Class Interfaces, Design, and Implementation 7

There is no single development, in either technology or management technique, which by itself promises even one order-of-magnitude improvement within a decade in productivity, in reliability, in simplicity.
Fred P. Brooks
No Silver Bullet — Essence and Accident in Software Engineering

One tenet of object-oriented programming is that a library of well-designed classes makes it easier to design and write programs. The acronym *COTS*, for *commercial, off-the shelf* software, is often used to identify the process of reusing (commercial) libraries of classes and code. Classes are often easier to reuse in programs than non-class code. For example, we've used the `Dice` and `RandGen` classes and the functions from `"prompt.h"` in many programs. In almost every program we've used `cout` and `cin`, objects that are part of the hierarchy of stream classes. As programmers and designers, we need to be familiar with what classes are available and with patterns of design that we can use. Reusing concepts is often as important as reusing code.

In this chapter we'll discuss the design and implementation of a class-based program for administering on-line quizzes. We'll see that careful planning makes it possible to reuse the same class interface so that different kinds of quizzes can be given. We'll study another example of class interface reuse in a program that simulates random walks in one- and two-dimensions.

7.1 Designing Classes: From Requirements to Implementation

Choosing classes and member functions is a difficult design process. It's very hard to choose the right number of classes with the right member functions so that the classes are cohesive, loosely coupled, easy to modify, and yield an elegant and correctly working program. Many programmers and computer scientists are good algorithmic thinkers but bad class designers, and vice versa. You shouldn't expect to become an accomplished designer early in your studies, but you can build expertise by studying other designs, and by modifying existing designs before creating your own. In this way you'll learn as an apprentice learns any craft.[1]

[1]Programming is both an art and a science. To some, it's only a science or only an art/craft. In my view there are elements of both in becoming an accomplished programmer. You must understand science and mathematics, but good design is not solely a scientific enterprise.

7.1.1 Requirements

The **requirements** of a problem or programming task are the constraints and demands asked by the person or group requesting a programming solution to a problem. As a designer/programmer your task in determining requirements is to interact with the user (here user means both the person using the program and the person hiring you) to solicit information and feedback about how the program will be used, what it must accomplish, and how it interacts with other programs and users. In this book and in most early courses, the requirements of a problem are often spelled out explicitly and in detail. However, sometimes you must infer requirements or make a best guess (since you don't have a real user/software client with whom to interact.)

The **specification** of the quiz problem from Section 6.2.2 is reproduced below. From the specification you may be able to infer the requirements. We'll use the specification as a list of requirements and move toward designing classes.

> 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.

Thinking about this specification leads to the following requirements (in no particular order).

1. More than one kind of question can be used in a quiz.
2. Several students can take a quiz sharing a keyboard.
3. Students may be allowed more than one chance to answer a question.

With a real client you would probably get the chance to ask questions about the requirements. Should a score be reported as in Programs 6.2 and 6.3? Should the scores be automatically recorded in a file? Should the user have the choice of what kind of quiz to take? We'll go forward with the requirements we've extracted from the problem specification. We'll try to design a program that permits unanticipated demands (features?) to be incorporated.

As we develop classes we'll keep the examples simple and won't go deeply into all the issues that arise during design. Our goal here is to see the process simply, glossing over many details but giving a real picture of the design process. In later chapters and future courses you'll delve more deeply into problems and issues of designing classes.

7.1.2 Nouns as Classes

The nouns in a specification are usually good candidates for classes. In the specification above the nouns that seem important include:

> quiz, question, problem, student, computer, keyboard, chance

Our program doesn't need to deal with the nouns computer and keyboard, so we'll use the other nouns as candidates for classes. As you become more experienced, you'll develop a feel for separating important nouns/classes from less important ones. You'll learn to identify some candidate class nouns as synonyms for others. For this quiz program we'll develop three classes: quiz, question, and student. A question object will represent a kind of question factory that can generate new problems. For example, an arithmetic question class might generate problems like "what is $2 + 2$?" or "what is 3×7?" On the other hand, an English literature question class might generate problems like "Who wrote *Charlotte's Web*?" or "In what work does the character Holden Caulfield appear?". As you'll see, a problem will be a part of the question class rather than a separate class.

7.1.3 Verbs as Member Functions (Methods)

The first step in designing identified classes is determining class behavior and responsibility. A class's public member functions determine its **behavior**. In some object-oriented languages member functions are called **methods**; I'll often use the terms member function and method interchangeably. Sometimes you may think at first that a method belongs to one class, but during the design process it will seem better to place it in another class. There isn't usually one right way to design classes or a program. The **responsibilities** of a class are the methods associated with the class and the interactions between classes. Sometimes candidate class methods can be found as verbs in a specification. Often, however, you'll need to anticipate how a program and classes are used to find methods.

7.1.4 Finding Verbs Using Scenarios

It's not always clear what member functions are needed. Sometimes creating **scenarios** of how a program works helps determine class behavior and responsibility. A scenario is a description, almost like a dialog, between the user and the program or between classes in a program.

In the quiz example scenarios could include:

- Two students sit at a keyboard, each is asked to enter her name, then a quiz is given and students alternate providing answers to questions.
- When a quiz is given, the student determines the number of questions that will be asked before the quiz starts. If two people are taking a quiz together, both are asked the same number of questions.
- Students have two chances to respond to a question. A simple "correct" or "incorrect" is given as feedback to each student response. If a student doesn't type a correct response, the correct answer is given.
- At the end of a quiz, each student taking the quiz is given a score.

Some verbs from these scenarios follow (long, descriptive names are chosen to make the verbs more clear).

EnterName, ChooseNumberOfQuestions, ChooseKindOfQuestion, RespondToQuestion, GetCorrectAnswer, GetScore, AskQuestion, ProvideFeedback

Which of these verbs goes with which class? In assigning responsibilities we'll need to return to the scenarios to see how the classes interact in a program. At this point we're concentrating only on public behavior of the classes, not on private state or implementation of the classes.

Program Tip 7.1: Concentrate on behavior rather than on state in initial class design. You can change how a class is implemented without affecting client programs that use the class, but you cannot change the member functions (e.g., the parameters used) without affecting client programs.

Client programs depend on the **interface** provided in a header file. If the interface changes, client programs must change too. Client programs should not rely on how a class is implemented. By writing code that conforms to an interface, rather than to an implementation, changes in client code will be minimized when the implementation changes.[2]

Pause to Reflect

7.1 The method `Dice::Roll` in *dice.cpp*, Program 6.1 uses a local `RandGen` variable to generate simulated random dice rolls. If the `RandGen` class is changed, does a client program like *roll.cpp*, Program 5.11 change? Why?

7.2 What are the behaviors of the class `CTimer` declared in the header file *ctimer.h*, Program G.5 and used in the client code *usetimer.cpp*, Program 6.12?

7.3 Write a specification for a class that simulates a coin. "Tossing" the coin results in either heads or tails.

7.4 Write a specification and requirements for a program to help a library with overdue items (libraries typically loan more than books). Make up whatever you don't know about libraries, but try to keep things realistic. Develop some scenarios for the program.

7.5 Suppose you're given an assignment to write a program to simulate the gambling game roulette using a computer (see Exercise 9 at the end of this chapter for an explanation of the game). Write a list of requirements for the game; candidate classes drawn from nouns used in your description; potential methods; and scenarios for playing the game.

[2]Client programs may depend indirectly on an implementation, that is, on how fast a class executes a certain method. Changes in class performance may not affect the correctness of a client program, but the client program will be affected.

Mary Shaw *(b. 19??)*

Mary Shaw is Professor of Computer Science at Carnegie Mellon University. Her research interests are in the area of software engineering, a subfield of computer science concerned with developing software using well-defined tools and techniques. In [EL94] Shaw says this about software engineering:

Science often grows hand-in-hand with related engineering. Initially, we solve problems any way we can. Gradually a small set of effective techniques enters the folklore and passes from one person to another. Eventually the best are recognized, codified, taught, perhaps named. Better understanding yields theories to explain the techniques, to predict or analyze results, and to provide a base for systematic extension. This improves practice through better operational guidance and tools that automate details. The software developer, thus freed from certain kinds of detail, can tackle bigger, more complex problems.

In discussing her current research interests, Shaw combines the themes of both language and architecture. She describes her research in the following:

Software now accounts for the lion's share of the cost of developing and using computer systems. My research is directed at establishing a genuine engineering discipline to support the design and development of software systems and reduce the costs and uncertainties of software production. My current focus is on design methods, analytic techniques, and tools used to construct complete software systems from subsystems and their constituent modules. This is the software architecture level of design, which makes me a software architect. (This is from her World Wide Web home page at Carnegie Mellon.)

In 1993 Shaw received the Warnier prize for contributions to software engineering. Among her publications are guides to bicycling and canoeing in western Pennsylvania.

7.1.5 Assigning Responsibilities

Not all responsibilities will be assigned to a class. Some will be free functions or code that appears in `main`, for example. In my design, I decided on the following assignments of responsibilities to classes.

- **Student**
  - Construct using name (ask for name in `main`)
  - RespondTo a question
  - GetScore
  - GetName (not in scenario, but useful accessor)
- **Quiz**
  - ChooseKindOfQuestion
  - AskQuestion of/GiveQuestion to a student
- **Question**
  - Create/Construct question type
  - AskQuestion
  - GetCorrectAnswer

These assignments are not the only way to assign responsibilities for the quiz program. In particular, it's not clear that a `Student` object should be responsible for determining its own score. It might be better to have the `Quiz` track the score for each student taking the quiz. However, we'll think about how scores are kept (this is state, and we shouldn't think about state at this stage, but we can think of which class is responsible for keeping the state). If `Quiz` keeps score, then it may be harder to keep score for three, four, or more students. If each `Student` keeps score, we may be able to add students more easily.

We haven't assigned to any class the responsibilities of determining the number of questions and of providing feedback. We'll prompt the student for the number of questions in `main` and feedback will be part of either `Quiz::GiveQuestionTo` or `Student::RespondTo`. We're using the **scope resolution operator** :: to associate a method with a class since this makes it clear how responsibilities are assigned.

7.1.6 Implementing and Testing Classes

At this point you could write a header (.h) file for each class. This helps solidify our decisions and writing code usually helps in finding flaws in the initial design. Design is not a sequential process, but is a process of **iterative enhancement**. At each step, you may need to revisit previous steps and rethink decisions that you thought were obviously correct. As you begin to implement the classes, you may develop scenarios unanticipated in the first steps of designing classes.

Program Tip 7.2: One cornerstone of iterative enhancement is adding code to a working program. This means that the software program grows, it doesn't spring forth fully functional. The idea is that it's easier to test small pieces and add functionality to an already tested program, than it is to test many methods or a large program at once.

Ideally we'll test each class separately from the other classes, but some classes are strongly coupled and it will be difficult to test one such class without having the other class already implemented and tested. For example, testing the `Student::RespondTo` method probably requires passing a question to this method that the student can respond to. If we don't have a question what can we do? We can use **stub functions** that are not fully functional (e.g., the function might be missing parameters) but that generate output we'll use to test our scenarios. We might use the stub shown as Program 7.1.

Program 7.1 studentstub.cpp

```
void Student::RespondTo( missing Question parameter )
{
    string answer;
    cout << endl << "type answer after question " << endl;
    cout << "what is your favorite color? ";
    cin >> answer;
    if (answer == "blue")
    {   cout << "that is correct" << endl;
    myCorrect++;
    }
    else
    {   cout << "No! your favorite color is blue" << endl;
    }
}
```

studentstub.cpp

We could use this stub function to test the other member functions `Student::Name()` and `Student::Score()`. Program 7.2 shows a test program for the class `Student`.

