Chapter 1 Designing Interfaces: A Random Number Library

AND SUCH A WALL AS I WOULD HAVE YOU THINK THAT HAD IN IT A CRANNIED HOLE, OR CHINK THROUGH WHICH THE LOVERS, PYRAMUS AND THISBE DID WHISPER OFTEN, VERY SECRETLY

— Shakespeare, A Midsummer Night's Dream, 1595-1596

OBJECTIVES

- To appreciate the principal criteria used to evaluate the design of an interface.
- To discover how paograms can simulate random behavior through the use of pseudo-random
- To understand the behavior of the library function `rand`.
- To learn how you can use arithmetic operations to change the range of the pseudo-random number sequence.
- To recognize the common types of interface entries.
- To learn the syntactic rules and conventions required to write an interface header file.
- To be able to use the facilities provided by the `random.h` interface.

In Chapter 7, you introduced to the concept of an interface. Moreover, working with the `graphics.h` interface gave you a chance to think about what goes into an interface and how to use one in your programming. But to understand interfaces fully, you must also learn how to implement them. This chapter gives you a chance to design a new interface together with its underlying implementation.

Depending on how broadly you view the problem, writing an interface can be either very simple or extremely challenging. If you consider only C's syntax and structure, there are not many new rules to learn. You already know how to write comments and function prototypes, which are the principal components of an interface. As is true with algorithms, however, the challenge comes not in coding the interface but in designing it. The important question is not so much how to write an interface but rather how to write a good one.

Designing a good interface is a subtle problem that requires you to balance many competing design criteria. This chapter examines those criteria and illustrates their application. To make the illustrations concrete, this chapter also walks you through the development of a library package that provides access to a simple random number abstraction.

1-1 Interface design

Programming is hard because programs reflect the complexity of the problems they solve. As long as we use computers to solve problems of ever-increasing sophistication, the process of programming will need to become more sophisticated as well.

Writing a program to solve a large or difficult problem forces you to manage an enormous amount of complexity. There algorithms to design, special cases to consider, user requirements to meet, and innumerable details to get right. To make programming manageable, you must reduce the complexity of the programming process as much as possible.

In Chapter 5, you learned how to use functions and procedures to reduce some of the complexity. Interfaces offer a similar reduction in programming complexity but at a higher level of detail. A function gives its caller access to a set of steps that together implement a single operation. An interface gives its client access to a set of functions that together implement a programming abstraction. The extent to which the interface simplifies the programming process, however, depends largely on how well it is designed.

To design an effective interface, you must balance several criteria. In general, you should try to develop interfaces that are

- *Unified.* A single interface should define a consistent abstraction with a clear unifying theme. If a function does not fit within that theme, it should be defined in a separate interface.
- *Simple.* To the extent that the underlying implementation is itself complex, the interface must seek to hide that complexity from the client.
- *Sufficient.* When clients use an abstraction, the interface must provide sufficient functionality to meet their needs. If some critical operation is missing from an interface, clients may decide to abandon it and develop their own, more powerful abstraction. As important as simplicity is, the designer must avoid simplifying an interface to the point that it becomes useless.
- *General.* A well-designed interface to should be flexible enough to meet the needs of many different clients. An interface that performs a narrowly defined set of operations for one client is not as useful as one that can be used in many different situations.
- *Stable.* The functions defined in an interface should continue to have precisely the same structure and effect, even if the underlying implementation changes. Making changes in the behavior of an interface forces clients to change their programs, which compromises the value of interface.

The sections that follow discuss each of these criteria in detail.

The importance of a unifying theme

Unity gives strength.

— **Aesop, The Bundle of Sticks, sixth century B.C.**

A central feature of a well-designed interface is that it presents a unified and consistent abstraction. In part, this criterion implies that the functions within a library should be chosen so that they reflect a coherent theme. For example, the math library consists of mathematical functions, the standard I/O library provides functions to perform input and output, and the graphics library provides functions for drawing pictures on the screen. Each function exported by these interfaces fits the purpose of that interface. For example, you would not expect to find `sqrt` in the `graphics.h` interface, even though graphical applications will often call `sqrt` to compute the length of a diagonal line. The `sqrt` function fits much more naturally into the framework of the math library.

The principle of a unifying theme also influences the design of the functions within a library interface. The functions within an interface should behave in as consistent a way as possible. Differences in the ways its functions work make using an interface much harder for the client. For example, all the functions in the graphics library use coordinates specified in inches and angles specified in degrees. If the implementor of the library had decided to add a function that required a different unit of measurement, clients would have to remember what units to use for each function. Similarly, the functions `DrawLine` and `DrawArc` in the graphics library were each designed so that drawing begins at the current position of the pen. Doing so means that the underlying conceptual model has a consistent structure that makes it easier to understand the library and its operation.

Simplicity and the principle of information hiding

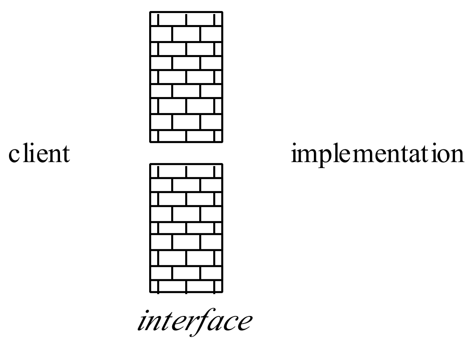
Embrace simplicity.

— **Lao-tzu, The Way of Lao-tzu, ca. 550 B.C**

Because a primary goal of using interfaces is to reduce the complexity of the programming process, it makes sense that simplicity is a desirable criterion in the design of an interface of an interface. In general, an interface should be as easy to use as possible. The underlying implementation may perform extremely intricate operations, but the client should nonetheless be able to think about those operations in a simple, more abstract way.

To a certain extent, an interface acts as a reference guide to a particular library abstraction. When you want to know how to use the math library, you go to the `math.h` interface to find out how to do so. The interface contains precisely the information that you, as a client, need to know—and no more. For clients, getting too much information can be as bad as getting to little, because additional detail is likely to make the interface more difficult to understand. Often, the real value of an interface lies not in the information it reveals but rather in the information it *hides*.

When you design an interface, you should try to protect the client from as many of the complicating details of the implementation as possible. In that respect, it is perhaps best to think of an interface not primarily as a communication channel between the client and the implementation, but instead as a wall that divides them.



Like the wall that divided the lovers Pyramus and Thisbe in Greek mythology, the wall representing an interface has a small chink that allows the client and the implementation th communicate. The main purpose of the wall, however, is to keep the two sides apart. Because we conceive of it as lying at the border of the abstraction represented by the library, an interface is sometimes called an **abstraction boundary**. Ideally, all the complexity involved in the realization of a library lies on the implementation side of the wall. The interface is successful if it keeps that complexity away from the client side. Keeping details confined to the implementation domain is called **information hiding**.

The principle of information hiding has important practical implications for interface design. When you write an interface, you should be sure you don't reveal details of the implementation, even in the commentary. Especially if you are writing an interface and an implementation at the same time, you may be tempted to document in your interface all the clever ideas you used to write the implementation. Try to resist that temptation. The interface is written for the benefit of the client and should contain only what the client needs to know.

Similarly, you should design the functions in an interface so that they are as simple as possible. If you can reduce the number of arguments or find a way to eliminate confusing special cases, it will be easier for the client to understand how to use those functions. Moreover, it is usually good practice to limit the total number of functions exported by interface, so that the client does not become lost in a mass of functions, unable to make sense of the whole.

Meeting the needs of your clients

Everything should be as simple as possible, but no simpler.

— **attributed to Albert Einstein**

Simplicity is only part of the story. You can easily make an interface simple just by throwing away any parts of it that are hard or complicated. There is a good chance you will also make the interface useless. Sometimes clients need to perform tasks that have some inherent complexity. Denying your clients the tools they require just to make the interface simpler is not an effective strategy. Your interface must provide sufficient functionality to serve the clients' needs. Learning to strike the right balance between simplicity and completeness in interface design is one of the fundamental challenges in programming.

In many case, the clients of an interface are concerned not only with whether a particular function is available but also with the efficiency of the underlying implementation. For example, if a programmer is developing a system for air-traffic control and needs to call functions provided by a library interface, those functions must return the correct answer quickly. Late answers may be just as devastating as wrong answers.

For the most part, efficiency is a concern for the implementation rather than the interface. Even so, you will often find it valuable to think about implementation strategies while you are designing the interface itself. Suppose, for example, that you are faced with a choice of two designs. If you determine that one of them would be much easier to implement efficiency, it makes sense—assuming there are no compelling reasons to the contrary—to choose that design.

The advantages of general tools

Give us the tolls and we will finish the job.

— Winston Churchill, radio address, 1941

An interface that is perfectly adapted to a particular clients's needs may not be useful to others. A good library abstraction serves the needs of many different clients. To do so, it must be general enough to solve a wide range of problems and not be limited to one highly specific purpose. By choosing a design that offers your clients flexibility in how they use the abstraction, you can create interfaces that are widely used.

