

CHAPTER

17

RECURSION

Objectives

- To describe what a recursive function is and the benefits of using recursion (§17.1).
- To develop recursive programs for recursive mathematical functions (§§17.2–17.3).
- To explain how recursive function calls are handled in a call stack (§§17.2–17.3).
- To think recursively (§17.4).
- To use an overloaded helper function to derive a recursive function (§17.5).
- To solve selection sort using recursion (§17.5.1).
- To solve binary search using recursion (§17.5.2).
- To solve the Towers of Hanoi problem using recursion (§17.6).
- To solve the Eight Queens problem using recursion (§17.7).
- To understand the relationship and difference between recursion and iteration (§17.8).
- To know tail-recursive functions and why they are desirable (§17.9).



17.1 Introduction



Recursion is a technique that leads to elegant solutions to problems that are difficult to program using simple loops.

string permutations

Suppose you wish to print all permutations of a string. For example, for a string **abc**, its permutations are **abc**, **acb**, **bac**, **bca**, **cab**, and **cba**. How do you solve this problem? There are several ways to do so. An intuitive and effective solution is to use recursion.

Eight Queens problem

The classic Eight Queens puzzle is to place eight queens on a chessboard such that no two can attack each other (i.e., no two queens are on the same row, same column, or same diagonal), as shown in Figure 17.1. How do you write a program to solve this problem? There are several ways to solve this problem. An intuitive and effective solution is to use recursion.

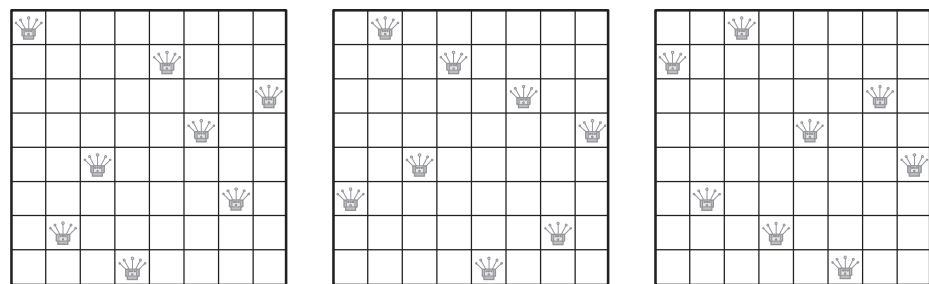


FIGURE 17.1 The Eight Queens problem can be solved using recursion.

recursive function

To use recursion is to program using *recursive functions*—functions that invoke themselves. Recursion is a useful programming technique. In some cases, it enables you to develop a natural, straightforward, simple solution to an otherwise difficult problem. This chapter introduces the concepts and techniques of recursive programming and illustrates by examples how to “think recursively.”

17.2 Example: Factorials



A recursive function is one that invokes itself.

Many mathematical functions are defined using recursion. We begin with a simple example that illustrates recursion.

The factorial of a number **n** can be recursively defined as follows:

$$\begin{aligned} 0! &= 1; \\ n! &= n \times (n - 1)!; \quad n > 0 \end{aligned}$$

How do you find **n!** for a given **n**? It is easy to find **1!** because you know that **0!** is **1** and **1!** is **1 × 0!**. Assuming you know that **(n - 1)!**, **n!** can be obtained immediately using **n × (n - 1)!**. Thus, the problem of computing **n!** is reduced to computing **(n - 1)!**. When computing **(n - 1)!**, you can apply the same idea recursively until **n** is reduced to **0**.

Let **factorial(n)** be the function for computing **n!**. If you call the function with **n = 0**, it immediately returns the result. The function knows how to solve the simplest case, which is referred to as the *base case* or the *stopping condition*. If you call the function with **n > 0**, it reduces the problem into a subproblem for computing the factorial of **n - 1**. The subproblem is essentially the same as the original problem but is simpler or smaller than the original. Because the subproblem has the same property as the original, you can call the function with a different argument, which is referred to as a *recursive call*.

base case or stopping condition

recursive call

The recursive algorithm for computing **factorial(n)** can be simply described as follows:

```
if (n == 0)
    return 1;
else
    return n * factorial(n - 1);
```

A recursive call can result in many more recursive calls, because the function is dividing a subproblem into new subproblems. For a recursive function to terminate, the problem must eventually be reduced to a stopping case. At this point the function returns a result to its caller. The caller then performs a computation and returns the result to its own caller. This process continues until the result is passed back to the original caller. The original problem can now be solved by multiplying **n** by the result of **factorial(n - 1)**.