Program 7.2 mainstub.cpp

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

int main()
{
    string name = PromptString("enter name: ");
    int numQuest = PromptRange("number of questions: ",1,10);
    Student st(name);
    int k;
    for(k=0; k < numQuest; k++)
    {   st.RespondTo();   // question parameter missing
    }
    cout << st.Name() << ", your score is "
         << st.Score() << " out of " << numQuest << endl;
    return 0;
}
```

mainstub.cpp

In testing the class `Student` I created a program *teststudent.cpp* like *mainstub.cpp* above. I put the class interface/declaration (.h file) and implementation/definition (.cpp file) in *teststudent.cpp* rather than in separate files (although the program above shows a `#include"student.h"`, that's not how I originally wrote the test program). After the class was tested, I cut-and-pasted the code segments into the appropriate student.h and student.cpp files. Although not shown here, a test program similar to the one above is available as *teststudent.cpp* in the on-line programs available for this book (but see *quiz.cpp*, Program 7.8 for a complete program with classes `Student` and `Quiz`). A run of *teststudent.cpp* (or *mainstub.cpp*, Program 7.2) follows.

OUTPUT

```
enter name: Owen
number of questions:  between 1 and 10: 3

type answer after question
what is your favorite color? red
No! your favorite color is blue

type answer after question
what is your favorite color? blue
that is correct

type answer after question
what is your favorite color? yellow
No! your favorite color is blue
Owen, your score is 1 out of 3
```

After testing the `Student` class we can turn to the `Quiz` class. In general the order in which classes should be implemented and tested is not always straightforward. In [Ben88] John Bentley offers the following "tips" from Al Schapira:

Program Tip 7.3: Always do the hard part first. If the hard part is impossible, why waste time on the easy part? Once the hard part is done, you're home free.

Program Tip 7.4: Always do the easy part first. What you think at first is the easy part often turns out to be the hard part. Once the easy part is done, you can concentrate all your efforts on the hard part.

7.1.7 Implementing the Class `Quiz`

There are two behaviors in the list of responsibilities for the class `Quiz`: choosing the kind of question and giving the question to a student. The kind of question will be an integral part of the class `Question`. It's not clear what the class `Quiz` can do in picking a type of question, but if there were different kinds of questions perhaps the `Quiz` class could choose one. Since we currently have only one type of question we'll concentrate on the second responsibility: giving a question to a student.

In designing and implementing the function `Quiz::GiveQuestionTo` we must decide how the `Quiz` knows which student to ask. There are three possibilities. The important difference between these possibilities is the responsibility of creating `Student` objects.

1. A `Quiz` object knows about all the students and asks the appropriate student. In this case all `Student` objects would be private data in the `Quiz` class, created by the `Quiz`.
2. The student of whom a question will be asked is passed as an argument to the `Quiz::GiveQuestionTo` member function. In this case the `Student` object is created somewhere like `main` and passed to a `Quiz`.
3. The student is created in the function `Quiz::GiveQuestionTo` and then asked a question.

These are the three ways in which a `Quiz` member function can access any kind of data, and in particular a `Student` object. The three ways correspond to how `Student` objects are defined and used:

1. As instance variables of the class `Quiz` since private data is global to all `Quiz` methods, so is accessible in `Quiz::GiveQuestionTo`.
2. As parameter(s) to `Quiz::GiveQuestionTo`. Parameters are accessible in the function to which they're passed.
3. As local variables in `Quiz::GiveQuestionTo` since local variables defined in a function are accessible in the function.

In our quiz program, the third option is not a possibility. Variables defined within a function are not accessible outside the function, so `Student` objects defined within the function `Quiz::GiveQuestionTo` are not accessible outside the function. This means no scores could be reported, for example. If we choose the first option, the `Quiz` class must provide some mechanism for getting student information since the students will be private in the `Quiz` class and not accessible, for example, in `main` to print scores unless the `Quiz` class provides accessor functions for students.

The second option makes the most sense. `Student` objects can be defined in `main`, as can a `Quiz` object. We can use code like the following to give a quiz to two students.

```
 int main()
 {
    Student owen("Owen");
    Student susan("Susan");
    Quiz q;
    q.GiveQuestionTo(owen);
    q.GiveQuestionTo(susan);

    cout << owen.Name()  << " score = "
         << owen.Score() << endl;
    cout << susan.Name() << " score = "
         << susan.Score() << endl;
    return 0;
}
```

This code scenario corresponds to one of the original requirements: allow two students to take a quiz at the same time using the same program. The code should also provide a clue as to how the `Student` parameter is passed to `Quiz::GiveQuestionTo`, by value, by reference, or by const-reference.

If you think carefully about the code, you'll see that the score reported for each student must be calculated or modified as part of having a question asked. This means the score of a student changes (potentially) when a question is asked. For changes to be communicated, the `Student` parameter must be a reference parameter. A value parameter is a copy, so any changes will not be communicated. A const-reference parameter cannot be changed, so the number of correct responses cannot be updated. Reference parameters are used to pass values back from functions (and sometimes to pass values in as well), so the `Student` parameter must be passed by reference.

We'll design the function `Quiz::GiveQuestionTo()` to permit more than one attempt, one of the original program requirements. The code is shown in Program 7.3.

Program 7.3 quizstub.cpp

```
void Quiz::GiveQuestionTo(Student & s)
// postcondition: student s asked a question
{
  cout << endl << "Ok, " << s.Name() << " it's your turn" << endl;
  cout << "type answer after question " << endl;

  myQuestion.Create();
  if (! s.RespondTo(myQuestion))
  { cout << "try one more time" << endl;
    if (! s.RespondTo(myQuestion))
    {  cout << "correct answer is " << myQuestion.Answer() << endl;
    }
  }
}
```

quizstub.cpp

This code shows some of the methods of the class `Question`. From the code, and the convention of using the prefix `my` for private data, you should be able to reason that the object `myQuestion` is private data in `Quiz` and that methods for the `Question` class include `Question::Create()` and `Question::Answer()`. The other method listed in the original responsibilities for `Question`, which we'll call `Question::Ask()` is responsible for asking the question. As we'll see, this method is called in `Student::RespondTo()`.

Pause to Reflect

7.6 If `myQuestion` is an instance variable of the class `Quiz`, where is `myQuestion` constructed?

7.7 Why is `s`, the parameter of `Quiz::GiveQuestionTo()` a reference parameter? Why can't it be a const reference parameter (think about the scenarios and what happens to parameter `s` after a question is given.)

7.8 As shown in Program 7.3, the function `Student::RespondTo()` returns a `bool` value. Based on the value's use, what is an appropriate postcondition for the function?

7.9 How is the function `Student::RespondTo()` in Program 7.3 different from the version used in *mainstub.cpp*, Program 7.2? Is it appropriate that the function changed?

7.10 What is the prototype of the method `Student::RespondTo()` as it is used in Program 7.3? In particular, how is the parameter passed: by value, by reference, or by const reference (and why)?

7.11 `Question::Answer()` returns the correct answer in some printable form. When do you think the correct answer is determined?

7.1.8 Implementing the Class `Question`

In testing the function `Quiz::GiveQuestionTo` above, I didn't have the class `Question` implemented. I could have implemented a simple version of the class; a version good enough for testing other classes. Alternatively I could use output statements in place of calling the `Question` methods, much like the quiz about favorite colors was used in testing the class `Student`. Since a simple version of the class is useful in testing other classes, I implemented the version shown in *question.h*, Program 7.4. Note that each member function is implemented in the class rather than as a separate function outside the class. In general, the class declaration (interface) should be kept separate from the definition (implementation). For a test implementation like this one, which will eventually be replaced by separate .h and .cpp files, making all the code part of the class declaration is acceptable practice. When function definitions are included in a class declaration, the functions are called **inline** functions. In general you should not use inline member functions, but should define them in a separate .cpp file.

Program Tip 7.5: Some programmers use inline member functions for “small” classes — those classes that have few member functions and few instance variables. However, as you’re learning to design and implement classes it’s a good idea to use the generally accepted practice of separating a class’s interface from its implementation by using separate .h and .cpp files.

Program 7.4 question.h

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

// simple Question class for testing other classes

class Question
{
  public:
    Question()
    {   // nothing to initialize
    }
    void Create()
    {   // the same question is used every time
    }
    void Ask()
    {   cout << "what is your favorite color? ";
    }
    string Answer() const
    {   return "blue";
    }
};
```

question.h

Our test version of `Student::RespondTo` can be modified to use the simple `Question` class as shown. The output of the program will not change from the original version in *teststudent.cpp*.

```
void Student::RespondTo(Question & q)
{
   string answer;
   cout << endl << "type answer after question " << endl;
   q.Ask();
   cin >> answer;

   if (answer == q.Answer())
   {   cout << "that is correct" << endl;
       myCorrect++;
   }
   else
```

```
    {   cout << "No! your favorite color is "
             << q.Answer() << endl;
    }
}
```

With this simple version of `Question` done, we can test the implementations of `Student` and `Quiz` completely. Then we can turn to a complete implementation of a `Question` class for implementing quizzes in arithmetic as called for in the requirements for this problem.

7.1.9 Sidebar: Converting `int` and `double` Values to `strings`

In our test-version of the `Question` class the function `Question::Answer()` returns a string. As we turn to the final implementation it seems like we'll need to change this return type to be an `int` since the answer to a question about an arithmetic operation like addition is almost certainly an integer, not a string. There's a compelling reason to leave the return type as `string`, however. One of the original requirements was to design and implement a program that allows quizzes about a wide variety of topics, such as English literature and rock and roll songs in the original list of topics. The answers to these questions will almost certainly be strings rather than numbers. How can we accommodate all possible quiz answers?

If we could convert `int` values to strings, such as the number 123 to the string `"123"`, we could continue to use strings to represent answers. Since almost any kind of answer can be represented as a string, we'd like to use strings. For example, the string `"3.14159"` prints just like the `double` value 3.14159. We built functions for converting integers to an English representation in *numtoeng.cpp*, Program 4.10 and in *digits.cpp*, Program 5.5. These programs converted an int value like 123 to `"one hundred twenty three"` and `"one two three"`, respectively. We could use these as a basis for writing our own conversion functions. Fortunately, there are functions already written that convert numeric values to equivalent strings and vice versa. These functions are demonstrated in *numtostring.cpp*, Program 7.5.

As shown in the output, the functions `tostring`, `atoi`, and `atof` do no error checking (non-numeric strings are converted to zero by both `atoi` and `atof`.) These conversion functions are part of the string processing functions accessible using *strutils.h* given in Howto G as Program G.8.[3]

With the conversion functions from *strutils.h* now in our programming tool kit, we can tackle the problem of implementing the `Question` class for questions about arithmetic problems.

[3]The functions `atoi` and `atof` are **adapter functions** for standard conversion functions with the same names in `<cstdlib>` (or `<stdlib.h>`). The functions `atoi` and `atof` in `<cstdlib>` take C-style, char * strings as parameters, so functions accepting string parameters are provided in `"strutils.h"` as adapters for the standard functions.

Program 7.5 numtostring.cpp

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

#include "strutils.h"   // for tostring, atoi

// illustrate string to int/double conversion and vice versa

int main()
{
    int    ival;
    double dval;
    string s;

    cout << "enter an int ";
    cin >> ival;

    s = tostring(ival);
    cout << ival << " as a string is " << s << endl;

    cout << "enter a double ";
    cin >> dval;
    cout << dval << " as a string is " << tostring(dval) << endl;

    cout << "enter an int (to store in a string) ";
    cin >> s;
    ival = atoi(s);
    cout << s << " as an int is " << ival << endl;

    cout << "enter a double (to store in a string) ";
    cin >> s;
    cout << s << " as a double is " << atof(s) << endl;

    return 0;
}
```

numtostring.cpp

OUTPUT

```
prompt> numtostring
enter an int 1789
1789 as a string is 1789
enter a double 2.7182
2.7182 as a string is 2.7182
enter an int (to store in a string) -639
-639 as an int is -639
enter a double (to store in a string) 17e2
17e2 as a double is 1700
prompt> numtostring
enter an int -123
-123 as a string is -123
enter a double 17e2
1700 as a string is 1700
enter an int (to store in a string) 23skidoo
23skidoo as an int is 23
enter a double (to store in a string) pi
pi as a double is 0
```

The member function `Question::Ask()` must ask the question last created by the function `Question::Create()`. Since these functions are called independently by client programs, the `Create` function must store information in private, state variables of the `Question` class. These state variables are then used by `Question::Ask()` to print the question. We'll use simple addition problems like "what is 20 + 13?". We'll store the two numbers that are part of a question in instance variables `myNum1` and `myNum2`. Values will be stored in these variables by `Question::Create()` and the values will be accessed in `Question::Ask()`. We'll also store the answer in the instance variable `myAnswer` so that it can be accessed in the accessor function `Question::Answer()`.

Program Tip 7.6: Instance variables are useful for communicating values between calls of different member functions. The values might be set in one function and accessed in a different function. Sometimes instance variables are used to maintain values between calls of the same function.

As the last step in our design we'll think about frequent uses of the class that we can make easier (or at least simpler). Client code will often check if a student response is correct, using code like this:

```
if (response == q.Answer()) // correct
```

We'll make this easier by using a bool-valued function `Question::IsCorrect` so that checking code will change to this:

```
if (q.IsCorrect(response)) // correct
```

This opens the possibility of changing how the function `Question::Answer` works. For example, we could allow `albany` to be a match for `Albany` by making `IsCorrect` ignore the case of the answers. We could even try to allow for misspellings. We might also try to prevent clients from calling `Answer`, but allow them to check if an answer is correct. We'll leave the `Answer` function in place for now, but in designing classes the goal of hiding information and minimizing access to private state should be emphasized. Consider the unnecessary information revealed in some campus debit-card systems. If a student buys some food, and the register shows a balance of $1,024.32 to everyone in the checkout line, too much information has been revealed. The only information that's needed to complete the purchase is whether the student has enough money in her account to cover the purchase. It's fine for the everyone to see "Purchase OK," but it's not acceptable for everyone to see all balances. A student balance, for example, could be protected by using a password to access this sensitive information.

Finally, we decide which functions are accessors and which are mutators. Accessor functions don't change state, so they should be created as `const` functions. The final class declaration is shown as *mathquest.h*, Program 7.6.

Program 7.6 mathquest.h

