



Recursion 14

14.1 RECURSIVE FUNCTIONS FOR TASKS 791

Case Study: Vertical Numbers 791

A Closer Look at Recursion 797

Pitfall: Infinite Recursion 799

Stacks for Recursion 800

Pitfall: Stack Overflow 802

Recursion Versus Iteration 802

14.2 RECURSIVE FUNCTIONS FOR VALUES 804

General Form for a Recursive Function That Returns a Value 804


Programming Example: Another Powers Function 804

14.3 THINKING RECURSIVELY 809

Recursive Design Techniques 809

Case Study: Binary Search—An Example of Recursive Thinking 810

Programming Example: A Recursive Member Function 818



After a lecture on cosmology and the structure of the solar system, William James was accosted by a little old lady.

"Your theory that the sun is the center of the solar system, and the earth is a ball which rotates around it has a very convincing ring to it, Mr. James, but it's wrong. I've got a better theory," said the little old lady.

"And what is that, madam?" inquired James politely.

"That we live on a crust of earth which is on the back of a giant turtle."

Not wishing to demolish this absurd little theory by bringing to bear the masses of scientific evidence he had at his command, James decided to gently dissuade his opponent by making her see some of the inadequacies of her position.

"If your theory is correct, madam," he asked, "what does this turtle stand on?"

"You're a very clever man, Mr. James, and that's a very good question," replied the little old lady, "but I have an answer to it. And it is this: the first turtle stands on the back of a second, far larger, turtle, who stands directly under him."

"But what does this second turtle stand on?" persisted James patiently.

To this the little old lady crowed triumphantly. "It's no use, Mr. James—it's turtles all the way down."

J. R. ROSS, *Constraints on Variables in Syntax*

INTRODUCTION

You have encountered a few cases of circular definitions that worked out satisfactorily. The most prominent examples are the definitions of certain C++ statements. For example, the definition of a *while* statement says that it can contain other (smaller) statements. Since one of the possibilities for these smaller statements is another *while* statement, there is a kind of circularity in that definition. The definition of the *while* statement, if written out in complete detail, will contain a reference to *while* statements. In mathematics, this kind of circular definition is called a recursive definition. In C++, a function may be defined in terms of itself in the same way. To put it more precisely, a function definition may contain a call to itself. In such cases, the function is said to be recursive. This chapter discusses recursion in C++ and more generally discusses recursion as a programming and problem-solving technique.

PREREQUISITES

Sections 14.1 and 14.2 use material only from Chapters 2 through 5. Section 14.3 uses material from Chapters 2 through 7 and 10.

14.1 RECURSIVE FUNCTIONS FOR TASKS

I remembered too that night which is at the middle of the Thousand and One Nights when Scheherazade (through a magical oversight of the copyist) begins to relate word for word the story of the Thousand and One Nights, establishing the risk of coming once again to the night when she must repeat it, and thus to infinity.

JORGE LUIS BORGES, *The Garden of Forking Paths*

When you are writing a function to solve a task, one basic design technique is to break the task into subtasks. Sometimes it turns out that at least one of the subtasks is a smaller example of the same task. For example, if the task is to search an array for a particular value, you might divide this into the subtask of searching the first half of the array and the subtask of searching the second half of the array. The subtasks of searching the halves of the array are “smaller” versions of the original task. Whenever one subtask is a smaller version of the original task to be accomplished, you can solve the original task using a recursive function. It takes a little training to easily decompose problems this way, but once you learn the technique, it can be one of the quickest ways to design an algorithm, and, ultimately, a C++ function. We begin with a simple case study to illustrate this technique.

Recursion

In C++, a function definition may contain a call to the function being defined. In such cases, the function is said to be **recursive**.

CASE STUDY Vertical Numbers

In this case study we design a recursive *void* function that writes numbers to the screen with the digits written vertically, so that, for example, 1984 would be written as



1
9
8
4

Problem Definition

The declaration and header comment for our function is as follows:

```
void write_vertical(int n);  
//Precondition: n >= 0.  
//Postcondition: The number n is written to the screen  
//vertically with each digit on a separate line.
```

Algorithm Design

One case is very simple. If n , the number to be written out, is only one digit long, then just write out the number. As simple as it is, this case is still important, so let's keep track of it.

Simple Case: If $n < 10$, then write the number n to the screen.

Now let's consider the more typical case in which the number to be written out consists of more than one digit. Suppose you want to write the number 1234 vertically so that the result is

```
1
2
3
4
```

One way to decompose this task into two subtasks is the following:

1. Output all the digits except the last digit like so:


```
1
2
3
```

2. Output the last digit, which in this example is 4.

Subtask 1 is a smaller version of the original task, so we can implement this subtask with a recursive call. Subtask 2 is just the simple case we listed earlier. Thus, an outline of our algorithm for the function `write_vertical` with parameter n is given by the following pseudocode:

```
if (n < 10)
{
    cout << n << endl;
}
else //n is two or more digits long:
{
    write_vertical(the number n with the last digit removed);
    cout << the last digit of n << endl;
}
```

Recursive subtask



In order to convert this pseudocode into the code for a C++ function, all we need to do is translate the following two pieces of pseudocode into C++ expressions:

the number n with the last digit removed

and

the last digit of n

These expressions can easily be translated into C++ expressions using the integer division operators `/` and `%` as follows:

```
n / 10 //the number n with the last digit removed
n % 10 //the last digit of n
```

For example, `1234 / 10` evaluates to 123, and `1234 % 10` evaluates to 4.

Several factors influenced our selection of the two subtasks we used in this algorithm. One was that we could easily compute the argument for the recursive call to `write_vertical` (shown in color) that we used in the pseudocode. The number `n` with the last digit removed is easily computed as `n/10`. As an alternative, you might have been tempted to divide the subtasks as follows:

1. Output the first digit of `n`.
2. Output the number `n` with the first digit removed.

This is a perfectly valid decomposition of the task into subtasks, and it can be implemented recursively. However, it is difficult to calculate the result of removing the first digit from a number, while it is easy to calculate the result of removing the last digit from a number.

Another reason for choosing these sorts of decompositions is that one of the subcases does not involve a recursive call. A successful definition of a recursive function always includes at least one case that does not involve a recursive call (as well as one or more cases that do involve at least one recursive call). This aspect of the recursive algorithm is discussed in the subsections that follow this case study.

Coding

We can now put all the pieces together to produce the recursive function `write_vertical` shown in Display 14.1. In the next subsection we will explain more details of how recursion works in this example.

DISPLAY 14.1 A Recursive Output Function (part 1 of 2)

```
1  //Program to demonstrate the recursive function write_vertical.
2  #include <iostream>
3  using namespace std;
4
5  void write_vertical(int n);
6  //Precondition: n >= 0.
7  //Postcondition: The number n is written to the screen vertically
8  //with each digit on a separate line.
9
10 int main( )
11 {
12     cout<< "write_vertical(3):" <<endl;
13     write_vertical(3);
14
```

(continued)

DISPLAY 14.1 A Recursive Output Function *(part 2 of 2)*

```
15     cout<< "write_vertical(12):" <<endl;
16     write_vertical(12);
17
18     cout<< "write_vertical(123):" <<endl;
19     write_vertical(123);
20
21     return 0;
22 }
23
24 //uses iostream:
25 void write_vertical(int n)
26 {
27     if (n < 10)
28     {
29         cout << n << endl;
30     }
31     else //n is two or more digits long:
32     {
33         write_vertical(n / 10);
34         cout << (n % 10) << endl;
35     }
36 }
```

Sample Dialogue

```
write_vertical(3):
3
write_vertical(12):
1
2
write_vertical(123):
1
2
3
```

Tracing a Recursive Call

Let's see exactly what happens when the following function call is made:

```
write_vertical(123);
```

When this function call is executed, the computer proceeds just as it would with any function call. The argument 123 is substituted for the parameter *n* in the function definition, and the body of the function is executed. After the substitution of 123 for *n*, the code to be executed is as follows:

```

if (123 < 10)
{
    cout << 123 << endl;
}
else //n is two or more digits long:
{
    write_vertical(123 / 10); ← Computation will
    cout << (123 % 10) << endl; stop here until the
                                recursive call returns.
}

```

Since 123 is not less than 10, the logical expression in the *if-else* statement is *false*, so the *else* part is executed. However, the *else* part begins with the following function call:

```
write_vertical(n / 10);
```

which (since *n* is equal to 123) is the call

```
write_vertical(123 / 10);
```

which is equivalent to

```
write_vertical(12);
```

When execution reaches this recursive call, the current function computation is placed in suspended animation and this recursive call is executed. When this recursive call is finished, the execution of the suspended computation will return to this point, and the suspended computation will continue from this point.

