

B Recursion

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

A function definition that includes a call to itself is said to be recursive. Like most modern programming languages, C++ allows functions to be recursive. If used with a little care, recursion can be a useful programming technique. This chapter introduces the basic techniques needed for defining successful recursive functions. There is nothing in this chapter that is truly unique to C++. If you are already familiar with recursion you can safely skip this chapter.

This chapter only uses material from Chapters 1 to 5. Sections 13.1 and 13.2 do not use any material from Chapter 5, so you can cover recursion any time after Chapter 4. If you have not read Chapter 11, you may find it helpful to review the section of Chapter 1 on namespaces.

B.1

Recursive void Functions

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 a list for a particular value, you might divide this into the subtask of searching the first half of the list and the subtask of searching the second half of the list. The subtasks of searching the halves of the list 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. We begin with a simple example 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**.

Example

VERTICAL NUMBERS

Display B.1 contains a demonstration program for a recursive function named `writeVertical` that takes one (nonnegative) `int` argument and writes that `int` to the screen, with the digits going down the screen one per line. For example, the invocation

```
writeVertical(1234);
```

would produce the output

```
1
2
3
4
```

The task to be performed by `writeVertical` may be broken down into the following two cases:

- *Simple case:* If $n < 10$, then write the number `n` to the screen.

After all, if the number is only one digit long, the task is trivial.

- *Recursive case:* If $n \geq 10$, then do two subtasks:

- 1 Output all the digits except the last digit.
- 2 Output the last digit.

For example, if the argument were 1234, the first subtask would output


```
1
2
3
```

and the second subtask would output 4. This decomposition into subtasks can be used to derive the function definition.

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 above. Thus, an outline of our algorithm for the function `writeVertical` with parameter `n` is given by the following pseudocode:

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

Recursive subtask



If you observe the following identities, it is easy to convert this pseudocode to a complete C++ function definition:

$n/10$ is the number n with the last digit removed.
 $n\%10$ is the last digit of n .

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

The complete code for the function is as follows:

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



Display 13.1 A Recursive void Function

```
1 //Program to demonstrate the recursive function writeVertical.
2 #include <iostream>
3 using std::cout;
4 using std::endl;

5 void writeVertical(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 int main( )
10 {
11     cout << "writeVertical(3):" << endl;
12     writeVertical(3);

13     cout << "writeVertical(12):" << endl;
14     writeVertical(12);

15     cout << "writeVertical(123):" << endl;
16     writeVertical(123);

17     return 0;
18 }

19 //uses iostream:
20 void writeVertical(int n)
21 {
22     if (n < 10)
23     {
24         cout << n << endl;
25     }
26     else //n is two or more digits long:
27     {
28         writeVertical(n/10);
29         cout << (n%10) << endl;
30     }
31 }
```

SAMPLE DIALOGUE

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

■ TRACING A RECURSIVE CALL

Let's see exactly what happens when the following function call is made (as in Display 13.1):

```
writeVertical(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 , and the body of the function is executed. After the substitution of 123 for n , the code to be executed is equivalent to the following:

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

Computation will stop here until the recursive call returns.

Since 123 is not less than 10, the `else` part is executed. However, the `else` part begins with the following function call:

```
writeVertical(n/10);
```

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

```
writeVertical(123/10);
```

which is equivalent to

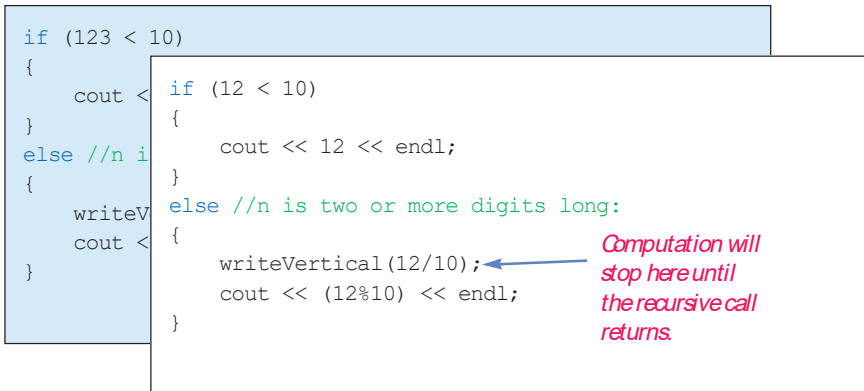
```
writeVertical(12);
```

When execution reaches this recursive call, the current function computation is placed in suspended animation and the 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 there.

The recursive call