```
#ifndef _MATHQUEST_H
#define _MATHQUEST_H

// ask a question involving arithmetic
//
// This class conforms to the naming conventions
// of quiz questions in "A Computer Science Tapestry" 2e,
// this convention requires the following functions:
//
// void Create()      – ask a new question
// void Ask() const   – ask the last question Create()'d
//
// bool IsCorrect(const string& answer) const
//      – return true iff answer is correct to last (Create()) question
// string Answer() const
//      – return the answer to the last (Create()) question
//
#include <string>
using namespace std;

class Question
{
  public:
    Question();

    bool IsCorrect(const string& answer) const;
    string Answer()                     const;
    void Ask()                          const;

    void Create();  // create a new question

  private:

    string myAnswer;  // store the answer as a string here
    int myNum1;       // numbers used in question
    int myNum2;
};

#endif
```

mathquest.h

The final class definition/implementation is shown as *mathquest.cpp*, Program 7.7. Some new syntax is shown in Program 7.7 for initializing instance variables in a constructor. In previous constructors like the `Dice` constructor in *dice.cpp*, Prog 6.1, instance variables were assigned values in the body of the constructor using syntax identical to variable assignment in other contexts.

The code in the *Question::Question* constructor uses an **initializer list** to give initial values to all instance variables. Each instance variable must be constructed. Construction of instance variables takes place before the body of the constructor executes. When parameters must be supplied to a variable at construction time, the values are supplied in an initializer list that appears between the constructor header and the body of the constructor. A single colon `':'` is used to begin the initializer list and each item in the list is separated from other items by a comma `','` — but note that the last item is not followed by a comma since commas separate items (this is a fence post problem.) Since some instance variables require parameters at construction time, such as a `Dice` variable requires a parameter, I'll use initializer lists for constructors in code shown from now on. When an instance variable doesn't need a constructor you can show it with a parameterless constructor as shown for `myVar3` in the syntax diagram. Alternatively you can omit this constructor call, but then one of the instance variables won't appear in the list. I'll try to be consistent in initializing all instance variables. Instance variables are initialized in the order in which they appear in a class declaration, and *not* in the order in which they appear in the initializer list. To avoid problems, make the order of construction in the initializer list the same as the order in which instance variables appear in the private section of a class declaration. Some compilers will catch inconsistent orderings and issue a warning.

Syntax: initializer list

```
ClassName::ClassName (parameters)
  :  myVar1(parameters),
     myVar2(parameters),
     myVar3( ),
     myVarN(parameters)
{
  code as needed for further initialization
}
```

Program 7.7 mathquest.cpp

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

#include "mathquest.h"
#include "randgen.h"
#include "strutils.h"


Question::Question()
   : myAnswer("*** error ***"),
     myNum1(0),
     myNum2(0)
{
    // nothing to initialize
}

void Question::Create()
{
    RandGen gen;

    myNum1 = gen.RandInt(10,20);
    myNum2 = gen.RandInt(10,20);
    myAnswer = tostring(myNum1 + myNum2);
}

void Question::Ask() const
{
    const int WIDTH = 7;
    cout << setw(WIDTH) << myNum1 << endl;
    cout << "+" << setw(WIDTH-1) << myNum2 << endl;
    cout << "——-" << endl;
    cout << setw(WIDTH-myAnswer.length()) << " ";
}

bool Question::IsCorrect(const string& answer) const
{
    return myAnswer == answer;
}

string Question::Answer() const
{
    return myAnswer;
}
```

mathquest.cpp

Program 7.8, *quiz.cpp*, uses all the classes in a complete quiz program. The class declarations and definitions for `Student` and `Quiz` are included in *quiz.cpp* rather than in separate .h and .cpp files. The `Question` class is separated into separate files to make it easier to incorporate new kinds of questions.

Program 7.8 quiz.cpp

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

// quiz program for illustrating class design and implementation

class Student
{
  public:
    Student(const string& name);   // student has a name

    int    Score() const;          // # correct
    string Name()  const;          // name of student

    bool RespondTo(Question & q);  // answer a question, update stats

  private:

    string myName;        // my name
    int    myCorrect;     // my # correct responses
};

Student::Student(const string& name)
  : myName(name),
    myCorrect(0)
{
    // initializer list does the work
}

bool Student::RespondTo(Question & q)
// postcondition: q is asked, state updated to reflect responses
//                return true iff question answered correctly
{
    string answer;
    q.Ask();
    cin >> answer;

    if (q.IsCorrect(answer))
    {   myCorrect++;
        cout << "yes, that's correct" << endl;
        return true;
    }
    else
    {   cout << "no, that's not correct" << endl;
        return false;
    }
}
```

```
int Student::Score() const
// postcondition: returns # correct
{
    return myCorrect;
}

string Student::Name() const
// postcondition: returns name of student
{
    return myName;
}

class Quiz
{
  public:
    Quiz();
    void GiveQuestionTo(Student & s); // ask student a question

  private:

    Question myQuestion; // question generator
};

Quiz::Quiz()
  : myQuestion()
{
  // nothing to do here
}

void Quiz::GiveQuestionTo(Student & s)
// postcondition: student s asked a question
{
    cout << endl << "Ok, " << s.Name() << " it's your turn" << endl;
    cout << "type answer after question " << endl;

    myQuestion.Create();
    if (! s.RespondTo(myQuestion))
    {   cout << "try one more time" << endl;
        if (! s.RespondTo(myQuestion))
        {    cout << "correct answer is " << myQuestion.Answer() << endl;
        }
    }
}

int main()
{
    Student owen("Owen");
    Student susan("Susan");
    Quiz q;
    int qNum = PromptRange("how many questions: ",1,5);
    int k;
    for(k=0; k < qNum; k++)
    {   q.GiveQuestionTo(owen);
        q.GiveQuestionTo(susan);
    }
    cout << owen.Name()  << " score:\t" << owen.Score()
         << " out of " << qNum
         << " = " << double(owen.Score())/qNum * 100 << "%" << endl;
    cout << susan.Name() << " score:\t" << susan.Score()
         << " out of " << qNum
         << " = " << double(susan.Score())/qNum * 100 << "%" << endl;

    return 0;
}
```

quiz.cpp

O U T P U T

```
prompt> quiz
how many questions:  between 1 and 5: 3

Ok, Owen it's your turn
type answer after question
      19
+     17
-------
      36
yes, that's correct

Ok, Susan it's your turn
type answer after question

      11
+     16
-------
      27
yes, that's correct

Ok, Owen it's your turn
type answer after question

      17
+     15
-------
      34
no, that's not correct
try one more time
```

output continued

OUTPUT

```
     17
+    15
-------
     32
yes, that’s correct

Ok, Susan it’s your turn
type answer after question
     20
+    17
-------
     37
yes, that’s correct

Ok, Owen it’s your turn
type answer after question
     16
+    17
-------
     23
no, that’s not correct
try one more time
     16
+    17
-------
     27
no, that’s not correct
correct answer is 33

Ok, Susan it’s your turn
type answer after question
     15
+    17
-------
     32
yes, that’s correct
Owen score:     2 out of 3 = 66.6667%
Susan score:    3 out of 3 = 100%
```

Pause to Reflect

7.12 What (simple) modifications can you make to the sample `Question` class in *question.h*, Program 7.4 so that one of two colors is chosen randomly as the favorite color. The color should be chosen in `Question::Create()` and used in the other methods, `Ask()` and `Answer()`.

7.13 Why is the string `"pi"` converted to the `double` value zero by `atof` in the sample run of Program 7.5, *numtostring.cpp*?

7.14 Does conversion of `"23skidoo"` to the `int` value 23 mirror how the string would be read if the user typed `"23skidoo"` if prompted by the following:

```
int num;
cout << "enter value ";
cin >> num;
```

7.15 Why is the function `Question::Ask()` declared as `const` in *mathquest.h*, Program 7.6?

7.16 The declaration for a class `Game` is partially shown below.

```
class Game
{
  public:
    Game();
    ...
  private:
    Dice myCube;
    int  myBankRoll;
};
```

The constructor should make `myCube` represent a six-sided `Dice` and should initialize `myBankRoll` to 5000. Explain why an initializer list is required because of `myCube` and show the syntax for the constructor `Game::Game()` (assuming there are only the two instance variables shown in the class).

7.17 The statements for reporting quiz scores for two students in `quiz.cpp`, Program 7.8 duplicate the code used for the output. Write a function that can be called to generate the output for either student, so that the statements below replace the score-producing output statements in `quiz.cpp`.

```
reportScores(owen,qNum);
reportScores(susan,qNum);
```

7.18 What question is asked if a client program calls `Question::Ask()` without ever calling `Question::Create()`?

7.2 A Conforming Interface: a new `Question` Class

We want to develop a new `Question` class for a different kind of quiz. As an illustration, we'll develop a class for asking questions about state capitals for U.S. states. We don't have the programming tools needed to easily store the state/capital pairs within the new `Question` class.[4] Instead, we'll put the state/capital pairs in a text file and read the text file repeatedly using a `WordStreamIterator` object. To generate a random state/capital question we'll skip a random number of lines of the file when reading it. Suppose the first five lines of the text file are as follows.

```
Alabama   Montgomery
Alaska   Juneau
Arizona   Phoenix
Arkansas   Little_Rock
California   Sacramento
```

If we skip two lines of the file we'll ask what the capital of Arizona is; if we skip four lines we'll ask about the capital of California; and if we don't skip any lines we'll ask about Alabama.

7.2.1 Using the New `Question` Class

We'll call the new class `Question`. This will allow us to use the class without changing the program *quiz.cpp*, Program 7.8, except to replace the `#include"mathquest.h"` line by `#include"capquest.h"`. Since the class about capitals declared in *capquest.h* has the same interface, (i.e., the same class name and the same public member functions) as the class declared in *mathquest.h*, the client program in *quiz.cpp* doesn't change at all. A run of the program *quiz.cpp*, Program 7.8, modified to include `"capquest.h"` is shown below.

The downside of this approach is that we can't let the user choose between math questions and capital questions while the program is running; the choice must be made before the program is compiled. This is far from ideal, because we wouldn't expect a real student-user to compile a program to take a quiz using a computer. However, until we study inheritance in Chapter 13 we don't really have any other options.

[4]Perhaps the simplest way to do this is to use a vector or array, but the method used in the `Question` class developed in this chapter is fairly versatile without using a programming construct we haven't yet studied.

OUTPUT

```
prompt> quiz
how many questions:  between 1 and 5: 2

it's your turn Owen
type answer after question
the capital of Wisconsin is Madison
yes, that's correct

it's your turn Susan
type answer after question
the capital of Washington is Seattle
no, try one more time
the capital of Washington is Tacoma
no, correct answer is Olympia

it's your turn Owen
type answer after question
the capital of Utah is Salt_Lake_City
yes, that's correct

it's your turn Susan
type answer after question

the capital of New_Mexico is Albuquerque
no, try one more time
the capital of New_Mexico is Santa_Fe
yes, that's correct
Owen score:     2 out of 2 = 100%
Susan score:    1 out of 2 = 50%
```

7.2.2 Creating a Program

Before looking briefly at the new implementation of `Question`, we'll review the process of creating a working C++ program. This will help you understand how the different `Question` classes work with the quiz program.

Three steps are needed to generate an executable program from source files.

1. The **preprocessing** step handles all `#include` directives and some others we haven't studied. A **preprocessor** is used for this step.
2. The **compilation** step takes input from the preprocessor and creates an **object file** (see Section 3.5) for each .cpp file. A **compiler** is used for this step.

3. One or more object files are combined with libraries of compiled code in the **linking** step. The step creates an executable program by linking together system-dependent libraries as well as client code that has been compiled. A **linker** is used for this step.

7.2.3 The Preprocessor

The preprocessor is a program run on each source file before the source file is compiled. A source file like *hello.cpp*, Program 2.1 is translated into something called a **translation unit** which is then passed to the compiler. The source file isn't physically changed by the preprocessor, but the preprocessor does use **directives** like `#include` in creating the translation unit that the compiler sees. Each preprocessor directive begins with a sharp (or number) sign # that must be the first character on the line.

Processing `#include` Statements. A `#include` statement literally cut-and-pastes the code in the file specified into the translation unit that is passed to the compiler. For example, the preprocessor directive `#include<iostream>` causes the preprocessor to find the file named `iostream` and insert it into the translation unit. This means that what appears to be a seven line program like the following might actually generate a translation unit that causes the compiler to compile 10,000 lines of code.