The recursive call

```
write_vertical(12);
```

is handled just like any other function call. The argument 12 is substituted for the parameter *n* and the body of the function is executed. After substituting 12 for *n*, there are two computations, one suspended and one active, as follows:

<pre> if (123 < 10) { cout << 123 << endl; } else //n is two or more digits long: { write_vertical(123 / 10); cout << (123 % 10) << endl; } </pre>	<pre> if (12 < 10) { cout << 12 << endl; } else //n is two or more digits long: { write_vertical(12 / 10); ← Computation will stop cout << (12 % 10) << endl; here until the recursive call returns. } </pre>
---	--

Since 12 is not less than 10, the Boolean expression in the *if-else* statement is *false* and so the *else* part is executed. However, as you already saw, the *else* part begins with a recursive call. The argument for the recursive call is *n* / 10, which

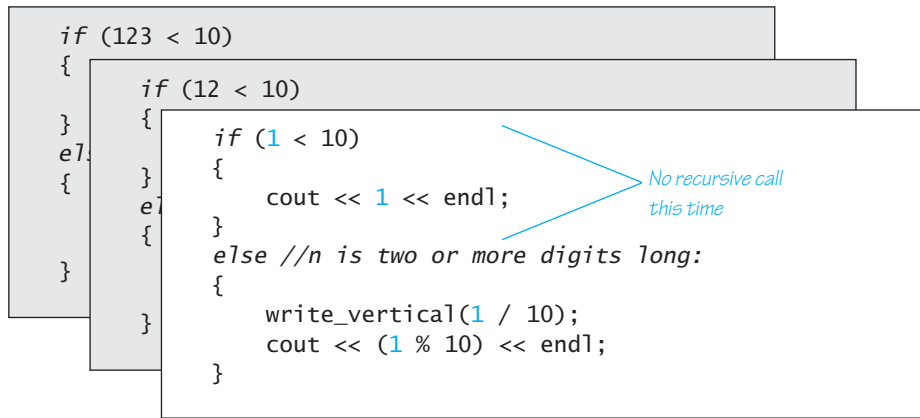
in this case is equivalent to $12 / 10$. So this second computation of the function `write_vertical` is suspended and the following recursive call is executed:

```
write_vertical(12/ 10);
```

which is equivalent to

```
write_vertical(1);
```

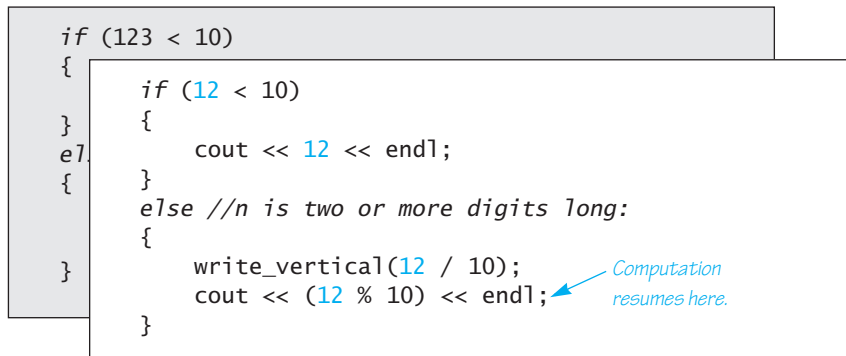
At this point there are two suspended computations waiting to resume and the computer begins to execute this new recursive call, which is handled just like all the previous recursive calls. The argument 1 is substituted for the parameter `n`, and the body of the function is executed. At this point, the computation looks like the following:



Output the digit 1

When the body of the function is executed this time, something different happens. Since 1 is less than 10, the Boolean expression in the `if-else` statement is `true`, so the statement before the `else` is executed. That statement is simply a `cout` statement that writes the argument 1 to the screen, and so the call `write_vertical(1)` writes 1 to the screen and ends without any recursive call.

When the call `write_vertical(1)` ends, the suspended computation that is waiting for it to end resumes where that suspended computation left off, as shown by the following:



When this suspended computation resumes, it executes a `cout` statement that outputs the value `123 % 10`, which is 2. That ends that computation, but there is yet another suspended computation waiting to resume. When this last suspended computation resumes, the situation is as follows:

Output the
digit 2

```
if (123 < 10)
{
    cout << 123 << endl;
}
else //n is two or more digits long:
{
    write_vertical(123 / 10);
    cout << (123 % 10) << endl;
}
```

Computation
resumes here.

When this last suspended computation resumes, it outputs the value `123 % 10`, which is 3, and the execution of the original function call ends. And, sure enough, the digits 1, 2, and 3 have been written to the screen one per line, in that order.

Output the digit 3

A Closer Look at Recursion

The definition of the function `write_vertical` uses recursion. Yet we did nothing new or different in evaluating the function call `write_vertical(123)`. We treated it just like any of the function calls we saw in previous chapters. We just substituted the argument 123 for the parameter `n` and then executed the code in the body of the function definition. When we reached the recursive call

```
write_vertical(123 / 10);
```

we simply repeated this process one more time.

The computer keeps track of recursive calls in the following way. When a function is called, the computer plugs in the arguments for the parameter(s) and begins to execute the code. If it should encounter a recursive call, then it temporarily stops its computation. This is because it must know the result of the recursive call before it can proceed. It saves all the information it needs to continue the computation later on and proceeds to evaluate the recursive call. When the recursive call is completed, the computer returns to finish the outer computation.

How recursion
works

The C++ language places no restrictions on how recursive calls are used in function definitions. However, in order for a recursive function definition to be useful, it must be designed so that any call of the function must ultimately terminate with some piece of code that does not depend on recursion. The function may call itself, and that recursive call may call the function again.

How recursion
ends

The process may be repeated any number of times. However, the process will not terminate unless eventually one of the recursive calls does not depend on recursion. The general outline of a successful recursive function definition is as follows:

- One or more cases in which the function accomplishes its task by using recursive calls to accomplish one or more smaller versions of the task.
- One or more cases in which the function accomplishes its task without the use of any recursive calls. These cases without any recursive calls are called **base cases** or **stopping cases**.

Often, an *if-else* statement determines which of the cases will be executed. A typical scenario is for the original function call to execute a case that includes a recursive call. That recursive call may in turn execute a case that requires another recursive call. For some number of times each recursive call produces another recursive call, but eventually one of the stopping cases should apply. *Every call of the function must eventually lead to a stopping case, or else the function call will never end because of an infinite chain of recursive calls.* (In practice, a call that includes an infinite chain of recursive calls will usually terminate abnormally rather than actually running forever.)

The most common way to ensure that a stopping case is eventually reached is to write the function so that some (positive) numeric quantity is decreased on each recursive call and to provide a stopping case for some “small” value. This is how we designed the function `write_vertical` in Display 14.1. When the function `write_vertical` is called, that call produces a recursive call with a smaller argument. This continues with each recursive call producing another recursive call until the argument is less than 10. When the argument is less than 10, the function call ends without producing any more recursive calls and the process works its way back to the original call and then ends.

General Form of a Recursive Function Definition

The general outline of a successful recursive function definition is as follows:

- One or more cases that include one or more recursive calls to the function being defined. These recursive calls should solve “smaller” versions of the task performed by the function being defined.
- One or more cases that include no recursive calls. These cases without any recursive calls are called **base cases** or **stopping cases**.

PITFALL Infinite Recursion

In the example of the function `write_vertical` discussed in the previous subsections, the series of recursive calls eventually reached a call of the function that did not involve recursion (that is, a stopping case was reached). If, on the other hand, every recursive call produces another recursive call, then a call to the function will, in theory, run forever. This is called **infinite recursion**. In practice, such a function will typically run until the computer runs out of resources and the program terminates abnormally. Phrased another way, a recursive definition should not be “recursive all the way down.” Otherwise, like the lady’s explanation of the universe given at the start of this chapter, a call to the function will never end, except perhaps in frustration.

Examples of infinite recursion are not hard to come by. The following is a syntactically correct C++ function definition, which might result from an attempt to define an alternative version of the function `write_vertical`:

```
void new_write_vertical(int n)
{
    new_write_vertical(n / 10);
    cout << (n % 10) << endl;
}
```

If you embed this definition in a program that calls this function, the compiler will translate the function definition to machine code and you can execute the machine code. Moreover, the definition even has a certain reasonableness to it. It says that to output the argument to `new_write_vertical`, first output all but the last digit and then output the last digit. However, when called, this function will produce an infinite sequence of recursive calls. If you call `new_write_vertical(12)`, that execution will stop to execute the recursive call `new_write_vertical (12/10)`, which is equivalent to `new_write_vertical(1)`. The execution of that recursive call will, in turn, stop to execute the recursive call

```
new_write_vertical(1/10);
```

which is equivalent to

```
new_write_vertical(0);
```

That, in turn, will stop to execute the recursive call `new_write_vertical (0/10)`; which is also equivalent to

```
new_write_vertical(0);
```

and that will produce another recursive call to again execute the same recursive function call `new_write_vertical(0)`; and so on, forever. Since the definition of `new_write_vertical` has no stopping case, the process will proceed forever (or until the computer runs out of resources). ■

SELF-TEST EXERCISES

1. What is the output of the following program?

```
#include <iostream>
using namespace std;
void cheers(int n);

int main()
{
    cheers(3);
    return 0;
}

void cheers(int n)
{
    if (n == 1)
    {
        cout << "Hurray\n";
    }
    else
    {
        cout << "Hip ";
        cheers(n - 1);
    }
}
```

2. Write a recursive *void* function that has one parameter which is a positive integer and that writes out that number of asterisks '*' to the screen all on one line.
3. Write a recursive *void* function that has one parameter, which is a positive integer. When called, the function writes its argument to the screen backward. That is, if the argument is 1234, it outputs the following to the screen:
4321
4. Write a recursive *void* function that takes a single *int* argument *n* and writes the integers 1, 2, . . . , *n*.
5. Write a recursive *void* function that takes a single *int* argument *n* and writes integers *n*, *n*-1, . . . , 3, 2, 1. (*Hint*: Notice that you can get from the code for Self-Test Exercise 4 to that for Self-Test Exercise 5, or vice versa, by an exchange of as little as two lines.)

