Chapter 5
Introduction to Recursion

And often enough, our faith beforehand in a certain result
is the only thing that makes the result come true.
— William James, *The Will To Believe,* 1897

Objectives

- To be able to define the concept of *recursion* as a programming strategy distinct from other forms of algorithmic decomposition.
- To recognize the paradigmatic form of a recursive function.
- To understand the internal implementation of recursive calls.
- To appreciate the importance of the *recursive leap of faith*.
- To understand the concept of *wrapper* functions in writing recursive programs.
- To be able to write and debug simple recursive functions at the level of those presented in this chapter.

Most algorithmic strategies used to solve programming problems have counterparts outside the domain of computing. When you perform a task repeatedly, you are using iteration. When you make a decision, you exercise conditional control. Because these operations are familiar, most people learn to use the control statements **`for`**, **`while`**, and **`if`** with relatively little trouble.

Before you can solve many sophisticated programming tasks, however, you will have to learn to use a powerful problem-solving strategy that has few direct counterparts in the real world. That strategy, called **recursion,** is defined as any solution technique in which large problems are solved by reducing them to smaller problems *of the same form*. The italicized phrase is crucial to the definition, which otherwise describes the basic strategy of stepwise refinement. Both strategies involve decomposition. What makes recursion special is that the subproblems in a recursive solution have the same form as the original problem.

If you are like most beginning programmers, the idea of breaking a problem down into subproblems of the same form does not make much sense when you first hear it. Unlike repetition or conditional testing, recursion is not a concept that comes up in day-to-day life. Because it is unfamiliar, learning how to use recursion can be difficult. To do so, you must develop the intuition necessary to make recursion seem as natural as all the other control structures. For most students of programming, reaching that level of understanding takes considerable time and practice. Even so, learning to use recursion is definitely worth the effort. As a problem-solving tool, recursion is so powerful that it at times seems almost magical. In addition, using recursion often makes it possible to write complex programs in simple and profoundly elegant ways.

5.1 A simple example of recursion

To gain a better sense of what recursion is, let’s imagine you have been appointed as the funding coordinator for a large charitable organization that is long on volunteers and short on cash. Your job is to raise $1,000,000 in contributions so the organization can meet its expenses.

If you know someone who is willing to write a check for the entire $1,000,000, your job is easy. On the other hand, you may not be lucky enough to have friends who are generous millionaires. In that case, you must raise the $1,000,000 in smaller amounts. If the average contribution to your organization is $100, you might choose a different tack: call 10,000 friends and ask each of them for $100. But then again, you probably don’t have 10,000 friends. So what can you do?

As is often the case when you are faced with a task that exceeds your own capacity, the answer lies in delegating part of the work to others. Your organization has a reasonable supply of volunteers. If you could find 10 dedicated supporters in different parts of the country and appoint them as regional coordinators, each of those 10 people could then take responsibility for raising $100,000.

Raising $100,000 is simpler than raising $1,000,000, but it hardly qualifies as easy. What should your regional coordinators do? If they adopt the same strategy, they will in turn delegate parts of the job. If they each recruit 10 fundraising volunteers, those people will only have to raise $10,000. The delegation process can continue until the volunteers are able to raise the money on their own; because the average contribution is $100, the volunteer fundraisers can probably raise $100 from a single donor, which eliminates the need for further delegation.

If you express this fundraising strategy in pseudocode, it has the following structure:

```
void CollectContributions(int n) {
    if (n <= 100) {
        Collect the money from a single donor.
    } else {
        Find 10 volunteers.
        Get each volunteer to collect n/10 dollars.
        Combine the money raised by the volunteers.
    }
}
```

The most important thing to notice about this pseudocode translation is that the line

Get each volunteer to collect **n/10** *dollars.*

is simply the original problem reproduced at a smaller scale. The basic character of the task—raise *n* dollars—remains exactly the same; the only difference is that *n* has a smaller value. Moreover, because the problem is the same, you can solve it by calling the original function. Thus, the preceding line of pseudocode would eventually be replaced with the following line:

```
CollectContributions(n / 10);
```

It's important to note that the `CollectContributions` function ends up calling itself if the contribution level is greater than $100. In the context of programming, having a function call itself is the defining characteristic of recursion.

The structure of the `CollectContributions` procedure is typical of recursive functions. In general, the body of a recursive function has the following form:

```
if (test for simple case) {
    Compute a simple solution without using recursion.
} else {
    Break the problem down into subproblems of the same form.
    Solve each of the subproblems by calling this function recursively.
    Reassemble the solutions to the subproblems into a solution for the whole.
}
```

This structure provides a template for writing recursive functions and is therefore called the **recursive paradigm.** You can apply this technique to programming problems as long as they meet the following conditions:

1. You must be able to identify **simple cases** for which the answer is easily determined.
2. You must be able to identify a **recursive decomposition** that allows you to break any complex instance of the problem into simpler problems of the same form.

The `CollectContributions` example illustrates the power of recursion. As in any recursive technique, the original problem is solved by breaking it down into smaller subproblems that differ from the original only in their scale. Here, the original problem is to raise $1,000,000. At the first level of decomposition, each subproblem is to raise $100,000. These problems are then subdivided in turn to create smaller problems until the problems are simple enough to be solved immediately without recourse to further subdivision. Because the solution depends on dividing hard problems into simpler ones, recursive solutions are often called **divide-and-conquer** strategies.

5.2 The factorial function

Although the `CollectContributions` example illustrates the concept of recursion, it gives little insight into how recursion is used in practice, mostly because the steps that make up the solution, such as finding 10 volunteers and collecting money, are not easily represented in a C++ program. To get a practical sense of the nature of recursion, you need to consider problems that fit more easily into the programming domain.

For most people, the best way to understand recursion is to start with simple mathematical functions in which the recursive structure follows directly from the statement of the problem and is therefore easy to see. Of these, the most common is the factorial function—traditionally denoted in mathematics as $n!$—which is defined as the product of the integers between 1 and n. In C++, the equivalent problem is to write an implementation of a function with the prototype

```
int Fact(int n);
```

that takes an integer `n` and returns its factorial.

As you probably discovered in an earlier programming course, it is easy to implement the `Fact` function using a `for` loop, as illustrated by the following implementation:

```
int Fact(int n) {
    int product;

    product = 1;
    for (int i = 1; i <= n; i++) {
        product *= i;
    }
    return product;
}
```

This implementation uses a `for` loop to cycle through each of the integers between 1 and `n`. In the recursive implementation this loop does not exist. The same effect is generated instead by the cascading recursive calls.

Implementations that use looping (typically by using `for` and `while` statements) are said to be **iterative.** Iterative and recursive strategies are often seen as opposites because they can be used to solve the same problem in rather different ways. These strategies, however, are not mutually exclusive. Recursive functions sometimes employ iteration internally, and you will see examples of this technique in Chapter 6.

The recursive formulation of `Fact`

The iterative implementation of `Fact`, however, does not take advantage of an important mathematical property of factorials. Each factorial is related to the factorial of the next smaller integer in the following way:

$$n! = n \times (n-1)!$$

Thus, 4! is $4 \times 3!$, 3! is $3 \times 2!$, and so on. To make sure that this process stops at some point, mathematicians define 0! to be 1. Thus, the conventional mathematical definition of the factorial function looks like this:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1)! & \text{otherwise} \end{cases}$$

This definition is recursive, because it defines the factorial of n in terms of the factorial of $n - 1$. The new problem—finding the factorial of $n - 1$—has the same form as the original problem, which is the fundamental characteristic of recursion. You can then use the same process to define $(n - 1)!$ in terms of $(n - 2)!$. Moreover, you can carry this process forward step by step until the solution is expressed in terms of $0!$, which is equal to 1 by definition.