The desire to ensure that an interface remains general has an important practical implication. When you are writing a program, you will often discover that you need a particular tool. If you decide that the tool is important enough to go into a library, you then need to change your mode of thought. When you design the interface for that library, you have to forget about the application that cause you to want the tool in the first place and instead design such a tool for the most general possible audience.

You encountered the need for this shift in perspective in the section on "Looking for common patterns" in Chapter 7. From the perspective of a client, you needed a function to draw windows for a house. To build the tool, however, you had to think more generally. The result was the function `DrawGrid`, which can be used in many deferent situations.

The value of stability

People change and forget to tell each other. Too bad—causes so many mistakes

— Lillian Hellman, Toys in the Attic, 1959

Interfaces have another property that makes them critically important to programming: they tend to be stable over long periods of time. Stable interfaces can dramatically simplify the problem of maintaining large programming systems by establishing clear boundaries of responsibility. As long as the interface does not change, both implementors and clients are relatively free to make changes on their own side of the abstraction boundary.

For example, suppose that you are the implementor of the math library. In the course of your work, you discover a clever new algorithm for calculating the sqrt function that cuts in half the time required to calculate a square root .If you can say to your clients that you have a new implementation of sqrt that works just as it did before, only faster, they will probably be pleased. If, on the other hand, you were to say that the name of the function had changed or that its use involved certain new restrictions, your clients would be justifiably annoyed. To use your “improved” implementation of square root, they would be forced to change their programs. Changing programs is a time-consuming, error-prone activity, and many clients would happily give up the extra efficiency for the convenience of being able to leave their programs alone.

Interface, however, simplify the task of program maintenance only if they remain stable. Programs change frequently as new algorithms are discovered or as the requirements of applications change. Throughout such evolution, however, the interfaces must remain as constant as possible. In a well-designed system, changing the details of an implementation is a straightforward process. The complexity involved in making that change is localized on the implementation side of the abstraction boundary. On the other hand, changing an interface often produces a global upheaval that requires changing every program that depends on it. Thus, interface changes should be undertaken very rarely and then only with the active participation of clients.

Some interface changes, however, are more drastic than others. For example, adding an entirely new function to an interface is usually a relatively straightforward process, since no clients already depend on that function. Changing an interface in such a way that existing programs will continue to run without modification is called **extending** the interface. If you find that you need to make evolutionary changes over the lifetime of an interface, it is usually best to make those changes by extension.

1-2 Generating random numbers by computer

To illustrate the foregoing principles of interface design, the rest of this chapter focuses on the problem of how to write programs that make seemingly random choices. Being able to simulate random behavior is necessary, for example, if you want to write a computer game that involves flipping a coin or rolling a die, but is also useful in more practical contexts.

Getting programs to behave in a random way involves a certain amount of complexity. For the benefit of client programmers, you want to hide that complexity behind an interface. In this chapter, you will have the opportunity to focus your attention on that interface from each of the possible perspectives—those of the interface designer, the implementor, and the client.

Deterministic versus nondeterministic behavior

Until now, all programs described in this text have behaved **deterministically**, which means that their actions are completely predictable given any set of input values. The behavior of such programs is repeatable. If a program produces one result when you run it today, it will produce the same result tomorrow.

In some programming applications, such as games or simulations, it is important that the behavior of your programs not be so predictable. For example, a computer game that always had the same outcome would be boring. In order to build a program that behaves randomly, you need some mechanism for representing a random process, such as flipping a coin or tossing a die, in the context of your programs. Programs that simulate such random events are called **nondeterministic** programs.

Random versus pseudo-random numbers

Partly because early computers were used primarily for numerical applications, the idea of generating randomness using a computer is often expressed in terms of being able to generate a **random number** in a particular range. From a theoretical perspective, a number is random if there is no way to determine in advance what value it will have among a set of equally probable possibilities. For example, rolling a die generates a random number between 1 and 6. If the die is fair, there is no way to predict which number will come up. The six possible values are equally likely.

Although the idea of a random number makes intuitive sense, it is a difficult notion to represent inside a computer. Computers operate by following a sequence of instructions in memory and therefore function in a deterministic mode. How is it possible to generate unpredictable results by following a deterministic set of rules? If a number is generated by a deterministic process, any user should be able to work through that same set of rules and anticipate the computer's response.

Yet computers do in fact use a deterministic procedure to generate what we call random numbers. This strategy works because, even though the user could, in theory, follow the same set of rules and anticipate the computer's response, no one actually bothers to do so. In most practical applications, it doesn't matter if the numbers are truly random; all that matters is that the numbers appear to be random. For numbers to appear random, they should (1) behave like random numbers form a statistical point of view and (2) be sufficiently difficult to predict in advance that no user would bother. "Random" numbers generated by an algorithmic process inside a computer are referred to as **pseudo-random numbers** to underscore the fact that no truly random activity is involved.

Generating pseudo-random numbers in ANSI C

The ANSI C library includes a function `rand` that produces pseudo-random numbers as part of the `stdlib.h` interface. The prototype for `rand` as given in the interface is

```
int rand (void)
```

which indicates that `rand` takes no arguments and returns an integer that is a pseudo-random value—a different result is returned on each call to rand. The result of rand is guaranteed to be nonnegative and no larger than the constant RAND-MAX, which is also defined in the stdlib.h interface. Thus each time rand is called, it returns a different integer between 0 and RAND_MAX, inclusive.

The value of RAND_MAX, depends on the computer system. In the typical Macintosh environment, RAND_MAX is 32,767. On a typical Unix workstation, it is 2,147,483,647. When you write programs that work with random numbers, you should not make any assumptions about the precise value of RAND_MAX. Instead, your programs should be prepared to use whatever value of RAND_MAX the system defines. If you are careful in doing so, you can take a program that works on one system and recompile it so that it works on another.

Running the program randtest.c given in Figure 8-1 shown how rand behaves.

FIGURE 8-1 randtest.c

```
/*
* File: randtest.c
* -----------------
* This program tests the ANSI rand function.
*/

#include <stdio.h>
#include <stdlib.h>
#include "genlib.h"

/*
* Constants
* -----------
* Ntrials – Number of trials
*/

#define NTrials 10

/* Main program */

main ()
{
    int i, r;
    printf("On this computer, RAND_MAX = %d.\n", RAND_MAX);
    printf("Here are the results of %d calls to rand:\n", NTrials);
    for (i = 0; i < NTrials; i++) {
            r = rand ();
            printf ("%10d\n", r);
    }
}
```

On the computer in my office, randtest.c generates the following output:

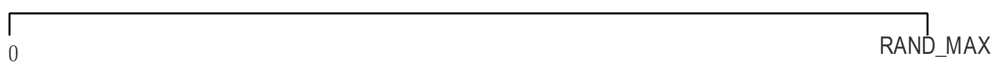
```
On this computer, RAND_MAX = 32767.
Here are the results of 10 calls to rand:
         346
         130
       10982
        1090
       11656
        7117
       17595
        6415
       22948
       31126
```

You can see that the program is generating numbers, all of which are positive and none of which is greater than 32,767, which the sample run shows as the value of RAND_MAX for this computer system. Because these are pseudo-random numbers, you know that there must be some pattern, but it is unlikely that you can discern one. From your point of view, the numbers appear to be random, because you don't know what the underlying pattern is.

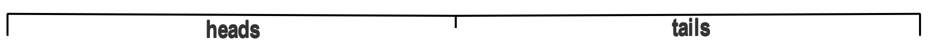
Changing the range of random numbers

The rand library function gives you a mechanism for generating pseudo-random numbers, but it rarely gives you precisely the range of values you need to fit a particular application. It generates numbers that are uniformly distributed over the range between 0 and RAND_MAX. Depending on your application, you are likely to want is a number that falls in some other range, usually much smaller. For example, if you are trying to simulate flipping a coin, you need to convert this large range of random number possibilities into a range containing only two outcomes: heads and tails. Similarly, if you are trying to represent rolling a die, then you need to convert the pseudo-random number returned by rand into numbers between 1 and 6, inclusive.

To make this sort of conversion, you need to reinterpret each random number produced by rand so that it covers a different range. The rand function generates numbers that lie somewhere on the number line between 0 and RAND_MAX:



If you want to simulate a coin toss, you can divide this line up so that half of it represents heeds and the other half represents tails:



You could easily use this insight to develop the cointest.c program shown in Figure 8-2, which simulates tossing a coin.

FIGURE 8-2 cointest.c

```
/*
```

```
* File: cointest.c
* ------------------
* This program simulates flipping a coin.
*/

#include <stdio.h>
#include <stdlib.h>
#include "genlib.h"

/*
* Constants
* -----------
* Ntrials – Number of trials
*/

#define NTrials 10

/* Main program */

main ()
{
        int i;

        for (i = 0; i < NTrials; i++) {
                if (rand () <= RAND_MAX / 2) {
                        printf ("Heads\n");
                } else {
                        printf ("Tails\n");
                }
        }
}
```

COMMON PITFALLS
When converting the result of rand to a more restricted range of integers, do not try to use the remainder operator. The only random property that you are allowed to count on when using rand is the position of the result along the

This program prints out either the string "Heads" or the string "Tails", with each outcome occurring approximately 50 percent of the time,. If you test the program, you get the following sample run:

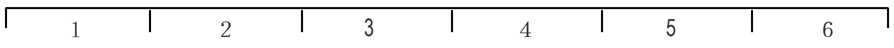
```
Heads
Heads
Heads
Heads
Heads
Heads
Tails
```

There is a reasonable mixture of heads and tails, and you can discern no easily detectable pattern.