Listing 17.1 is a complete program that prompts the user to enter a nonnegative integer and displays the factorial for the number.

LISTING 17.1 ComputeFactorial.cpp

```
1  #include <iostream>
2  using namespace std;
3
4  // Return the factorial for a specified index
5  int factorial(int);
6
7  int main()
8  {
9      // Prompt the user to enter an integer
10     cout << "Please enter a non-negative integer: ";
11     int n;
12     cin >> n;
13
14     // Display factorial
15     cout << "Factorial of " << n << " is " << factorial(n);
16
17     return 0;
18 }
19
20 // Return the factorial for a specified index
21 int factorial(int n)
22 {
23     if (n == 0) // Base case
24         return 1;
25     else
26         return n * factorial(n - 1); // Recursive call
27 }
```

Please enter a nonnegative integer: 5
Factorial of 5 is 120



The **factorial** function (lines 21–27) is essentially a direct translation of the recursive mathematical definition for the factorial into C++ code. The call to **factorial** is recursive because it calls itself. The parameter passed to **factorial** is decremented until it reaches the base case of **0**.

You see how to write a recursive function. How does recursion work? Figure 17.2 illustrates the execution of the recursive calls, starting with **n = 4**. The use of stack space for recursive calls is shown in Figure 17.3.

how does it work?

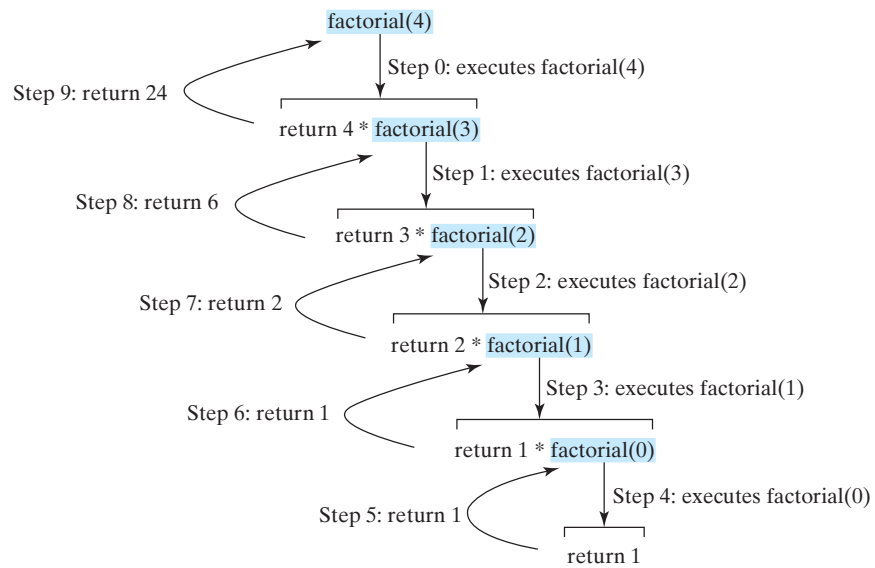


FIGURE 17.2 Invoking **factorial(4)** spawns recursive calls to **factorial**.