```
writeVertical(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:



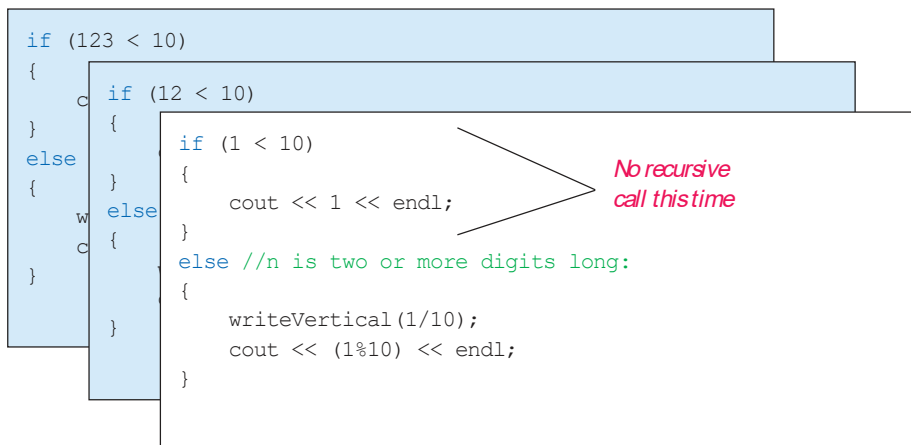
Since 12 is not less than 10, the `else` part is executed. However, as you already saw, the `else` part begins with a recursive call. The argument for the recursive call is `n/10`, which in this case is equivalent to `12/10`. So this second computation of the function `writeVertical` is suspended and the following recursive call is executed:

```
writeVertical(12/10);
```

which is equivalent to

```
writeVertical(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:



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 `writeVertical(1)` writes 1 to the screen and ends without any recursive call.

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

```

if (123 < 10)
{
    if (12 < 10)
    {
        cout << 12 << endl;
    }
    else //n is two or more digits long:
    {
        writeVertical(12/10); ← Computation resumes here.
        cout << (12%10) << endl;
    }
}

```

When this suspended computation resumes, it executes a `cout` statement that outputs the value `12%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:

```

if (123 < 10)
{
    cout << 123 << endl;
}
else //n is two or more digits long:
{
    writeVertical(123/10); ← Computation resumes here.
    cout << (123%10) << endl;
}

```

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

■ A CLOSER LOOK AT RECURSION

The definition of the function `writeVertical` uses recursion. Yet we did nothing new or different in evaluating the function call `writeVertical(123)`. We treated it just like any of the function calls we saw in previous chapters. We simply 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

```
writeVertical(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, it temporarily stops its computation 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. 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 in order to return a value. The general outline of a successful recursive function definition is as follows:

how
recursion
ends

- One or more cases in which the function accomplishes its task by using one or more 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.

base case
or stopping
case

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 `writeVertical` in Display 13.1. When the function `writeVertical` 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 `writeVertical` 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.