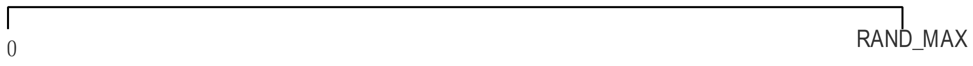
In thinking about how to convert the result of rand into two possibilities, many new programmers may be tempted to adopt what seems initially like a simpler approach—using the remainder operator. If you divide the result of rand by 2 and take the remainder, the result is either 0 or 1. In a program, you could define 0 to be heads and 1 to be tails. This strategy is dangerous because there is no guarantee that the result of rand will be randomly distributed between even and odd numbers. The only guarantee is that the magnitude of the result will be randomly distributed along the number line between 0 and RAND-MAX.

One common implementation of rand provides a vivid illustration of how serious an error this approach to generating random numbers can be. On many computer systems, the rand function is implemented in such a way that the result alternates between even and odd values. The results are still randomly scattered on the number line in terms of how far along the line they fall. Even so, a program that uses the remainder operator to simulate a coin flip ends up generating heads and tails in a strictly alternating pattern.

What about simulating a die roll? If you use the strategy of the cointest.c example, all you need to do is overlay the outcomes



on the number line



Suppose you tried to handle this task in a brute-force way be following the structure of the cointest.c example. The result of doing so is shown in Figure 8-3.

FIGURE 8-3 First attempt at RollDie

```
int RollDie (void)
{
        if (rand () < RAND_MAX/ 6) {
                return (1);
        } else if (rand () < RAND_MAX* 2 / 6) {
                return (2);
        } else if (rand () < RAND_MAX* 3 / 6) {
                return (3);
        } else if (rand () < RAND_MAX* 4 / 6) {
                return (4);
        } else if (rand () < RAND_MAX* 5 / 6) {
                return (5);
        } else {
                return (6);
        }
}
```

This implementation contains several errors

Unfortunately, this implementation of the RollDie function has a few serious problems. Because they are the sort of problems you might run into in your own coding, they are worth considering closely.

The first problem in the code is that you have made an assumption that was easy and natural to make although nonetheless unwarranted in the context[1]. The program was supposed to express the following English idea:

- If the random number generated is less than 1/6 of the maximum, return the value1.
- Otherwise, if the number is less than 2/6 of the maximum, return the value 2.
- Otherwise, if the number is less than 3/6 of the maximum, return the value 3, and so on.

[1] In his book, Zen and the Art of Motorcycle Maintenance, Robert Pirsig calls this sort of error—one in which a seemingly reasonable assumption leads to false conclusions—a gumption trap. Gumption traps come up often in programming, and Pirsig's book offers at least as many useful insights to debugging programs as it does to repairing motorcycles.

The problem is that your code doesn't quite capture this idea. By repeatedly calling the function rand, you will generate a new random number in each of the if statements. The structure of the function depends on the assumption that the random number remains the same each time. To see that this is in fact the case, look at the buggy implementation of RollDie and try to understand under what conditions it will rerun the answer 2. In order for the function to return 2, the first if statement must come out FALSE and the second one must come out TRUE. The condition in the first if statement is FALSE five times out of six. The second if statement is written so that it returns TRUE one third of the time, because the call to rand returns an entirely new random value. In statistics, the probability of two independent events occurring is the product of the individual probabilities, so the probability that RollDie returns 2 is

COMMON PITFALLS

When you call a function that produces a pseudo-random number, it is important to remember that the function will generate a different value each time it is called. If you want to keep track of a particular value, you must store the result of the function in a variable.

$$\frac{5}{6}\times\frac{1}{3}=\frac{5}{18}$$

Five chances out of 18 is almost twice as large as one chance in six, meaning that the RollDie function is much more likely to return 2 than it should be. To solve at least this one problem, you need to declare a variable to hold the result of the call to rand and then test that variable in each line, as shown in Figure 8-4.

FIGURE 8-4 Second attempt at RollDie

```
int RollDie (void)
{
        int r;

        if (r < RAND_MAX / 6) {
                return (1);
        } else if (r < RAND_MAX * 2 / 6) {
                return (2);
        } else if (r < RAND_MAX * 3 / 6) {
                return (3);
        } else if (r < RAND_MAX * 4 / 6) {
                return (4);
        } else if (r < RAND_MAX * 5 / 6) {
                return (5);
        } else {
                return (6);
        }
}
```

This implementation is still incorrect.

Unfortunately, this implementation is still buggy. The second problem, however, is more subtle. On most systems, RAND_MAX is given the value it has for a reason. The usual value chosen for RAND_MAX is not merely the maximum possible result for the rand function, but also the largest positive value that the system can represent using type int. This limitation causes a serious problem in the proposed implementation of RollDie, because the program is written in such a way that intermediate results may be larger than the maximum integer size. Even though the final result of

```
RAND_MAX
```

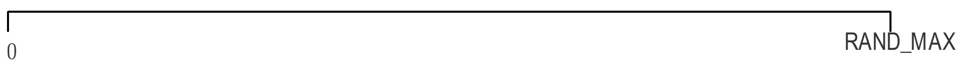
fits in a value of type int, C's rules of precedence indicate that RAND_MAX is first multiplied by 2 and then divided by 6. Generating an integer outside of the allowable range is called an **arithmetic overflow**. If such an overflow occurs, the program will not produce the intended answer. You can fix this problem by writing

```
RAND_MAX / 6 * 2
```

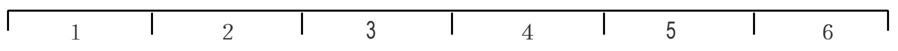
There is, however, a better approach.

The real problem with the RollDie implementation is that the procedure is much too complicated. The code tests for each of the six possible outcomes as a separate case. What you need is some mathematical insight that will allow you to eliminate the special cause altogether.

Look once more at the geometric problem. What you need to do is to convent the number line



into the discrete intervals



This time, rather than using if statements, it makes more sense to use arithmetic operations to accomplish the task. Before deciding how arithmetic operations apply o this situation, however, it is useful to generalize the problem so that your solution technique can serve a wider variety of applications.

Generalizing the problem

To simulate rolling a die, you generate a random integer between 1 and 6. If you want a program to "pick a card, and card" you want it to choose a number between 1 and 52. To model a European roulette wheel, you would want it to pick a number between 0 and 36. In general, what you need is not a function that chooses a number between 0 and RAND_MAX but one that chooses a random integer between two limits that you supply. The function you need might be defined using the following prototype:

```
int RandomInteger (int low, int high)
```

In other words, if you give this function two integers, it will return a random integer that lies between those endpoints, including each endpoint in the range. Thus, to simulate a roll of a die, you would call

```
RandomInteger (1, 6)
```

and , for a spin of the European roulette wheel, you would call

```
RandomInteger (0, 36)
```

Such a general tool has many uses, and it will be to your advantage to put this tool in a library so that you can use it again and again.

You already know how to generate a random number in the interval 0 to RAND_MAX. To convert this to a random number in a more restricted range, you can use the following four-step process:

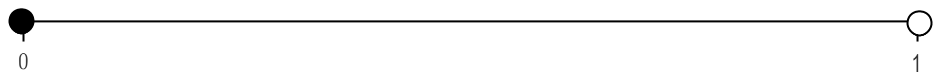
1. *Normalize* the integer result from `rand` by converting it into a floating-point number `d` in the range 0≤`d`<1.
2. *Scale* the value `d` by multiplying it by the size of the desired range, so that it spans the correct number of integers.
3. *Truncate* the number back to an integer by throwing away any fraction. This step gives you a random integer with a lower bound of 0.
4. *Translate* the integer so that the range begins at the desired lower bound.

To normalize the value, you first need to convert your result to a double and then divide it by the number of elements in the range. The numbers run from 0 to `RAND_MAX`, inclusive, so that the number of possible outcomes is `RAND_MAX` plus 1 (there are `RAND_MAX` values between 1 and `RAND_MAX`, and you also need to account for the value 0).