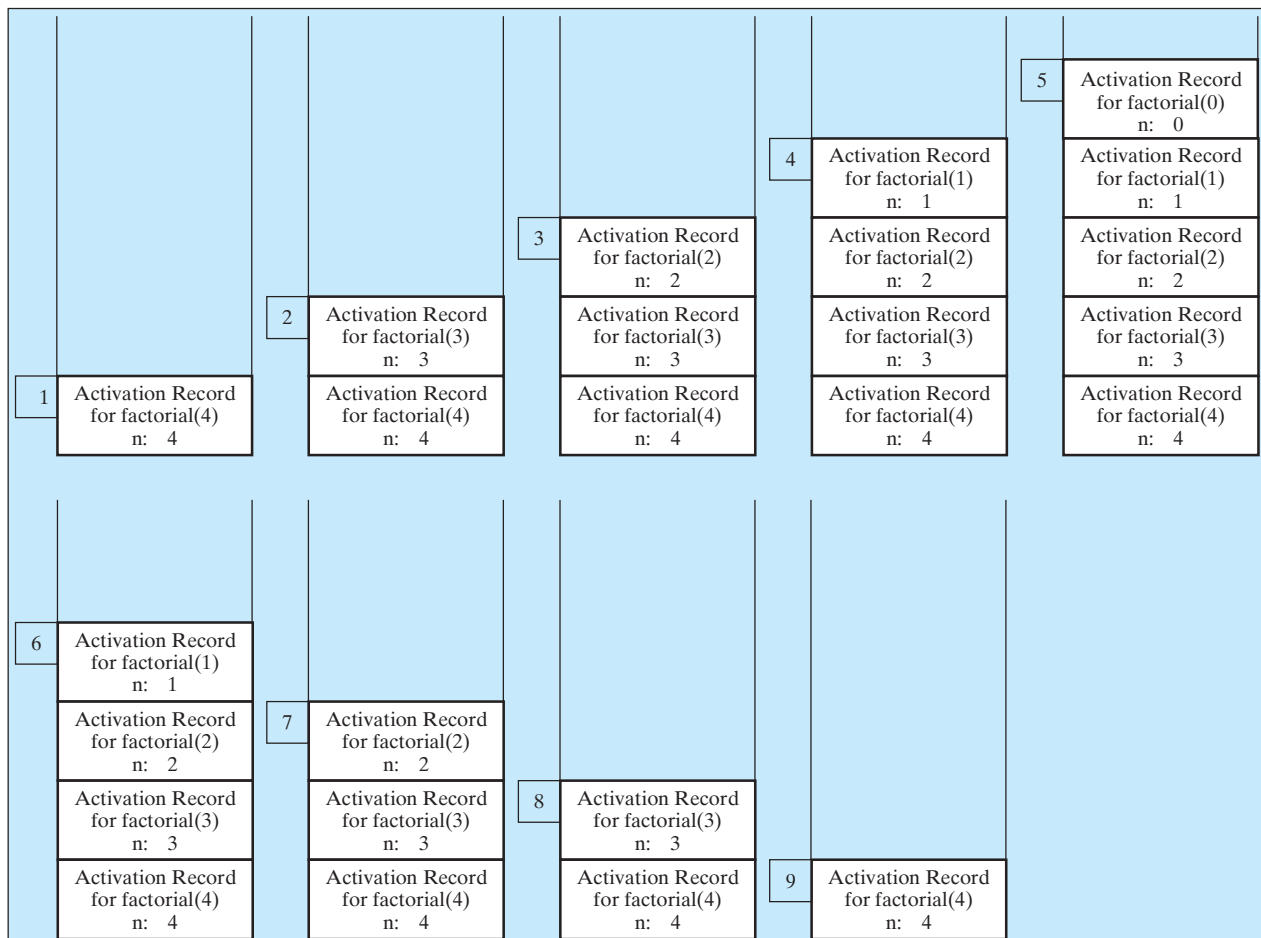


FIGURE 17.3 When **factorial(4)** is being executed, the **factorial** function is called recursively, causing memory space to dynamically change.

**Caution**

Infinite recursion can occur if recursion does not reduce the problem in a manner that allows it to eventually converge into the base case or a base case is not specified. For example, suppose you mistakenly write the **factorial** function as follows:

infinite recursion

```
int factorial(int n)
{
    return n * factorial(n - 1);
}
```

The function runs infinitely and causes the stack overflow.

**Pedagogical Note**

It is simpler and more efficient to implement the **factorial** function using a loop. However, the recursive **factorial** function is a good example to demonstrate the concept of recursion.

**Note**

The example discussed so far shows a recursive function that invokes itself. This is known as *direct recursion*. It is also possible to create *indirect recursion*. This occurs when function **A** invokes function **B**, which in turn invokes function **A**. There can even be several more functions involved in the recursion. For example, function **A** invokes function **B**, which invokes function **C**, which invokes function **A**.

direct recursion
indirect recursion

- 17.1** What is a recursive function? Describe the characteristics of recursive functions. What is an infinite recursion?
- 17.2** Show the output of the following programs and identify base cases and recursive calls.