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

```
void newWriteVertical(int n)
{
    newWriteVertical(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 `newWriteVertical`, 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 `newWriteVertical(12)`, that execution will stop to execute the recursive call `newWriteVertical(12/10)`, which is equivalent to `newWriteVertical(1)`. The execution of that recursive call will, in turn, stop to execute the recursive call

```
newWriteVertical(1/10);
```

infinite
recursion

which is equivalent to

```
newWriteVertical(0);
```

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

```
newWriteVertical(0);
```

and that will produce another recursive call to again execute the same recursive function call `newWriteVertical(0);`, and so on, forever. Since the definition of `newWriteVertical` 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 std::cout;
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 that 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 that 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 Exercise 4 to that for this exercise, or vice versa, by an exchange of as little as two lines.)

■ STACKS FOR RECURSION

stack

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.

last-in/
first-out

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 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 value returned by the recursive call. But before computing the recursive call, it saves enough information so that when it does finally determine the value returned by 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 subsection entitled “Tracing a Recursive Call.” Even though we did not call it a stack at the time, the figures of computations placed one on top of the other illustrate 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. Thus, 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.

Needless to say, computers do not have stacks of paper. This is just an analogy. The computer uses portions of memory rather than pieces of paper. The content 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. (These 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 activation frame contained a complete copy of the function definition.)

activation
frame

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, 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.

stack
overflow

■ 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 13.2 contains a nonrecursive



Display 13.2 Iterative Version of the Function in Display 13.1

```

1  //Uses iostream:
2  void writeVertical(int n)
3  {
4      int nsTens = 1;
5      int leftEndPiece = n;
6      while (leftEndPiece > 9)
7      {
8          leftEndPiece = leftEndPiece/10;
9          nsTens = nsTens*10;
10     }
11     //nsTens is a power of ten that has the same number
12     //of digits as n. For example, if n is 2345, then
13     //nsTens is 1000.

14     for (int powerOf10 = nsTens;
15          powerOf10 > 0; powerOf10 = powerOf10/10)
16     {
17         cout << (n/powerOf10) << endl;
18         n = n%powerOf10;
19     }
20 }
```

iterative
version

version of the function given in Display 13.1. 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 `writeVertical` given in Display 13.1 is replaced by the version given in Display 13.2, 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.

efficiency

A recursively written function will usually run slower and use more storage than an equivalent iterative version. 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.

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.

13.2 Recursive Functions That Return a Value

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 that 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.

Example

ANOTHER POWERS FUNCTION

Chapter 3 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 `result` equal to 8.0 :

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

The function `pow` takes two arguments of type `double` and returns a value of type `double`. Display 13.3 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 `result2` equal to 8, since 2^3 is 8:

```
int result2 = power(2, 3);
```

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



Display 13.3 The Recursive Function power

```
1  //Program to demonstrate the recursive function power.
2  #include <iostream>
3  #include <cstdlib>
4  using std::cout;
5  using std::endl;

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

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

14     return 0;
15 }

16 //uses iostream and cstdlib:
17 int power(int x, int n)
18 {
19     if (n < 0)
20     {
21         cout << "Illegal argument to power.\n";
22         exit(1);
23     }

24     if (n > 0)
25         return ( power(x, n - 1)*x );
26     else // n == 0
27         return (1);
28 }
```

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
```

`pow`. The function `pow` returns a value of type `double`, which is only an approximate quantity. The function `power` returns a value of type `int`, which is an exact quantity. 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:

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

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 13.3 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 `result3` equal to 1:

```
int result3 = 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 above `return` statement as follows:

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

Thus, the final value returned for `power(2, 1)` is 2. So, the following would set the value of `result4` equal to 2:

```
int result4 = 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 computations 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 B.4.