```
#include<iostream>
using namespace std;
int main()
{
    cout << "hello world" << endl;
    return 0;
}
```

I tried the program above with three different C++ environments. The size of the translation unit ranged from 2,986 lines using g++ with Linux, to 16,075 using Borland CBuilder, to 17,261 using Metrowerks Codewarrior.

Compilers are fast. At this stage of your programming journey you don't need to worry about minimizing the use of the `#include` directive, but in more advanced courses you'll learn techniques that help keep compilation times fast and translation units small.

Where are `include` Files Located? The preprocessor looks in a specific list of directories to find include files. This list is typically called the **include path**. In most environments you can alter the include path so that the preprocessor looks in different directories. In many environments you can specify the order of the directories that are searched by the preprocessor.

Program Tip 7.7: If the preprocessor cannot find a file specified, you'll probably get a warning. In some cases the preprocessor will find a different file than the one you intend; one that has the same name as the file you want to include. This can lead to compilation errors that are hard to fix. If your system lets you examine the translation unit produced by the preprocessor you may be able to tell what files were included. You should do this only when you've got real evidence that the wrong header file is being included.

Most systems look in the directory in which the .cpp file that's being preprocessed is located. More information about setting options in your programming environment can be found in Howto I.

Other Preprocessor Directives. The only other preprocessor directive we use in this book is the **conditional compilation** directive. Each header file begins and ends with preprocessor directives as follows (see also *dice.h*, Program G.3). Suppose the file below is called *foo.h*.

```
#ifndef _FOO_H
#define _FOO_H

header file for Foo goes here

#endif
```

The first line tells the preprocessor to include the file *foo.h* in the current translation unit only if the symbol `_FOO_H` is *not* defined. The *n* in `ifndef` means "if NOT defined", then proceed. The first thing that happens if the symbol `_FOO_H` is not defined, is that it becomes defined using the directive `#define`. The final directive `#endif` helps limit the extent of the first `#ifndef`. Every `#ifndef` has a matching `#endif`. The reason for bracketing each header file with these directives is to prevent the same file from being included twice in the same translation unit. This could easily happen, for example, if you write a program in which you include both `<iostream>` and `"date.h"`. The header file `"date.h"` also includes `<iostream>`. When you include one file, you also include all the files that it includes (and all the files that they include, and all the files that they include). Using the `#ifndef` directive prevents an infinite chain of inclusions and prevents the same file from being included more than once.

Occasionally it's useful to be able to prevent a block of code from being compiled. You might do this, for example, during debugging or development to test different versions of a function. The directive `#ifdef` causes the preprocessor to include a section of a file only if a specific symbol is defined.

```
#ifdef FOO
void TryMe(const string& s)
{   cout << s << " is buggy" << endl;
}
#endif
```

```
  void TryMe(const string& s)
 {   cout << s << "is correct" << endl;
 }
```

In the code segment above, the call `TryMe("rose")` generates `rose is correct` as output. The first version (on top) of `TryMe` isn't compiled, because the preprocessor doesn't include it in the translation unit passed to the compiler unless the symbol `FOO` is defined. You can, of course, define the symbol `FOO` if you want to. Some programmers use `#ifdef 0` to block out chunks of code since zero is never defined.

7.2.4 The Compiler

The input to the compiler is the translation unit generated by the preprocessor from a source file. The compiler generates an **object file** for each compiled source file. Usually the object file has the same prefix as the source file, but ends in .o or .obj. For example, the source file *hello.cpp* might generate *hello.obj* on some systems. In some programming environments the object files aren't stored on disk, but remain in memory. In other environments, the object files are stored on disk. It's also possible for the object files to exist on disk for a short time, so that the linker can use them. After the linking step the object files might be automatically erased by the programming environment.

Object files are typically larger than the corresponding source file, but may be smaller than the translation unit corresponding to the source file. Many compilers have options that generate **optimized code**. This code will run faster, but the compiler will take longer to generate the optimized code. On some systems you won't be able to use a debugger with optimized code.

Program Tip 7.8: Turn code optimization off. Unless you are writing an application that must execute very quickly, and you've used profiling and performance tools that help pinpoint execution bottlenecks, it's probably not worth optimizing your programs. In some systems, debuggers may get confused when using optimized code, and it's more important for a program to be correct than for it to be fast.

Since the compiler uses the translation unit provided by the preprocessor to create an object file, any changes in the translation unit from a .cpp source file will force the .cpp file to be recompiled. For example, if the header file *question.h* is changed, then the source program *quiz.cpp*, Program 7.8 will need to be recompiled. Since the file *question.h* is part of the translation unit generated from *quiz.cpp*, the recompilation is necessary because the translation unit changed. In general, a source file has several **compilation dependencies**. Any header file included by the source file generates a dependency. For example, Program 7.8, *quiz.cpp* has four direct dependencies:

- `<iostream>` and `string`, two system dependencies.
- `"prompt.h"` and `"mathquest.h"`, two non-system dependencies.

There may be other indirect dependencies introduced by these. Since both `"prompt.h"` and `"mathquest.h"` include `<string>`, another dependency would be introduced, but `<string>` is already a dependency.

Program Tip 7.9: You should try to minimize the number of dependencies for each source file. Since a change in a dependency will force the source file to be recompiled, keeping the number dependencies small means you'll need to recompile less often during program development.

Notice that *mathquest.cpp*, Program 7.7 depends directly on the files `randgen.h` and `strutils.h`. These two files are *not* dependencies for *quiz.cpp* since they're not part of the translation unit for *quiz.cpp*.

Libraries. Often you'll have several object files that you use in all your programs. For example, the implementations of `iostream` and `string` functions are used in nearly all the programs we've studied. Many programs use the classes declared in `prompt.h`, `dice.h`, `date.h` and so on. Each of these classes has a corresponding object file generated by compiling the .cpp file. To run a program using all these classes the object files need to be combined in the linking phase. However, nearly all programming environments make it possible to combine object files into a library which can then be linked with your own programs. Using a library is a good idea because you need to link with fewer files and it's usually simple to get an updated library when one becomes available.

7.2.5 The Linker

The linker combines all the necessary object files and libraries together to create an executable program. Libraries are always needed, even if you are not aware of them. Standard libraries are part of every C++ environment and include classes and functions for streams, math, and so on. Often you'll need to use more than one library. For example, I use a library called *tapestry.lib* for all the programs in this book. This library contains the object files for classes `Dice`, `Date`, `RandGen` and functions from `strutils` among many others. The suffix `.lib` is typically used for libraries.

You aren't usually aware of the linker as you begin to program because the libraries are linked in automatically. However, as soon as you begin to write programs that use several .cpp files, you'll probably encounter linker errors.

For example, if I try to create an executable program from *quiz.cpp*, Program 7.8, but I forget to link in the code from the class `Question` in *mathquest.cpp*, Program 7.7, the following errors are generated. The first two errors using Metrowerks Codewarrior follow:

```
Link Error : Undefined symbol: ?Ask@Question@@QBEXXZ
(Question::Ask) in file: quiz.cpp
```

```
Link Error : Undefined symbol:?IsCorrect@Question@@QBE_NABV?
$basic_string@DU?$char_traits@D@std@@V?$
                                       allocator@D@2@@std@@@Z
(Question::IsCorrect) in file:quiz.cpp
```

Using Microsoft Visual C++ the first two errors follow:

```
quiz.obj : error LNK2001: unresolved external symbol
 "public: void__thiscall Question::Ask(void)const "
          (?Ask@Question@@QBEXXZ)
quiz.obj : error LNK2001: unresolved external symbol
"public: bool__thiscall Question::IsCorrect
(class std::basic_string<char,struct std::char_traits<char>,
class std::allocator<char> > const &)const "
(?IsCorrect@Question@@QBE_NABV?$basic_string@DU
```

These errors may be hard to understand. The key thing to note is that they are **linker errors**. Codewarrior specifically identifies the errors as linker errors. If you look at the Visual C++ output you'll see a clue that the linker is involved; the errors are identified as `error LNK2001`.

Program Tip 7.10: If you get errors about unresolved references, or undefined/unresolved external symbols, then you've got a linker error. This means that you need to combine the object files from different .cpp files together. In most C++ environments this is done by adding the .cpp file to a project, or by changing a Makefile to know about all the .cpp files that must be linked together.

String Compilation and Linker Errors. The other reason the errors are hard to read is because of the standard class `string`. The `string` class is complicated because it is intended to be an industrial-strength class used with several character sets (e.g., ASCII and UNICODE) at some point. The `string` class is actually built on top of a class named `basic_string` which you may be able to identify in some of the linker errors above.

7.2.6 A New `Question` Class

The new question class in *capquest.h* has the same public member functions as, but a different private section from, the class in *mathquest.h*. Part of *capquest.h* follows.

```
class Question
{
  public:
    Question(const string& filename);
    bool IsCorrect(const string& answer) const;
    string Answer()                      const;
    void Ask()                           const;

    void Create();  // create a new question

  private:

    string myAnswer;           // answer (state capital)
    string myQuestion;         // the state
    WordStreamIterator myIter; // iterates over file
};
```

The instance variable `myIter` processes the file of states and capitals, choosing one line at random as the basis for a question each time `Question::Create()` is called (see Program 7.9, *capquest.cpp*.) The instance variable `myQuestion` replaces the two instance variables `myNum1` and `myNum2` from *mathquest.h*, Program 7.6. The method `Question::Create()` in *capquest.cpp* does most of the work. In creating the new `Question` class three goals were met.

- Using the same interface (public methods) as the class in *mathquest.h* helped in writing the new class. When I wrote the new class I concentrated only on the implementation since the interface was already done.
- The client program `quiz.cpp` did not need to be rewritten. It did need to be recompiled after changing `#include"mathquest.h"` to use `"capquest.h"`.
- The new class `Question` can be used for questions other than states and capitals. The modifications are straightforward and discussed in the following Pause and Reflect exercises.

Program 7.9 capquest.cpp

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

#include "randgen.h"
#include "strutils.h"

Question::Question(const string& filename)
   : myAnswer("*** error ***"),
     myQuestion("*** error ***")
{
    myIter.Open(filename.c_str());
}

void Question::Create()
{
    RandGen gen;

    int toSkip = gen.RandInt(0,49);   // skip this many lines
    int k;
    myIter.Init();
    for(k=0; k < toSkip; k++)
    {   myIter.Next();   // skip the state
        myIter.Next();   // and the capital
    }
    myQuestion = myIter.Current();
    myIter.Next();
    myAnswer = myIter.Current();
}

void Question::Ask() const
{
    cout << "the capital of " << myQuestion << " is ";
}

bool Question::IsCorrect(const string& answer) const
{
    return myAnswer == answer;
}

string Question::Answer() const
{
    return myAnswer;
}
```

capquest.cpp

Pause to Reflect

7.19 Why are the state New York and the capital Little Rock stored in the data file as `New_York` and `Little_Rock`, respectively (why aren't spaces used)?

7.20 The class `Question` declared in *capquest.h* uses a `WordStreamIterator` instance variable, so it has `#include"worditer.h"` at the top of the file. This means that *quiz.cpp*, Program 7.8, depends directly on `"worditer.h"` and indirectly on `<string>` since `<string>` is included in *worditer.h*. What prevents the file `<string>` from being included multiple times when the preprocessor creates a translation unit for *quiz.cpp*?

7.21 The file *capquest.cpp*, Program 7.9 includes `"randgen.h"`. Does *quiz.cpp* depend on `"randgen.h"`? Why?

7.22 If the class `RandGen` declared in `"randgen.h"` is rewritten so that the header file changes, does *quiz.cpp* need to be recompiled? Relinked (to create an executable program about state capitals)? Why?

7.23 The constant 49 is **hardwired** into the definition of `Question::Create()` for skipping lines in the file of states and capitals. Explain how `myIter` could be used in the constructor of the class to count the lines in the file so that the number 49 would be computed by the class itself at run time.

7.24 Suppose you want to create a quiz based on artists/groups and their records. Data are stored in a text file as follows:

```
Lawn_Boy Phish
A_Live_One Phish
Automatic_for_the_People R.E.M.
Broken Nine_Inch_Nails
The_Joshua_Tree U2
Nick_of_Time Bonnie_Raitt
```

The idea is to ask the user to identify the group that made an album. How can you change the class `Question` in *capquest.h* and *capquest.cpp* so that it can be used to give both state/capital and group/recording quizzes. With the right modifications you should be able to use questions of either type in the same quiz program. (Hint: the new `Question` class constructor could have two parameters, one for the file of data and one for the prompt for someone taking the quiz.)

7.3 Random Walks

When you can measure what you are speaking about,
and express it in numbers, you know something about it…
Lord Kelvin
Popular Lectures and Addresses

We must never make experiments to confirm our ideas, but simply to control them.
Claude Bernard
Bulletin of New York Academy of Medicine, vol. IV, p. 997

In this section we'll explore some programs and classes that are simulations of natural and mathematical events. We'll also use the pattern of iteration introduced with the `WordStreamIterator` class in *worditer.h*, Program G.6 (see Howto G) and used in *maxword.cpp*, Program 6.11. We'll design and implement several classes. Classes for one- and two-dimensional random walks will share a common interface, just as the class `Question` declared in both *mathquest.h* and *capquest.h* did. Because of this common interface, a class for observing random walks (graphically or by printing the data in the walk to a file) will be able to observe both walks. First we'll write a simple program to simulate random walks, then we'll design and implement a class based on this program. Comparing the features of both programs will add to your understanding of object-oriented programming. We'll also study `structs`, a C++ feature for storing data that can be used instead of a class.