From your perspective as a programmer, the practical impact of the mathematical definition is that it provides a template for a recursive implementation. In C++, you can implement a function **`Fact`** that computes the factorial of its argument as follows:

```
int Fact(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * Fact(n - 1);
    }
}
```

If **`n`** is 0, the result of **`Fact`** is 1. If not, the implementation computes the result by calling **`Fact(n - 1)`** and then multiplying the result by **`n`**. This implementation follows directly from the mathematical definition of the factorial function and has precisely the same recursive structure.

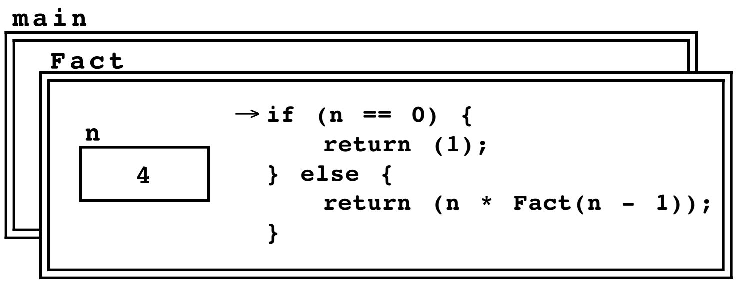
Tracing the recursive process

If you work from the mathematical definition, writing the recursive implementation of **`Fact`** is straightforward. On the other hand, even though the definition is easy to write, the brevity of the solution may seem suspicious. When you are learning about recursion for the first time, the recursive implementation of **`Fact`** seems to leave something out. Even though it clearly reflects the mathematical definition, the recursive formulation makes it hard to identify where the actual computational steps occur. When you call **`Fact`**, for example, you want the computer to give you the answer. In the recursive implementation, all you see is a formula that transforms one call to **`Fact`** into another one. Because the steps in that calculation are not explicit, it seems somewhat magical when the computer gets the right answer.

If you follow through the logic the computer uses to evaluate any function call, however, you discover that no magic is involved. When the computer evaluates a call to the recursive **`Fact`** function, it goes through the same process it uses to evaluate any other function call. To visualize the process, suppose that you have executed the statement

```
f = Fact(4);
```

as part of the function **`main`**. When **`main`** calls **`Fact`**, the computer creates a new stack frame and copies the argument value into the formal parameter **`n`**. The frame for **`Fact`** temporarily supersedes the frame for **`main`**, as shown in the following diagram:



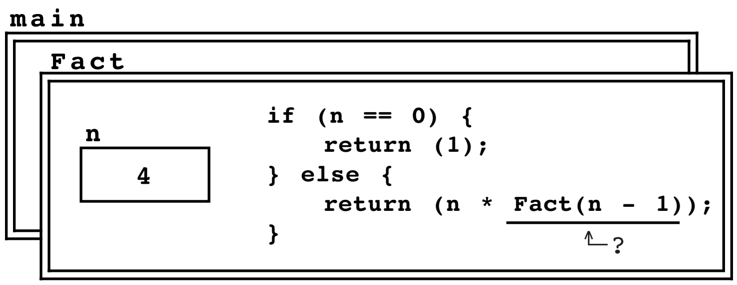
```
→if (n == 0) {
     return (1);
 } else {
     return (n * Fact(n - 1));
 }
```

In the diagram, the code for the body of **`Fact`** is shown inside the frame to make it easier to keep track of the current position in the program, which is indicated by an arrow. In the current diagram, the arrow appears at the beginning of the code because all function calls start at the first statement of the function body.

The computer now begins to evaluate the body of the function, starting with the **`if`** statement. Because **`n`** is not equal to 0, control proceeds to the **`else`** clause, where the program must evaluate and return the value of the expression

```
    n * Fact(n - 1)
```

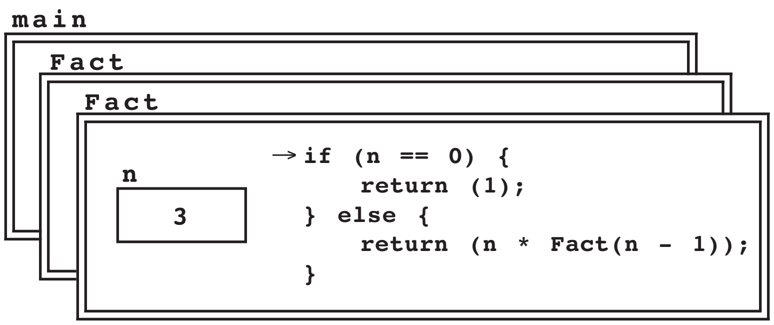
Evaluating this expression requires computing the value of **`Fact(n - 1)`**, which introduces a recursive call. When that call returns, all the program has to do is to multiply the result by **`n`**. The current state of the computation can therefore be diagrammed as follows:



```
if (n == 0) {
    return (1);
} else {
    return (n * Fact(n - 1));
}
```

As soon as the call to **`Fact(n - 1)`** returns, the result is substituted for the expression underlined in the diagram, allowing computation to proceed.

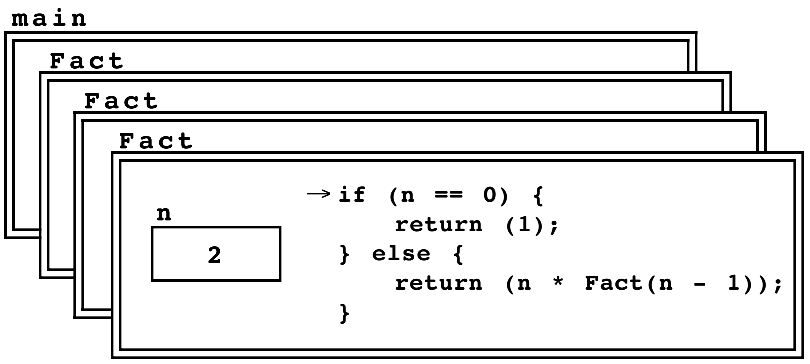
The next step in the computation is to evaluate the call to **`Fact(n - 1)`**, beginning with the argument expression. Because the current value of **`n`** is 4, the argument expression **`n - 1`** has the value 3. The computer then creates a new frame for **`Fact`** in which the formal parameter is initialized to this value. Thus, the next frame looks like this:



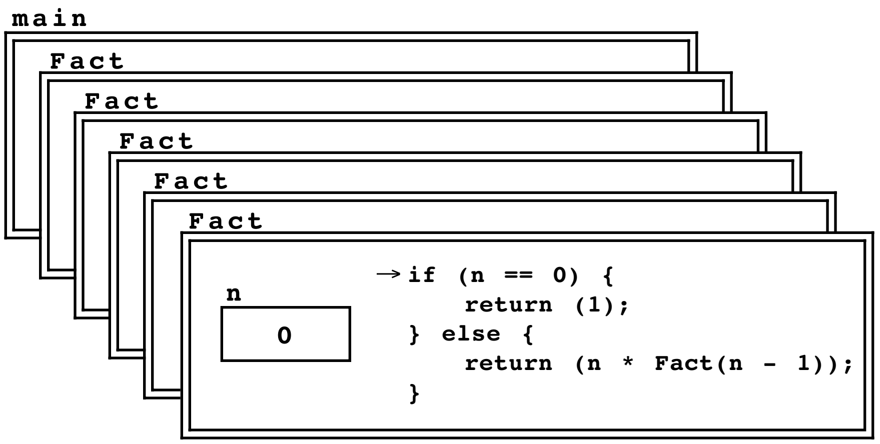
```
→if (n == 0) {
     return (1);
 } else {
     return (n * Fact(n - 1));
 }
```

There are now two frames labeled **Fact**. In the most recent one, the computer is just starting to calculate **Fact(3)**. In the preceding frame, which the newly created frame hides, the **Fact** function is awaiting the result of the call to **Fact(n - 1)**.