Self-Test Exercises

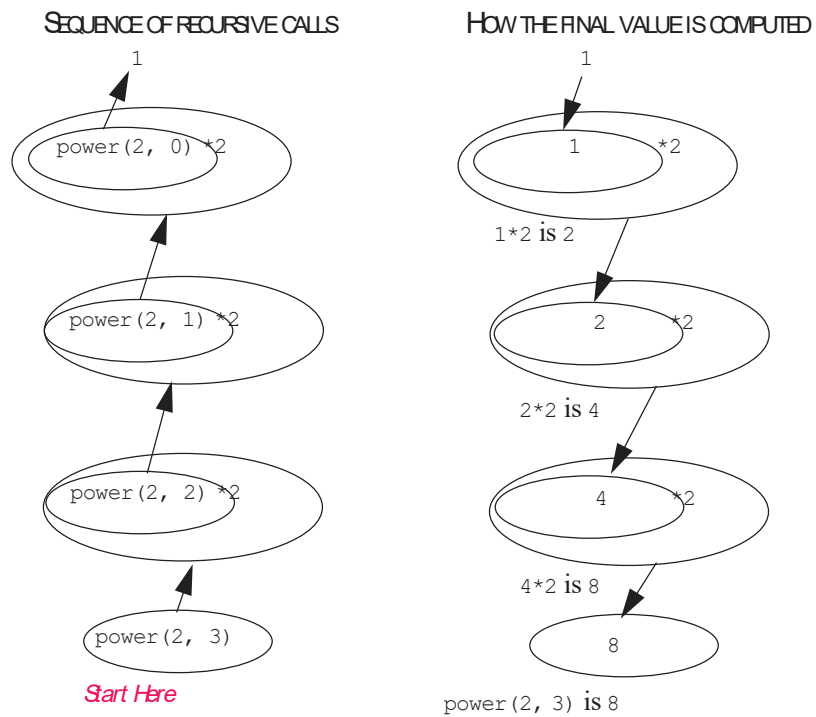
12. What is the output of the following program?

```
#include <iostream>
using std::cout;
using std::endl;

int mystery(int n);
//Precondition n >= 1.

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

int mystery(int n)
{
```

Display 13.4 Evaluating the Recursive Function Call `power(2, 3)`

```

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 std::cout;
using std::endl;

int rose(int n);
//Precondition: n >= 0.

int main()
{
    cout << rose(4) << endl;
}

```

```

        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` are 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 $1/(x^n)$.

13.3 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 13.3. 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 designing 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 you need do is check that the following three properties are satisfied:

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.

criteria for
functions
that return
a value

For example, consider the function `power` in Display 13.3.

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

```
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

$$x^{n-1} * x$$

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

That's all you need to check to be sure that the definition of `power` is correct. (The above 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 mathematical induction 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:

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.

criteria for
void
functions

■ BINARY SEARCH

This subsection develops a recursive function that searches an array to determine 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.

The indexes of the array `a` are the integers 0 through `finalIndex`. To make the task of searching the array easier, we will assume that the array is sorted. Hence, we know the following:

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

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.

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, work your way toward the end of the book. This idea produces our first draft of an algorithm:

algorithm—
first version

```
found = false; //so far.
mid = approximate midpoint between 0 and finalIndex;
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[finalIndex];
```

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[finalIndex];
```

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 `finalIndex`. 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 `finalIndex`, we can express the pseudocode more precisely, as follows:

```
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];
```

algorithm—
first
refinement

To search the entire array, the algorithm would be executed with `first` set equal to 0 and `last` set equal to `finalIndex`. 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`.

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`, 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 13.5.

stopping
case

algorithm—
final version

Display 13.5 Pseudocode for Binary Search

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

ALGORITHM TO SEARCH a[first] **THROUGH** a[last]

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

TO LOCATE THE VALUE KEY:

```
if (first > last) //A stopping case
    found = false;
else
{
    mid = approximate midpoint between first and last;
    if (key == a[mid]) //A stopping case
    {
        found = false;
        location = mid;
    }
    else if key < a[mid] //A case with recursion
        search a[first] through a[mid - 1];
    else if key > a[mid] //A case with recursion
        search a[mid + 1] through a[last];
}
```

CODING

Now we can routinely translate the pseudocode into C++ code. The result is shown in Display 13.6. The function `search` is an implementation of the recursive algorithm given in Display 13.5. A diagram of how the function performs on a sample array is given in Display 13.7.

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 `search` 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 `finalIndex`. However, the recursive calls will set them to values other than 0 and `finalIndex`.



Display 13.6 Recursive Function for Binary Search (part 1 of 2)

```
1  //Program to demonstrate the recursive function for binary search.
2  #include <iostream>
3  using std::cin;
4  using std::cout;
5  using std::endl;
6  const int ARRAY_SIZE = 10;