**Check Point**

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

int f(int n)
{
    if (n == 1)
        return 1;
    else
        return n + f(n - 1);
}

int main()
{
    cout << "Sum is " << f(5) << endl;

    return 0;
}
```

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

void f(int n)
{
    if (n > 0)
    {
        cout << n % 10;
        f(n / 10);
    }
}

int main()
{
    f(1234567);

    return 0;
}
```

- 17.3** Write a recursive mathematical definition for computing 2^n for a positive integer n .
- 17.4** Write a recursive mathematical definition for computing x^n for a positive integer n and a real number x .
- 17.5** Write a recursive mathematical definition for computing $1 + 2 + 3 + \dots + n$ for a positive integer.
- 17.6** How many times is the **factorial** function in Listing 17.1 invoked for **factorial(6)**?



17.3 Case Study: Fibonacci Numbers

In some cases, recursion enables you to create an intuitive, straightforward, simple solution to a problem.

The **factorial** function in the preceding section easily could be rewritten without using recursion. In some cases, however, using recursion enables you to give a natural, straightforward, simple solution to a program that would otherwise be difficult to solve. Consider the well-known Fibonacci series problem, as follows:

The series:	0	1	1	2	3	5	8	13	21	34	55	89	. . .
Indices:	0	1	2	3	4	5	6	7	8	9	10	11	

The Fibonacci series begins with **0** and **1**, and each subsequent number is the sum of the preceding two numbers in the series. The series can be defined recursively as follows:

```
fib(0) = 0;
fib(1) = 1;
fib(index) = fib(index - 2) + fib(index - 1); index >= 2
```

The Fibonacci series was named for Leonardo Fibonacci, a medieval mathematician, who originated it to model the growth of the rabbit population. It can be applied in numeric optimization and in various other areas.

How do you find **fib(index)** for a given **index**? It is easy to find **fib(2)** because you know **fib(0)** and **fib(1)**. Assuming that you know **fib(index - 2)** and **fib(index - 1)**, **fib(index)** can be obtained immediately. Thus, the problem of computing **fib(index)** is reduced to computing **fib(index - 2)** and **fib(index - 1)**. When computing **fib(index - 2)** and **fib(index - 1)**, you apply the idea recursively until **index** is reduced to **0** or **1**.

The base case is **index = 0** or **index = 1**. If you call the function with **index = 0** or **index = 1**, it immediately returns the result. If you call the function with **index >= 2**, it divides the problem into two subproblems for computing **fib(index - 1)** and **fib(index - 2)** using recursive calls. The recursive algorithm for computing **fib(index)** can be simply described as follows:

```
if (index == 0)
    return 0;
else if (index == 1)
    return 1;
else
    return fib(index - 1) + fib(index - 2);
```

Listing 17.2 is a complete program that prompts the user to enter an index and computes the Fibonacci number for the index.

LISTING 17.2 ComputeFibonacci.cpp

```
1 #include <iostream>
2 using namespace std;
3
4 // The function for finding the Fibonacci number
5 int fib(int);
6
7 int main()
8 {
9     // Prompt the user to enter an integer
10    cout << "Enter an index for the Fibonacci number: ";
11    int index;
```



```

12  cin >> index;
13
14  // Display factorial
15  cout << "Fibonacci number at index " << index << " is "
16      << fib(index) << endl;
17
18  return 0;
19 }
20
21 // The function for finding the Fibonacci number
22 int fib(int index)
23 {
24     if (index == 0) // Base case                base case
25         return 0;
26     else if (index == 1) // Base case            base case
27         return 1;
28     else // Reduction and recursive calls
29         return fib(index - 1) + fib(index - 2);    recursion
30 }

```

Enter an index for the Fibonacci number: 7

Fibonacci number at index 7 is 13



The program does not show the considerable amount of work done behind the scenes by the computer. Figure 17.4, however, shows successive recursive calls for evaluating **fib(4)**. The original function, **fib(4)**, makes two recursive calls, **fib(3)** and **fib(2)**, and then returns **fib(3) + fib(2)**. But in what order are these functions called? In C++, operands for the binary **+** operator may be evaluated in any order. Assume it is evaluated from the left to right. The labels in Figure 17.4 show the order in which functions are called.