The current computation, however, is the one required to complete the topmost frame. Once again, **n** is not 0, so control passes to the **else** clause of the **if** statement, where the computer must evaluate **Fact(n - 1)**. In this frame, however, **n** is equal to 3, so the required result is that computed by calling **Fact(2)**. As before, this process requires the creation of a new stack frame, as shown:



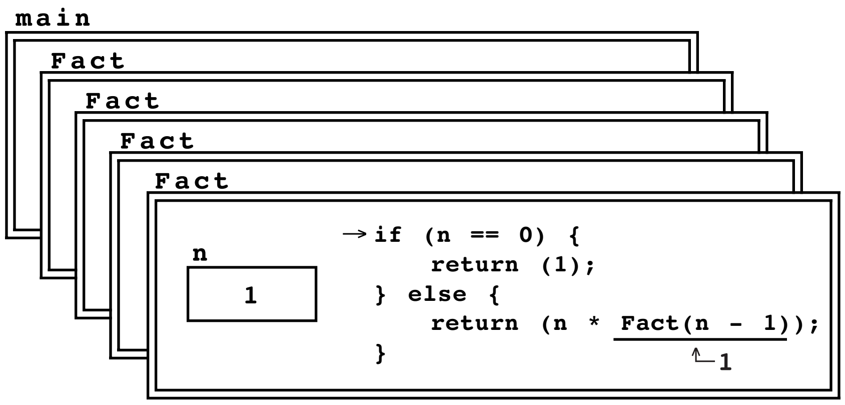
Following the same logic, the program must now call **Fact(1)**, which in turn calls **Fact(0)**, thereby creating two new stack frames. The resulting stack configuration looks like this:



At this point, however, the situation changes. Because the value of **n** is 0, the function can return its result immediately by executing the statement

```
return 1;
```

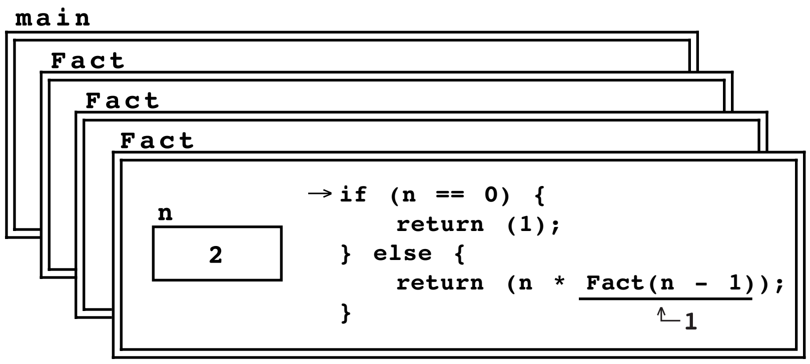
The value 1 is returned to the calling frame, which resumes its position on top of the stack, as shown:



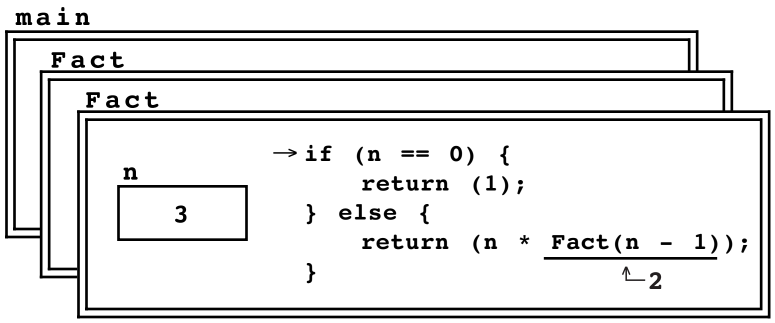
From this point, the computation proceeds back through each of the recursive calls, completing the calculation of the return value at each level. In this frame, for example, the call to `Fact(n - 1)` can be replaced by the value 1, so that the result at this level can be expressed as follows:

```
return n * [1]);
```

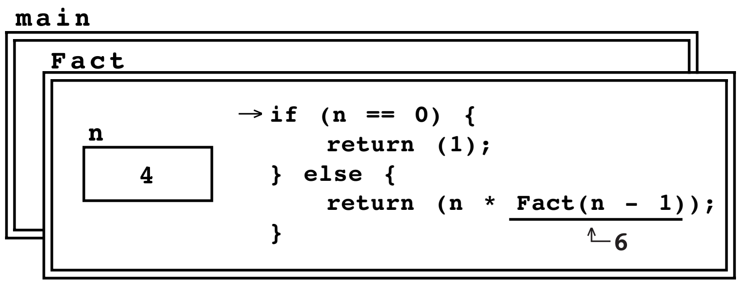
In this stack frame, **`n`** has the value 1, so the result of this call is simply 1. This result gets propagated back to its caller, which is represented by the top frame in the following diagram:



Because **`n`** is now 2, evaluating the **`return`** statement causes the value 2 to be passed back to the previous level, as follows:



At this stage, the program returns 3 × 2 to the previous level, so that the frame for the initial call to **`Fact`** looks like this:



The final step in the calculation process consists of calculating 4 x 6 and returning the value 24 to the main program.

The recursive leap of faith

The point of the long `Fact(4)` example in the preceding section is to show you that the computer treats recursive functions just like all other functions. When you are faced with a recursive function, you can—at least in theory—mimic the operation of the computer and figure out what it will do. By drawing all the frames and keeping track of all the variables, you can duplicate the entire operation and come up with the answer. If you do so, however, you will usually find that the complexity of the process ends up making the problem much harder to understand.

When you try to understand a recursive program, you must be able to put the underlying details aside and focus instead on a single level of the operation. At that level, you are allowed to assume that any recursive call automatically gets the right answer as long as the arguments to that call are simpler than the original arguments in some respect. This psychological strategy—assuming that any simpler recursive call will work correctly—is called the **recursive leap of faith.** Learning to apply this strategy is essential to using recursion in practical applications.

As an example, consider what happens when this implementation is used to compute `Fact(n)` with `n` equal to 4. To do so, the recursive implementation must compute the value of the expression

```
n * Fact(n - 1)
```

By substituting the current value of `n` into the expression, you know that the result is

```
4 * Fact(3)
```

Stop right there. Computing `Fact(3)` is simpler than computing `Fact(4)`. Because it is simpler, the recursive leap of faith allows you to assume that it works. Thus, you should assume that the call to `Fact(3)` will correctly compute the value of 3!, which is 3 x 2 x 1, or 6. The result of calling `Fact(4)` is therefore 4 x 6, or 24.

As you look at the examples in the rest of this chapter, try to focus on the big picture instead of the morass of detail. Once you have made the recursive decomposition and identified the simple cases, be satisfied that the computer can handle the rest.

5.3 The Fibonacci function

In a mathematical treatise entitled *Liber Abbaci* published in 1202, the Italian mathematician Leonardo Fibonacci proposed a problem that has had a wide influence on many fields, including computer science. The problem was phrased as an exercise in population biology—a field that has become increasingly important in recent years. Fibonacci's problem concerns how the population of rabbits would grow from generation to generation if the rabbits reproduced according to the following, admittedly fanciful, rules:

- Each pair of fertile rabbits produces a new pair of offspring each month.
- Rabbits become fertile in their second month of life.
- Old rabbits never die.

If a pair of newborn rabbits is introduced in January, how many pairs of rabbits are there at the end of the year?

You can solve Fibonacci's problem simply by keeping a count of the rabbits at each month during the year. At the beginning of January, there are no rabbits, since the first pair is introduced sometime in that month, which leaves one pair of rabbits on February 1. Since the initial pair of rabbits is newborn, they are not yet fertile in February, which means that the only rabbits on March 1 are the original pair of rabbits. In March, however, the original pair is now of reproductive age, which means that a new pair of rabbits is born. The new pair increases the colony's population—counting by pairs—to two on April 1. In April, the original pair goes right on reproducing, but the rabbits born in March are as yet too young. Thus, there are three pairs of rabbits at the beginning of May. From here on, with more and more rabbits becoming fertile each month, the rabbit population begins to grow more quickly.