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 int main( )
13 {
14     int a[ARRAY_SIZE];
15     const int finalIndex = ARRAY_SIZE - 1;

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

16     int key, location;
17     bool found;
18     cout << "Enter number to be located: ";
19     cin >> key;
20     search(a, 0, finalIndex, key, found, location);

21     if (found)
22         cout << key << " is in index location "
23             << location << endl;
24     else
25         cout << key << " is not in the array." << endl;

26     return 0;
27 }
28 void search(const int a[], int first, int last,
29            int key, bool& found, int& location)
30 {
31     int mid;
32     if (first > last)
33     {
34         found = false;
35     }
```

Display 13.6 Recursive Function for Binary Search (*part 2 of 2*)

```

36     else
37     {
38         mid = (first + last)/2;

39         if (key == a[mid])
40         {
41             found = true;
42             location = mid;
43         }
44         else if (key < a[mid])
45         {
46             search(a, first, mid - 1, key, found, location);
47         }
48         else if (key > a[mid])
49         {
50             search(a, mid + 1, last, key, found, location);
51         }
52     }
53 }
```

CHECKING THE RECURSION

The subsection entitled “Recursive Design Techniques” 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 13.6.

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`, which 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.

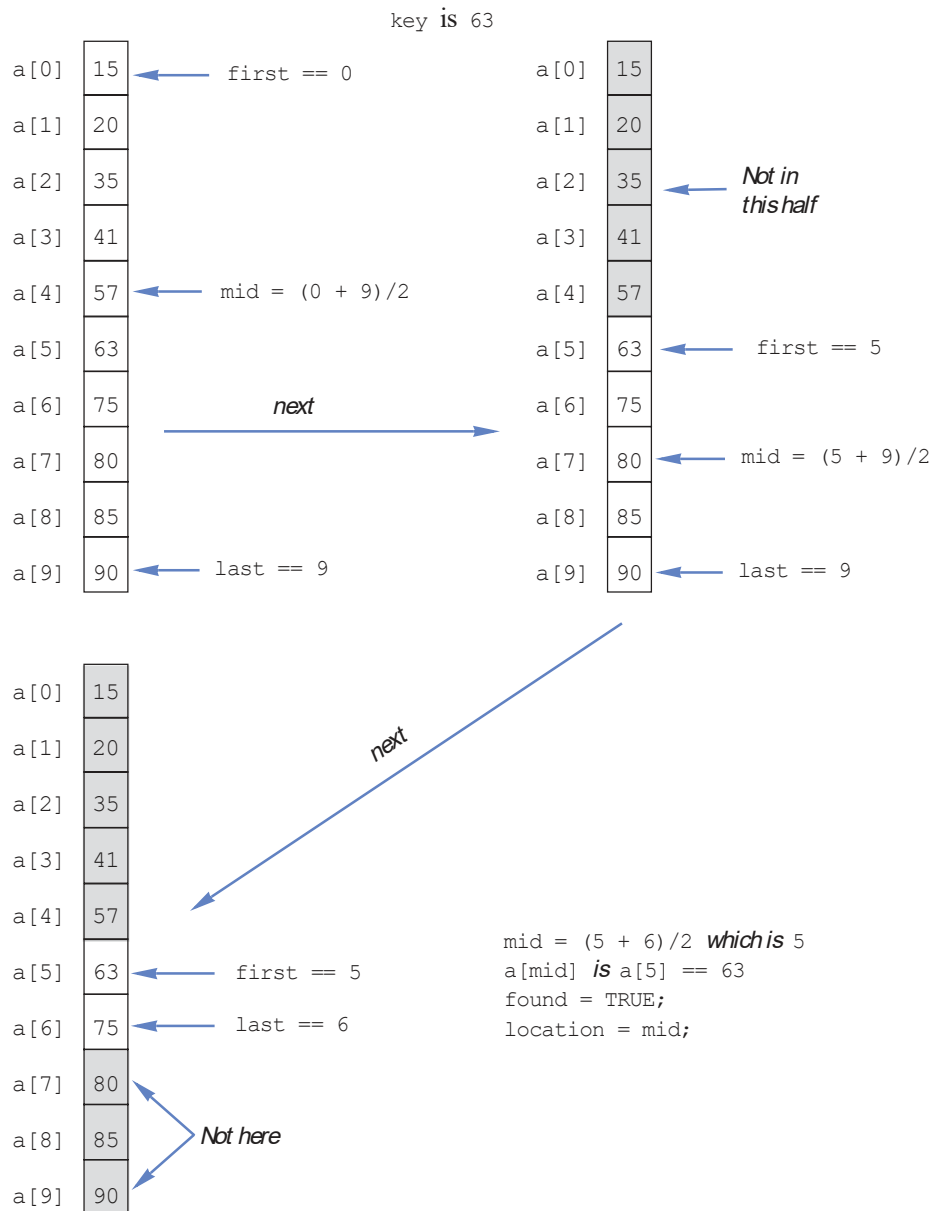
If `first > last`, there are no array elements between `a[first]` and `a[last]` and so `key` is not in this segment of the array. (Nothing is in this segment of the array!) So, if `first > last`, the function `search` correctly sets `found` equal to `false`.

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]`.

Display 13.7 Execution of the Function search



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 only search 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 with 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 only need to compare 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 13.8. 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.

Display 13.8 Iterative Version of Binary Search



FUNCTION DECLARATION