Stacks for Recursion

In order to keep track of recursion, and a number of other things, most computer systems make use of a structure called a *stack*. A **stack** is a very specialized kind of memory structure that is analogous to a stack of paper. In

this analogy there is an inexhaustible supply of extra blank sheets of paper. To place some information in the stack, it is written on one of these sheets of paper and placed on top of the stack of papers. To place more information in the stack, a clean sheet of paper is taken, the information is written on it, and this new sheet of paper is placed on top of the stack. In this straightforward way, more and more information may be placed on the stack.

Getting information out of the stack is also accomplished by a very simple procedure. The top sheet of paper can be read, and when it is no longer needed, it is thrown away. There is one complication: Only the top sheet of paper is accessible. In order to read, say, the third sheet from the top, the top two sheets must be thrown away. Since the last sheet that is put on the stack is the first sheet taken off the stack, a stack is often called a **last-in/first-out (LIFO)** memory structure.

Using a stack, the computer can easily keep track of recursion. Whenever a function is called, a new sheet of paper is taken. The function definition is copied onto this sheet of paper, and the arguments are plugged in for the function parameters. Then the computer starts to execute the body of the function definition. When it encounters a recursive call, it stops the computation it is doing on that sheet in order to compute the recursive call. But before computing the recursive call, it saves enough information so that, when it does finally complete the recursive call, it can continue the stopped computation. This saved information is written on a sheet of paper and placed on the stack. A new sheet of paper is used for the recursive call. The computer writes a second copy of the function definition on this new sheet of paper, plugs in the arguments for the function parameters, and starts to execute the recursive call. When it gets to a recursive call within the recursively called copy, it repeats the process of saving information on the stack and using a new sheet of paper for the new recursive call. This process is illustrated in the earlier subsection entitled "Tracing a Recursive Call." Even though we did not call it a stack in that section, the illustrations of computations placed one on top of the other demonstrate the actions of the stack.

This process continues until some recursive call to the function completes its computation without producing any more recursive calls. When that happens, the computer turns its attention to the top sheet of paper on the stack. This sheet contains the partially completed computation that is waiting for the recursive computation that just ended. So, it is possible to proceed with that suspended computation. When that suspended computation ends, the computer discards that sheet of paper, and the suspended computation that is below it on the stack becomes the computation on top of the stack. The computer turns its attention to the suspended computation that is now on the top of the stack, and so forth. The process continues until the computation on the bottom sheet is completed. Depending on how many recursive calls are made and how the function definition is written, the stack may grow and shrink in any fashion. Notice that the sheets in the stack can only be accessed in a last-in/first-out fashion, but that is exactly what is needed to keep track of recursive calls. Each suspended version is waiting for the completion of the version directly above it on the stack.



VideoNote
Recursion and the Stack

Needless to say, computers do not have stacks of paper of this kind. This is just an analogy. The computer uses portions of memory rather than pieces of paper. The contents of one of these portions of memory (“sheets of paper”) is called an **activation frame**. These activation frames are handled in the last-in/first-out manner we just discussed. (The activation frames do not contain a complete copy of the function definition, but merely reference a single copy of the function definition. However, an activation frame contains enough information to allow the computer to act as if the frame contained a complete copy of the function definition.)

Stack

A **stack** is a *last-in/first-out* memory structure. The first item referenced or removed from a stack is always the last item entered into the stack. Stacks are used by computers to keep track of recursion (and for other purposes).

PITFALL Stack Overflow

There is always some limit to the size of the stack. If there is a long chain in which a function makes a recursive call to itself, and that call results in another recursive call, and that call produces yet another recursive call, and so forth, then each recursive call in this chain will cause another activation frame to be placed on the stack. If this chain is too long, then the stack will attempt to grow beyond its limit. This is an error condition known as a **stack overflow**. If you receive an error message that says *stack overflow*, it is likely that some function call has produced an excessively long chain of recursive calls. One common cause of stack overflow is infinite recursion. If a function is recursing infinitely, then it will eventually try to make the stack exceed any stack size limit. ■

Recursion Versus Iteration

Recursion is not absolutely necessary. In fact, some programming languages do not allow it. Any task that can be accomplished using recursion can also be done in some other way without using recursion. For example, Display 14.2 contains a nonrecursive version of the function given in Display 14.1. The nonrecursive version of a function typically uses a loop (or loops) of some sort in place of recursion. For that reason, the nonrecursive version is usually referred to as an **iterative version**. If the definition of the function `write_vertical` given in Display 14.1 is replaced by the version given in Display 14.2, then the output will be the same. As is true in this case, a recursive version of a function can sometimes be much simpler than an iterative version.

DISPLAY 14.2 Iterative Version of the Function in Display 14.1

```

1  //Uses iostream:
2  void write_vertical(int n)
3  {
4      int tens_in_n = 1;
5      int left_end_piece = n;
6      while (left_end_piece > 9)
7      {
8          left_end_piece = left_end_piece/10;
9          tens_in_n = tens_in_n * 10;
10     }
11     //tens_in_n is a power of ten that has the same number
12     //of digits as n. For example, if n is 2345, then
13     //tens_in_n is 1000.
14
15     for (int power_of_10 = tens_in_n;
16          power_of_10 > 0; power_of_10 = power_of_10/10)
17     {
18         cout << (n/power_of_10) <<endl;
19         n = n % power_of_10;
20     }
21 }

```

A recursively written function will usually run slower and use more storage than an equivalent iterative version. Although the iterative version of `write_vertical` given in Display 14.2 looks like it uses more storage and does more computing than the recursive version in Display 14.1, the two versions of `write_vertical` actually use comparable storage and do comparable amounts of computing. In fact, the recursive version may use more storage and run somewhat slower, because the computer must do a good deal of work manipulating the stack in order to keep track of the recursion. However, since the system does all this for you automatically, using recursion can sometimes make your job as a programmer easier and can sometimes produce code that is easier to understand. As you will see in the examples in this chapter and in the Self-Test Exercises and Programming Projects, sometimes a recursive definition is simpler and clearer; other times, an iterative definition is simpler and clearer.

SELF-TEST EXERCISES

6. If your program produces an error message that says *stack overflow*, what is a likely source of the error?
7. Write an iterative version of the function `cheers` defined in Self-Test Exercise 1.

8. Write an iterative version of the function defined in Self-Test Exercise 2.
9. Write an iterative version of the function defined in Self-Test Exercise 3.
10. Trace the recursive solution you made to Self-Test Exercise 4.
11. Trace the recursive solution you made to Self-Test Exercise 5.

14.2 RECURSIVE FUNCTIONS FOR VALUES

To iterate is human, to recurse divine.

ANONYMOUS

General Form for a Recursive Function That Returns a Value

The recursive functions you have seen thus far are all *void* functions, but recursion is not limited to *void* functions. A recursive function can return a value of any type. The technique for designing recursive functions that return a value is basically the same as for *void* functions. An outline for a successful recursive function definition that returns a value is as follows.

- One or more cases in which the value returned is computed in terms of calls to the same function (that is, using recursive calls). As was the case with *void* functions, the arguments for the recursive calls should intuitively be “smaller.”
- One or more cases in which the value returned is computed without the use of any recursive calls. These cases without any recursive calls are called **base cases** or **stopping cases** (just as they were with *void* functions).

This technique is illustrated in the next Programming Example.