A random walk is a model built on mathematical and physical concepts that is used to explain how molecules move in an enclosed space. It's also used as the basis for several mathematical models that predict stock market prices. First we'll investigate a random walk in one dimension and then move to higher dimensions.

June 7, 1999 10:10 owltex Sheet number 53 Page number 310 magenta black

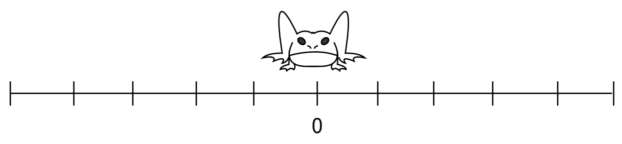


Figure 7.1 Initial position of a frog in a one-dimensional random walk.

7.3.1 One-Dimensional Random Walks

Suppose a frog lives on a lily pad and there are lily pads stretching in a straight line in two directions. The frog "walks" by flipping a coin. If the coin comes up heads, the frog jumps to the right, otherwise the frog jumps to the left. Each time the frog jumps, it jumps one unit, but the length of the jump might change. This jumping process is repeated for a specific number of steps and then the walk stops. The initial configuration for such a random walk is shown in Figure 7.1. We can gather several interesting statistics from a random walk when it is complete (and sometimes during the walk). In a walk of *n* steps we might be interested in how far from the start the frog is at the end of the walk. Also of interest are the furthest points from the start reached by the frog (both east and west or positive and negative if the walk takes place on the x-axis) and how often the frog revisits the "home" lily pad.

We'll look at a simple program for simulating random walks, then think about designing a class that encapsulates a walk, but be more general than the walk we've described. The size of a frog's world might be limited, for example, if the frog lives in a drain pipe.

We'll use a two-sided `Dice` object to represent the coin that determines what direction the frog jumps. Program 7.10, *frogwalk.cpp*, simulates a one-dimensional random walk. The program uses the C++ **switch** instead of an `if/else` statement. The `switch` statement is the final control statement we'll use in our programs. A switch statement is often shorter than the corresponding sequence of cascaded `if/else` statements, but it's also easier to make programming errors when writing code using `switch` statements. We'll discuss the statement after the program listing.

With a graphical display, the frog could be shown moving to the left and right. Alternatively, a statement that prints the position of the frog could be included within the `for` loop. This would provide clues as to whether the program is working correctly. In the current program, the only output is the final position of the frog. Without knowing what this position should be in terms of a mathematical model, it's hard to determine if the program accurately models a one-dimensional random walk.

Program 7.10 frogwalk.cpp

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

// simulate one-dimensional random walk
// Owen Astrachan, 8/13/94, modified 5/1/99

int main()
{
    int numSteps = PromptRange("enter # of steps",0,20000);
    int position = 0;         // "frog" starts at position 0
    Dice die(2);              // used for "coin flipping"
    int k;
    for(k=0; k < numSteps; k++)
    {   switch (die.Roll())
        {
          case 1:
            position++;    // step to the right
            break;
          case 2:
            position--;    // step to the left
            break;
        }
    }
    cout << "final position = " << position << endl;
    return 0;
}
```

frogwalk.cpp

OUTPUT

```
prompt> frogwalk
enter # of steps between 0 and 20000: 1000
final position = 32
prompt> frogwalk
enter # of steps between 0 and 20000: 1000
final position = -14
prompt> frogwalk
enter # of steps between 0 and 20000: 1000
final position = 66
```

7.3.2 Selection with the `switch` Statement

In Exercise 9 of Chapter 4 a program was specified for drawing different heads as part of a simulated police sketch program. The following function `Hair` comes from one version of this program:

```
void Hair(int choice)
// precondition: 1 <= choice <= 3
// postcondition: prints hair in style specified by choice
{
    if (1 == choice)
    {   cout << "  |||||||///////// " << endl;
    }
    else if (2 == choice)
    {   cout << "  ||||||||||||||||  " << endl;
    }
    else if (3 == choice)
    {   cout << "  |______________|  " << endl;
    }
}
```

The cascaded `if`/`else` statements work well. In some situations, however, an alternative conditional statement can lead to code that is shorter and sometimes more efficient. You shouldn't be overly concerned about this kind of efficiency, but in a program differentiating among 100 choices instead of three the efficiency might be a factor. The **switch** statement provides an alternative method for writing the code in `Hair`.

```
void Hair(int choice)
// precondition: 1 <= choice <= 3
// postcondition: prints hair in style specified by choice
{
    switch(choice)
    {
        case 1:
            cout << "  |||||||///////// " << endl;
            break;
        case 2:
            cout << "  ||||||||||||||||  " << endl;
            break;
        case 3:
            cout << "  |______________|  " << endl;
            break;
    }
}
```

Each **case label**, such as `case 1`, determines what statements are executed based on the value of the expression used in the `switch` test (in this example, the value of the variable `choice`). There should be one case label for each possible value of the switch test expression.

All of the labels are *constants* that represent *integer* values known at compile time. Examples include `13`, `53 - 7`, `true`, and `'a'`. It's not legal to use `double` values like 2.718, `string` values like `"spam"`, or expressions that use variables like `2*choice` for case labels in a `switch` statement.

Syntax: switch statement

```
switch (expression)
{
  case constant₁:
    statement list;
    break;
  case constant₂:
    statement list;
    break;
...
  default :
    statement list;
}
```

If the value of *expression* in the `switch` test matches a case label, then the corresponding statements are executed. The `break` causes flow of control to continue with the statement following the `switch`. If no matching case label is found, the `default` statements, if present, are executed. Most programmers put the default statement last inside a `switch`, but a few argue that it should be the first label. There are no "shortcuts" in forming cases. You cannot write `case 1,2,3:`, for example, to match either one, two, or three. For multiple matches, each case is listed separately as follows:

```
case 1 :
case 2 :
case 3 :
   statement list
   break;
```

In the `switch` statement shown in `Hair`, exactly one `case` statement is executed; the `break` causes control to continue with the statement following the `switch`. (Since there is no following statement in `Hair`, the function exits and the statement after the call of `Hair` is executed next.) In general, a `break` statement is required, or control will **fall through** from one `case` to the next.

Program Tip 7.11: It's very easy to forget the `break` needed for each `case` statement, so when you write `switch` statements, be very careful.

Program Tip 7.12: As a general design rule, don't include more than two or three statements with each `case` label. If more statements are needed, put them in a function and call the function. This will make the `switch` statement easier to read.

Stumbling Block

A missing `break` statement often causes hard-to-find errors. If the `break` corresponding to `case 2` in the function `Hair` is removed, and the value of `choice` is 2, two lines of output will be printed.

(*Warning! Incorrect code follows!*)

```
void Hair(int choice)
// precondition: 1 <= choice <= 3
// postcondition: prints hair in style specified by choice
{
    switch(choice)
    {
        case 1:
            cout << "  ||||||///////// " << endl;
            break;
        case 2:
            cout << "  ||||||||||||||||  " << endl;
        case 3:
            cout << "  |______________|  " << endl;
            break;
    }
}
```

OUTPUT

```
||||||||||||||||
|______________|
```

Because there is no `break` following the hair corresponding to `case 2`, execution falls through to the next statement, and the output statement corresponding to `case 3` is executed.

The efficiency gained from a `switch` statement occurs because only one expression is evaluated and the corresponding `case` statements are immediately executed. In a sequence of `if`/`else` statements it is possible for all the `if` tests to be evaluated. As mentioned earlier, it's not worth worrying about this level of efficiency until you've timed a program and know what statements are executed most often. The `switch` statement does make some code easier to read, and the efficiency gains can't hurt.

7.3.3 A Random Walk Class

Program 7.10, *frogwalk.cpp*, is short. It's not hard to reason that it correctly simulates a one-dimensional random walk. However, modifying the program to have more than one frog hopping on the lily pads is cumbersome because the program is not designed to be extended in this way. If we encapsulate the state and behavior of a random-walking frog in a class, it will be easier to have more than one frog in the same program. With a class we may be able to have different random-walkers jump with different probabilities, that is, one walker might jump left 50% of the time, another 75% of the time. Using a class will also make it easier to extend the program to simulate a two-dimensional walk.

We'll use a class `RandomWalk` whose interface is shown in *walk.h*, Program 7.11. Member functions `Init`, `HasMore`, and `Next` behave similarly to their counterparts in the `WordStreamIterator` class (see Program 6.11, *maxword.cpp*) and the *StringSetIterator* class (see *maxword2.cpp*.) This usage of the **iterator** pattern is somewhat different from what we've used in previous classes and programs, but we use the same names since the random walk is an iterative process. There are two differences in the use of an iterator here.

- The iterator functions are part of the class `RandomWalk` rather than belonging to a separate class. In the other uses the iterator class was separate from the class being iterated over.
- In the `StringSetIterator` and `WordStreamIterator` classes the collection being iterated over was complete when the iterators execute. For the `RandomWalk` class the iterating functions create the random walk — using the functions again results in a different random walk rather than reiterating over the same walk.

Program 7.11 walk.h

```
#ifndef _RANDOMWALK_H
#define _RANDOMWALK_H

// Owen Astrachan, 6/20/96, modified 5/1/99
// class for implementing a one dimensional random walk
//
// constructor specifies number of steps to take, random walk
// goes left or right with equal probability
//
// two methods for running simulation:
//
// void Simulate()  - run a complete simulation
//
// Init(); HasMore(); Next() - idiom for starting and iterating
//                             one step at a time
// accessor functions:
//
// int Position()    - returns x coordinate
//                     (# steps left/right from origin)
// int Current()     - alias for GetPosition()
//
// int TotalSteps()  - returns total # steps taken

class RandomWalk
```

```
{
  public:
    RandomWalk(int maxSteps);  // constructor, parameter = max # steps
    void Init();               // take first step of walk
    bool HasMore();            // returns false if walk finished, else true
    void Next();               // take next step of random walk

    void Simulate();           // take all steps in simulation

    int Position()   const;     // returns position (x coord) of walker
    int Current()    const;     // same as position
    int TotalSteps() const;     // returns total # of steps taken

  private:

    void TakeStep();           // simulate one step of walk
    int myPosition;            // current x coordinate
    int mySteps;               // # of steps taken
    int myMaxSteps;            // maximum # of steps allowed
};

#endif
```

walk.h

A parameter to the `RandomWalk` constructor specifies the number of steps in the walk. A `main` function that uses the class follows.

```
int main()
{
  int numSteps = PromptRange("enter # steps",0,1000000);

  RandomWalk frog(numSteps);
  frog.Simulate();
  cout << "final position = " << frog.GetPosition() << endl;
}
```

In this program an entire simulation takes place immediately using the member function `Simulate`. The output from this program is the same as the output from *frogwalk.cpp*. Using the `RandomWalk` class makes it easier to simulate more than one random walk at the same time. In *frogwalk2.cpp*, Program 7.12, two random walkers are defined. The program keeps track of how many times the walkers are located at the same position during the walk. It would be very difficult to write this program based on *frogwalk.cpp*, Program 7.10. Since the number of steps in the simulation is a parameter to the `RandomWalk` constructor, variables `frog` and `toad` must be defined *after* you enter the number of steps. One alternative would be to have a member function `SetSteps` used to set the number of steps in the simulation.

Program 7.12 frogwalk2.cpp

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

```
#include "prompt.h"
#include "walk.h"

// simulate two random walkers at once
// Owen Astrachan, 6/29/96, modified 5/1/99

int main()
{
     int numSteps = PromptRange("enter # steps",0,30000);

     RandomWalk frog(numSteps);       // define two random walkers
     RandomWalk toad(numSteps);
     int samePadCount = 0;            // # times at same location

     frog.Init();                     // initialize both walks
     toad.Init();

     while (frog.HasMore() && toad.HasMore())
     {   if (frog.Current() == toad.Current())
         {   samePadCount++;
         }
         frog.Next();
         toad.Next();
     }
     cout << "frog position = " << frog.Position() << endl;
     cout << "toad position = " << toad.Position() << endl;
     cout << "# times at same location = " << samePadCount << endl;
     return 0;
}
```

frogwalk2.cpp

Because both random walkers take the same number of steps, it isn't necessary to have checks using both `frog.HasMore()` and `toad.HasMore()`, but since both walkers must be initialized using `Init` and updated using `Next`, we use `HasMore` for both to maintain symmetry in the code.[5]

Reviewing Program Tip 7.1 we find that it's good advice to concentrate first on class methods and behavior, then move to instance variables and state.

[5]Checking both `HasMore` functions will be important if we modify the classes to behave differently. Write programs anticipating that they'll change.

OUTPUT