```
void search(const int a[], int lowEnd, int highEnd,
            int key, bool& found, int& location);
//Precondition: a[lowEnd] through a[highEnd] are sorted in increasing
//order.
//Postcondition: If key is not one of the values a[lowEnd] through
//a[highEnd], then found == false; otherwise, a[location] == key and
//found == true.
```

FUNCTION DEFINITION

```
void search(const int a[], int lowEnd, int highEnd,
            int key, bool& found, int& location)
{
    int first = lowEnd;
    int last = highEnd;
    int mid;

    found = false; //so far
    while ( (first <= last) && !(found) )
    {
        mid = (first + last)/2;
        if (key == a[mid])
        {
            found = true;
            location = mid;
        }
        else if (key < a[mid])
        {
            last = mid - 1;
        }
        else if (key > a[mid])
        {
            first = mid + 1;
        }
    }
}
```

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 the numbers 1 through n.
```

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

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 check to see 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 check that the function is correct.
- When designing 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. using std::cout;
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)
```



```

    }
}

//testing code for both Exercises 4 and 5
int main( )
{
    cout << "calling writeUp(" << 10 << ")\n";
    writeUp(10);
    cout << endl;
    cout << "calling writeDown(" << 10 << ")\n";
    writeDown(10);
    cout << endl;
    return 0;
}

/* Test results
calling writeUp(10)
1 2 3 4 5 6 7 8 9 10
calling writeDown(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. `using std::cout;`
`void cheers(int n)`
`{`
 `while (n > 1)`
 `{`
 `cout << "Hip ";`
 `n--;`
 `}`
 `cout << "Hurray\n";`
`}`
8. `using std::cout;`
`void stars(int n)`
`{`
 `for (int count = 1; count <= n; count++)`
 `cout << '*';`
`}`
9. `using std::cout;`
`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 Self-Test Exercise 4. If $n = 3$, the code to be executed is

```

if (3 >= 1)
{
    writeDown(3 - 1);
}

```

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

```

if (2 >= 1)
{
    writeDown(2 - 1)
}

```

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

```

if (1 >= 1)
{
    writeDown(1 - 1)
}

```

On the final recursion, $n = 0$ and the `true` clause is not executed:

```

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

```

The recursion unwinds, the `cout << n << " ";` line of code is executed for each recursive call that was on the stack, with $n = 3$, then $n = 2$, and finally $n = 1$. The output is 3 2 1.