Computing terms in the Fibonacci sequence

At this point, it is useful to record the population data so far as a sequence of terms, indicated here by the subscripted value t_i, each of which shows the number of rabbit pairs at the beginning of the ith month from the start of the experiment on January 1. The sequence itself is called the **Fibonacci sequence** and begins with the following terms, which represent the results of our calculation so far:

t_0	t_1	t_2	t_3	t_4
0	1	1	2	3

You can simplify the computation of further terms in this sequence by making an important observation. Because rabbits in this problem never die, all the rabbits that were around in the previous month are still around. Moreover, all of the fertile rabbits have produced a new pair. The number of fertile rabbit pairs capable of reproduction is simply the number of rabbits that were alive in the month before the previous one. The net effect is that each new term in the sequence must simply be the sum of the preceding two. Thus, the next several terms in the Fibonacci sequence look like this:

t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}
0	1	1	2	3	5	8	13	21	34	55	89	144

The number of rabbit pairs at the end of the year is therefore 144.

From a programming perspective, it helps to express the rule for generating new terms in the following, more mathematical form:

$$t_n = t_{n-1} + t_{n-2}$$

An expression of this type, in which each element of a sequence is defined in terms of earlier elements, is called a **recurrence relation.**

The recurrence relation alone is not sufficient to define the Fibonacci sequence. Although the formula makes it easy to calculate new terms in the sequence, the process has to start somewhere. In order to apply the formula, you need to have at least two terms in hand, which means that the first two terms in the sequence—t_0 and t_1—must be defined explicitly. The complete specification of the terms in the Fibonacci sequence is therefore

$$t_n = \begin{cases} n & \text{if } n \text{ is 0 or 1} \\ t_{n-1} + t_{n-2} & \text{otherwise} \end{cases}$$

This mathematical formulation is an ideal model for a recursive implementation of a function **`Fib(n)`** that computes the *n*th term in the Fibonacci sequence. All you need to do is plug the simple cases and the recurrence relation into the standard recursive paradigm. The recursive implementation of **`Fib(n)`** is shown in Figure 5-1, which also includes a test program that displays the terms in the Fibonacci sequence between two specified indices.

Gaining confidence in the recursive implementation

Now that you have a recursive implementation of the function **`Fib`**, how can you go about convincing yourself that it works? You can always begin by tracing through the logic. Consider, for example, what happens if you call **`Fib(5)`**. Because this is not one of the simple cases enumerated in the **`if`** statement, the implementation computes the result by evaluating the line

```
return Fib(n - 1) + Fib(n - 2);
```

which is in this case equivalent to

```
return Fib(4) + Fib(3);
```

At this point, the computer calculates the result of **`Fib(4)`**, adds that to the result of calling **`Fib(3)`**, and returns the sum as the value of **`Fib(5)`**.

But how does the computer go about evaluating **`Fib(4)`** and **`Fib(3)`**? The answer, of course, is that it uses precisely the same strategy. The essence of recursion is to break problems down into simpler ones that can be solved by calls to exactly the same function. Those calls get broken down into simpler ones, which in turn get broken down into even simpler ones, until at last the simple cases are reached.

On the other hand, it is best to regard this entire mechanism as irrelevant detail. Remember the recursive leap of faith. Your job at this level is to understand how the call to **`Fib(5)`** works. In the course of walking though the execution of that function, you have managed to transform the problem into computing the sum of **`Fib(4)`** and **`Fib(3)`**. Because the argument values are smaller, each of these calls represents a simpler case. Applying the recursive leap of faith, you can assume that the program correctly computes each of these values, without going through all the steps yourself. For the purposes of validating the recursive strategy, you can just look the answers up in the table. **`Fib(4)`** is 3 and **`Fib(3)`** is 2, so the result of calling **`Fib(5)`** is $3 + 2$, or 5, which is indeed the correct answer. Case closed. You don't need to see all the details, which are best left to the computer.

Figure 5-1 Recursive implementation of the Fibonacci function

```
/*
 * File: fib.cpp
 * -------------
 * This program lists the terms in the Fibonacci sequence with
 * indices ranging from MIN_INDEX to MAX_INDEX.
 */

#include "genlib.h"
#include <iostream>

/*
 * Constants
 * ---------
 * MIN_INDEX -- Index of first term to generate
 * MAX_INDEX -- Index of last term to generate
 */

const int MIN_INDEX  = 0;
const int MAX_INDEX = 12;

/* Private function prototypes */

int Fib(int n);

/* Main program */

int main() {
    cout << "This program lists the Fibonacci sequence." << endl;
    for (int i = MIN_INDEX; i <= MAX_INDEX; i++) {
        cout << "Fib(" << i << ")";
        if (i < 10) cout << " ";
        cout << " = " << Fib(i) << endl;
    }
    return 0;
}

/*
 * Function: Fib
 * Usage: t = Fib(n);
 * ------------------
 * This function returns the nth term in the Fibonacci sequence
 * using a recursive implementation of the recurrence relation
 *
 *      Fib(n) = Fib(n - 1) + Fib(n - 2)
 */

int Fib(int n) {
    if (n < 2) {
        return n;
    } else {
        return Fib(n - 1) + Fib(n - 2);
    }
}
```

Efficiency of the recursive implementation

If you do decide to go through the details of the evaluation of the call to `Fib(5)`, however, you will quickly discover that the calculation is extremely inefficient. The recursive decomposition makes many redundant calls, in which the computer ends up calculating the same term in the Fibonacci sequence several times. This situation is illustrated in Figure 5-2, which shows all the recursive calls required in the calculation of `Fib(5)`. As you can see from the diagram, the program ends up making one call to `Fib(4)`, two calls to `Fib(3)`, three calls to `Fib(2)`, five calls to `Fib(1)`, and three calls to `Fib(0)`. Given that the Fibonacci function can be implemented efficiently using iteration, the enormous explosion of steps required by the recursive implementation is more than a little disturbing.

Recursion is not to blame

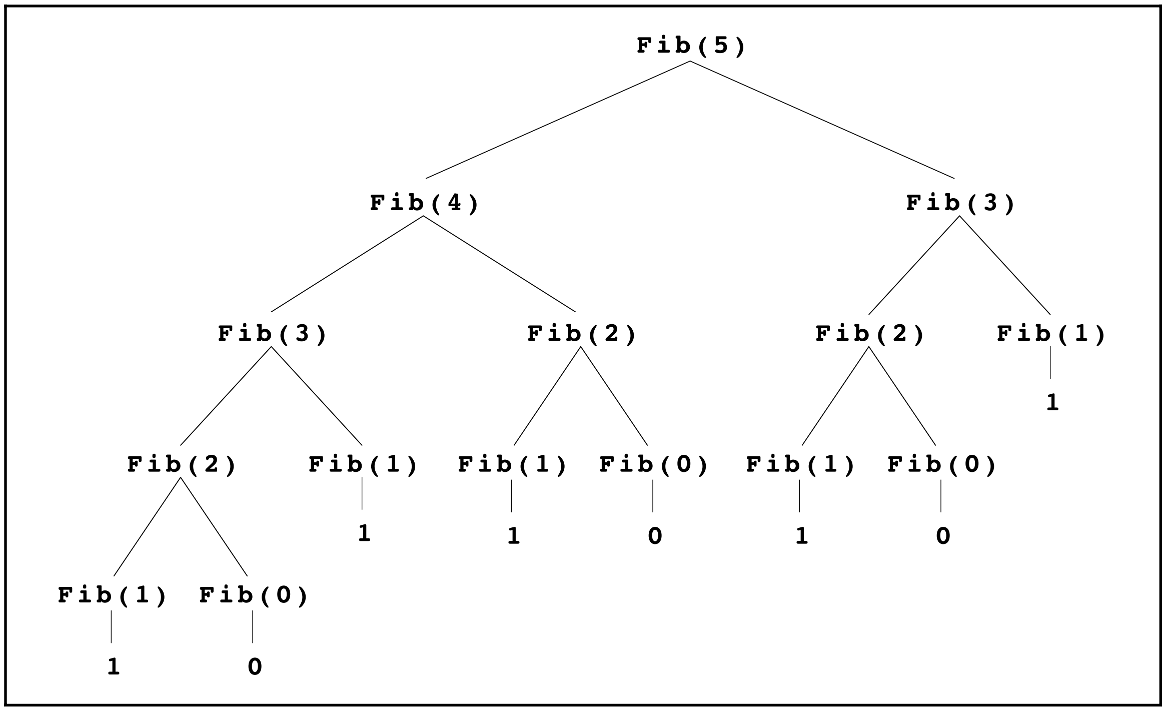
On discovering that the implementation of `Fib(n)` given in Figure 5-1 is highly inefficient, many people are tempted to point their finger at recursion as the culprit. The problem in the Fibonacci example, however, has nothing to do with recursion per se but rather the way in which recursion is used. By adopting a different strategy, it is possible to write a recursive implementation of the `Fib` function in which the large-scale inefficiencies revealed in Figure 5-2 disappear completely.