PROGRAMMING EXAMPLE

Another Powers Function

In Chapter 4 we introduced the predefined function `pow` that computes powers. For example, `pow(2.0, 3.0)` returns $2.0^{3.0}$, so the following sets the variable `x` equal to 8.0 :

```
double x = pow(2.0, 3.0);
```

The function `pow` takes two arguments of type *double* and returns a value of type *double*. Display 14.3 contains a recursive definition for a function that is similar but that works with the type *int* rather than *double*. This new function is called `power`. For example, the following will set the value of `y` equal to 8 , since 2^3 is 8 :

```
int y = power(2, 3);
```


DISPLAY 14.3 The Recursive Function power

```

1  //Program to demonstrate the recursive function power.
2  #include <iostream>
3  #include <cstdlib>
4  using namespace std;

5  int power(int x, int n);
6  //Precondition: n >= 0.
7  //Returns x to the power n.

8  int main( )
9  {
10     for (int n = 0; n < 4; n++)
11         cout << "3 to the power " << n
12             << " is " << power(3, n) << endl;

13     return 0;
14 }

15 //uses iostream and cstdlib:
16 int power(int x, int n)
17 {
18     if (n < 0)
19     {
20         cout << "Illegal argument to power.\n";
21         exit(1);
22     }

23     if (n > 0)
24         return ( power(x, n - 1) * x);
25     else // n == 0
26         return (1);
27 }

```

Sample Dialogue

```

3 to the power 0 is 1
3 to the power 1 is 3
3 to the power 2 is 9
3 to the power 3 is 27

```

Our main reason for defining the function `power` is to have a simple example of a recursive function, but there are situations in which the function `power` would be preferable to the function `pow`. The function `pow` returns values of type *double*, which are only approximate quantities. The function `power` returns values of type *int*, which are exact quantities. In some situations, you might need the additional accuracy provided by the function `power`.

The definition of the function `power` is based on the following formula:

x^n is equal to $x^{n-1} * x$

Translating this formula into C++ says that the value returned by `power(x, n)` should be the same as the value of the expression

```
power(x, n - 1) * x
```

The definition of the function `power` given in Display 14.3 does return this value for `power(x, n)`, provided $n > 0$. The case where n is equal to 0 is the stopping case. If n is 0, then `power(x, n)` simply returns 1 (since x^0 is 1).

Let's see what happens when the function `power` is called with some sample values. First consider the following simple expression:

```
power(2, 0)
```

When the function is called, the value of x is set equal to 2, the value of n is set equal to 0, and the code in the body of the function definition is executed. Since the value of n is a legal value, the *if-else* statement is executed. Since this value of n is not greater than 0, the *return* statement after the *else* is used, so the function call returns 1. Thus, the following would set the value of y equal to 1:

```
int y = power(2, 0);
```

Now let's look at an example that involves a recursive call. Consider the expression

```
power(2, 1)
```

When the function is called, the value of x is set equal to 2, the value of n is set equal to 1, and the code in the body of the function definition is executed. Since this value of n is greater than 0, the following *return* statement is used to determine the value returned:

```
return ( power(x, n - 1) * x );
```

which in this case is equivalent to

```
return ( power(2, 0) * 2 );
```

At this point the computation of `power(2, 1)` is suspended, a copy of this suspended computation is placed on the stack, and the computer then starts a new function call to compute the value of `power(2, 0)`. As you have already seen, the value of `power(2, 0)` is 1. After determining the value of `power(2, 0)`, the computer replaces the expression `power(2, 0)` with its value of 1 and resumes the suspended computation. The resumed computation determines the final value for `power(2, 1)` from the *return* statement above as follows:

```
power(2, 0) * 2 is 1 * 2, which is 2.
```

Thus, the final value returned for `power(2, 1)` is 2. The following would therefore set the value of z equal to 2:

```
int z = power(2, 1);
```

Larger numbers for the second argument will produce longer chains of recursive calls. For example, consider the statement

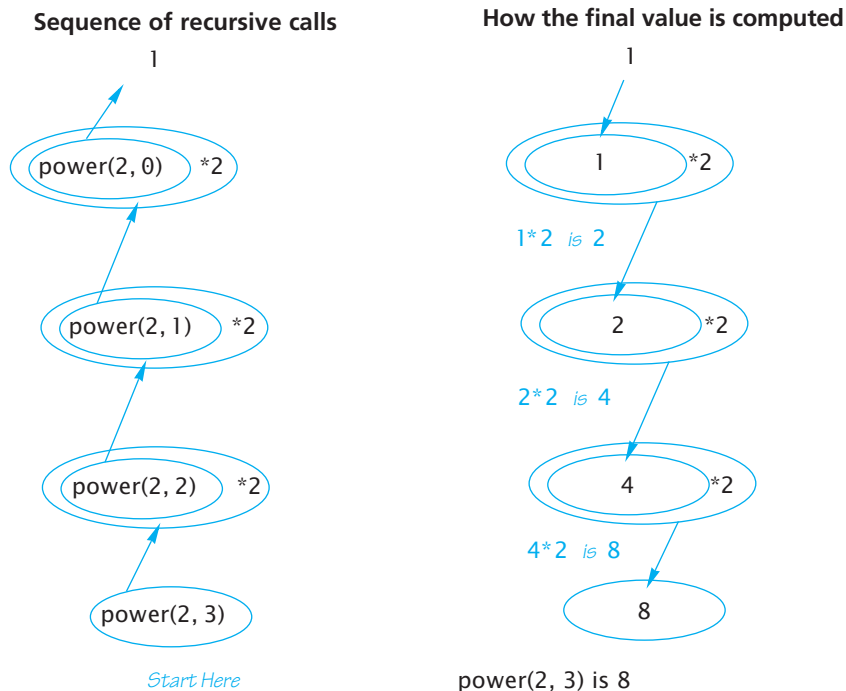
```
cout << power(2, 3);
```

The value of `power(2, 3)` is calculated as follows:

```
power(2, 3) is power(2, 2) * 2
power(2, 2) is power(2, 1) * 2
power(2, 1) is power(2, 0) * 2
power(2, 0) is 1 (stopping case)
```

When the computer reaches the stopping case, `power(2,0)`, there are three suspended computations. After calculating the value returned for the stopping case, it resumes the most recently suspended computation to determine the value of `power(2,1)`. After that, the computer completes each of the other suspended computations, using each value computed as a value to plug into another suspended computation, until it reaches and completes the computation for the original call, `power(2,3)`. The details of the entire computation are illustrated in Display 14.4.

DISPLAY 14.4 Evaluating the Recursive Function Call `power(2, 3)`



SELF-TEST EXERCISES

12. What is the output of the following program?

```
#include <iostream>
using namespace std;
int mystery(int n);
//Precondition  $n \geq 1$ .

int main()
{
    cout << mystery(3);
    return 0;
}

int mystery(int n)
{
    if (n <= 1)
        return 1;
    else
        return (mystery(n - 1) + n);
}
```

13. What is the output of the following program? What well-known mathematical function is rose?

```
#include <iostream>
using namespace std;
int rose(int n);
//Precondition:  $n \geq 0$ .

int main()
{
    cout << rose(4);
    return 0;
}

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

14. Redefine the function power so that it also works for negative exponents. In order to do this, you will also have to change the type of the value returned to *double*. The function declaration and header comment for the redefined version of power is as follows:

```
double power(int x, int n);
```

```
//Precondition: If  $n < 0$ , then  $x$  is not 0.
//Returns  $x$  to the power  $n$ .
```

(Hint: x^{-n} is equal to $1/(x^n)$.)

14.3 THINKING RECURSIVELY

There are two kinds of people in the world: those who divide the world into two kinds of people and those who do not.

ANONYMOUS

Recursive Design Techniques

When defining and using recursive functions you do not want to be continually aware of the stack and the suspended computations. The power of recursion comes from the fact that you can ignore that detail and let the computer do the bookkeeping for you. Consider the example of the function `power` in Display 14.3. The way to think of the definition of `power` is as follows:

`power(x, n)` returns `power(x, n - 1) * x`

Since x^n is equal to $x^{n-1} * x$, this is the correct value to return, provided that the computation will always reach a stopping case and will correctly compute the stopping case. So, after checking that the recursive part of the definition is correct, all you need check is that the chain of recursive calls will always reach a stopping case and that the stopping case always returns the correct value.

When you design a recursive function, you need not trace out the entire sequence of recursive calls for the instances of that function in your program. If the function returns a value, all that you need do is confirm that the following three properties are satisfied:

Criteria for
functions that
return a value

1. There is no infinite recursion. (A recursive call may lead to another recursive call and that may lead to another and so forth, but every such chain of recursive calls eventually reaches a stopping case.)
2. Each stopping case returns the correct value for that case.
3. For the cases that involve recursion: *If* all recursive calls return the correct value, *then* the final value returned by the function is the correct value.

For example, consider the function `power` in Display 14.3:

1. *There is no infinite recursion:* The second argument to `power(x,n)` is decreased by 1 in each recursive call, so any chain of recursive calls must eventually reach the case `power(x,0)`, which is the stopping case. Thus, there is no infinite recursion.

2. *Each stopping case returns the correct value for that case:* The only stopping case is `power(x, 0)`. A call of the form `power(x, 0)` always returns 1, and the correct value for x^0 is 1. So the stopping case returns the correct value.
3. *For the cases that involve recursion—if all recursive calls return the correct value, then the final value returned by the function is the correct value:* The only case that involves recursion is when $n > 1$. When $n > 1$, `power(x, n)` returns

$$\text{power}(x, n - 1) * x$$

To see that this is the correct value to return, note that: if `power(x, n-1)` returns the correct value, then `power(x, n-1)` returns x^{n-1} and so `power(x, n)` returns

$$x^{n-1} * x, \text{ which is } x^n$$

and that is the correct value for `power(x, n)`.

That's all you need to check in order to be sure that the definition of `power` is correct. (This technique is known as *mathematical induction*, a concept that you may have heard about in a mathematics class. However, you do not need to be familiar with the term in order to use this technique.)

We gave you three criteria to use in checking the correctness of a recursive function that returns a value. Basically, the same rules can be applied to a recursive *void* function. If you show that your recursive *void* function definition satisfies the following three criteria, then you will know that your *void* function performs correctly:

Criteria for void functions

1. There is no infinite recursion.
2. Each stopping case performs the correct action for that case.
3. For each of the cases that involve recursion: *If all recursive calls perform their actions correctly, then the entire case performs correctly.*

CASE STUDY Binary Search—An Example of Recursive Thinking

In this case study we develop a recursive function that searches an array to find out whether it contains a specified value. For example, the array may contain a list of numbers for credit cards that are no longer valid. A store clerk needs to search the list to see if a customer's card is valid or invalid. In Chapter 7 (Display 7.10) we discussed a simple method for searching an array by simply checking every array element. In this section we will develop a method that is much faster for searching a sorted array.

The indexes of the array `a` are the integers 0 through `final_index`. In order to make the task of searching the array easier, we assume that the array is sorted. Hence, we know the following:

$$a[0] \leq a[1] \leq a[2] \leq \dots \leq a[\text{final_index}]$$

When searching an array, you are likely to want to know both whether the value is in the list and, if it is, where it is in the list. For example, if we are searching for a credit card number, then the array index may serve as a record number. Another array indexed by these same indexes may hold a phone number or other information to use for reporting the suspicious card. Hence, if the sought-after value is in the array, we will want our function to tell where that value is in the array.

Problem Definition

We will design our function to use two call-by-reference parameters to return the outcome of the search. One parameter, called *found*, will be of type *bool*. If the value is found, then *found* will be set to *true*. If the value is found, then another parameter, called *location*, will be set to the index of the value found. If we use *key* to denote the value being searched for, the task to be accomplished can be formulated precisely as follows:

Precondition: *a[0]* through *a[final_index]* are sorted in increasing order.

Postcondition: if *key* is not one of the values *a[0]* through *a[final_index]*, then *found == false*; otherwise, *a[location] == key* and *found == true*.

Algorithm Design

Now let us proceed to produce an algorithm to solve this task. It will help to visualize the problem in very concrete terms. Suppose the list of numbers is so long that it takes a book to list them all. This is in fact how invalid credit card numbers are distributed to stores that do not have access to computers. If you are a clerk and are handed a credit card, you must check to see if it is on the list and hence invalid.

How would you proceed? Open the book to the middle and see if the number is there. If it is not and it is smaller than the middle number, then work backward toward the beginning of the book. If the number is larger than the middle number, you work your way toward the end of the book. This idea produces our first draft of an algorithm:

```
found = false; //so far.
mid = approximate midpoint between 0 and final_index;
if (key == a[mid])
{
    found = true;
    location = mid;
}
else if (key < a[mid])
    search a[0] through a[mid - 1];
else if (key > a[mid])
    search a[mid + 1] through a[final_index];
```

Algorithm—first
version

Since the searchings of the shorter lists are smaller versions of the very task we are designing the algorithm to perform, this algorithm naturally lends

itself to the use of recursion. The smaller lists can be searched with recursive calls to the algorithm itself.

Our pseudocode is a bit too imprecise to be easily translated into C++ code. The problem has to do with the recursive calls. There are two recursive calls shown:

```
search a[0] through a[mid - 1];
```

and

```
search a[mid + 1] through a[final_index];
```

More parameters

To implement these recursive calls, we need two more parameters. A recursive call specifies that a subrange of the array is to be searched. In one case it is the elements indexed by 0 through mid-1. In the other case it is the elements indexed by mid+1 through final_index. The two extra parameters will specify the first and last indexes of the search, so we will call them first and last. Using these parameters for the lowest and highest indexes, instead of 0 and final_index, we can express the pseudocode more precisely as follows:

Algorithm—first refinement

```
To search a[first] through a[last] do the following:
found = false; //so far.
mid = approximate midpoint between first and last;
if (key == a[mid])
{
    found = true;
    location = mid;
}
else if (key < a[mid])
    search a[first] through a[mid - 1];
else if (key > a[mid])
    search a[mid + 1] through a[last];
```

To search the entire array, the algorithm would be executed with first set equal to 0 and last set equal to final_index. The recursive calls will use other values for first and last. For example, the first recursive call would set first equal to 0 and last equal to the calculated value mid-1.

Stopping case algorithm—final version

As with any recursive algorithm, we must ensure that our algorithm ends rather than producing infinite recursion. If the sought-after number is found on the list, then there is no recursive call and the process terminates, but we need some way to detect when the number is not on the list. On each recursive call, the value of first is increased or the value of last is decreased. If they ever pass each other and first actually becomes larger than last, then we will know that there are no more indexes left to check and that the number key is not in the array. If we add this test to our pseudocode, we obtain a complete solution as shown in Display 14.5.

Coding

Now we can routinely translate the pseudocode into C++ code. The result is shown in Display 14.6. The function search is an implementation of the recursive algorithm given in Display 14.5. A diagram of how the function performs on a sample array is given in Display 14.7.

DISPLAY 14.5 Pseudocode for Binary Search

```
int a[Some_Size_Value];
```

Algorithm to search a[first] through a[last]

```
1  //Precondition:
2  //a[first] <= a[first + 1] <= a[first + 2] <= ... <= a[last]
```

To locate the value key:

```
1  if (first > last) //A stopping case
2      found = false;
3  else
4      {
5          mid = approximate midpoint between first and last;
6          if (key == a[mid]) //A stopping case
7              {
8                  found = true;
9                  location = mid;
10             }
11         else if key < a[mid] //A case with recursion
12             search a[first] through a[mid - 1];
13         else if key > a[mid] //A case with recursion
14             search a[mid + 1] through a[last];
15     }
```

DISPLAY 14.6 Recursive Function for Binary Search (part 1 of 2)

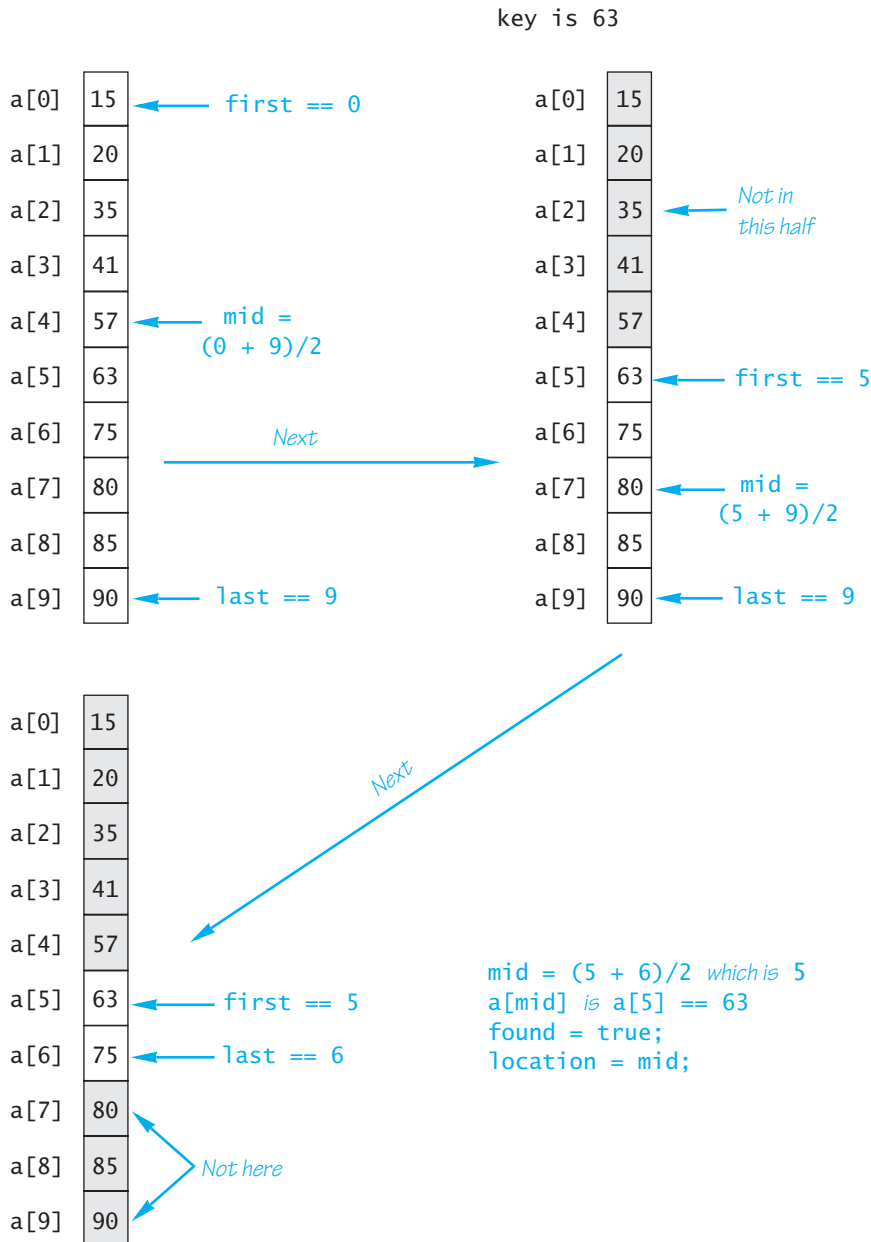
```
1  //Program to demonstrate the recursive function for binary search.
2  #include <iostream>
3  using namespace std;
4  const int ARRAY_SIZE = 10;
5
6
7  void search(const int a[], int first, int last,
8             int key, bool& found, int& location);
9  //Precondition: a[first] through a[last] are sorted in increasing order.
10 //Postcondition: if key is not one of the values a[first] through a[last],
11 //then found == false; otherwise, a[location] == key and found == true.
12
13
14 int main( )
15 {
16     int a[ARRAY_SIZE];
17     constint final_index = ARRAY_SIZE - 1;
18
```

(continued)

DISPLAY 14.6 Recursive Function for Binary Search *(part 2 of 2)*

<This portion of the program contains some code to fill and sort the array a. The exact details are irrelevant to this example.>