```
prompt> frogwalk2
enter # steps between 0 and 30000: 10000
frog position = -6
toad position = -26
# times at same location = 87
prompt> frogwalk2
enter # steps between 0 and 30000: 10000
frog position = 16
toad position = 40
# times at same location = 392
```

For `RandomWalk` I first decided to use the iteration pattern of `Init`, `HasMore`, and `Next`. Since it may be useful to execute an entire simulation at once I decided to implement a `Simulate` function to do this. As we'll see, it will be easy to implement this function using the iterating member functions. Finally, the class must provide some accessor functions. In this case we need functions to determine the current location of a `RandomWalk` object and to determine the total number of steps taken.

Determining what data should be private is not always a simple task (see Program Tip 7.6 for some guidance.) You'll often need to revise initial decisions and add or delete data members as the design of the class evolves. As a general guideline, private data should be an intrinsic part of what is modeled by the class. For example, the current position of a `RandomWalk` object is certainly an intrinsic part of a random walk. The `Dice` object used to determine the direction to take at each step is not intrinsic. The state of one `Dice` object does not need to be accessed by different member functions, nor does the state need to be maintained over several invocations of the same function. Even if a `Dice` object is used in several member functions, there is no compelling reason for the same `Dice` object to be used across more than one function.

When you implement a class you should use the same process of iterative enhancement we used in previous programs. For classes this means you might not implement all member functions at once. For example, you could leave a member function out of the public section at first and add it later when the class is partially complete. Alternatively, you could include a declaration of the function, but implement it as an empty **stub function** with no statements.

When I implemented `RandomWalk` I realized that there would be code duplicated in `Init` and `Next` since both functions simulate one random step. Since it's a good idea to avoid code duplication whenever possible, I decided to factor the duplicate code out into another function called `TakeStep` called from both `Init` and `Next`.[6] This kind of **helper function** should be declared in the private section so that it is not accessible to client programs. Member functions, however, can call private helper functions.

[6]Actually, I wrote the code for `Init` and `Next` and then realized it was duplicated after the fact so I added the helper function.

It's not unreasonable to make `TakeStep` public so that client programs could use either the iteration member functions or the `TakeStep` function. Similarly you may decide that the function `Simulate` is superfluous since client programs can implement it by using `Init`, `HasMore`, and `Next` (see Program 7.13, *walk.cpp*). There is often a tension between including too many member functions in an effort to provide as much functionality as possible and too few member functions in an effort to keep the public interface simple and easy to use. There are usually many ways of writing a program, implementing a class, skinning a cat, and walking a frog.

In [Rie96], Arthur Riel offers two design heuristics we'll capture as one programming tip.

Program Tip 7.13: Minimize the number of methods in the interface (protocol) of a class. You should also implement a minimal public interface that all classes understand.

The `RandomWalk` member functions are fairly straightforward. All private data are initialized in the constructor; the function `RandomWalk::TakeStep()` simulates a random step and updates private data accordingly, and the other member functions are used to simulate a random walk or to access information about a walk, such as the current location of the simulated walker. The implementation is shown in Program 7.13.

Program 7.13 walk.cpp

```
#include "walk.h"
#include "dice.h"

RandomWalk::RandomWalk(int maxSteps)
  : myPosition(0),
    mySteps(0),
    myMaxSteps(maxSteps)
// postcondition: no walk has been taken, but walk is ready to go
{
    // work done in initializer list
}

void RandomWalk::TakeStep()
// postcondition: one step of random walk taken
{
    Dice coin(2);
    switch (coin.Roll())
    {
      case 1:
        myPosition--;
        break;
      case 2:
        myPosition++;
        break;
    }
    mySteps++;
}

void RandomWalk::Init()
// postcondition: first step of random walk taken
{
    myPosition = 0;
    mySteps = 0;
    TakeStep();
}

bool RandomWalk::HasMore()
// postcondition: returns true when random walk still going
//                i.e., when # of steps taken < max. # of steps
{
    return mySteps < myMaxSteps;
}

void RandomWalk::Next()
// postcondition: next step in random walk simulated
{
    TakeStep();
}

void RandomWalk::Simulate()
// postcondition: one simulation completed
{
    for(Init(); HasMore(); Next())
    {
    // simulation complete using iterator methods
    }
}

int RandomWalk::Position() const
// postcondition: returns position of walker (x coordinate)
{
    return myPosition;
}

int RandomWalk::Current() const
// postcondition: retrns position of walker (x coordinate)
{
    return Position();
}

int RandomWalk::TotalSteps() const
// postcondition: returns number of steps taken by walker
{
    return mySteps;
}
```

walk.cpp

Each member function requires only a few lines of code. The brevity of the functions makes it easier to verify that they are correct. As you design your own classes, try to keep the implementations of each member function short. Using private helper functions can help both in keeping code short and in factoring out common code.

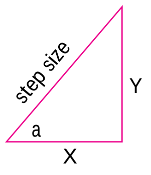
Program Tip 7.14: Use private helper functions to avoid code-duplication in public methods. The helper functions should be private because client programs don't need to know how a class is implemented, and helper functions are an implementation technique.

7.3.4 A Two-Dimensional Walk Class

In this section we'll extend the one-dimensional random walk to two dimensions. A two-dimensional random walk is a more realistic model of a large molecule moving in a gas or liquid, although it is still much simpler than the physical forces that govern molecular motion. Nevertheless, the two-dimensional walk provides insight into the phenomenon known as *Brownian motion*, named after the botanist Robert Brown who, in the early 1800s, investigated pollen grains moving in water. His observations were modeled physically by Albert Einstein, whose hypotheses were confirmed by Jean-Baptiste Perrin, who won a Nobel prize for his work.

The class `RandomWalk2D` models a two-dimensional random walk, the implementation and use of which are shown in *brownian.cpp*, Program 7.14. In two dimensions, a molecule can move in any direction. This direction can be specified by a random number of degrees from the horizontal. A random number between 1 and 360 can be generated by a 360-sided dice. However, using a `Dice` object would constrain the molecule to use a direction that is an integer. We'd like molecules to be able to go in any direction, including angles such as 1.235157 and 102.3392. Instead of using a `Dice` object, we'll use an object from the class *RandGen*, specified in *randgen.h*, Program G.4. Since the sine and cosine functions `sin` and `cos` from `<cmath>` are needed for this simulation, and since these functions require an angle specified in radians[7] rather than degrees, we need to use random `double` values.

The geometry to translate a random direction into x and y distances follows:



cos(a) = X / step size
sin(a) = Y / step size

If a random angle a is chosen, the distance moved in the X-direction is $\cos(a) \times$ step size as shown in the diagram.

[7]There are 360 degrees in a circle and 2π radians in a circle. It's not necessary to understand radian measure, but $180^\circ = \pi$ radians. This means that $d^\circ = d(3.14159/180)$ radians. You can also use conversion functions `deg2rad` and `rad2deg` in *mathutils.h*, Program G.9 in Howto G.

The distance in the Y-direction is a similar function of the sine of the angle a. In the member function `RandomWalk2D::TakeStep()` these properties are used to update the coordinates of a molecule in simulating a two-dimensional random walk. The manner in which a direction is calculated changes in moving from one to two dimensions. We also need to change how a position is stored so that we can track both an x and y coordinate. We could use two instance variables, such as `myXcoord` and `myYcoord`. Instead, we'll use the `Point` class for representing points in two dimensions (the header file *point.h* is Program G.10 in Howto G). As we'll see in Section 7.4, `Point` acts like a class, but is in some ways different because it has public data. These are the principal differences between the class `RandomWalk` and `RandomWalk2D`:

- The implementation of the member function `TakeStep` to cope with a two-dimensional random direction.
- The change of type for instance variable `myPosition` from `int` to `Point` to cope with two dimensions.
- The change in return type for methods `Position` and `Current` from `int` to `Point`.

Program 7.14 brownian.cpp

```
#include <iostream>
#include <cmath>          // for sin, cos, sqrt
#include "randgen.h"
#include "prompt.h"
#include "mathutils.h"    // for PI
#include "point.h"
using namespace std;

// simluate two-dimensional random walk
// Owen Astrachan, 6/20/95, modified 6/29/96, modified 5/1/99

class RandomWalk2D
{
  public:
     RandomWalk2D(long maxSteps,
                  int size);    // # of steps, size of one step
    void Init();                // take first step of walk
    bool HasMore();             // returns false if walk finished, else true
    void Next();                // take next step of random walk
    void Simulate();            // complete an entire random walk

    long  TotalSteps() const;     // total # of steps taken by molecule
    Point Position()   const;     // current position
    Point Current()    const;     // alias for Position

  private:
    void TakeStep();       // simulate one step of walk
    Point myPosition;      // coordinate of current position
    long  mySteps;         // # of steps taken
    int   myStepSize;      // size of step
    long  myMaxSteps;      // maximum # of steps allowed
};

RandomWalk2D::RandomWalk2D(long maxSteps,int size)
  : myPosition(),
    mySteps(0),
    myStepSize(size),
    myMaxSteps(maxSteps)
// postcondition: walker initialized
{

}

void RandomWalk2D::TakeStep()
// postcondition: one step of random walk taken
{
    RandGen gen;                    // random number generator
    double randDirection = gen.RandReal(0,2*PI);

    myPosition.x += myStepSize * cos(randDirection);
    myPosition.y += myStepSize * sin(randDirection);
    mySteps++;
}

void RandomWalk2D::Init()
// postcondition: Init step of random walk taken
{
    mySteps = 0;
    myPosition = Point(0,0);
    TakeStep();
}

bool RandomWalk2D::HasMore()
// postcondition: returns false when random walk is finished
//                i.e., when # of steps taken >= max. # of steps
//                return true if walk still in progress
{
    return mySteps < myMaxSteps;
}

void RandomWalk2D::Next()
// postcondition: next step in random walk simulated
{
    TakeStep();
}

void RandomWalk2D::Simulate()
{
    for(Init(); HasMore(); Next())
    {
        // simulation complete using iterator methods
    }
}
```

```
long RandomWalk2D::TotalSteps() const
// postcondition: returns number of steps taken by molecule
{
     return mySteps;
}

Point RandomWalk2D::Position() const
// postcondition: return molecule’s position
{
   return myPosition;
}

Point RandomWalk2D::Current() const
// postcondition: return molecule’s position
{
   return myPosition;
}

int main()
{
    long numSteps= PromptRange("enter # of random steps",1L,1000000L);
    int stepSize=  PromptRange("size of one step",1,20);
    int trials=    PromptRange("number of simulated walks",1,1000);
    RandomWalk2D molecule(numSteps,stepSize);

    int k;
    double total = 0.0;
    Point p;
    for(k=0; k < trials; k++)
    {
        molecule.Simulate();
        p = molecule.Position();
        total += p.distanceFrom(Point(0,0));  // total final distance from origin
    }
    cout << "average distance from origin = " << total/trials << endl;
    return 0;
}
```

brownian.cpp

OUTPUT