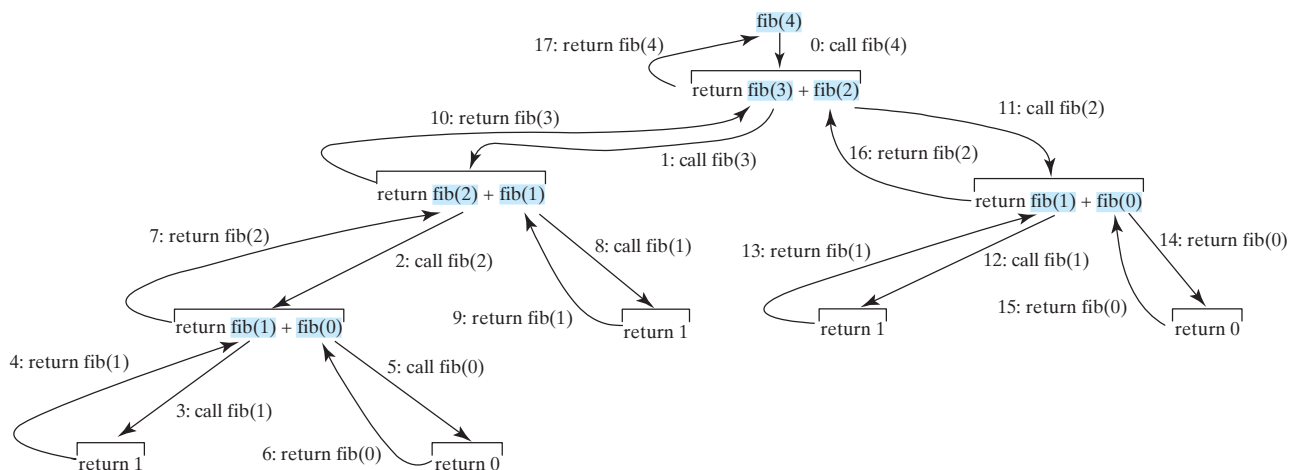


FIGURE 17.4 Invoking **fib(4)** spawns recursive calls to **fib**.

As shown in Figure 17.4, there are many duplicated recursive calls. For instance, **fib(2)** is called twice, **fib(1)** three times, and **fib(0)** twice. In general, computing **fib(index)** requires twice as many recursive calls as are needed for computing **fib(index - 1)**. As you try larger index values, the number of calls substantially increases, as shown in Table 17.1.

TABLE 17.1 Number of Recursive Calls in **fib** (index)

index	2	3	4	10	20	30	40	50
# of calls	3	5	9	177	21891	2,692,537	331,160,281	2,075,316,483

**Pedagogical Note**

The recursive implementation of the **fib** function is very simple and straightforward, but not efficient. See Programming Exercise 17.2 for an efficient solution using loops. The recursive **fib** function is a good example to demonstrate how to write recursive functions, though it is not practical.

**Check Point****17.7** How many times is the **fib** function in Listing 17.2 invoked for **fib(6)**?**17.8** Show the output of the following two programs:

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

void f(int n)
{
    if (n > 0)
    {
        cout << n << " ";
        f(n - 1);
    }
}

int main()
{
    f(5);

    return 0;
}
```

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

void f(int n)
{
    if (n > 0)
    {
        f(n - 1);
        cout << n << " ";
    }
}

int main()
{
    f(5);

    return 0;
}
```

17.9 What is wrong in the following function?

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

void f(double n)
{
    if (n != 0)
    {
        cout << n;
        f(n / 10);
    }
}

int main()
{
    f(1234567);

    return 0;
}
```


17.4 Problem Solving Using Recursion

If you think recursively, you can solve many problems using recursion.

The preceding sections presented two classic recursion examples. All recursive functions have the following characteristics:

- The function is implemented using an **if-else** or a **switch** statement that leads to different cases.
- One or more *base cases* (the simplest case) are used to stop recursion.
- Every recursive call reduces the original problem, bringing it increasingly closer to a base case until it becomes that case.



recursion characteristics

if-else

base cases

reduction

In general, to solve a problem using recursion, you break it into subproblems. If a subproblem resembles the original problem, you can apply the same approach to solve the subproblem recursively. This subproblem is almost the same as the original problem in nature with a smaller size.

Recursion is everywhere. It is fun to *think recursively*. Consider drinking coffee. You may describe the procedure recursively as follows:

think recursively