```
19     int key, location;
20     bool found;
21     cout << "Enter number to be located: ";
22     cin >> key;
23     search(a, 0, final_index, key, found, location);
24
25     if (found)
26         cout << key << " is in index location "
27             << location << endl;
28     else
29         cout << key << " is not in the array." << endl;
30
31     return 0;
32 }
33 void search(const int a[], int first, int last,
34            int key, bool& found, int& location)
35 {
36     int mid;
37     if (first > last)
38     {
39         found = false;
40     }
41     else
42     {
43         mid = (first + last)/2;
44
45         if (key == a[mid])
46         {
47             found = true;
48             location = mid;
49         }
50         else if (key < a[mid])
51         {
52             search(a, first, mid - 1, key, found, location);
53         }
54         else if (key > a[mid])
55         {
56             search(a, mid + 1, last, key, found, location);
57         }
58     }
59 }
```

DISPLAY 14.7 Execution of the Function `search`

Solve a more general problem

Notice that the function `search` solves a more general problem than the original task. Our goal was to design a function to search an entire array. Yet the function will let us search any interval of the array by specifying the index bounds `first` and `last`. This is common when designing recursive functions. Frequently, it is necessary to solve a more general problem in order to be able to express the recursive algorithm. In this case, we only wanted the answer in the case where `first` and `last` are set equal to 0 and `final_index`. However, the recursive calls will set them to values other than 0 and `final_index`.

Checking the Recursion

In the subsection entitled “Recursive Design Techniques,” we gave three criteria that you should check to ensure that a recursive *void* function definition is correct. Let’s check these three things for the function `search` given in Display 14.6.

1. *There is no infinite recursion:* On each recursive call, the value of `first` is increased or the value of `last` is decreased. If the chain of recursive calls does not end in some other way, then eventually the function will be called with `first` larger than `last`, and that is a stopping case.
2. *Each stopping case performs the correct action for that case:* There are two stopping cases: when `first > last` and when `key==a[mid]`. Let’s consider each case.

If `first > last`, there are no array elements between `a[first]` and `a[last]`, and so `key` is not in this segment of the array. (Nothing is in this segment of the array!) So, if `first > last`, the function `search` correctly sets `found` equal to *false*.

If `key==a[mid]`, the algorithm correctly sets `found` equal to *true* and `location` equal to `mid`. Thus, both stopping cases are correct.

3. *For each of the cases that involve recursion, if all recursive calls perform their actions correctly, then the entire case performs correctly:* There are two cases in which there are recursive calls: when `key < a[mid]` and when `key > a[mid]`. We need to check each of these two cases.

First suppose `key < a[mid]`. In this case, since the array is sorted, we know that if `key` is anywhere in the array, then `key` is one of the elements `a[first]` through `a[mid-1]`. Thus, the function need only search these elements, which is exactly what the recursive call

```
search(a, first, mid - 1, key, found, location);
```

does. So if the recursive call is correct, then the entire action is correct.

Next, suppose `key > a[mid]`. In this case, since the array is sorted, we know that if `key` is anywhere in the array, then `key` is one of the elements `a[mid+1]` through `a[last]`. Thus, the function need search only these elements, which is exactly what the recursive call

```
search(a, mid + 1, last, key, found, location);
```

does. So if the recursive call is correct, then the entire action is correct. Thus, in both cases the function performs the correct action (assuming that the recursive calls perform the correct action).

The function `search` passes all three of our tests, so it is a good recursive function definition.

Efficiency

The binary search algorithm is extremely fast compared to an algorithm that simply tries all array elements in order. In the binary search, you eliminate about half the array from consideration right at the start. You then eliminate a quarter, then an eighth of the array, and so forth. These savings add up to a dramatically fast algorithm. For an array of 100 elements, the binary search will never need to compare more than 7 array elements to the key. A simple serial search could compare as many as 100 array elements to the key and on the average will compare about 50 array elements to the key. Moreover, the larger the array is, the more dramatic the savings will be. On an array with 1000 elements, the binary search will need to compare only about 10 array elements to the key value, as compared to an average of 500 for the simple serial search algorithm.

An iterative version of the function `search` is given in Display 14.8. On some systems, the iterative version will run more efficiently than the recursive version. The algorithm for the iterative version was derived by mirroring the recursive version. In the iterative version, the local variables `first` and `last` mirror the roles of the parameters in the recursive version, which are also named `first` and `last`. As this example illustrates, it often makes sense to derive a recursive algorithm even if you expect to later convert it to an iterative algorithm.

Iterative version

DISPLAY 14.8 Iterative Version of Binary Search (part 1 of 2)

Function Declaration

```
1 void search(const int a[], int low_end, int high_end,
2 int key, bool& found, int& location);
3 //Precondition: a[low_end] through a[high_end] are sorted in increasing
4 //order.
5 //Postcondition: If key is not one of the values a[low_end] through
6 //a[high_end], then found == false; otherwise, a[location] == key and
7 //found == true.
```

Function Definition

```
1 void search(const int a[], int low_end, int high_end,
2 int key, bool& found, int& location)
3 {
4     int first = low_end;
5     int last = high_end;
```

(continued)

DISPLAY 14.8 Iterative Version of Binary Search (part 2 of 2)

```
6      int mid;
7
8      found = false; //so far
9      while ( (first <= last) && !(found) )
10     {
11         mid = (first + last)/2;
12         if (key == a[mid])
13         {
14             found = true;
15             location = mid;
16         }
17         else if (key < a[mid])
18         {
19             last = mid -1;
20         }
21         else if (key > a[mid])
22         {
23             first = mid + 1;
24         }
25     }
26 }
```

PROGRAMMING EXAMPLE**A Recursive Member Function**

A member function of a class can be recursive. Member functions can use recursion in the same way that ordinary functions do. Display 14.9 contains an example of a recursive member function. The class `BankAccount` used in that display is the same as the class named `BankAccount` that was defined in Display 10.6, except that we have overloaded the member function name `update`. The first version of `update` has no arguments and posts one year of simple interest to the bank account balance. The other (new) version of `update` takes an *int* argument that is some number of years. This member function updates the account by posting the interest for that many years. The new version of `update` is recursive; has one parameter, called `years`; and uses the following algorithm:

If the number of years is 1, then *//Stopping case*:

call the other function named `update` (the one with no arguments).

If the number of years is greater than 1, then *//Recursive case*:

make a recursive call to post `years-1` worth of interest, and then call the other function called `update` (the one with no arguments) to post one more year's worth of interest.

DISPLAY 14.9 A Recursive Member Function (part 1 of 2)

```

1  //Program to demonstrate the recursive member function update(years).
2  #include <iostream>
3  using namespace std;
4
5  //Class for a bank account:
6  class BankAccount
7  {
8  public:
9      BankAccount(int dollars, int cents, double rate);
10     //Initializes the account balance to $dollars.cents and
11     //initializes the interest rate to rate percent.
12
13     BankAccount(int dollars, double rate);
14     //Initializes the account balance to $dollars.00 and
15     //initializes the interest rate to rate percent.
16
17     BankAccount( );
18     //Initializes the account balance to $0.00 and
19     //initializes the interest rate to 0.0%.
20
21     void update( );
22     //Postcondition: One year of simple interest
23     //has been added to the account balance.
24
25     void update(int years);
26     //Postcondition: Interest for the number of years given has been added to the
27     //account balance. Interest is compounded annually.
28
29     double get_balance( );
30     //Returns the current account balance.
31
32     double get_rate( );
33     //Returns the current account interest rate as a percentage.
34
35     void output(ostream& outs);
36     //Precondition: If outs is a file output stream, then outs has already
37     //been connected to a file.
38     //Postcondition: Balance & interest rate have been written to the stream outs.
39 private:
40     double balance;
41     double interest_rate;
42     double fraction(double percent); //Converts a percentage to a fraction.
43 };
44
45 int main( )
46 {
47     BankAccount your_account(100, 5);
48     your_account.update(10);
49     cout.setf(ios::fixed);

```

The class `BankAccount` in this program is an improved version of the class `BankAccount` given in Display 10.6.

Two different functions with the same name

(continued)

DISPLAY 14.9 A Recursive Member Function (part 2 of 2)

