate an output file stream with a named disk file.

- If you enter a nonnumeric value when a numeric value (e.g., an `int` or a `double`) is expected, the extraction will fail and the nonnumeric character remains unprocessed on the input stream.
- Types sometimes need to be cast, or changed, to another type. Casting often causes values to change; that is when casting from a `double` to an `int`, truncation occurs. A new cast operator, `static_cast`, should be used if your compiler supports it.
- Constants for the largest `int` and `double` values are accessible and can be found in the header files `<limits.h>` and `<float.h>`, respectively. The constants defining system extreme values are `INT_MAX`, `INT_MIN`, `LONG_MAX`, `LONG_MIN`, `DBL_MAX`, and `DBL_MIN`.
- Finding extreme (highest and lowest) values is a typical fence post problem. Initializing with the first value is usually a good approach, but sometimes a value of "infinity" is available for initialization (e.g., `INT_MAX`).
- The class `CTimer` can be used to time program segments. The granularity of it's underlying clock may differ among different computers.
- The `WordStreamIterator` class encapsulates file-reading so that the same file can be easily read many times within the same program.
- The `StringSet` class is used to represent sets of strings (no duplicates). An associated class `StringSetIterator` allows access to each value in a set.

6.7 Exercises

6.1 Create a data file in the format

```
firstname lastname testscore
firstname lastname testscore
```

where the first two entries on a line are `string` values and the last entry is an `int` test score in the range 0–100. For example:

```
Owen Astrachan 95
Dave Reed 56
Steve Tate 99
Dave Reed 77
Steve Tate 92
Owen Astrachan 88
Mike Clancy 100
Mike Clancy 95
Dave Reed 47
```

Write a program that prompts for a name and then reads the text file and computes and outputs the average test score for the person whose name is entered. Use the following `while` statement to read entries from an `ifstream` variable `input`.

```
string first, last;
int score;
while (input >> first >> last >> score)
{
   // read one line, process it
}
```

6.2 Implement a class similar to the class `Dice` but like the child's game *Magic 8-Ball*. Call the class `Fortune`. A `Fortune` object should represent a many-sided fortune-teller. You can choose six sides, or eight sides, or even twenty sides like the "real" Magic 8-ball, but the number of sides is fixed. It is not specified at construction as it is for the class `Dice`. Each time the object is "rolled," (or shaken, or asked to tell the future) a different fortune is returned. For example, consider the code below and the sample output.

```
#include "fortune.h"

int main()
{
   int rolls = PromptRange("# of fortunes ", 1, 10);
   Fortune f;
   int k;
   for(k=0; k < rolls; k++)
   {   cout << f.Shake() << endl;
   }
   return 0;
}
```

OUTPUT

```
prompt> testfortune
# of fortunes 4
Reply Hazy, Try Again
My Reply is No
Concentrate and Ask Again
Signs Point to Yes
```

Be creative with your fortunes, and develop a program that illustrates all the member functions of your class. For an added challenge, make the class behave so that after it has told more than 100 fortunes it breaks and tells the same one every time.

6.3 Create a class `WordDice` similar to the class from the previous exercise, but with a constructor that takes a file name and reads strings from the specified file. The strings can be stored in a `StringSet` instance variable. One of the strings is returned at random each time the function `WordDice::Roll` is called.
For example, the code segment below might print any one of seven different colors if

the data file `"spectrum.dat"` contains the lines:

```
red orange yellow
green blue indigo violet
```

The code fragment using this file follows:

```
WordDice wd("spectrum.dat");
cout << wd.Roll() << endl;
cout << wd.Roll() << endl;
cout << wd.Roll() << endl;
```

OUTPUT

```
prompt> testwordie
red
green
yellow
```

You should test the program with different data files. For an added challenge, test the program by rolling a `WordDice` object as many times as needed until all the different words are "rolled." Print the number of rolls needed to generate all the possible words.

6.4 Create a data file where each line has the format

```
item size retail-price-sold-for
```

For example, a file might contain information from a clothing store (prices aren't meant to be realistic):

```
coat small 110.00
coat large 130.00
shirt medium 22.00
dress tiny 49.00
pants large 78.50
coat large 140.00
```

Write a program that prompts the user for the name of a data file and then prompts for the name of an item, the size of the item, and the wholesale price paid for the item. The program should generate several statistics as output:

- Average retail price paid for the item
- Total profit made for selling the item
- Percentage of all sales accounted for by the specified item and size, both by price and by units sold
- Percentage of all item sales, where the item is the same as specified, both by price and by units sold

For example, in the data file above, if the wholesale price of a large coat is $100.00, then the output should include:

- Average retail price for large coats is $135.00.
- Total profit is $70.00.
- Percentage of all sales is one-third (2 out of 6).
- Percentage of all coat sales is two-thirds (2 out of 3).