As is often the case when using recursion, the key to finding a more efficient solution lies in adopting a more general approach. The Fibonacci sequence is not the only sequence whose terms are defined by the recurrence relation

$$t_n = t_{n-1} + t_{n-2}$$

Depending on how you choose the first two terms, you can generate many different sequences. The traditional Fibonacci sequence

Figure 5-2 Steps in the calculation of `Fib(5)`



$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots$$

comes from defining $t_0 = 0$ and $t_1 = 1$. If, for example, you defined $t_0 = 3$ and $t_1 = 7$, you would get this sequence instead:

$$3, 7, 10, 17, 27, 44, 71, 115, 186, 301, 487, 788, 1275, \ldots$$

Similarly, defining $t_0 = -1$ and $t_1 = 2$ gives rise to the following sequence:

$$-1, 2, 1, 3, 4, 7, 11, 18, 29, 47, 76, 123, 199, \ldots$$

These sequences all use the same recurrence relation, which specifies that each new term is the sum of the previous two. The only way the sequences differ is in the choice of the first two terms. As a general class, the sequences that follow this pattern are called **additive sequences.**

This concept of an additive sequence makes it possible to convert the problem of finding the n^{th} term in the Fibonacci sequence into the more general problem of finding the n^{th} term in an additive sequence whose initial terms are t_0 and t_1. Such a function requires three arguments and might be expressed in C++ as a function with the following prototype:

```
int AdditiveSequence(int n, int t0, int t1);
```

If you had such a function, it would be easy to implement **`Fib`** using it. All you would need to do is supply the correct values of the first two terms, as follows:

```
int Fib(int n) {
    return AdditiveSequence(n, 0, 1);
}
```

The body consists of a single line of code that does nothing but call another function, passing along a few extra arguments. Functions of this sort, which simply return the result of another function, often after transforming the arguments in some way, are called **wrapper** functions. Wrapper functions are extremely common in recursive programming. In most cases, a wrapper function is used—as it is here—to supply additional arguments to a subsidiary function that solves a more general problem.

From here, the only remaining task is to implement the function **`AdditiveSequence`**. If you think about this more general problem for a few minutes, you will discover that additive sequences have an interesting recursive character of their own. The simple case for the recursion consists of the terms t_0 and t_1, whose values are part of the definition of the sequence. In the C++ implementation, the value of these terms are passed as arguments. If you need to compute t_0, for example, all you have to do is return the argument **`t0`**.

But what if you are asked to find a term further down in the sequence? Suppose, for example, that you want to find t_6 in the additive sequence whose initial terms are 3 and 7. By looking at the list of terms in the sequence

t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	
3	7	10	17	27	44	71	115	186	301	. . .

you can see that the correct value is 71. The interesting question, however, is how you can use recursion to determine this result.

The key insight you need to discover is that the *n*th term in any additive sequence is simply the *n*–1st term in the additive sequence which begins one step further along. For example, t_6 in the sequence shown in the most recent example is simply t_5 in the additive sequence

t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	
7	10	17	27	44	71	115	186	301	. . .

that begins with 7 and 10.

This discovery makes it possible to implement the function **`AdditiveSequence`** as follows:

```
int AdditiveSequence(int n, int t0, int t1) {
    if (n == 0) return t0;
    if (n == 1) return t1;
    return AdditiveSequence(n - 1, t1, t0 + t1);
}
```

If you trace through the steps in the calculation of **`Fib(5)`** using this technique, you will discover that the calculation involves none of the redundant computation that plagued the earlier recursive formulation. The steps lead directly to the solution, as shown in the following diagram:

```
Fib(5)
  = AdditiveSequence(5, 0, 1)
    = AdditiveSequence(4, 1, 1)
      = AdditiveSequence(3, 1, 2)
        = AdditiveSequence(2, 2, 3)
          = AdditiveSequence(1, 3, 5)
            = 5
```

Even though the new implementation is entirely recursive, it is comparable in efficiency to the standard iterative version of the Fibonacci function.

5.4 Other examples of recursion

Although the factorial and Fibonacci functions provide excellent examples of how recursive functions work, they are both mathematical in nature and may therefore convey the incorrect impression that recursion is applicable only to mathematical functions. In fact, you can apply recursion to any problem that can be decomposed into simpler problems of the same form. It is useful to consider a few additional examples, including several that are far less mathematical in their character.

Detecting palindromes

A **palindrome** is a string that reads identically backward and forward, such as **`"level"`** or **`"noon"`**. Although it is easy to check whether a string is a palindrome by iterating through its characters, palindromes can also be defined recursively. The insight you need to do so is that any palindrome longer than a single character must contain a shorter palindrome in its interior. For example, the string **`"level"`** consists of the palindrome **`"eve"`** with an **`"l"`** at each end. Thus, to check whether a string is a palindrome—assuming the string is sufficiently long that it does not constitute a simple case—all you need to do is

1. Check to see that the first and last characters are the same.
2. Check to see whether the substring generated by removing the first and last characters is itself a palindrome.

If both conditions apply, the string is a palindrome.

The only other question you must consider before writing a recursive solution to the palindrome problem is what the simple cases are. Clearly, any string with only a single character is a palindrome because reversing a one-character string has no effect. The one-character string therefore represents a simple case, but it is not the only one. The empty string—which contains no characters at all—is also a palindrome, and any recursive solution must operate correctly in this case as well.

Figure 5-3 contains a recursive implementation of the predicate function **`IsPalindrome(str)`** that returns **`true`** if and only if the string **`str`** is a palindrome. The function first checks to see whether the length of the string is less than 2. If it is, the string is certainly a palindrome. In not, the function checks to make sure that the string meets both of the criteria listed earlier.

This implementation in Figure 5-3 is somewhat inefficient, even though the recursive decomposition is easy to follow. You can write a more efficient implementation of **`IsPalindrome`** by making the following changes:

• *Calculate the length of the argument string only once.* The initial implementation calculates the length of the string at every level of the recursive decomposition, even though the structure of the solution guarantees that the length of the string decreases by two on every recursive call. By calculating the length of the string at the beginning and passing it down through each of the recursive calls, you can eliminate many calls to the **`length`** method. To avoid changing the prototype for **`IsPalindrome`**, you need

Figure 5-3 Recursive implementation of `IsPalindrome`

```
/*
 * Function: IsPalindrome
 * Usage: if (IsPalindrome(str)) . . .
 * ----------------------------------
 * This function returns true if and only if the string is a.
 * palindrome. This implementation operates recursively by noting
 * that all strings of length 0 or 1 are palindromes (the simple
 * case) and that longer strings are palindromes only if their first
 * and last characters match and the remaining substring is a
 * palindrome.
 */

bool IsPalindrome(string str) {
    int len = str.length();
    if (len <= 1) {
        return true;
    } else {
        return (str[0] == str[len - 1]
                && IsPalindrome(str.substr(1, len - 2)));
    }
}
```

to define **IsPalindrome** as a wrapper function and have it pass the information to a second recursive function that does all the actual work.