```

42     cout.setf(ios::showpoint);
43     cout.precision(2);
44     cout << "If you deposit $100.00 at 5% interest, then\n"
45           << "in ten years your account will be worth $"
46           << your_account.get_balance( ) << endl;
47     return 0;
48 }
49
50 void BankAccount::update( )
51 {
52     balance = balance + fraction(interest_rate)*balance;
53 }
54
55 void BankAccount::update(int years)
56 {
57     if (years == 1)
58     {
59         update( );
60     }
61     else if (years > 1)
62     {
63         update(years - 1);
64         update( );
65     }
66 }

```

Overloading (that is, calls to another function with the same name)

Recursive function call

<Definitions of the other member functions are given in Display 10.5 and Display 10.6, but you need not read those definitions in order to understand this example.>

Sample Dialogue

```

If you deposit $100.00 at 5% interest, then
in ten years your account will be worth $162.89

```

It is easy to see that this algorithm produces the desired result by checking the three points given in the subsection entitled "Recursive Design Techniques."

1. *There is no infinite recursion:* Each recursive call reduces the number of years by 1 until the number of years eventually becomes 1, which is the stopping case. So there is no infinite recursion.
2. *Each stopping case performs the correct action for that case:* The one stopping case is when `years==1`. This case produces the correct action, since it simply calls the other overloaded member function called `update`, and we checked the correctness of that function in Chapter 10.

3. *For the cases that involve recursion, if all recursive calls perform correctly, then the entire case performs correctly:* The recursive case—that is, `years>1`—works correctly, because if the recursive call correctly posts `years-1` worth of interest, then all that is needed is to post one additional year's worth of interest and the call to the overloaded zero-argument version of `update` will correctly post one year's worth of interest. Thus, *if the recursive call performs the correct action, then the entire action for the case of `years>1` will be correct.*

In this example, we have overloaded `update` so that there are two different functions named `update`: one that takes no arguments and one that takes a single argument. Do not confuse the calls to the two functions named `update`. These are two different functions that, as far as the compiler is concerned, just coincidentally happen to have the same name. When the definition of the function `update` with one argument includes a call to the version of `update` that takes no arguments, that is not a recursive call. Only the call to the version of `update` with the *exact* same function declaration is a recursive call. To see what is involved here, note that we could have named the version of `update` that takes no argument `post_one_year()`, instead of naming it `update()`, and then the definition of the recursive version of `update` would read as follows:

Overloading

```
void BankAccount::update(int years)
{
    if (years == 1)
    {
        post_one_year();
    }
    else if (years > 1)
    {
        update(years - 1);
        post_one_year();
    }
}
```

Recursion and Overloading

Do not confuse recursion and overloading. When you overload a function name, you are giving two different functions the same name. If the definition of one of these two functions includes a call to the other, that is not recursion. In a recursive function definition, the definition of the function includes a call to the *exact* same function with the *exact same definition*, not to some other function that coincidentally uses the same name. It is not too serious an error if you confuse overloading and recursion, since they are both legal. It is simply a question of getting the terminology straight so that you can communicate clearly with other programmers and so that you understand the underlying processes.

SELF-TEST EXERCISES

15. Write a recursive function definition for the following function:

```
int squares(int n);
//Precondition: n >= 1
//Returns the sum of the squares of numbers 1 through n.
```

For example, `squares(3)` returns 14 because $1^2 + 2^2 + 3^2$ is 14.

16. Write an iterative version of the one-argument member function `BankAccount::update(int years)` that is described in Display 14.9.

CHAPTER SUMMARY

- If a problem can be reduced to smaller instances of the same problem, then a recursive solution is likely to be easy to find and implement.
- A recursive algorithm for a function definition normally contains two kinds of cases: one or more cases that include at least one recursive call and one or more stopping cases in which the problem is solved without any recursive calls.
- When writing a recursive function definition, always confirm that the function will not produce infinite recursion.
- When you define a recursive function, use the three criteria given in the subsection “Recursive Design Techniques” to confirm that the function is correct.
- When you design a recursive function to solve a task, it is often necessary to solve a more general problem than the given task. This may be required to allow for the proper recursive calls, since the smaller problems may not be exactly the same problem as the given task. For example, in the binary search problem, the task was to search an entire array, but the recursive solution is an algorithm to search any portion of the array (either all of it or a part of it).

Answers to Self-Test Exercises

1. Hip Hip Hurray
2.

```
void stars(int n)
{
    cout << '*';
    if (n > 1)
        stars(n - 1);
}
```

The following is also correct but is more complicated:

```
void stars(int n)
{
    if (n <= 1)
    {
        cout << '*';
    }
    else
    {
        stars(n - 1);
        cout << '*';
    }
}
```

3. `void backward(int n)`
- ```
{
 if (n < 10)
 {
 cout << n;
 }
 else
 {
 cout << (n % 10); //write last digit
 backward(n / 10); //write the other digits backward
 }
}
```

4. and 5. The answer to 4 is `write_up(int n);`. The answer to 5 is `write_down(int n);`.

```
#include <iostream>
using namespace std;
void write_down(int n)
{
 if (n >= 1)
 {
 cout << n << " ";
 write_down(n - 1);
 }
}
```

```
void write_up(int n)
{
 if (n >= 1)
 {
 write_up(n - 1);
 cout << n << " ";
 }
}
```

```

//testing code for both #4 and #5
int main()
{
 cout << "calling write_up(" << 10 << ")\n";
 write_up(10);
 cout << endl;
 cout << "calling write_down(" << 10 << ")\n";
 write_down(10);
 cout << endl;
 return 0;
}
/* Test results
calling write_up(10)
1 2 3 4 5 6 7 8 9 10
calling write_down(10)
10 9 8 7 6 5 4 3 2 1
*/

```

6. An error message that says *stack overflow* is telling you that the computer has attempted to place more activation frames on the stack than are allowed on your system. A likely cause of this error message is infinite recursion.

7. `void` cheers(`int` n)

```

{
 while (n > 1)
 {
 cout << "Hip ";
 n--;
 }
 cout << "Hurray\n";
}

```

8. `void` stars(`int` n)

```

{
 for (int count = 1; count <= n; count++)
 cout << '*';
}

```

9. `void` backward(`int` n)

```

{
 while (n >= 10)
 {
 cout << (n % 10); //write last digit
 n = n / 10; //discard the last digit
 }
 cout << n;
}

```

10. Trace for Exercise 4: If  $n = 3$ , the code to be executed is

```
if (3 >= 1)
{
 write_up(3 - 1);
 cout << 3 << " ";
}
```

On the next recursion,  $n = 2$ ; the code to be executed is

```
if (2 >= 1)
{
 write_up(2 - 1);
 cout << 2 << " ";
}
```

On the next recursion,  $n = 1$  and the code to be executed is

```
if (1 >= 1)
{
 write_up(1 - 1);
 cout << 1 << " ";
}
```

On the final recursion,  $n = 0$  and the code to be executed is

```
if (0 >= 1) // condition false, body skipped
{
 // skipped
}
```

The recursions unwind; the output (obtained while recursion was winding up) is 1 2 3.

11. Trace for Exercise 5: If  $n = 3$ , the code to be executed is

```
if (3 >= 1)
{
 cout << 3 << " ";
 write_down(3 - 1);
}
```

Next recursion,  $n = 2$ , the code to be executed is

```
if (2 >= 1)
{
 cout << 2 << " ";
 write_down(2 - 1)
}
```

Next recursion,  $n = 1$ , the code to be executed is

```
if (1 >= 1)
```

```

{
 cout << 1 << " ";
 write_down(1 - 1)
}

```

Final recursion,  $n = 0$ , and the "true" clause is not executed:

```

if (0 >= 1) // condition false
{
 // this clause is skipped
}

```

The output is 3 2 1.

12. 6

13. The output is 24. The function is the factorial function, usually written  $n!$  and defined as follows:

$n!$  is equal to  $n * (n - 1) * (n - 2) * \dots * 1$

14. //Uses *iostream* and *cstdlib*:

```

double power(int x, int n)
{
 if (n < 0 && x == 0)
 {
 cout << "Illegal argument to power.\n";
 exit(1);
 }

 if (n < 0)
 return (1/power(x, -n));
 else if (n > 0)
 return (power(x, n - 1)*x);
 else // n == 0
 return (1.0);
}

```

15. *int* squares(*int* n)