6.5 Write a program based on the word game *Madlibs.* The input to Madlibs is a vignette or brief story, with words left out. Players are asked to fill in missing words by prompting for adjectives, nouns, verbs, and so on. When these words are used to replace the missing words, the resulting story is often funny when read aloud.
In the computerized version of the game, the input will be a text file with certain words annotated by enclosing the words in brackets. These enclosed words will be replaced after prompting the user for a replacement. All words are written to another text file (use an `ofstream` variable).[17] Since words will be read and written one at a time, you'll need to keep track of the number of characters written to the output file so that you can use an `endl` to finish off, or flush, lines in the output file. For example, in the sample run below, output lines are flushed using `endl` after writing 50 characters (the number of characters can be accumulated using the `string` member function `length`.)
The output below is based on an excerpt from *Romeo and Juliet* annotated for the game. Punctuation must be separated from words that are annotated so that the brackets can be recognized (using `substr`). Alternatively, you could search for brackets using `find` and maintain the punctuation.
The text file `mad.in` is

```
But soft! What [noun] through yonder window [verb] ?
It is the [noun] , and [name] is the [noun] !
Arise, [adjective] [noun] , and [verb] the [adjective]
[noun] , Who is already [adjective] and
[another_adjective] with [emotion]
```

The output is shown on the next page. Because we don't have the programming tools to read lines from files, the lines in the output aren't the same as the lines in the input. In the following run, the output file created is reread to show the user the results.

[17]You may need to call the member function `close` on the `ofstream` object. If the output file is truncated so that not all data is written, call `close` when the program has finished writing to the stream.

OUTPUT

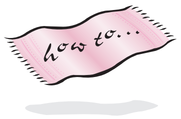
```
prompt> madlibs enter madlibs file: mad.in
name for output file: mad.out
enter noun: fish
enter verb: jumps
enter noun: computer
enter name: Susan
enter noun: porcupine
enter adjective: wonderful
enter noun: book
enter verb: run
enter adjective: lazy
enter noun: carwash
enter adjective: creative
enter another_adjective: pretty
enter emotion: anger

But soft! What fish through yonder window jumps ?
It is the computer, and Susan is the porcupine !
Arise, wonderful book , and run the lazy carwash ,
Who is already creative and pretty with anger
```

6.6 Write a program to compute the average of all the numbers stored in a text file. Assume the numbers are integers representing test scores, for example:

```
70 85 90
92 57 100 88
87 98
```

First use the extraction operator, `>>`. Then use a `WordStreamIterator` object. Since `WordStreamIterator::Current` returns a string, you'll need to convert the string to the corresponding integer; that is, the `string` `"123"` should be converted to the `int` 123. The function `atoi` in `"strutils.h"` in Howto G will convert the string.



```
int atoi(string s)
// pre: s represents an int, that is "123", "-457", etc.
// post: returns int equivalent of s
//       if s isn't properly formatted (that is "12a3")
//       then 0 (zero) is returned
```

6.7 The **standard deviation** of a group of numbers is a statistical measure of how much the numbers spread out from the average (the average is also called the *mean*). A low standard deviation means most of the numbers are near the mean. If numbers are denoted as $(x_1, x_2, x_3, \ldots, x_n)$, then the mean is denoted as $\overline{x}$. The standard deviation is the square root of the **variance**. (The standard deviation is usually denoted by the Greek letter sigma, σ, and the variance is denoted by σ^2.

The mathematical formula for calculating the variance is

$$\begin{aligned}\sigma^2 &= \frac{1}{n-1}[(x_1-\bar{x})^2+(x_2-\bar{x})^2+\cdots(x_n-\bar{x})^2] \\ &= \frac{1}{n-1}[\sum_{i=1}^{n}(x_i-\bar{x})^2]\end{aligned}$$

Using algebra this formula can be rearranged to yield the formula

$$\sigma^2 = \frac{1}{n-1}[\sum_{i=1}^{n}x_i^2 - \frac{1}{n}(\sum_{i=1}^{n}x_i)^2] \qquad (6.1)$$

This formula does not involve the mean, so it can be computed with a single pass over all the data rather than two passes (one to calculate the mean, the other to calculate the variance).
Write a program to compute the variance and standard deviation using both formulae. Although these formulae are mathematically equivalent, they often yield very different answers because of errors introduced by floating-point computations. Use the technique from the previous exercise so that you can read a file of data twice using a `WordStreamIterator` object. If the data consist of floating-point values instead of integers, you can use the function `atof` to convert a string to the `double` value it represents, such as `atof("123.075") == 123.075`.