```
prompt> brownian
enter # of random steps between 1 and 1000000: 1024
size of one step between 1 and 20: 1
number of simulated walks between 1 and 1000: 100
average distance from origin = 26.8131
prompt> brownian
enter # of random steps between 1 and 1000000: 1024
size of one step between 1 and 20: 4
number of simulated walks between 1 and 1000: 100
average distance from origin = 108.861
```

If the output of one simulation is printed and used in a plotting program, a graph of the random walk can be made. Two such graphs are shown in Figs. 7.2 and 7.3. Note that the molecule travels in completely different areas of the plane. However, the molecule's final distance from the origin doesn't differ drastically between the two runs. The distance from the origin of a point (x, y) is calculated by the formula $\sqrt{x^2 + y^2}$. The distances are accumulated in Program 7.14 using the method `Point::distanceFrom()` so that the average distance can be output.

The paths of the walk shown in the plots are interesting because they are **self-similar.** If a magnifying glass is used for a close-up view of a particular part of the walk, the picture will be similar to the overall view of the walk. Using a more powerful magnifying glass doesn't make a difference; the similarity still exists. This is a fundamental property of **fractals,** a mathematical concept that is used to explain how seemingly random phenomena aren't as random as they initially seem.

The results of both random walks illustrate one of the most important relationships of statistical physics. In a random walk, the average (expected) distance D from the start of a walk of N steps, where each step is of length L, is given by the following equation:

$$D = \sqrt{N} \times L \tag{7.1}$$

The results of the simulated walks above don't supply enough data to validate this relationship, but the data are supportive. In the exercises you'll be asked to explore this further.

Pause to Reflect

7.25 Modify *frogwalk.cpp*, Program 7.10, so the user enters a distance from the origin—say, 142—and the program simulates a walk until this distance is reached (in either the positive or negative direction). The program should output the number of steps needed to reach the distance.

7.26 Only one simulation is performed in Program 7.10. The code for that one simulation could be moved to a function. Write a prototype for such a function that returns both the final distance from the start as well as the maximum distance from the start reached during the walk.

June 7, 1999 10:10 owltex Sheet number 69 Page number 326 magenta black

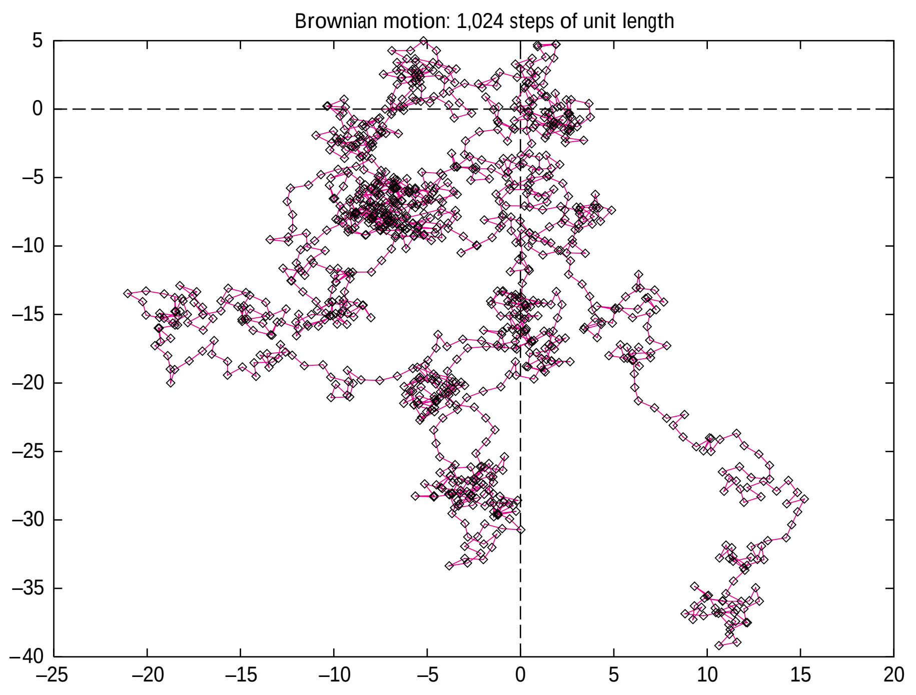


Figure 7.2 Fractal characteristics of two-dimensional random walks.

7.27 Can you find an expression for use in *frogwalk.cpp*, Program 7.10, so that no `switch` or `if/else` statement is needed when the position is updated? For example: `position += die.Roll()` would add either 1 or 2 to the value of `position`. What's needed is an expression that will add either −1 or 1 with equal probability.

7.28 A two-dimensional walk on a lattice constrains the random walker to take steps in the compass point directions: north, east, south, west. How can the class `RandomWalk` be modified to support a frog that travels on lattice points? How can the class `RandomWalk2D` be modified?

7.29 If you modified the random walking classes `RandomWalk2D` and `RandomWalk` with code to track the number of times the walker returned to the starting position, either (0,0) or 0 respectively, would you expect the results to be similar?

7.30 Suppose the one-dimensional walker is restricted to walking in a circle instead of on an infinite line. Outline a modification to the class `RandomWalk` so that the number of "lily pads" on a circle is specified as well as the number of steps in a walk. Strive for a modification that entails minimal change to the class.

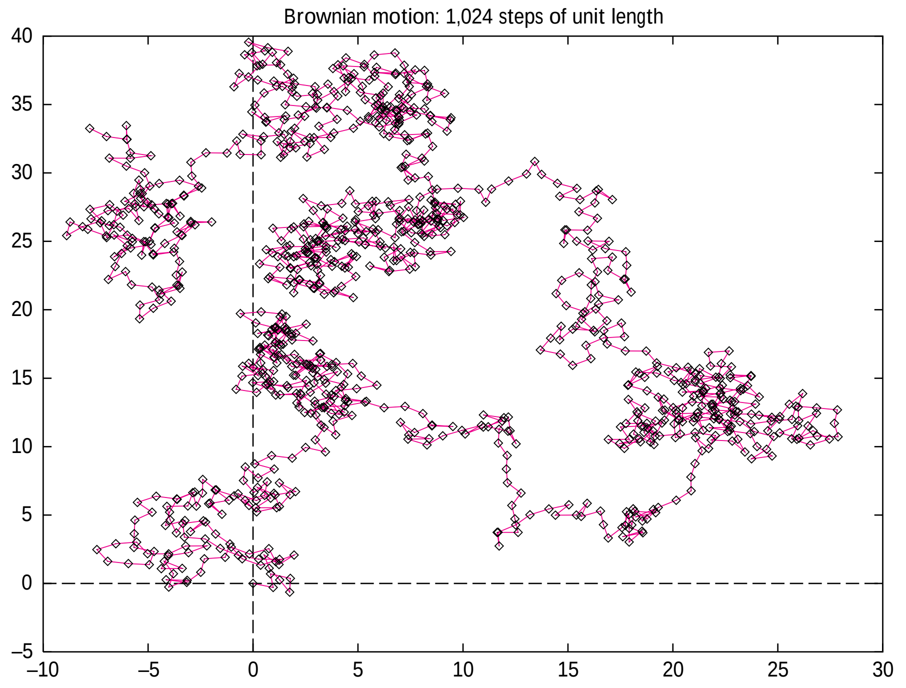


Figure 7.3 Fractal characteristics of two-dimensional random walks (*continued*).

7.3.5 The Common Interface in `RandomWalk` and `RandomWalk2D`

Because the methods of `RandomWalk` and `RandomWalk2D` have the same names, we can modify Program 7.12, `frogwalk2.cpp` very easily. That program keeps track of how many times two walkers have the same position (we used the metaphor of two frogs sharing the same lily pad). The only difference between the one-dimensional walk class declared in *walk.h* and the two-dimensional class whose declaration and definition are both given in *brownian.cpp*, Program 7.14 is that the functions `Current()` and `Position()` return an `int` in the one-dimensional case and a `Point` in the two-dimensional case. As we'll see in Section 7.4, `Point` objects can be compared for equality and printed, so the only change needed to the code in *frogwalk2.cpp* to accommodate two-dimensional walkers is a change in the `#include` from `"walk.h"` to `"walk2d.h"`. Here I'm assuming that the class `RandomWalk2D` has been defined and implemented in .h and .cpp files rather than in *brownian.cpp*. Actually a small change must be made in the constructor calls of `frog` and `toad` since the size of the step is specified for the two-dimensional walkers.

Program 7.15 twodwalk.cpp

```
#include <iostream>
using namespace std;
#include "prompt.h"
#include "walk2d.h"

// simulate two random walkers at once
// Owen Astrachan, 6/29/96, modified 5/1/99

int main()
{
    int numSteps = PromptRange("enter # steps",0,30000);

    RandomWalk2D frog(numSteps,1);  // define two random walkers
    RandomWalk2D toad(numSteps,1);
    int samePadCount = 0;            // # times at same location

    frog.Init();                     // initialize both walks
    toad.Init();

    while (frog.HasMore() && toad.HasMore())
    {   // if (frog.Current() == toad.Current())
        if (frog.Current().distanceFrom(toad.Current()) < 1.0)
        {   samePadCount++;
    }
        frog.Next();
    toad.Next();
    }
    cout << "frog position = " << frog.Position() << endl;
    cout << "toad position = " << toad.Position() << endl;
    cout << "# times at same location = " << samePadCount << endl;
    return 0;
}
```

twodwalk.cpp

OUTPUT

```
prompt> twodwalk
enter # steps between 0 and 30000: 20000
frog position = (138.376, 118.173)
toad position = (59.5489, -61.5897)
# times at same location = 0

prompt> twodwalk
enter # steps between 0 and 30000: 20000
frog position = (-57.0885, 53.7944)
toad position = (-6.07683, 142.7)
# times at same location = 0
```

It's probably not surprising that the two-dimensional walkers never occupy the same position. Even if the walkers are very close to each other it's extraordinarily unlikely that the `double` values representing both *x* and *y* coordinates will be exactly the same. This is due in part to accumulated round-off errors introduced when small `double` values are added together. In general you should avoid comparing `double` values for exact equality, but use a function like `FloatEqual` in *mathutils.h*, Program G.9 and discussed in Howto G.

A simple change in Program 7.15, *twodwalk.cpp*, can track if two walkers are very close rather than having exactly the same position. Using `Point::distanceFrom()` (see Program 7.14, *brownian.cpp*) lets us do this if we change the `if` test as follows.

```
if (frog.Current().distanceFrom(toad.Current()) < 1.0)
```

Two runs with this test show a change in behavior.

OUTPUT

```
prompt> twodwalk
enter # steps between 0 and 30000: 20000
frog position = (-37.9018, 68.9209)
toad position = (-4.6354, 18.2154)
# times at same location = 6

prompt> twodwalk
enter # steps between 0 and 30000: 20000
frog position = (-125.509, 98.8204)
toad position = (82.7206, -24.1438)
# times at same location = 11
```

7.4 structs as Data Aggregates

Suppose you're writing a function to find the number of words in a text file that have fewer than four letters, between four and seven letters, and more than seven letters. The prototype for such a function might be:

```
void fileStats(const string& filename, int& smallCount,
               int& medCount, int& largeCount)
// postcondition: return word counts for text-file filename
//            smallCount  = # words with length() < 4
//            medCount    = # words with 4 <= length() <= 7
//            largeCount  = # words with 7 < length()
```

It's easy to imagine a more lengthy and elaborate set of statistics for a text file; the parameter list for a modified `fileStats` function would quickly become cumbersome.

We could write a class instead, with instance variables recording each count or other statistic. However, if we write a single member function to get all the statistics, we have the same prototype as the function `fileStats` shown above. If we use one member function for each statistic, that quickly gets cumbersome in a different way.

Instead of using several related parameters, we can group the related parameters together so that they can be treated as a single structure. A class works well as a way to group related data together, but if we adhere to the guideline in Program Tip 6.2, all data should be private with public accessor functions when clients need access to some representation of a class' state. Object-oriented programmers generally accept this design guideline and implement accessor and mutator methods for retrieving and updating state data.

Sometimes, rather than using a class to encapsulate both data (state) and behavior, a **struct** is used. In C++ a struct is similar to a class but is used for storing related data together. Structs are implemented almost exactly like classes, but the word `struct` replaces the word `class`. The only difference between a struct and a class in C++ is that by default all data and functions in a struct are public whereas the default in a class is that everything is private. We'll use structs to combine related data together so that the data can be treated as a single unit. A struct used for this purpose is described in the C++ standard as *plain old data*, or *pod.*

In the file statistics example we could use this declaration:

```
struct FileStats
{
    string fileName;   // name of text file
    int    smallCount; // # words with length() < 4
    int    medCount;   // # words with 4 <= length() <= 7
    int    largeCount; // # words with 7 < length()
};
```

Since the combined data have different types, that is `string` and `int`, a struct is often called a **heterogeneous aggregate,** a means of encapsulating data of potentially different types into one new type. As a general design rule we won't require any member functions in a struct and will rely on all data fields being public by default. As we'll see, it may be useful to implement some member functions, including constructors, but we won't insist on these as we do for the design and implementation of a new class. In general, we'll use structs when we want to group data (state) and perhaps some behavior (functions) together, but we won't feel obligated to use the same kinds of design rules that we use when we design classes (e.g., all data are private). You should know that other programmers use structs in a different way and do not include constructors or other functions in structs. Since constructors often make programs shorter and easier to develop without mistakes, we'll use them when appropriate.

Using the struct `FileStats` we might have the following code:

```
void computeStats(FileStats& fs)
// precondition:  fs.fileName is name of a text file
// postcondition: data fields of fs represent statistics
{  // code here
}
int main()
{
    FileStats fs;
    fs.fileName = "poe.txt";
    computeStats(fs);
    cout << "# large words in " << fs.fileName
         << " = " << fs.largeCount << endl;
    return 0;
}
```

Program Tip 7.15: If you're designing a class with little or no behavior, but just data that are accessed and modified, consider implementing the class as a struct. A class should have behavior beyond setting and retrieving the value of each instance variable. Using structs for encapsulating data (with helper functions when necessary, such as for construction and printing) is a good compromise when development of a complete class seems like overkill.

7.4.1 structs for Storing Points

We've used objects of type Point in programs for simulating two-dimensional random walks (see Program 7.14, *brownian.cpp*, for an example.) The type Point declared in *point.h*, Program G.10 in Howto G is implemented as struct rather than a class. With our design guidelines, a struct allows us to make the data public. For Point the data are x and y coordinates. Using a struct means we don't need to provide methods for getting and setting the coordinates, but can access them directly as shown in Program 7.16, *usepoint.cpp*.

Program 7.16 usepoint.cpp

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

#include "point.h"

int main()
{
    Point p;
    Point q(3.0, 4.0);

    // print the points
    cout << "p = " << p << " q = " << q << endl;

    q.x *= 2;
```

```
        q.y *= 2;
        cout << "q doubled = " << q << endl;

        p = q;
        if (p == q)
        {   cout << "points are now equal" << endl;
        }
        else
        {   cout << "points are NOT equal" << endl;
        }

        p = Point(0,0);
        cout << q.distanceFrom(p) << " = distance of q from " << p << endl;

        return 0;
    }
```

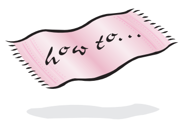
usepoint.cpp

OUTPUT

```
prompt> usepoint
p = (0, 0) q = (3, 4)
q doubled = (6, 8)
points are now equal
10 = distance of q from (0, 0)
```

The data members of the structs p and q are accessed with a dot notation just as member functions of a class are accessed. However, because the data fields are public, they can be updated and accessed without using member functions. Sometimes the decision to use either a struct, or several variables, or a class will not be simple. Using a struct instead of several variables makes it easy to add more data at a later time.