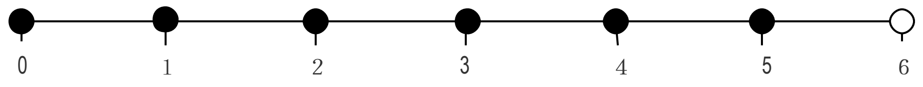
As noted in the section on "Assignment statements" in Chapter 2, you can use a type cast to specify an explicit conversion from one type to another. Type casts are written by enclosing the name of the desired new type in parentheses and writing it before the value to be converted .In this case, for example, you can convert the result of `rand` into a number `d` between 0 and 1 by writing:

```
d = (double) rand () / ((double) RAND_MAX + 1);
```

The numerator in this fraction must be less than the denominator so that the end result will always be strictly less than 1. Therefore, at the end of this process, you have a random real number that is at least 0 but always strictly less than 1. In mathematics, a range of real numbers that can be equal to one endpoint but not he other is called a **half-open interval**. In diagrams, the endpoint that is not included in the range is indicated using an open circle. Thus the range of possibilities for the variable `d` is diagrammed as follows:



The next step is to scale this value so that the range stretches to cover the correct number of integers. For example, to simulate the die roll, you need to multiply the value by 6, so that the new scaled range looks like this:



Note that here are six integers covered by the range: the integers 0, 1, 2, 3, 4, and 5. The value 6 itself lies outside of the range of possibilities, since the value `d` can never be as large as 1.

In the general case, you wan to multiply the normalized random number by the number of elements in the range, which is given by the expression

```
(high – low + 1)
```

The "extra" 1 in this expression is necessary because the range is inclusive and therefore contains both endpoints. Subtraction gives the distance between two integers, which is one less than the number of integers contained in the inclusive range. There are six outcomes for a die roll—1, 2, 3, 4, 5, and 5—but 6 minus 1 is only 5. To compute the number of outcomes, you need to subtract the smallest value from the largest and then add 1.

Next, you truncate the real number black to an integer. In C, if you convert a double to an int using a type cast, the conversion is done by throwing away any fractional part. Thus, if you take a real number that you know to be greater than or equal to 0 but strictly less than 6, you will get one of the integers 0, 1, 2, 3, 4, or 5.

The last step in the process is to translate the result so that it lies in the desired range. You have the correct number of integer outcomes; the only problem is than they start at 0. To obtain the correct set of possibilities, you simply add the value of the lower bound.

You can put all of these steps together and write the implementation o the function RandomInteger as follows:

```
int RandomInteger (int low, int high)
{
        int k;
        double d;

        d = (double) rand () / ((double) RAND_MAX + 1);
        k = (int) (d * (high – low + 1);
        return (low + k);
}
```

1-3 Saving tools in libraries

The ReandomInteger function is useful enough that you should put it in a library. The first step in this process is to create an interface. Once you complete the interface, you then write a corresponding implementation in a separate file. In most cases, the files used for an interface and its implementation have the same name except for the file type. Thus, if the interface is named random.h, you would ordinarily use random.h as the name of the implementation file.

The contents of an interface

The basic structure of an interface is illustrated by the graphics.h example introduced in Chapter 7. Like all the interfaces introduced in this text, graphics.h consists primarily of comments written for the benefit of clients who use that library. These comments are a critical part of the interface and should never be neglected when you are designing one.

A single definition exported by an interface to its clients is called an **interface entry**. Interface entries come in several different forms, of which the following are the most common:

- *Function prototypes*. As interface must contain the prototype of every function it makes available to the client.
- *Constant definitions*. As interface will often use #define to define a constant that

the clients will need to know. For example, the stdlib.h interface defines the constant RAND_MAX to tell its clients the maximum value returned by the rand function.

- *Type definitions.* Although you do not yet know how to define new types yourself, it is useful to know that interfaces often define new types for use by clients. For example, the genlib.h interface defines the types bool and string . Defining types in an interface is an extremely important technique in modern programming. You will see several examples of interfaces that export types beginning in Chapter 9.

In addition to these entries and their associated comments, every interface you write should contain three lines that are used to help the compiler keep track of the interfaces it has read. After the initial comments, but before any of the actual entries, every interface should contain the lines

```
#ifndef _name_h
#define _neme_h
```

where name is the name of the interface file. The last line of the interface must be

```
#endif
```

In complicated programs, a single interface may be included many times through a variety of paths. So that the complier will not read through the same interface each time, the line

```
#ifndef _name_H
```

cause the compiler to skip all of the text up to the #endif line if the symbol _name_h has been previously defined. On the first time through the interface it hasn't, so the compiler goes on reading. Immediately thereafter, however, the compiler encounters the line

```
#define _name_h
```

SYNTAX for an interface file

#ifndef _*name*_h
#define _*name*_h
any required #include lines
interface entries
#endif

Where:
name is the name of the library.
the #include lines section is used only if the interface itself requires other libraries and consists of standard #include lines
interface entries represents the function prototypes, constants, and types exported by the library
Comments should appear throughout the interface to provide clients with the information they need to use the library.

which defines the symbol _name_h. if the compiler should later start to read the same interface, _name_h will already have been defined, and the compiler knows that it can ignore the entire contents of the interface.

Whether or not you understand precisely how this technique works, the rule is clear. Whenever you write an interface, you must include the #ifndef, #define, and #endif lines, as shown in the syntax box. This sort of stylized pattern that is included every time you write a particular type of file is often called **boilerplate**. These lines are the boilerplate for interfaces. You don't really need to understand them; you just need to make sure that they are always there.

In addition to the boilerplate, a interface will sometimes need to include other interfaces using thee same #include lines that you have already used in your own programs, The rules for when such lines are required are discussed in the section on “Including header files in an interface” late in this chapter.

Writing the random.h **interface**

If you apply the rules from the preceding section to the problem of writing the random.h interface, you should realize that your first responsibility is to write an initial comment explaining what the library provides and who might use it. After the comment, you must include the boilerplate for interfaces, which in this case is

```
#ifndef _random_h
#define _random_h
```

The next thing to write in the random.h interface is a comment about the RandomInteger procedure:

```
/*
* Function: RandomInteger
* Usage: n = RandomInteger (low, high);
* ------------------------------------------------
* This function returns a random integer in the range
* low to high, inclusive.
*/
```

This comment provides the client with the information necessary to use the function. The usage line, for example, illustrates a sample call to the function, which is often particularly helpful to the client. The comment also contains an English description of what the function does but no discussion of how the function does it.

The next component of the interface is the prototype for the function itself:

```
    int RandomInteger (int low, int high);
```

This line is the only one in the interface that has any real significance to the compiler; the others are either comments or boilerplate.

The last line in the interface is simple the #endif line that is part of the boilerplate for interface.

The portion of the random.h interface discussed so far appears in Figure 8-5. As noted in the caption, the interface in Figure 8-5 is only a preliminary version. Later in the chapter, new functions are added to this interface that extend its capabilities.

FIGURE 8-5 Preliminary version of random.h

```
/*
* File: random.h
* -----------------
* This file contains a preliminary version of a library
* interface to produce pseudo-random numbers.
*/

#ifndef _random_h
#define _random_h
```

```
/*
* Function: RandomInteger
* Usage: n = RandomInteger (low, high);
*-------------------------------------------------
* This function returns a random integer in the range
* low to high, inclusive.
*/

int RandomInteger (int low, int high);

#endif
```

The random.h implementation

The implementation for the ranndom.h interface goes in a separate file, random.c. For the interface as it now exists, the corresponding implementation file is shown in Figure 8-6.

FIGURE 8-6 Preliminary version of random.c

```
/*
* File: random.c
* ----------------
* This file implements the preliminary random.h interface.
*/

#include <stdio.h>
#inlcude <stdlib.h>

#include "genlib.h"
#include "random.h"

/*
* Function: RandomInteger
* ------------------------------
* This function first obtains a random integer in
* the range [0…RAND_MAX] by applying four steps:
* (1) Generate a real number between 0 and 1.
* (2) Scale it to the appropriate range size.
* (3) Truncate the value to an integer.
* (4) Translate it to the appropriate starting point.
*/

int RandomInteger (int low, int high)
{
        int k;
        double d;

        d = (double) rand () / ((double) RAND_MAX+1);
        k = (int) (d * (high –low + 1);
        return (low + k);
}
```

The implementation begins with an initial comment, which is simply a reference to the interface. The next section lists the #include files required for the compilation. You always want stdio.h and genlib.h, and you need stdlib.h so that you have access to the function rand. Finally, every implementation needs to include its own interface so the compiler can check the prototypes against the actual definitions.

After the #include lines, the next section consists of the implementations of the functions exported by the interface, along with any comments that would be useful to the programmers who may need to maintain this program in the future.

Like all other forms of expository writing, comments must be written so that they take account of their audience. When you write comments, you must put yourself in the role of the reader so that you can understand what information that reader will want to see. Comments in the .c file have a different audience than their counterparts in the .h file. The comments in the implementation are written for another implementor who may have to modify the implementation in some way. They therefore must explain how the implementation works and provide any details that late maintainers would want to know. Comments in the interface, on the other hand, are written for the client. A client should never have to read the comments inside the implementation. The comments in the interface should be sufficient.

Constructing a client program

You can test the random.c implementation by writing the program dicetest.c shown in Figure 8-7. The main program makes use of your new random number library, so you need to include the line

```
#include “random.h”
```

in the dicetest.c file so that it can use the RandomInteger function.