6.8 The **hailstone** sequence, sometimes called the **3**n+**1** sequence, is defined by a function $f(n)$:

$$f(n) = \begin{cases} n/2 & \text{if } n \text{ is even} \\ 3\times n+1 & \text{otherwise, if } n \text{ is odd}\end{cases} \qquad (6.2)$$

We can use the value computed by f as the argument of f as shown below; the successive values of n form the hailstone sequence.[18]

```
while (n != 1)
{   n = f(n);
}
```

Although it is conjectured that this loop always terminates, no one has been able to prove it. However, it has been verified by computer for an enormous range of numbers. Several sequences are shown below with the initial value of n on the left.

```
7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1

11 34 17 52 26 13 40 20 10 5 16 8 4 2 1

22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1

14 7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
```

[18]It's called a *hailstone sequence* because the numbers go up and down, mimicking the process that forms hail.

```
8 4 2 1

9 28 14 7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
```

Write a program to find the value of n that yields the longest sequence. Prompt the user for two numbers, and limit the search for n to all values between the two numbers.

6.9 Use the `CTimer` class test two methods for computing powers outlined in Section 5.1.7. The first method outlined there makes n multiplications to compute x^n; the second method makes roughly $\log_2(n)$ multiplications, that is, 10 multiplications to compute x^{1024} (here x is a `double` value but n is an `int`.)
Write two functions, with different names but the same parameter lists, for computing x^n based on the two methods. Call these functions thousands of times each with different values of n. For example, you might calculate 3.0^{50}, 3.0^{100}, 3.0^{150} and so on. You'll need to do several calculations for a fixed n to make a `CTimer` object register. Plot the values with values of n on the x-axis and time (in seconds) on the y-axis. If you have access to a spreadsheet program you can make the plots automatically by writing the data to an output file.
You should also compare the time required by these two methods, with the time using the function `pow` from `<cmath>`. Finally, you should test both methods of exponentiation using `BigInt` values rather than `double` values for the base (the exponent can still be an integer.) You should try to explain the timings you observe with `BigInt` values which should be different from the timings observed for `double` values.

6.10 Data files for several of Shakespeare's plays are available on the web pages associated with this book (and may be included in a CD you can get with the book's programs.) Write a program that reads the words from at least five different plays, putting the words from each play in a `StringSet` object. You should find the words that are in the intersection of all the plays. Finding the intersection may take a while, so test the program with small data files before trying Shakespeare's plays.
After you've found the words in common to all five plays (or more plays) find the top ten most frequently occurring of these words. There are many ways to do this. One method is to find the most frequently occurring word using code from Program 6.16, *maxword3.cpp*. After this word is found, remove it from the set of common words and repeat the process. You can use this method to rank order (most frequent to least frequent) all the words in common to the plays, but this will take a long time using the `WordStreamIterator` class.

6.11 Do the last exercise, but rather than reading a file of words many times (e.g., once for each word in the list of common words) adopt a different approach. First read all the words from a file into a list, using the class `StringList` from *clist.h*. Program 6.17, *listcount.cpp* shows how `StringList` is used. The only function that's needed other than iterating functions is the function `cons` that attaches an element to the front of a list and returns the new list (the old list is not changed).

OUTPUT

```
prompt> hamlet.txt

first 31,949 words removed

DENMARK
OF
PRINCE
HAMLET,
OF
TRAGEDY
THE
1604
31957 words read
```

Program 6.17 listcount.cpp

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

#include "clist.h"
#include "prompt.h"

int main()
{
    string filename = PromptString("enter filename ");
    ifstream input(filename.c_str());
    string word;

    StringList slist;
    while (input >> word)
    {   slist = cons(word,slist);
    }

    StringListIterator it(slist);
    for(it.Init(); it.HasMore(); it.Next())
    {   cout << it.Current() << endl;
    }
    cout << slist.Size() << " words read " << endl;

    return 0;
}
```

listcount.cpp

Since new words are added to the front using `cons`, the words are stored in the list so that the first word read is the last word in the list. Using the class `StringList` can make string processing programs faster that using the class `WordStreamIterator`

because strings are read from memory rather than from disk.
For example, on my 300 MHz Pentium, using *maxword.cpp*, Program 6.11 takes approximately 53 seconds to process `poe.txt`. Using *maxword3.cpp*, Program 6.16 takes approximately 28 seconds. Replacing the inner `WordStreamIterator` by a `StringListIterator` reduces the time to 3.4 seconds because memory is so much faster than disk.