- *Don't make a substring on each call*. Instead of calling **substr** to make copy of the interior of the string, you can pass the first and last position for the substring as parameters and allow those positions to define the subregion of the string being checked.

The revised implementation of **IsPalindrome** appears in Figure 5-4.

Binary search

When you work with arrays or vectors, one of the most common algorithmic operations consists of searching the array for a particular element. For example, if you were working with arrays of strings, it would be extremely useful to have a function

Figure 5-4 More efficient implementation of IsPalindrome

```
/*
 * Function: IsPalindrome
 * Usage: if (IsPalindrome(str)) . . .
 * -----------------------------------
 * This function returns true if and only if the character string
 * str is a palindrome.  This level of the implementation is
 * just a wrapper for the CheckPalindrome function, which
 * does the real work.
 */

bool IsPalindrome(string str) {
    return CheckPalindrome(str, 0, str.length() - 1);
}

/*
 * Function: CheckPalindrome
 * Usage: if (CheckPalindrome(str, firstPos, lastPos)) . . .
 * ----------------------------------------------------------
 * This function returns true if the characters from firstPos
 * to lastPos in the string str form a palindrome.  The
 * implementation uses the recursive insight that all
 * strings of length 0 or 1 are palindromes (the simple
 * case) and that longer strings are palindromes only if
 * their first and last characters match and the remaining
 * substring is a palindrome.  Recursively examining the
 * interior substring is performed by adjusting the indexes
 * of the range to examine.  The interior substring
 * begins at firstPos+1 and ends at lastPos-1.
 */

bool CheckPalindrome(string str, int firstPos, int lastPos) {
    if (firstPos >= lastPos) {
        return true;
    } else {
        return (str[firstPos] == str[lastPos]
                && CheckPalindrome(str, firstPos + 1, lastPos - 1));
    }
}
```

```
    int FindStringInArray(string key, string array[], int n);
```

that searches through each of the `n` elements of `array`, looking for an element whose value is equal to `key`. If such an element is found, `FindStringInArray` returns the index at which it appears (if the key appears more than once in the array, the index of any matching is fine). If no matching element exists, the function returns –1.

If you have no specific knowledge about the order of elements within the array, the implementation of `FindStringInArray` must simply check each of the elements in turn until it either finds a match or runs out of elements. This strategy is called the **linear search algorithm,** which can be time-consuming if the arrays are large. On the other hand, if you know that the elements of the array are arranged in alphabetical order, you can adopt a much more efficient approach. All you have to do is divide the array in half and compare the key you're trying to find against the element closest to the middle of the array, using the order defined by the ASCII character codes, which is called **lexicographic order.** If the key you're looking for precedes the middle element, then the key—if it exists at all—must be in the first half. Conversely, if the key follows the middle element in lexicographic order, you only need to look at the elements in the second half. This strategy is called the **binary search algorithm.** Because binary search makes it possible for you to discard half the possible elements at each step in the process, it turns out to be much more efficient than linear search for sorted arrays.

The binary search algorithm is also a perfect example of the divide-and-conquer strategy. It is therefore not surprising that binary search has a natural recursive implementation, which is shown in Figure 5-5. Note that the function `FindStringInSortedArray` is implemented as a wrapper, leaving the real work to the recursive function `BinarySearch`, which takes two indices—`low` and `high`—that limit the range of the search.

The simple cases for `BinarySearch` are

1. *There are no elements in the active part of the array.* This condition is marked by the fact that the index `low` is greater than the index `high`, which means that there are no elements left to search.
2. *The middle element (or an element to one side of the middle if the array contains an even number of elements) matches the specified key.* Since the key has just been found, `FindStringInSortedArray` can simply return the index of the middle value.

If neither of these cases applies, however, the implementation can simplify the problem by choosing the appropriate half of the array and call itself recursively with an updated set of search limits.

Mutual recursion

In each of the examples considered so far, the recursive functions have called themselves directly, in the sense that the body of the function contains a call to itself. Although most of the recursive functions you encounter are likely to adhere to this style, the definition of *recursion* is actually somewhat broader. To be recursive, a function must call itself at some point during its evaluation. If a function is subdivided into subsidiary functions, the recursive call can actually occur at a deeper level of nesting. For example, if a function f calls a function g, which in turn calls f, the function calls are still considered to be recursive. Because the functions f and g call each other, this type of recursion is called **mutual recursion.**

Figure 5-5 Divide-and-conquer implementation of binary search

```
/*
 * Function: FindStringInSortedArray
 * Usage: index = FindStringInSortedArray(key, array, n);
 * ------------------------------------------------------
 * This function searches the array looking for the specified
 * key. The argument n specifies the effective size of the
 * array, which must be sorted according to lexicographic
 * order.  If the key is found, the function returns the
 * index in the array at which that key appears. (If the key
 * appears more than once in the array, any of the matching
 * indices may be returned).  If the key does not exist in
 * the array, the function returns -1.  In this implementation,
 * FindStringInSortedArray is simply a wrapper; all the work
 * is done by the recursive function BinarySearch.
 */

int FindStringInSortedArray(string key, string array[], int n) {
    return BinarySearch(key, array, 0, n - 1);
}

/*
 * Function: BinarySearch
 * Usage: index = BinarySearch(key, array, low, high);
 * ---------------------------------------------------
 * This function does the work for FindStringInSortedArray.
 * The only difference is that BinarySearch takes both the
 * upper and lower limit of the search.
 */

int BinarySearch(string key, string array[], int low, int high) {
    if (low > high) return -1;
    int mid = (low + high) / 2;
    if (key == array[mid]) return mid;
    if (key < array[mid]) {
        return BinarySearch(key, array, low, mid - 1);
    } else {
        return BinarySearch(key, array, mid + 1, high);
    }
}
```

As a simple example, let's investigate how to use recursion to test whether a number is even or odd. If you limit the domain of possible values to the set of **natural numbers,** which are defined simply as the set of nonnegative integers, the even and odd numbers can be characterized as follows:

- A number is *even* if its predecessor is odd.
- A number is *odd* if is not even.
- The number 0 is even by definition.

Even though these rules seem simplistic, they constitute the basis of an effective, if inefficient, strategy for distinguishing odd and even numbers. A mutually recursive implementation of the predicate functions `IsEven` and `IsOdd` appears in Figure 5-6.

Figure 5-6 Mutually recursive definitions of `IsEven` and `IsOdd`

```
/*
 * Function: IsEven
 * Usage: if (IsEven(n)) . . .
 * ---------------------------
 * This function returns true if n is even.  The number 0
 * is considered even by definition; any other number is
 * even if its predecessor is odd.  Note that this function
 * is defined to take an unsigned argument and is therefore
 * not applicable to negative integers.
 */

bool IsEven(unsigned int n) {
    if (n == 0) {
        return true;
    } else {
        return IsOdd(n - 1);
    }
}

/*
 * Function: IsOdd
 * Usage: if (IsOdd(n)) . . .
 * --------------------------
 * This function returns true if n is odd, where a number
 * is defined to be odd if it is not even.  Note that this
 * function is defined to take an unsigned argument and is
 * therefore not applicable to negative integers.
 */

bool IsOdd(unsigned int n) {
    return !IsEven(n);
}
```

5.5 Thinking recursively

For most people, recursion is not an easy concept to grasp. Learning to use it effectively requires considerable practice and forces you to approach problems in entirely new ways. The key to success lies in developing the right mindset—learning how to think recursively. The remainder of this chapter is designed to help you achieve that goal.

Maintaining a holistic perspective

In Chapter 2 of *The Art and Science of C,* I devote one section to the philosophical concepts of holism and reductionism. Simply stated, **reductionism** is the belief that the whole of an object can be understood merely by understanding the parts that make it up. Its antithesis is **holism,** the position that the whole is often greater than the sum of its parts. As you learn about programming, it helps to be able to interleave these two perspectives, sometimes focusing on the behavior of a program as a whole, and at other times delving into the details of its execution. When you try to learn about recursion, however, this balance seems to change. Thinking recursively requires you to think holistically. In the recursive domain, reductionism is the enemy of understanding and invariably gets in the way.