FIGURE 8-7 dicetest.c

```
/*
* File: dicetest.c
* --------------------
* This program simulates rolling a die.
*/

#include <stdio.h>
#include “genlib.h”
#include “random.h”

/*
* Constants
* -----------
* Ntrials – Number of trials
*/

#define Ntrials 10

/* Function prototypes */

int RollDie (void);

/* Main programm */

main ()
{
        int i;
        for (i = 0; i    <NTrials; i++) {
                printf ("%d\n", RollDie ());
        }
    }

/*
* Function: RollDie
* Usage: die = RollDie ();
* ---------------------------
* This function generates and returns a random integer in the
* range 1 to 6, representing the roll of a six-sided die.
*/

int RollDie (void)
{
        return (RandomInteger (1, 6);
}
```

Let's quickly test the program to make sure that it works. Running the program gives the following result:

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

Once again, the numbers are all in the correct range and appear random. The number 2 comes up more often than the others, but it is statistically possible that the number 2 will come up four times by pure chance. Even so, you might want to investigate by running the pogrom again. This time it gives:

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

The disturbing observation is not simple that the number 2 came up just as many times on this second run. The entire result is exactly the same. In fact, every time you run this program, you get precisely the same result. This behavior on the part of your test program does not bode well for the prospect of writing interesting computer games.

Initializing the random number generator

The fact that the `dicetest.c` program produces the same sequence of numbers each time is not because of any bug in the implementation of RandomInteger. This behavior comes instead from the definition of the rand function in the standard ANSI libraries. Unless the caller takes specific action to change the standard mode of operation, the rand function always returns the same sequence of values on every execution of a program that calls it. Thus, every program presented so far in this chapter will have exactly the same effect each time it is run.

At first glance, you may find it hard to see any reason why rand might behave as it does, particularly since the rand function exists to simulate a nondeterministic process. As the stdlib.h interface is defined, the behavior of rand is entirely deterministic. There is, however, an extremely good reason to define rand in this way: Programs that behave deterministically are easier to debug.

To illustrate this fact, suppose you have just written a program to play an intricate game, such as Monopoly. As is always the case with newly written programs, the odds are good your program has a few bugs. In a complex program, bugs can be relatively obscure, in the sense that they only occur in rare situations. Suppose that you've been playing the game and find that the program starts behaving in a bizarre way, but you weren't alert enough to pay attention to all the relevant symptoms. You would like to run the program again and watch more carefully this time.

If the program is running in a nondeterministic way, a second run of the program will behave differently from the first. Bugs that showed up the first time may not occur on the second pass. In general, it is extremely difficult to reproduce the conditions that cause a program to fail if the program is behaving in a truly random fashion. If, on the other hand, the program is operating deterministically, it will do the same thing each time it is run. This behavior makes it possible for you to recreate a problem. During the debugging phase, the rand function is doing the right thing by returning the same sequence of values every time.

Even if the system definition of rand has advantages for debugging, it is still important to be able to change that behavior once the program is working. Understanding how to make this change, however, requires knowing a little more about the implementation of rand.

The ANSI libraries generate pseudo-random numbers by keeping track of the last number generated. Each time random is called, it takes the last number and performs a series of calculations using that number to produce the next one. Because you don't know what those calculations are, it is best to think of the entire operation as a black box where old numbers go in on one side and new pseudo-random numbers pop out on the other.

The randtest.c program described in the section on "Generating pseudo-random numbers in ANSI C" earlier in this chapter provides an illustration of the internal operation of rand. On the computer in my office, the first 10 calls to rand generate the numbers shown in this sample run:

```
On this computer, RAND_MAX = 32767.
Here are the results of 10 calls to rand:
16838
5758
10113
17515
31051
5627
23010
7419
16212
4086
```

The first call to rand produces the number 16838. The next call corresponds to putting 16838 into one end of the black box representing the internal implementation and having 5758 pop out on the side:

$$\begin{tikzcd}16838 \arrow[r] & \boxed{\text{black box}} \arrow[r] & 5758\end{tikzcd}$$

Similarly, on the next call to rand, the implementation puts 5758 into the black box, which returns 10113:

$$\begin{tikzcd}5758 \arrow[r] & \boxed{\text{black box}} \arrow[r] & 10113\end{tikzcd}$$

This same process is repeated on each call to rand. The computation inside the black box is designed so that (1) the numbers are uniformly distributed over the legal range, and (2) the sequence goes on for a long time before it begins to repeat.

But what about the first call to rand—the one that returns 16838? The implementation must have a starting point. There must be an integer, s, that goes into the black box and produces 16838:

$$\begin{tikzcd}s \arrow[r] & \boxed{\text{black box}} \arrow[r] & 16838\end{tikzcd}$$

This initial value—the value that is used to get the entire process started—is called a **seed** for the random number generator. The ANSI library implementation sets the initial seed to a constant value every time a program is started so that it always produces the same sequence. You can change the sequence by setting the seed to a different value. To do so, you need to call the function srand, which takes the new seed as an argument. To make sure the value of the new seed changes for each run of the program, the standard approach is to sue the value of the internal system clock as the initial seed. Because the time keeps changing, the random number sequence will change as well.

You can retriever the current value of the system clock by calling the function time, which is defined in the ANSI library interface time.h, and then converting the result to an integer. This technique allows you to write the following statement, which has the effect of initializing the pseudo-random number generator to some unpredictable point:

```
srand ((int) time (NULL));
```

Although it requires only a single line, the operation to set the random seed to an unpredictable value based on the system clock is relatively obscure. If this line were to appear in the client program, the client would have to understand the concept of a random number seed, the time function, and the meaning of the mysterious constant NULL. To make things simpler for the client, it would be much better to give this operation a simple name like Randomize and add it to the random number library. If you make this change, all the client needs to do is call

```
Randomize ();
```

which is certainly easier to explain.

The implementation of Randomize is simply

COMMON PITFALLS

When you write a program that works with random numbers, it is usually best to debug it without calling Randomize. When the program seems to be working well, you can insert a call to Randomize in the main program, after which the program will change its behavior from one run to another.

```
void Randomize (void)
{
        srand ((int) time (NULL));
}
```

1-4 Evaluating the design of the random.h interface

As part of the process of designing an interface, you should keep in mind the general principles guiding such design. In the case of the evolving random.h interface, for example, it is important to consider how well the current interface meets the five basic criteria outlined earlier in this chapter:

- *Is it unified*? The two functions, RandomInteger and Randomize, both fit under the unifying theme of providing access to a random number abstraction. Thus, the interface is unified.
- *Is it simple*? Although you have not had much opportunity to use the functions and see if they are in fact simple to use, the dicetest.c program gives some evidence that they are. Moreover, it is clear that the interface hides considerable complexity. Calling RandomInteger frees the client form having to worry about the internal steps of normalization, scaling, truncation and translation, since all those operations are performed by the implementation. Similarly, the Randomize function protects the client form all the internal details of seeding the random number generator. Thus, the interface certainly provides some measure of simplification.
- *Is it sufficient*? This question is always difficult to answer because it raises the companion question: sufficient for what? Though you probably cannot anticipate the needs for all clients, it is a good idea to try. The current version of the package is useful of clients who need random integers, but some clients would require other operations, such as some means of simulating random real numbers over a continuous range. The possibility suggests that some further design work may be required to meet this need.
- *Is it general*? The issue of generality is closely linked with that of sufficiency, but also includes the question of whether the interface design unconsciously incorporates any assumptions that are really in the domain of a particular client, thereby reducing its utility to others. For example, if the interface were defined to include functions that simulated a die roll, as opposed to allowing the client to build such functions on top of RandomInteger, that interface would likely be too narrow in its design. As it stands, however, the functions in the interface seem to meet the criterion of generality.
- *Is it stable*? The issue of stability is not so much a question for the design phase as for the long-term maintenance cycle of the package as a whole. The important question at this point is whether the interface design promotes long-term stability in some way. In general, an interface that satisfies the other requirements can probably remain stable, although preserving such stability requires good discipline on the part of those who are in charge of maintaining the library.

Thus, the only pending concern is that the random.h interface does not provide and the functions that clients are likely to need. In particular, the analysis of the design in the preceding section suggests that providing random real numbers would increase the utility of the random number library for some clients. It is therefore worth defining an additional function, presumable called RandomReal to go along with RandomInteger, that provides the necessary capability.

Generating random real numbers

As it happens, you have already used the rand function to generate a random real number as part of the implementation of RandomInteger. The first step in the process of generating a random integer was to generate a random floating-point number between 0 and 1. To implement the RandomReal function, one approach would be to do the same calculation and return the result. Such a design, however, violates to some extent the unifying principles that give the library consistency. Designed with that approach, RandomReal would take no arguments and return a floating-point value in a preset range. The RandomInteger function behaves differently. It takes two arguments and returns a value in the range defined by those inputs. For consistency, it is probably best that RandomReal have the same basic design. If it does, clients who know how RandomInteger works can correctly predict the structure of RandomReal. Thus, RandomReal should have the following prototype:

```
double RandomReal (double low, double high);
```

The implementation essentially consists of the first two lines of the implementation of RandomInteger, except that the scaling factor is now the actual distance between the endpoints instead of the number of integers contained in the at range. Thus the implementation of RandomReal is

```
double RandomReal (double low, double high)
{
        double d;

        d= (double) rand () / ((double) RAND_MAX + 1);
        return (low + d * (high – low));
}
```

Simulating a probabilistic event

In addition to random real numbers, there is another type of random variable that might be useful to include in a general abstraction for simulating random behavior. Suppose you are writing a program in which you want a certain ever to occur with random probability. For example, suppose that your program is intended to model an assembly line on which there is a defect that occurs, on average, in 1 out of every 1000 parts that travel down the line. In terms of the simulation, another way to think about this situation is that each part has a 1 in 1000 chance of being defective. In mathematics and statistics, probabilities are represented as numbers between 0 and 1, so the probability of a defect in this example is .001 (1/1000).