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

```

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

```

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

```

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

```



```
}
```

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

```
if (1 >= 1)
{
    cout << 1 << " ";
    writeUp(1 - 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 by working through the stack) is 1 2 3.

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 );
}
```

PROGRAMMING PROJECTS



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. The Fibonacci numbers are F_0 is 1, F_1 is 1, F_2 is 2, F_3 is 3, F_4 is 5, and in general

$$F_{i+2} = F_i + F_{i+1} \text{ for } i = 0, 1, 2, \dots$$

2. 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 the formula for $C(n, r)$ and write a recursive function that computes the value of the formula. Embed the function in a program and test it.



3. Write a recursive function that has an argument that is an array of characters and two arguments that are 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[1] == 'A' a[2] == 'B' a[3] == 'C' a[4] == 'D' a[5] == 'E'
```

and the bounds are 2 and 5, then after the function is run the array elements should be

```
a[1] == 'A' a[2] == 'E' a[3] == 'D' a[4] == 'C' a[5] == '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 that is an array that contains a string value; the function should reverse 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.

4. Write an iterative version of the recursive function in the previous project. Embed it in a program and test it.



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. Eschatology (concerns about the end of time) and theology will be left to those better qualified; our interest is limited to the recursive solution to the problem.

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 $n-1$ 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. (You need to have first done Programming Project 1.) In this exercise you will compare the efficiency of a recursive and an iterative function to compute the Fibonacci number.
 - a. Examine the recursive function computation of Fibonacci numbers. Note that each Fibonacci number is recomputed many times. To avoid this recomputation, do programming problem 1 iteratively, rather than recursively; that is, do the problem with a loop. You should compute each Fibonacci number once on the way to the number requested and discard the numbers when they are no longer needed.
 - b. Time the solution for project 1 and part a of this project in finding the 1st, 3rd, 5th, 7th, 9th, 11th, 13th, and 15th Fibonacci numbers. Determine how long each function takes. Compare and comment on your results.

Hints: If you are running Linux, you can use the Bash 'time' utility. It gives real time (as in wall clock time), user time (time measured by cpu cycles devoted to your program), and sys time (cpu cycles devoted to tasks other than your program). If you are running in some other environment, you will have to read your manual, or ask your instructor, to find out how to measure the time a program takes to run.

7. (You need to have first done Programming Project 6.) When computing a Fibonacci number using the most straight forward recursive function definition, the recursive solution recomputes each Fibonacci number too many times. To compute $F_{i+2} = F_i + F_{i+1}$, it computes all the numbers computed in F_i a second time in computing F_{i+1} . You can avoid this by saving the numbers in an array while computing F_i . Write another version of your recursive Fibonacci function based on this idea. In the recursive solution for calculating the N^{th} Fibonacci number, declare an array of size N . Array entry with index i stores the i^{th} ($i \leq N$) Fibonacci number as it is computed the first time. Then use the array to avoid the second (redundant) recalculation of the Fibonacci numbers. Time this solution as in Programming Project 6, and compare it to your results for the iterative solution.

For additional online Programming Projects, click the CodeMate icons below.

1.7



14 CHAPTER

Inheritance



14.1 INHERITANCE BASICS 584

Derived Classes 584

Constructors in Derived Classes 594

Pitfall: Use of Private Member Variables from the Base Class 596

Pitfall: Private Member Functions Are Effectively Not Inherited 598

The `protected` Qualifier 598

Redefinition of Member Functions 601

Redefining versus Overloading 603

Access to a Redefined Base Function 604

Functions That Are Not Inherited 605

14.2 PROGRAMMING WITH INHERITANCE 606

Assignment Operators and Copy Constructors in Derived Classes 606

Destructors in Derived Classes 607

Example: Partially Filled Array with Backup 608

Pitfall: Same Object on Both Sides of the Assignment Operator 617

Example: Alternate Implementation of `PFArrayDBak` 617

Tip: A Class Has Access to Private Members of All Objects of the Class 618

Tip: “Is a” versus “Has a” 620

Protected and Private Inheritance 621

Multiple Inheritance 622

CHAPTER SUMMARY 623

ANSWERS TO SELF-TEST EXERCISES 623

PROGRAMMING PROJECTS 625

Like mother, like daughter

Common saying

INTRODUCTION

Object-oriented programming is a popular and powerful programming technique. Among other things, it provides for a dimension of abstraction known as inheritance. This means that a very general form of a class can be defined and compiled. Later, more specialized versions of that class may be defined and can inherit the properties of the general class. This chapter covers inheritance in general and, more specifically, how it is realized in C++.