Program Tip 7.16: Be wary when you decide to use a struct rather than a class. When you use a struct, client programs will most likely depend directly on the implementation of the struct rather than only on the interface. If the implementation changes, all client programs may need to be rewritten rather than just recompiled or relinked as when client programs use only an interface rather than direct knowledge of an implementation.



If you reason carefully about the output from *usepoint.cpp* you'll notice several properties of Point. You can verify some of these by examining the header file *point.h* in Howto G.

- The default (parameterless) Point constructor initializes to the origin: (0,0).

- Point objects can be assigned to each other, as in `p = q`, and compared for equality, as in `if (p == q)`.
- A **temporary** (or **anonymous**) Point object can be created by calling a constructor and using the constructed Point in a statement. The following statement from *usepoint.cpp* creates a temporary Point representing the origin (0,0) and assigns it to p.

  ```
  p = Point(0,0);
  ```

  A temporary is also used in *brownian.cpp*, Program 7.14 to compute a walker's final distance from the origin.
- The method `Point::distanceFrom()` computes the distance of one point from another.

7.4.2 Operators for structs

In the programs using Point objects we printed points and compared them for equality. These operations are possible on Point objects because the relational operators and the output insertion `operator <<` are **overloaded** for the class Point. The definition of the overloaded stream insertion operator is shown below (see also *point.cpp* with the files provided for use with this book.)

```
ostream& operator << (ostream& os, const Point& p)
// postcondition: p inserted on output as (p.x,p.y)
{
    os << p.tostring();
    return os;
}
```

The parameter output represents any output stream, that is, either cout or an ofstream object. After the point p is inserted onto stream output, the stream is returned so that a chain of insertions can be made in one statement as shown in *usepoint.cpp*. A full description of how to overload the insertion operator and all other operators is found in Howto E

Program Tip 7.17: Many classes should have a member function named tostring that produces a representation of the class as a string. Using the tostring method makes it very simple to overload the stream insertion operator, but is also useful in other contexts.

If you use the graphics package associated with this book you'll probably use the tostring method to "print" on the graphics screen since the screen displays strings, but not streams.

As another example, here is the relational operator == for Point objects.

```
bool operator == (const Point& lhs, const Point& rhs)
{
    return lhs.x == rhs.x && lhs.y == rhs.y;
}
```

Note that the prototype for this function is declared in *point.h*, but the definition above is found in *point.cpp* just as methods are declared in a header file and implemented in the corresponding .cpp file.

Program Tip 7.18: When possible, design a class to behave as users will expect from the behavior of built-in types like `int` and `double`. This often means overloading relational operators, the stream insertion operator, and ensuring that objects can be assigned to each other.

As we'll see in Howto E and study in later chapters, overloaded operators can make the syntax of developing programs with new classes much simpler than if no overloaded operators were implemented.

7.5 Chapter Review

- The first step in developing programs and classes is to develop a specification and a list of requirements.
- Nouns in a problem statement or specification help identify potential classes; verbs help identify potential methods.
- When designing and implementing classes, first concentrate on behavior (methods), then concentrate on state (private data).
- Use scenarios to help develop classes and programs.
- Use stub functions when you want to test a class (or program) without implementing all the functions at once. Test classes in isolation from each other whenever possible.
- Factor out common code accessed by more than one function into another function that is called multiple times. For member functions, make these helping functions private so that they can be called from other member functions but not from client programs.
- Try to keep the bodies of each member function short so that the functions are easy to verify and modify.
- Functions `atoi`, `atof` allow conversion from strings to ints and doubles, respectively. These and the overloaded function `tostring` to convert from ints and doubles to strings are found in `strutils.h`.
- Keep classes to a single purpose. Use more than one class rather than combining different or unrelated behaviors in the same class.
- Program by conforming to known interfaces whenever possible. This reduces both

conceptual hurdles and potential recompilation and relinking.

- Creating a program from source files in C++ consists of preprocessing, compiling, and linking. Libraries are often accessed in the linking phase.
- Compilation errors and linking errors have different causes.
- Initializer lists are used to construct private data in a class. You should use initializer lists rather than assigning values in the body of a constructor.
- Random walks are useful models of many natural phenomenon with a basis in mathematics and statistical physics.
- The `switch` statement is an alternative to cascaded `if/else` statements.
- Structs are used as heterogeneous aggregates. When related data should be stored together without the programming and design overhead of implementing a class, structs are a useful alternative. Structs are classes in which the data are public by default. Structs can also have constructors and helper functions to make them easier to use.
- The insertion operator can be overloaded for programmer-defined types as can relational operators.

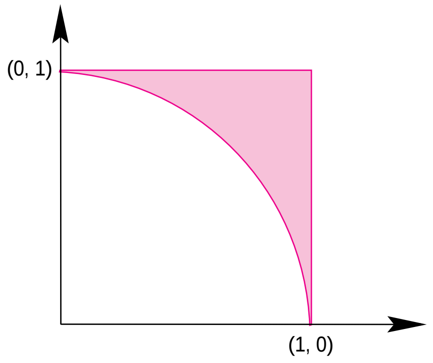
7.6 Exercises

7.1 Write a quiz program similar to *quiz.cpp*, Program 7.8, but using different levels of mathematical drill questions. Give the user a choice of easy, medium, or hard questions. An easy question involves addition of two-digit numbers, but no carry is required, so that `23 + 45` is ok, but `27 + 45` is not. A medium question uses addition, but a carry is always required. A hard question involves multiplication of a two-digit number by a one-digit number, but the answer must be less than one-hundred.

7.2 Modify Program 7.10, *frogwalk.cpp*, to keep track of all the locations that are visited more than once, not just the number of times the walkers are at the same location. To do this, use a `StringSet` object (see Programs 6.14 and 6.15 in Section 6.5.) Use the functions `tostring` from *strutils.h* to convert walker positions to strings so that they can be stored in a `StringSet`.
Then change the program so that two two-dimensional walkers are used as in *twodwalk.cpp*. You'll need to use `Point::tostring()` to store two-dimensional locations in a `StringSet`.

7.3 A result of Dirichlet (see [Knu98a], Section 4.5) says that if two numbers are chosen at random, the probability that their greatest common divisor equals 1 is $6/\pi^2$. Write a program that repeatedly chooses two integers at random and calculates the approximation to π. For best results use a `RandGen` variable `gen` (from *randgen.h*) and generate a random integer using `gen.RandInt(1,RAND_MAX)`.

7.4 A reasonable but rough approximation of the mathematical constant π can be obtained by simulating throwing darts. The simulated darts are thrown at a dart board in the shape of a square with a quarter-circle in it.



If 1000 darts are thrown at the square, and 785 land in the circle, then 785/1000 is an approximation for $\pi/4$ since the area of the circle (with radius 1) is $\pi/4$. The approximation for π is $4 \times 0.785 = 3.140$. Write a program to approximate π using this method. Use a unit square as shown in the figure, with corners at (0,0), (1,0), (1,1), and (0,1). Use the `RandGen` class specified in *randgen.h* and the member function `RandReal`, which returns a random `double` value in the range $[0 \, .. \, 1)$. For example, the following code segment generates random x and y values inside the square and increments a counter `hits` if the point (x, y) lies within the circle.

```
x = gen.RandReal();
y = gen.RandReal();

if (x*x + y*y <= 1.0)
{   hits++;
}
```

This works because the equation of the unit circle is $x^2 + y^2 = 1$. Allow the user to specify the number of darts (random numbers) thrown or use a varying number of darts to experiment with different approximations. This kind of approximation is called a **Monte Carlo** approximation.

7.5 This problem (adapted from [BRE71]) is a simplistic simulation of neutrons in a nuclear reactor. Neutrons enter the lead wall of a nuclear reactor and collide with the lead atoms in the wall. Each time a neutron collides with a lead atom it rebounds in a random direction (between 0 and 2π radians) before colliding with another lead atom, reentering the reactor, or leaving the wall. To simplify the simulation we'll assume that all neutrons enter the wall at a right angle; each neutron travels a distance d before either colliding, reentering the reactor, or leaving the wall; and the wall is $3d$ units thick. Figure 7.4 diagrams a wall; the reactor is at the bottom. The neutron at the left reenters the reactor, the neutron in the middle leaves the wall outside the reactor, and the neutron on the right is absorbed in the wall (assume that after 10 collisions within the wall a neutron is absorbed).

If p is the depth of penetration inside the wall, then p is changed after each collision by `p += d * cos(angle)` where `angle` is a random angle (see *brownian.cpp*, Program 7.14). If $p < 0$, then the neutron reenters the reactor, and if $3d < p$, then the

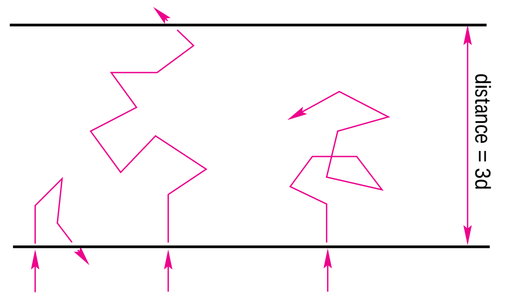


Figure 7.4 Collisions in a nuclear reactor.

neutron leaves the wall; otherwise it collides with another lead atom or is absorbed. Write a program to simulate this reactor. Use 10,000 neutrons in the simulation and determine the percentage of neutrons that return to the reactor, are absorbed in the wall, and penetrate the wall to leave the reactor. Use the simulation to determine the minimal wall thickness (as a multiple of d) required so that no more than 5% of the neutrons escape the reactor. To help test your simulation, roughly 26% of the neutrons should leave a $3d$-thick wall and roughly 22% should be absorbed.

7.6 Repeat the simulation from the previous exercise but assume the neutrons enter the wall at a random angle rather than at a right angle. Then implement a neutron observer class that records the movements of a neutron. Record the motion of 10 neutrons and graph the output if you have access to a plotting program.

7.7 Write a program to test the relationship $D = \sqrt{N} \times L$ from statistical physics, described in Section 7.10. Use a one-dimensional random walk and vary both the length of each step L and the number of steps N. You'll need to run several hundred experiments for each value; try to automate the process.
If you have access to a graphing program, graph the results. If you know about curve fitting, try to fit a curve to the results and see if the empirical observations match the theoretical equation. You can repeat this experiment for the two-dimensional random walk.

7.8 Write a program for two-dimensional random walks in which two frogs (or two molecules) participate at the same time. Keep track of the closest and furthest distances the molecules are away from each other during the simulation. Can you easily extend this to three frogs or four frogs?

7.9 Write a program to simulate a roulette game. In roulette you can place bets on which of 38 numbers is chosen when a ball falls into a numbered slot. The numbers range from 1 to 36, with special 0 and 00 slots. The 0 and 00 slots are colored green; each of the numbers 1 through 36 is red or black. The red numbers are 1, 3, 5, 7, 9, 12, 14, 16, 18, 19, 21, 23, 25, 27, 30, 32, 34, and 36. Gamblers can make several different kinds of bet, each of which pays off at different odds as listed in Table 7.1. A payoff of 1 to 1 means that a \$10.00 bet earns \$10.00 (plus the bet \$10.00 back); 17 to 1 means that a \$10.00 bet earns \$170.00 (plus the \$10.00 back). If the wheel spins 0 or 00, then all bets lose

Table 7.1 Roulette bets and payoff odds

bet	**payoff odds**
red/black	1 to 1
odd/even	1 to 1
single number	35 to 1
two consecutive numbers	17 to 1
three consecutive numbers	11 to 1

except for a bet on the single number 0/00 or on the two consecutive numbers 0 and 00. You may find it useful to implement a separate `Bet` class to keep track of the different kinds of bets and odds. For example, when betting on a number, you'll need to keep track of the number, but betting on red/black requires only that you remember the color chosen.

7.10 Design and implement a struct for representing points in three dimensions. Then program a random walk in three dimensions and determine how often two walkers are within 10 units of each other. Use a class `RandomWalk3D` patterned after the class `RandomWalk2D` in *brownian.cpp*, Program 7.14.