To maintain the holistic perspective, you must become comfortable adopting the recursive leap of faith, which was introduced in its own section earlier in this chapter.

Whenever you are writing a recursive program or trying to understand the behavior of one, you must get to the point where you ignore the details of the individual recursive calls. As long as you have chosen the right decomposition, identified the appropriate simple cases, and implemented your strategy correctly, those recursive calls will simply work. You don't need to think about them.

Unfortunately, until you have had extensive experience working with recursive functions, applying the recursive leap of faith does not come easily. The problem is that it requires to suspend your disbelief and make assumptions about the correctness of your programs that fly in the face of your experience. After all, when you write a program, the odds are good—even if you are an experienced programmer—that your program won't work the first time. In fact, it is quite likely that you have chosen the wrong decomposition, messed up the definition of the simple cases, or somehow messed things up trying to implement your strategy. If you have done any of these things, your recursive calls won't work.

When things go wrong—as they inevitably will—you have to remember to look for the error in the right place. The problem lies somewhere in your recursive implementation, not in the recursive mechanism itself. If there is a problem, you should be able to find it by looking at a single level of the recursive hierarchy. Looking down through additional levels of recursive calls is not going to help. If the simple cases work and the recursive decomposition is correct, the subsidiary calls will work correctly. If they don't, there is something you need to fix in the definition of the recursive function itself.

Avoiding the common pitfalls

As you gain experience with recursion, the process of writing and debugging recursive programs will become more natural. At the beginning, however, finding out what you need to fix in a recursive program can be difficult. The following is a checklist that will help you identify the most common sources of error.

- *Does your recursive implementation begin by checking for simple cases?* Before you attempt to solve a problem by transforming it into a recursive subproblem, you must first check to see if the problem is so simple that such decomposition is unnecessary. In almost all cases, recursive functions begin with the keyword **`if`**. If your function doesn't, you should look carefully at your program and make sure that you know what you're doing.[1]
- *Have you solved the simple cases correctly?* A surprising number of bugs in recursive programs arise from having incorrect solutions to the simple cases. If the simple cases are wrong, the recursive solutions to more complicated problems will inherit the same mistake. For example, if you had mistakenly defined **`Fact(0)`** as 0 instead of 1, calling **`Fact`** on any argument would end up returning 0.
- *Does your recursive decomposition make the problem simpler?* For recursion to work, the problems have to get simpler as you go along. More formally, there must be some **metric**—a standard of measurement that assigns a numeric difficulty rating to the problem—that gets smaller as the computation proceeds. For mathematical functions like **`Fact`** and **`Fib`**, the value of the integer argument serves as a metric. On each recursive call, the value of the argument gets smaller. For the **`IsPalindrome`** function, the appropriate metric is the length of the argument string, because the string gets shorter on each recursive call. If the problem instances do not get simpler, the

[1] At times, as in the case of the **`IsPalindrome`** implementation, it may be necessary to perform some calculations prior to making the simple-case test. The point is that the simple-case test must precede any recursive decomposition.

decomposition process will just keep making more and more calls, giving rise to the recursive analogue of the infinite loop, which is called **nonterminating recursion.**

- *Does the simplification process eventually reach the simple cases, or have you left out some of the possibilities?* A common source of error is failing to include simple case tests for all the cases that can arise as the result of the recursive decomposition. For example, in the **`IsPalindrome`** implementation presented in Figure 5-3, it is critically important for the function to check the zero-character case as well as the one-character case, even if the client never intends to call **`IsPalindrome`** on the empty string. As the recursive decomposition proceeds, the string arguments get shorter by two characters at each level of the recursive call. If the original argument string is even in length, the recursive decomposition will never get to the one-character case.
- *Do the recursive calls in your function represent subproblems that are truly identical in form to the original?* When you use recursion to break down a problem, it is essential that the subproblems be of the same form. If the recursive calls change the nature of the problem or violate one of the initial assumptions, the entire process can break down. As several of the examples in this chapter illustrate, it is often useful to define the publicly exported function as a simple wrapper that calls a more general recursive function which is private to the implementation. Because the private function has a more general form, it is usually easier to decompose the original problem and still have it fit within the recursive structure.
- *When you apply the recursive leap of faith, do the solutions to the recursive subproblems provide a complete solution to the original problem?* Breaking a problem down into recursive subinstances is only part of the recursive process. Once you get the solutions, you must also be able to reassemble them to generate the complete solution. The way to check whether this process in fact generates the solution is to walk through the decomposition, religiously applying the recursive leap of faith. Work through all the steps in the current function call, but assume that every recursive call generates the correct answer. If following this process yields the right solution, your program should work.

Summary

This chapter has introduced the idea of *recursion,* a powerful programming strategy in which complex problems are broken down into simpler problems of the same form. The important points presented in this chapter include:

- Recursion is similar to stepwise refinement in that both strategies consist of breaking a problem down into simpler problems that are easier to solve. The distinguishing characteristic of recursion is that the simpler subproblems must have the same form as the original.
- In C++, recursive functions typically have the following paradigmatic form:

```
if (test for simple case) {
    Compute a simple solution without using recursion.
} else {
    Break the problem down into subproblems of the same form.
    Solve each of the subproblems by calling this function recursively.
    Reassemble the solutions to the subproblems into a solution for the whole.
}
```

- To use recursion, you must be able to identify *simple cases* for which the answer is easily determined and a *recursive decomposition* that allows you to break any complex instance of the problem into simpler problems of the same type.

- Recursive functions are implemented using exactly the same mechanism as any other function call. Each call creates a new stack frame that contains the local variables for that call. Because the computer creates a separate stack frame for each function call, the local variables at each level of the recursive decomposition remain separate.
- Before you can use recursion effectively, you must learn to limit your analysis to a single level of the recursive decomposition and to rely on the correctness of all simpler recursive calls without tracing through the entire computation. Trusting these simpler calls to work correctly is called the *recursive leap of faith*.
- Mathematical functions often express their recursive nature in the form of a *recurrence relation,* in which each element of a sequence is defined in terms of earlier elements.
- Although some recursive functions may be less efficient than their iterative counterparts, recursion itself is not the problem. As is typical with all types of algorithms, some recursive strategies are more efficient than others.
- In order to ensure that a recursive decomposition produces subproblems that are identical in form to the original, it is often necessary to generalize the problem. As a result, it is often useful to implement the solution to a specific problem as a simple *wrapper* function whose only purpose is to call a subsidiary function that handles the more general case.
- Recursion need not consist of a single function that calls itself but may instead involve several functions that call each other in a cyclical pattern. Recursion that involves more than one function is called *mutual recursion*.
- You will be more successful at understanding recursive programs if you can maintain a holistic perspective rather than a reductionistic one.

Thinking about recursive problems in the right way does not come easily. Learning to use recursion effectively requires practice and more practice. For many students, mastering the concept takes years. But because recursion will turn out to be one of the most powerful techniques in your programming repertoire, that time will be well spent.

Review questions

1. Define the terms *recursive* and *iterative*. Is it possible for a function to employ both strategies?

2. What is the fundamental difference between recursion and stepwise refinement?

3. In the pseudocode for the **`CollectContributions`** function, the **`if`** statement looks like this:

   ```
   if (n <= 100)
   ```

   Why is it important to use the **`<=`** operator instead of simply checking whether **`n`** is exactly equal to 100?

4. What is the standard recursive paradigm?

5. What two properties must a problem have for recursion to make sense as a solution strategy?

6. Why is the term *divide and conquer* appropriate to recursive techniques?

7. What is meant by the *recursive leap of faith?* Why is this concept important to you as a programmer?

8. In the section entitled "Tracing the recursive process," the text goes through a long analysis of what happens internally when **`Fact(4)`** is called. Using this section as a model, trace through the execution of **`Fib(4)`**, sketching out each stack frame created in the process.