```

{
 if (n <= 1)
 return 1;
 else
 return (squares(n - 1) + n * n);
}

```

16. *void* BankAccount::update(*int* years)

```

{
 for (int count = 1; count <= years; count++)
 update();
}

```

## PRACTICE PROGRAMS

*Practice Programs can generally be solved with a short program that directly applies the programming principles presented in this chapter.*

1. Write a recursive function definition for a function that has one parameter *n* of type *int* and that returns the *n*th Fibonacci number. See Programming Project 6 in Chapter 3 for the definition of Fibonacci numbers. Embed the function in a program and test it.
2. Write a recursive version of the function `index_of_smallest` that was used in the sorting program in Display 7.12 of Chapter 7. Embed the function in a program and test it.
3. Write a recursive version of the search function in Display 7.10 of Chapter 7.
4. There are *n* people in a room, where *n* is an integer greater than or equal to 2. Each person shakes hands once with every other person. What is the total number of handshakes in the room? Write a recursive function to solve this problem, with the following header:

```
int handshake(int n)
```

where `handshake(n)` returns the total number of handshakes for *n* people in the room. To get you started, if there are only one or two people in the room, then:

```
handshake(1) = 0
handshake(2) = 1
```

5. Write a recursive function that returns `true` if an input string is a palindrome and `false` if it is not. You can do this by checking if the first character equals the last character, and if so, make a recursive call with the input string minus the first and last characters. You will have to define a suitable stopping condition. Test your function with several palindromes and non-palindromes.



VideoNote  
Solution to Practice  
Program 14.4

## PROGRAMMING PROJECTS

*Programming Projects require more problem-solving than Practice Programs and can usually be solved many different ways. Visit [www.myprogramminglab.com](http://www.myprogramminglab.com) to complete many of these Programming Projects online and get instant feedback.*

1. The formula for computing the number of ways of choosing *r* different things from a set of *n* things is the following:

$$C(n, r) = n! / (r! * (n - r)!)$$

The factorial function  $n!$  is defined by

$$n! = n * (n - 1) * (n - 2) * \dots * 1$$

Discover a recursive version of this formula and write a recursive function that computes the value of the formula. Embed the function in a program and test it.

2. Write a recursive function that has an argument that is an array of characters and two arguments that are bounds on array indexes. The function should reverse the order of those entries in the array whose indexes are between the two bounds. For example, if the array is

```
a[0] == 'A' a[1] == 'B' a[2] == 'C' a[3] == 'D' a[4] == 'E'
```

and the bounds are 1 and 4, then after the function is run the array elements should be

```
a[0] == 'A' a[1] == 'E' a[2] == 'D' a[3] == 'C' a[4] == 'B'
```

Embed the function in a program and test it. After you have fully debugged this function, define another function that takes a single argument which is an array that contains a string value and that reverses the spelling of the string value in the array argument. This function will include a call to the recursive definition you did for the first part of this project. Embed this second function in a program and test it.

3. Write an iterative version of the recursive function in Programming Project 1. Embed it in a program and test it.
4. Write a recursive function to sort an array of integers into ascending order using the following idea: Place the smallest element in the first position, then sort the rest of the array by a recursive call. This is a recursive version of the selection sort algorithm discussed in Chapter 7. (*Note:* Simply taking the program from Chapter 7 and plugging in a recursive version of `index_of_smallest` will not suffice. The function to do the sorting must itself be recursive and not merely use a recursive function.)
5. Towers of Hanoi: There is a story about Buddhist monks who are playing this puzzle with 64 stone disks. The story claims that when the monks finish moving the disks from one post to a second via the third post, time will end.

A stack of  $n$  disks of decreasing size is placed on one of three posts. The task is to move the disks one at a time from the first post to the second. To do this, any disk can be moved from any post to any other post, subject to the rule that you can never place a larger disk over a smaller disk. The (spare) third post is provided to make the solution possible. Your task is to write a recursive function that describes instructions for a solution to





this problem. We don't have graphics available, so you should output a sequence of instructions that will solve the problem.

(*Hint:* If you could move up  $n-1$  of the disks from the first post to the third post using the second post as a spare, the last disk could be moved from the first post to the second post. Then by using the same technique (whatever that may be) you can move the  $n-1$  disks from the third post to the second post, using the first disk as a spare. There! You have the puzzle solved. You only have to decide what the nonrecursive case is, what the recursive case is, and when to output instructions to move the disks.)

6. The game of "Jump It" consists of a board with  $n$  positive integers in a row, except for the first column, which always contains 0. These numbers represent the cost to enter each column. Here is a sample game board where  $n$  is 6:

|   |   |    |   |    |    |
|---|---|----|---|----|----|
| 0 | 3 | 80 | 6 | 57 | 10 |
|---|---|----|---|----|----|

The object of the game is to move from the first column to the last column with the lowest total cost. The number in each column represents the cost to enter that column. You always start the game in the first column and have two types of moves. You can either move to the adjacent column or jump over the adjacent column to land two columns over. The cost of a game is the sum of the costs of the columns visited.

In the board shown above, there are several ways to get to the end. Starting in the first column, our cost so far is 0. We could jump to 80, then jump to 57, then move to 10 for a total cost of  $80 + 57 + 10 = 147$ . However, a cheaper path would be to move to 3, jump to 6, then jump to 10, for a total cost of  $3 + 6 + 10 = 19$ .

Write a recursive solution to this problem that computes the lowest cost of the game and outputs this value for an arbitrarily large game board represented as an array. Your program doesn't have to output the actual sequence of jumps, only the lowest cost of this sequence. After making sure that your solution works on small arrays, test it on boards of larger and larger values of  $n$  to get a feel for the scalability and efficiency of your solution.

7. Suppose we can buy chocolate bars from the vending machine for \$1 each. Inside every chocolate bar is a coupon. We can redeem 7 coupons for 1 chocolate bar from the machine. We would like to know how many chocolate bars can be eaten, including those redeemed via coupon, if we have  $n$  dollars.

For example, if we have \$20, then we can initially buy 20 chocolate bars. This gives us 20 coupons. We can redeem 14 coupons for 2 additional chocolate bars. These two additional chocolate bars have 2 more coupons,

so we now have a total of 8 coupons when added to the 6 left over from the original purchase. This gives us enough to redeem for 1 final chocolate bar. As a result we now have 23 chocolate bars and 2 leftover coupons.

Write a recursive solution to this problem that inputs from the user the number of dollars to spend on chocolate bars and outputs how many chocolate bars you can collect after spending all your money and redeeming as many coupons as possible. Your recursive function will be based upon the number of coupons owned.

8. Some problems require finding all permutations (different orderings) of a set of items. For a set of  $n$  items  $\{a_1, a_2, a_3, \dots, a_n\}$  there are  $n!$  permutations. For example, given the set  $\{1, 2, 3\}$  there are six permutations:

$\{3, 2, 1\}$   $\{2, 3, 1\}$   $\{2, 1, 3\}$   $\{3, 1, 2\}$   $\{1, 3, 2\}$   $\{1, 2, 3\}$

Write a recursive function that generates all the permutations of a set of numbers. The general outline of a solution is given here, but the implementation is up to you. The program will require storing a set of permutations of numbers that you can implement in many ways (for example, linked lists of nodes, linked lists of vectors, arrays, etc.) Your program should call the recursive function with sets of several different sizes, printing the resulting set of permutations for each.

One solution is to first leave out the  $n$ th item in the set. Recursively find all permutations using the set of  $(n-1)$  items. If we insert the  $n$ th item into each position for all of these permutations, then we get a new set of permutations that includes the  $n$ th item. The base case is when there is only one item in the set, in which case the solution is simply the permutation with the single item.

For example, consider finding all permutations of  $\{1, 2, 3\}$ . We leave the 3 out and recursively find all permutations of the set  $\{1, 2\}$ . This consists of the permutations:

$\{1, 2\}$                        $\{2, 1\}$

Next we insert the 3 into every position for these permutations. For the first permutation, we insert the 3 in the front, between 1 and 2, and after 2. For the second permutation, we insert the 3 in the front, between 2 and 1, and after 1:

$\{3, 1, 2\}$   $\{1, 3, 2\}$   $\{1, 2, 3\}$   $\{3, 2, 1\}$   $\{2, 3, 1\}$   $\{2, 1, 3\}$

The resulting six permutations comprise all permutations of the set  $\{1, 2, 3\}$ .

9. The word ladder game was invented by Lewis Carroll in 1877. The idea is to begin with a start word and change one letter at a time until arriving at an end word. Each word along the way must be an English word.

For example, starting from FISH you can make a word ladder to MAST through the following ladder:

FISH, WISH, WASH, MASH, MAST

Write a program that uses recursion to find the word ladder given a start word and an end word, or determines if no word ladder exists. Use the file `words.txt` that is available online with the source code for the book as your dictionary of valid words. This file contains 87314 words. Your program does not need to find the shortest word ladder between words, any word ladder will do if one exists.