This chapter does not use any of the material presented in Chapter 12 (file I/O) or Chapter 13 (recursion). It also does not use the material in Section 7.3 of Chapter 7, which covers vectors. Section 14.1 also does not use any material from Chapter 10 (pointers and dynamic arrays).

14.1

Inheritance Basics

If there is anything that we wish to change in the child, we should first examine it and see whether it is not something that could better be changed in ourselves.

Carl Gustav Jung, *The Integration of the Personality*

derived class
base class

Inheritance is the process by which a new class—known as a derived class—is created from another class, called the base class. A derived class automatically has all the member variables and all the ordinary member functions that the base class has, and can have additional member functions and additional member variables.

■ DERIVED CLASSES

Suppose we are designing a record-keeping program that has records for salaried employees and hourly employees. There is a natural hierarchy for grouping these classes. These are all classes of people who share the property of being employees.

Employees who are paid an hourly wage are one subset of employees. Another subset consists of employees who are paid a fixed wage each month or week. Although the program may not need any type corresponding to the set of all employees, thinking in terms of the more general concept of employees

can be useful. For example, all employees have names and Social Security numbers, and the member functions for setting and changing names and Social Security numbers will be the same for salaried and hourly employees.

Within C++ you can define a class called `Employee` that includes all employees, whether salaried or hourly, and then use this class to define classes for hourly employees and salaried employees.

The class `Employee` will also contain member functions that manipulate the data fields of the class `Employee`. Displays 14.1 and 14.2 show one possible definition for the class `Employee`.

Display 14.1 Interface for the Base Class `Employee`

```

1
2 //This is the header file employee.h.
3 //This is the interface for the class Employee.
4 //This is primarily intended to be used as a base class to derive
5 //classes for different kinds of employees.
6 #ifndef EMPLOYEE_H
7 #define EMPLOYEE_H

8 #include <string>
9 using std::string;

10 namespace SavitchEmployees
11 {

12     class Employee
13     {
14     public:
15         Employee( );
16         Employee(string theName, string theSsn);
17         string getName( ) const;
18         string getSsn( ) const;
19         double getNetPay( ) const;
20         void setName(string newName);
21         void setSsn(string newSsn);
22         void setNetPay(double newNetPay);
23         void printCheck( ) const;
24     private:
25         string name;
26         string ssn;
27         double netPay;
28     };

29 } //SavitchEmployees

30 #endif //EMPLOYEE_H

```

Display 14.2 Implementation for the Base Class Employee (part 1 of 2)

```
1
2 //This is the file employee.cpp.
3 //This is the implementation for the class Employee.
4 //The interface for the class Employee is in the header file employee.h.
5 #include <string>
6 #include <cstdlib>
7 #include <iostream>
8 #include "employee.h"
9 using std::string;
10 using std::cout;

11 namespace SavitchEmployees
12 {
13     Employee::Employee( ) : name("No name yet"), ssn("No number yet"), netPay(0)
14     {
15         //deliberately empty
16     }

17     Employee::Employee(string theName, string theNumber)
18         : name(theName), ssn(theNumber), netPay(0)
19     {
20         //deliberately empty
21     }

22     string Employee::getName( ) const
23     {
24         return name;
25     }

26     string Employee::getSsn( ) const
27     {
28         return ssn;
29     }

30
31     double Employee::getNetPay( ) const
32     {
33         return netPay;
34     }

35     void Employee::setName(string newName)
36     {
37         name = newName;
38     }

39     void Employee::setSsn(string newSsn)
```