In this example, the outcome has only two possibilities: either there is a defect or there isn’t. The fact that there are two outcomes that represent the presence or absence of a condition suggests that it would be appropriate to represent the situation using a Boolean value. The value TRUE is used to signify that a defect has been detected, which should occur with probability .001.

In situations of this sort, it is helpful to have a predicate function that returns TRUE with some specified probability. If you had access to such a function, which might be named RandomChance, you could represent the assembly line example using the following code:

```
if (RandomChace (.001)) {
        printf (“A defect has occurred.\n”);
}
```

The advanta ge of this implementation over the one presented in the section on “Changing the range of random numbers” earlier in this chapter is that this one does not require the client to understand the operation of the rand function itself. The client can instead rely only on the functions defined in the higher-level random.h interface. The existence of the rand function can then be considered as a detail of concern only in the implementation.

The prototype and implementation for the RandomChance function are each very simple. The prototype, which becomes part of the interface, is

```
        bool RandomChance (doulbe p);
```

You can easily write the implementation in terms of RandomReal like this:

```
bool RandomChance (double p);
{
        return (RandomReal (0, 1) < p);
}
```

Including header files in an interface

COMMON PITFALLS
Each interface must include any header files that are necessary for the compiler to understand the interface itself. An interface does not, however, include header files that are only required by the underlying implementation. Those header files are included only in the .c file that implements the interface.

Adding RandomChance to the random.h interface, however, brings up an important issue. Randomchance is a predicate function and therefore returns a result of type bool. As noted in the section on “Boolean data” in Chapter 4, the type bool is not actually a part of C but is instead defined in the genlib.h interface. For the compiler to interpret correctly the prototype for RandomChance when it reads through the random.h interface, it needs access to the definition of bool in genlib.h .

To provide the compiler with the information it requires you need to include the genlib.h header file as part of the interface. Thus, right after the boilerplate lines

```
        #ifndef _random_h
        #define _random_h
```

the random.h header file must include the line

```
       #include “genlib.h”
```

which instructs the compiler to read the definitions in genlib.h. The compiler will therefore have read the definition for bool by the time it reaches the prototype for RandomChance later in the file.

Each interface must include only those header files that are required to compile the interface itself and not the corresponding implementation. For example, the implementation file random.c needs access to stdlib.h and time.h in order to use functions like rand, srand, and time. These functions, however, appear only in the implementation, not in the interface. Thus, the random.h interface does not need to include these header files ever though the random.c implementation does. By contrast, the type bool appears explicitly in the random.h interface, which means that the interface must include genlib.h.

Completing the implementation of the random-number package

All that remains to complete the definition of the random.h interface and the corresponding random.c implementation is to add the new functions defined in the last few sections to the preliminary versions of these files given in Figure 8-5 and 8-6, along with enough commentary to allow clients to understand the interface. Because all the sections of code have been shown individually, the complete versions of the interface and implementation do not appear in this chapter but in Appendix B.

1-5 Using the random-number package

Now that you have a random-number package, you can use it as often as you want. Whenever you decide to write a new computer game or any other application that involves random numbers, you'll have a set of tools you can use without having to remember the underlying details. All you need to do to use the random number library is to include the header file random.h in you program and make sure the library is available when you compile and run the program.

FIGURE 8-8 craps.c

```
/*
* File: craps.c
* ----------------
* This program plays the dice game called craps. For a discussion
* of the rules of carps, please see the GiveInstructions function.
*/

#include <stdio.h>
#include “genlib.h”
#include “simpio.h”
#include “strlib.h”

/* Function prototypes */

void GiveInstructions (void);
void playCrapsGame (void);
int RollTwoDice (void);
bool GetYesOrNo (string prompt);
```

```
/* Main program */

main ()
{
        Randomize ();
        if (GetYesOrNo ("Would you like instructions? ")) {
                GiveInstructions ();
}
while (TRUE) {
        playCrapsGame ();
        if (!GetYesOrNO ("Would you like to play again? ")) break;
}
}

/*
* Function: GiveInstructions
* Usage: GiveInstructions ();
* ---------------------------------
* This function welcomes the player to the game and gives
* instructions on the rules to craps.
*/

void GiveInstructions (void)
{
        printf ("Welcome to the craps table!\n\n");
        printf ("To play craps, you start by rolling a pair of dice\n");
        printf (" and looking at the total. If the total is 2, 3, or\n");
        printf ("12, that's called 'crapping out' and you lose. If\n");
        printf ("you roll a 7 or an 11, that's called a 'natural' an\n");
        printf ("you win. If you roll any other number, that number\n");
        printf ("becomes your 'point' and you keep on rolling until\n");
        printf ("you roll your point again (in which case you win");
        printf ("or a 7 (in which case you lose).\n");
}

/*
* Function: playCrapsGame
* Usage: playCrapsGame ();
* ---------------------------------
* This function plays one game of craps.
*/

void playCrapsGame (void)
{
        int total, point;

        printf ("\nHere we go!\n");
        total = RollTwoDice ();
        if (total == 7 || total == 11) {
                printf ("That's a natural. You win.\n");
        } else if (total == 2 || total == 3 || total == 12) {
                printf (" That's craps. You lose.\n");
        } else {
                point = total;
                printf ("Your point is %d.\n", point);
                while (TRUE) {
                        total = RollTwoDice ();
                        if (total == point) {
                                printf ("You made your point. You win.\n");
                                break;
                        } else if (total == 7) {
                                printf ("That's a seven. You lose. \n");
                                break;
                        }
                }
        }
}

/*
* Function: RollTwoDice
* Usage: total = RollTwoDice ();
* ------------------------------------
* This function rolls two dice and returns their sum. As part
* Of the implementation, the result is displayed on the screen.
*/

int RollTwoDice (void)
{
        int d1, d2, total;

        printf (Rolling the dice…\n");
        d1 = RandomInteger (1, 6);
        d2 = RandomInteger (1, 6);
        total = d1 + d2;
        printf ("You rolled %d and %d -- that 's %d.\n", d1, d2, total);
        return (total);
}

/*
* Function: GetYesOrNo
* Usage: if (GetYesOrNo (prompt)) …
* ------------------------------------------
* This function asks the user the question indicated by prompt
* and waits for a yes/no response. If the use answers "yes"
* or "no", the program returns TRUE or FALSE accordingly.
* If the user gives any other response, the program asks
* the question again.
*/

bool GetYesOrNo (string prompt)
{
        string answer;

        while (TRUE) {
                printf (%s", prompt);
                answer = GetLine ();
                if (StringEqual (answer, "yew")) return (TRUE);
                if (StringEqual (answer, "no")) return (FALSE);
                printf ("Please answer yes or no.\n");
        }
}
```

To illustrate the use of the package, a program called craps.c is shown in Figure 8-8. This program simulates the casino game of craps, which is played as follows. You start by rolling two six-sided dice and looking at the total. The game then breaks down into the following cases based on that first roll:

- You roiled a 2, 3, or 12. Rolling these numbers on your first roll is called crapping out and means theta you lose.
- You roiled a 7 or an 11. When either of these numbers comes up on your first roll, it is called a natural, and you win.
- You rolled one of the other numbers (4, 5, 6, 8, 9, or 10). In this case, the number you rolled is called you point, and you continue to roll the dice until either you roll your point a second time, in which case you win, or you roll a 7, in which case you lose. If you roll any other number (including 2, 3, 11, and 12, which are no longer treated

specially), you just keep on rolling until your point or a 7 appears.

The program itself is a straightforward translation of the English rules into C code. As you look through the raps.c program, you should notice the following features:

- ◆ The program includes the interface random.h so it can use the functions in that library. Moreover, the program uses only the functions in that library and never calls rand (or srand or time) directly. The random number sequence is initialized by calling Randomize, and each die roll is generated by a call to RandomInteger.
- ◆ The program is broken up into units that successively indicate greater detail. This decomposition helps to highlight the program structure and makes it possible for you to understand how the pieces fit together.
- ◆ The problem of rolling two dice comes up at several points in the program, so the combined action of simulating the roll of two dice, displaying the result, and remembering the total is encapsulated into the function RollTwoDice, which can be used in other contexts as well.

SUMMARY

In this chapter, you have had the chance to consider the process of writing an interface and its corresponding implementation. At one level, you have learned about the syntactic structure of an interface and the components it contains. You have also learned several more general principles of interface design—principles that will prove extremely important as you begin to solve larger tasks. Finally, you had the opportunity to see those design principles as they were applied to the construction of the random.h interface.