9. Modify Fibonacci's rabbit problem by introducing the additional rule that rabbit pairs stop reproducing after giving birth to three litters. How does this assumption change the recurrence relation? What changes do you need to make in the simple cases?

10. How many times is **`Fib(1)`** called when calculating **`Fib(n)`** using the recursive implementation given in Figure 5-1?

11. What would happen if you eliminated the **`if (n == 1)`** check from the function **`AdditiveSequence`**, so that the implementation looked like this:

```
int AdditiveSequence(int n, int t0, int t1) {
    if (n == 0) return t0;
    return AdditiveSequence(n - 1, t1, t0 + t1);
}
```

    Would the function still work? Why or why not?

12. What is a wrapper function? Why are they often useful in writing recursive functions?

13. Why is it important that the implementation of **`IsPalindrome`** in Figure 5-3 check for the empty string as well as the single character string? What would happen if the function didn't check for the single character case and instead checked only whether the length is 0? Would the function still work correctly?

14. Explain the effect of the function call

```
CheckPalindrome(str, firstPos + 1, lastPost - 1)
```

    in the **`IsPalindrome`** implementation given in Figure 5-4.

15. What is mutual recursion?

16. What would happen if you defined **`IsEven`** and **`IsOdd`** as follows:

```
bool IsEven(unsigned int n) {
    return !IsOdd(n);
}

bool IsOdd(unsigned int n) {
    return !IsEven(n);
}
```

    Which of the errors explained in the section "Avoiding the common pitfalls" is illustrated in this example?

17. The following definitions of `IsEven` and `IsOdd` are also incorrect:

```
bool IsEven(unsigned int n) {
    if (n == 0) {
        return true;
    } else {
        return IsOdd(n - 1);
    }
}

bool IsOdd(unsigned int n) {
    if (n == 1) {
        return true;
    } else {
        return IsEven(n - 1);
    }
}
```

Give an example that shows how this implementation can fail. What common pitfall is illustrated here?

Programming exercises

1. Spherical objects, such as cannonballs, can be stacked to form a pyramid with one cannonball at the top, sitting on top of a square composed of four cannonballs, sitting on top of a square composed of nine cannonballs, and so forth. Write a recursive function `Cannonball` that takes as its argument the height of the pyramid and returns the number of cannonballs it contains. Your function must operate recursively and must not use any iterative constructs, such as `while` or `for`.

2. Unlike many programming languages, C++ does not include a predefined operator that raises a number to a power. As a partial remedy for this deficiency, write a recursive implementation of a function

```
int RaiseIntToPower(int n, int k)
```

that calculates n^k. The recursive insight that you need to solve this problem is the mathematical property that

$$n^k = \begin{cases} 1 & \text{if } k = 0 \\ n \times n^{k-1} & \text{otherwise} \end{cases}$$

3. The **greatest common divisor** (g.c.d.) of two nonnegative integers is the largest integer that divides evenly into both. In the third century B.C., the Greek mathematician Euclid discovered that the greatest common divisor of *x* and *y* can always be computed as follows:

- If *x* is evenly divisible by *y,* then *y* is the greatest common divisor.
- Otherwise, the greatest common divisor of *x* and *y* is always equal to the greatest common divisor of *y* and the remainder of *x* divided by *y*.

Use Euclid's insight to write a recursive function `GCD(x, y)` that computes the greatest common divisor of *x* and *y*.

4. Write an iterative implementation of the function **`Fib(n)`**.

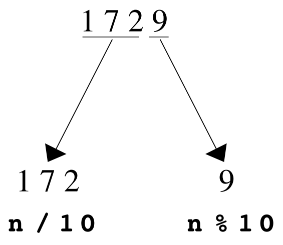
5. For each of the two recursive implementations of the function **`Fib(n)`** presented in this chapter, write a recursive function (you can call these **`CountFib1`** and **`CountFib2`** for the two algorithms) that counts the number of function calls made during the evaluation of the corresponding Fibonacci calculation. Write a main program that uses these functions to display a table showing the number of calls made by each algorithm for various values of **`n`**, as shown in the following sample run:

```
This program compares the performance of two
algorithms to compute the Fibonacci sequence.

Number of calls:
   N     Fib1   Fib2
  --     ----   ----
   0        1      2
   1        1      3
   2        3      4
   3        5      5
   4        9      6
   5       15      7
   6       25      8
   7       41      9
   8       67     10
   9      109     11
  10      177     12
  11      287     13
  12      465     14
```

6. Write a recursive function **`DigitSum(n)`** that takes a nonnegative integer and returns the sum of its digits. For example, calling **`DigitSum(1729)`** should return 1 + 7 + 2 + 9, which is 19.

   The recursive implementation of **`DigitSum`** depends on the fact that it is very easy to break an integer down into two components using division by 10. For example, given the integer 1729, you can divide it into two pieces as follows:



   Each of the resulting integers is strictly smaller than the original and thus represents a simpler case.

7. The **digital root** of an integer *n* is defined as the result of summing the digits repeatedly until only a single digit remains. For example, the digital root of 1729 can be calculated using the following steps:

Step 1:	$1 + 7 + 2 + 9$	$\rightarrow$	19
Step 2:	$1 + 9$	$\rightarrow$	10
Step 3:	$1 + 0$	$\rightarrow$	1

Because the total at the end of step 3 is the single digit 1, that value is the digital root.

Write a function **`DigitalRoot(n)`** that returns the digital root of its argument. Although it is easy to implement **`DigitalRoot`** using the **`DigitSum`** function from exercise 6 and a **`while`** loop, part of the challenge of this problem is to write the function recursively without using any explicit loop constructs.

8. The mathematical combinations function $C(n, k)$ is usually defined in terms of factorials, as follows:

$$C(n, k) \quad = \quad \frac{n!}{k! \times (n-k)!}$$

The values of $C(n, k)$ can also be arranged geometrically to form a triangle in which n increases as you move down the triangle and k increases as you move from left to right. The resulting structure,, which is called *Pascal's Triangle* after the French mathematician Blaise Pascal, is arranged like this:

$$\begin{array}{c}
C(0,0) \\
C(1,0) \quad C(1,1) \\
C(2,0) \quad C(2,1) \quad C(2,2) \\
C(3,0) \quad C(3,1) \quad C(3,2) \quad C(3,3) \\
C(4,0) \quad C(4,1) \quad C(4,2) \quad C(4,3) \quad C(4,4)
\end{array}$$

Pascal's Triangle has the interesting property that every entry is the sum of the two entries above it, except along the left and right edges, where the values are always 1. Consider, for example, the circled entry in the following display of Pascal's Triangle:

$$\begin{array}{c}
1 \\
1 \quad 1 \\
1 \quad 2 \quad 1 \\
1 \quad 3 \quad 3 \quad 1 \\
1 \quad 4 \quad 6 \quad 4 \quad 1 \\
1 \quad 5 \quad 10 \quad 10 \quad 5 \quad 1 \\
1 \quad 6 \quad \textcircled{15} \quad 20 \quad 15 \quad 6 \quad 1
\end{array}$$

This entry, which corresponds to $C(6, 2)$, is the sum of the two entries—5 and 10—that appear above it to either side. Use this relationship between entries in Pascal's Triangle to write a recursive implementation of the **`Combinations`** function that uses no loops, no multiplication, and no calls to **`Fact`**.

9. Write a recursive function that takes a string as argument and returns the reverse of that string. The prototype for this function should be

```
string Reverse(string str);
```

and the statement

```
cout << Reverse("program") << endl;
```

should display

```
margorp
```

Your solution should be entirely recursive and should not use any iterative constructs such as **`while`** or **`for`**.

10. The **`strutils.h`** library contains a function **`IntegerToString`**,. You might have wondered how the computer actually goes about the process of converting an integer into its string representation. As it turns out, the easiest way to implement this function is to use the recursive decomposition of an integer outlined in exercise 6. Rewrite the **`IntegerToString`** implementation so that it operates recursively without using use any of the iterative constructs such as **`while`** and **`for`**.