Important points introduced in this chapter include:

- ● The challenge of constructing an interface lies in the design of hte interface rather than its *coding*.
- ● A well-designed interface must be *unified*, *simple*, *sufficient*, *general*, and *stable*. Since these criteria sometimes conflict with each other, you must learn to strike an appropriate balance in your interface design.
- ● All the functions defined in an interface should fit a unifying theme and be as consistent as possible in their behavior.
- ● A main purpose of an interface is to keep the complexity of the implementation away from its clients. This principle is called *information hiding*.
- ● The abstraction represented by an interface must be powerful enough to satisfy the needs of its clients.
- ● An interface that is designed to be general enables many different clients to use the same library package.
- ● Clients must be able to rely on the stability of the interfaces they use. Changing an interface is a serious matter and not one to be undertaken lightly. On the other hand, maintaining a stable interface allows the implementor considerable freedom to change the underlying implementation.

- Programs can simulate random behavior by using an algorithmic process to generate a sequence of numbers that appears to be random. The numbers in such a sequence are called *pseudo-random numbers*.
- The ANSI library defines a function `rand` that returns a pseudo-random number between 0 and `RAND_MAX`.
- You can change the range of the pseudo-random numbers by applying simple arithmetic operations.
- The definitions exported by an interface are called *interface entries*. The most common interface entries are function prototype, *constant definitions*, and *type definitions*. The interface should also contain comments for each entry so the client can understand how to use that entry.
- To ensure that the compiler reads an interface only once, every interface should include these lines before the first interface entry:

  ```
  #ifndef _name_h
  #define _name_h
  ```

  and this line at the end of the interface file:

  ```
  #endif
  ```

- Unless you take special action, the `rand` function generates the same sequence of random numbers every time the program is run. To generate an unpredictable sequence, you must change the initial random-number seed. When you are using the `random.h` interface, the easiest way to set the seed is by calling the function `Randomize`.
- Each interface must include any header files that are necessary for the compiler to understand the interface itself. An interface should not include header files that are required only by the underlying implementation.
- If you want to work with pseudo-random numbers in your programs, you should use the `random.h` interface, which exports the functions `Randomize`, `RandomInteger`, `RandomReal`, and `RandomChance`. Using this interface is much simpler than working directly with `rand`.

REVIEW QUESTIONS

1. True or false: The hardest thing about writing an interface is following all of C's syntactic rules.
2. What are the five criteria for good interface design listed in this chapter?
3. What is an abstraction boundary?
4. Why is it important for an interface to be stable?
5. What is meant by the term *pseudo-random number*?
6. On most computers, how is the value of `RAND_MAX` chosen?
7. What four steps are necessary to convert the result of `rand` into an integer value with a different range?
8. How would you use `RandomInteger` to generate a pseudo-random number between 1 and 100?
9. By executing each of the statements by hand, determine whether `RandomInteger` works

with negative arguments. What are the possible results of calling the function RandomInteger (-5, 5)?

10. Could you use the multiple assignment statement

    ```
    d1 = d2 = RandomInteger (1, 6);
    ```

    to simulate the process of rolling two dice?
11. What are the three most common interface entries?
12. If you were defining an interface named magic.h, what would the interface boilerplate look like? What is the purpose of these lines?
13. True or false: the rand function ordinarily generates the same sequence of random numbers every time a program is run.
14. What is meant by the term *seed* in the context of random numbers?
15. What suggestion does this chapter offer for debugging a program involving random numbers?
16. What functions are defined in the final version of the random.h interface? In what context would you use each function?

PROGRAMMING EXERCISES

1. Run the randtest.c program on your computer system. What is the value of RAND_MAX on your machine?

2. Write a program that displays a random even number between 2 and 100.

3. Write a program that displays a random seven-digit phone number. The output should adhere to the following rules, which apply in the United States:

    - The output contains a hyphen between the third and fourth digit, as in 555-1968.
    - Neither of the first two digits is 0 or 1. This rule has actually been dropped in many parts of the United States, but you should nevertheless apply it for the purpose of this problem. Thus your program might generate the number 781-9902 but not the number 718-9902.

4. Write a program that displays the name of a card randomly chose from a complete deck of 52 playing cards. Each card consists of a rand (ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, jack, queen, king) and a suit (clubs, diamonds, hearts, spades). Your program should display the complete name of the card, as shown in the following sample run:

    ```
    Queen of Spades
    ```

5. Heads…
   Heads…
   Heads…
   A weaker man might be moved to re-examine his faith, if in nothing else at least in the

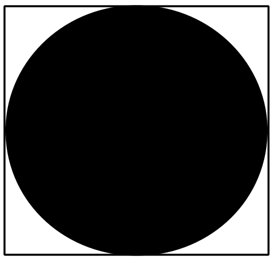
law of probability.

— **Rosencrantz and Guildenstern Are Dead, Tom Stoppard, 1967**

Write a program that simulates flipping a coin repeatedly and continues until three consecutive heads are tossed. At that point, your program should display the total number of coin flips that were made. The following is one possible sample run of the program:

```
tails
heads
heads
tails
tails
heads
tails
heads
heads
heads
It took 10 flips to get heads 3 consecutive times.
```

6. Although it is often easiest to think of random numbers in the context of games of chance, they have other, more practical uses in computer science and mathematics. For example, you can use random numbers to generate a rough approximation of the constant л by writing a simple program that simulates a dart board. Imagine that you have a dart board hanging on your wall. It consists of a circle painted on a square backdrop, as in the following diagram:



What happens if you throw a whole bunch of darts completely randomly, ignoring any darts that miss the board altogether? Some of the darts will fall inside the painted circle, but some will be outside the circle in the white corners of the square. Because you threw the darts randomly, the ratio of the number of darts that landed inside the circle to the total number of darts hitting the sequare should be approximately equal to the ratio between the two areas. The ratio of the areas is independent of the actual size of the dart board, as illustrated by the following formula:

$$\frac{\text{darts falling inside the circle}}{\text{darts falling inside the circle}} \cong \frac{\text{area of the circle}}{\text{area of the square}} = \frac{\pi r^2}{4r^2} = \frac{\pi}{4}$$

To simulate this process in a program, imagine that the dart board is drawn on the standard coordinate plane introduced in the section on “The underlying model for graphics.h” in Chapter 7, with its center at the origin and a radius of 1 unit. The process of throwing a dart randomly at the square can be modeled by generating two random number, x and y, each of which lies between –1 and 1. This (x, y) point always lies

$$\sqrt{x^2 + y^2} < 1$$

somewhere inside the square. The point (x, y) lies inside the circle if

This condition, however, can be simplified considerably by squaring each side of the

$$x^2 + y^2 < 1$$

inequality, which gives the following more efficient test:

If you perform this simulation many times and compute the fraction of darts that fall within the circle, The result will be somewhere in the neighborhood of л /4.

Write a program that simulates throwing 10,000 darts and then uses the simulation technique described in this exercise to generate and display an approximate value of л . Don't worry if your answer is correct only in the first few digits. The strategy used in this problem is not particularly accurate, even through it occasionally proves useful as a technique for making rough approximations. In mathematics, this technique is called Monte Carlo integration, after the capital city of Monaco.

7. Albert Einstein said that "I shall never believe that God plays dice with the world." Despite Einstein's metaphysical objections, the current models of physics, and particularly of quantum theory, are based on the idea that nature does indeed involve random processes. A radioactive atom, for example, does not decay for any specific reason that we mortals understand. Instead, that atom has a random probability of decaying within a period of time. Sometimes it does, sometimes it doesn't, and there is no way to know for sure.

   Because physicists consider radioactive decay a random process, it is not surprising that random numbers can be used to simulate that process. Suppose you start with a collection of atoms, each of which has a certain probability of decaying in any unit of time. You can then approximate the decay process by taking each atom in turn and deciding randomly whether it decays, considering the probability.

   Write a program that simulates the decay of a sample that contains 10,000 atoms of radioactive material, where each atom has a 50 percent chance of decaying in a year. The output of your program should be a table showing the year and the number of atoms remaining, such as the table shown in this sample run:

```
Year      Atoms left
-----     -----------
  0          10000
  1           4969
  2           2464
  3           1207
  4            627
  5            311
  6            166
  7             89
  8             40
  9             21
 10              8
 11              4
 12              1
 13              0
```

As the numbers indicate, roughly half the atoms in the sample decay each year. In physics, the conventional way to express this observation is to say that the sample has a *half-life* of one year.

8. As computers become more common in schools, it is important to find way to use the machines to aid in the teaching process. This need has led to the development of an educational software industry that has produced many programs that help teach concepts to children.

   As an example of an educational application, write a program that poses a series of simple arithmetic problems for a student to answer, as illustrated by the following sample run:

```
Welcome to Math Quiz!
What is 14 +  2? 16↵
That's the answer!
What is 17 – 15? 17↵
That's incorrect. Try a different answer: 15↵
That's incorrect. Try a different answer: 3↵
No, the answer is 2.
What is 20 – 16? 4↵
That's the answer!
What is 9 + 4? 11↵
That's incorrect. Try a different answer: 13↵
That's the answer!
What is 11 – 1? 10↵
That's the answer!
```

   Your program should meet these requirements:

   - It should ask a series of five questions. As with any such limit, the number of questions should be coded as a `#define` constant so that it can easily be changed.
   - Each question should consist of a single addition or subtraction problem involving just two numbers, such as "What is 2 + 3?" or "What is 11 – 7?". The type of problem—addition or subtraction—should be chosen randomly for each question.
   - To make sure the problems are appropriate for students in the first or second grade, none of the numbers involved, including the answer, should be less than 1 or greater than 20. This restriction means that your program should never ask questions like "What is 11 + 13?" or "What is 4 – 7?" because the answers are outside the legal range. Within these constraints, your program should choose the numbers randomly.
   - The program should give the student three chances to answer each question. If the student gives the correct answer, your program should indicate that fact in some properly congratulatory way and go on to the next question. If the student does not get the answer in three tries, the program should give the answer and go on to another problem.

9. Even though the program in exercise 8 was designed to offer encouragement when the student responds correctly, the monotonous repetition of a sentence like "That's the answer!" has the opposite effect after a while. To add variety to the interaction, modify your solution to exercise 8 so that it randomly chooses among four or five

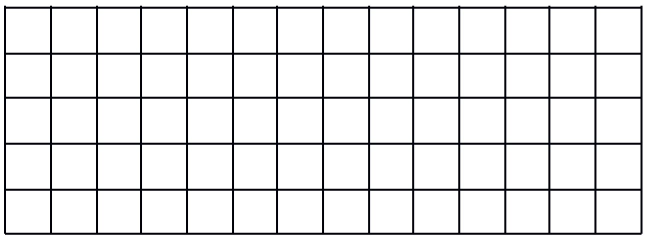
different messages when the student gets the right answer, as illustrated in this sample

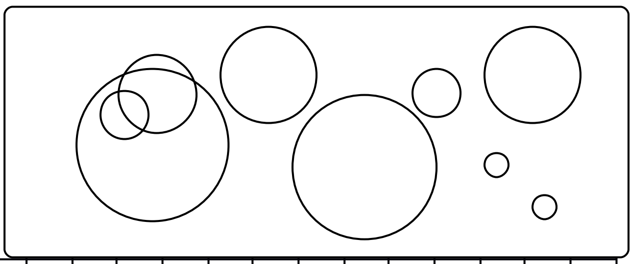
```
Welcome to Math Quiz!
What is 14 +   2? 16↵
Correct!
What is 17 – 15? 2↵
You got it. The answer is 2.
What is 20 – 16? 4↵
You got it. The answer is 4.
What is 9 + 4? 13↵
That's incorrect. Try a different answer: 13↵
Correct!
What is 11 – 1? 10↵
That's the answer!
```

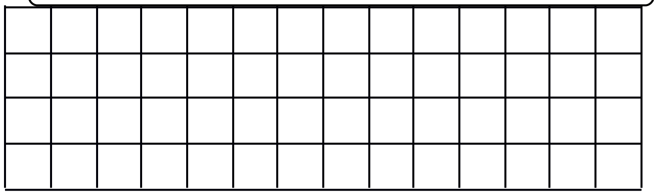
run:

10. Using the graphics library presented in Chapter 7, write a program that draws a set of 10 circles with different sizes and positions. Each circles should have a randomly chose radius between 0.05 and 0.5 inches and should be positioned at a random location in the drawing window, subject to the condition that the entire circle must fit inside the window without extending past the edge. The following sample run shows one possible outcome:

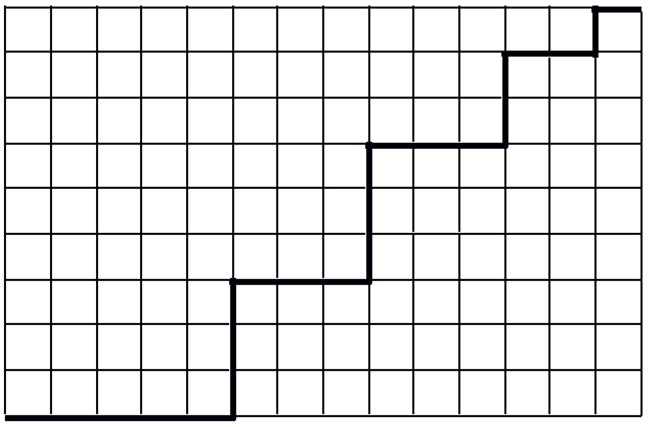
11. Imagine that you live in a well-planned city laid out so that its streets and avenues form blocks that are precisely square, as in this diagram:







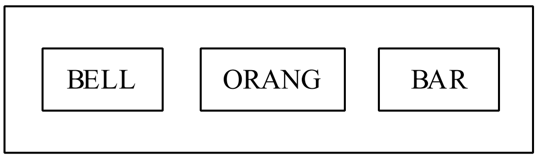
Suppose your office is located at the northeast corner of the map and you want to walk to your home in the southwest corner. Even if you don't want to back track or go out of your way, there are still many possible routes for getting home. At each intersection, you can choose randomly to go west or south. When you reach the southern or western edge of the map, you can just head home along that roadway. For example, the colored line in the following diagram shows one random route:



Write a program that uses the graphics library to trace out a random path through the city. You should start by moving the pen to the intersection in the upper right corner. From there, you draw a line either horizontally or vertically to get yourself to the next intersection, choosing the direction at random. Continue this process until you get home, making sure you don't run off the map.

Your program should not try to draw the entire map; it is enough just to show the path. If you want more practice using the graphics library, however, you could try to draw the entire figure shown in the example showing the random path. The make the heavy line for the random path, you need to draw two straight lines, one of each side of the actual grid lines.

12. In casinos from Monte Carlo to Las Vegas, one of the most common gambling devices is the slot machine—the "one-armed bandit." A typical slot machine has three wheels that spin around behind a narrow window. Each wheel is marked with the following symbols: CHERRY, LEMON, ORANGE, PLUM, BELL, and BAR. The window, however, allows you to see only one symbol on each wheel at a time. For example, the window might show the following configuration:



If you put a silver dollar into a slot machine and pull the handle on its side, the wheels spin around and eventually come to rest in some new configuration. If the configuration matches one of a set of winning patterns printed on the front of the slot machine, you get back some money. If not, you're out a dollar. The following table shows a typical set of winning patterns, along with their associated payoffs:

BAR	BAR	BAR	pays	$250
BELL	BELL	BELL/BAR	pays	$20
PLLUM	PLUM	PLUM/BAR	pays	$14
ORANGE	ORANGE	ORANGE/BAR	pays	$10

CHERRY	CHERRY	CHERRY	pays	$7
CHERRY	CHERRY	-	pays	$5
CHERRY	-	-	pays	$2

The notation BELL/BAR means that either a BELL or a BAR can appear in that position, and the dash means that any symbol at all can appear. Thus, getting a CHERRY in the first position is automatically good for two dollars, no matter what appears on the other wheels. Note that there is never any payoff for the LEMON symbol, even if you happen to line them up three of them.

Write a program that simulates playing a slot machine. Your program should provide the user with an initial stake of $50 and then let the user play until either the money runs out or the user decides to quit. During each round, your program should take away a dollar, simulate the spinning of the wheels, evaluate the result, and pay the user any appropriate winnings. For example, a user might be lucky enough to see the following sample run:

```
Would you like instructions? no↵
You have $50. Would you like to play? yes↵
PLUM      LEMON     LEMON     -- you lose
You have $49. Would you like to play? yes↵
PLUM      BAR       LEMON     -- you lose
You have $48. Would you like to play? yes↵
BELL      LEMON     ORANGE    -- you lose
You have $47. Would you like to play? yes↵
CHERRY    CHERRY    ORANGE    -- you win $5
You have $51. Would you like to play? yes↵
LEMON     ORANGE    BAR       -- you lose
You have $50. Would you like to play? yes↵
PLUM      BELL      PLUM      -- you lose
You have $49. Would you like to play? yes↵
BELL      BELL      BELL      -- you win $20
You have $68. Would you like to play? yes↵
CHERRY    PLUM      LEMON     -- you win $2
You have $69. Would you like to play? yes↵
ORANGE    BAR       PLUM      -- you lose
You have $68. Would you like to play? yes↵
ORANGE    PLUM      BELL      -- you lose
You have $67. Would you like to play? yes↵
BAR       BAR       BAR       -- you win $250
You have $316. Would you like to play? no↵
```

Even though doing so is not realistic (and would make the slot machine unprofitable for the casino), you should assume that each of the six symbols is equally likely on each wheel.

13. Chapter 7 defined several general functions for creating graphical figures that are useful enough to put into a separate library. These include DrawBox, DrawCenteredBOx, DrawCenteredCircle, DrawTriangle, and DrawGrid. Define a new interface gfigures.h that exports those five functions, and write the corresponding gfigures.h file to implement that interface. Rewrite the house.c program (Figure 7-7) so that it uses your new interface.

14. The calendar.c program in Chapter 5 includes several functions that are general enough to consider including in a library. It is easy to imagine, for example, that clients other