```
void drinkCoffee(Cup& cup)
{
    if (!cup.isEmpty())
    {
        cup.takeOneSip(); // Take one sip
        drinkCoffee(cup);
    }
}
```



Assume **cup** is an object for a cup of coffee with the instance functions **isEmpty()** and **takeOneSip()**. You can break the problem into two subproblems: one is to drink one sip of coffee and the other is to drink the rest of the coffee in the cup. The second problem is the same as the original problem but smaller in size. The base case for the problem is when **cup** is empty.

Let us consider a simple problem of printing a message for **n** times. You can break the problem into two subproblems: one is to print the message one time and the other is to print the message **n - 1** times. The second problem is the same as the original problem with a smaller size. The base case for the problem is **n == 0**. You can solve this problem using recursion as follows:

```
void nPrintln(const string& message, int times)
{
    if (times >= 1)
    {
        cout << message << endl;
        nPrintln(message, times - 1);
    } // The base case is times == 0
}
```

recursive call

Note that the `fib` function in Listing 17.2 returns a value to its caller, but the `nPrintln` function is `void` and does not return a value to its caller.

think recursively

Many of the problems presented in the early chapters can be solved using recursion if you *think recursively*. Consider the palindrome problem in Listing 5.16, `TestPalindrome.cpp`. Recall that a string is a palindrome if it reads the same from the left and from the right. For example, `mom` and `dad` are palindromes, but `uncle` and `aunt` are not. The problem to check whether a string is a palindrome can be divided into two subproblems:

- Check whether the first character and the last character of the string are equal.
- Ignore these two end characters and check whether the rest of the substring is a palindrome.

The second subproblem is the same as the original problem with a smaller size. There are two base cases: (1) the two end characters are not the same; (2) the string size is `0` or `1`. In case 1, the string is not a palindrome; and in case 2, the string is a palindrome. The recursive function for this problem can be implemented in Listing 17.3.

LISTING 17.3 RecursivePalindrome.cpp

include header file	1 <code>#include <iostream></code> 2 <code>#include <string></code> 3 <code>using namespace std;</code> 4
function header	5 <code>bool isPalindrome(const string& s)</code>
string length	6 { 7 <code>if (s.size() <= 1) // Base case</code> 8 <code>return true;</code> 9 <code>else if (s[0] != s[s.size() - 1]) // Base case</code> 10 <code>return false;</code>
recursive call	11 <code>else</code> 12 <code>return isPalindrome(s.substr(1, s.size() - 2));</code> 13 } 14
input string	15 <code>int main()</code> 16 { 17 <code>cout << "Enter a string: ";</code> 18 <code>string s;</code> 19 <code>getline(cin, s);</code> 20 21 <code>if (isPalindrome(s))</code> 22 <code>cout << s << " is a palindrome" << endl;</code> 23 <code>else</code> 24 <code>cout << s << " is not a palindrome" << endl;</code> 25 26 <code>return 0;</code> 27 }



```
Enter a string: aba ↵ Enter
aba is a palindrome
```



```
Enter a string: abab ↵ Enter
abab is not a palindrome
```

The `isPalindrome` function checks whether the size of the string is less than or equal to `1` (line 7). If so, the string is a palindrome. The function checks whether the first and the last elements of the string are the same (line 9). If not, the string is not a palindrome. Otherwise,

obtain a substring of `s` using `s.substr(1, s.size() - 2)` and recursively invoke `isPalindrome` with the new string (line 12).

17.5 Recursive Helper Functions

Sometimes you can find a solution to the original problem by defining a recursive function to a problem similar to the original problem. This new function is called a recursive helper function. The original problem can be solved by invoking the recursive helper function.



The preceding recursive `isPalindrome` function is not efficient, because it creates a new string for every recursive call. To avoid creating new strings, you can use the `low` and `high` indices to indicate the range of the substring. These two indices must be passed to the recursive function. Since the original function is `isPalindrome(const string& s)`, you have to create a new function `isPalindrome(const string& s, int low, int high)` to accept additional information on the string, as shown in Listing 17.4.

LISTING 17.4 RecursivePalindromeUsingHelperFunction.cpp

```

1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  bool isPalindrome(const string& s, int low, int high)           helper function
6  {
7      if (high <= low) // Base case
8          return true;
9      else if (s[low] != s[high]) // Base case
10         return false;
11     else
12         return isPalindrome(s, low + 1, high - 1);           recursive call
13 }
14
15 bool isPalindrome(const string& s)                             function header
16 {
17     return isPalindrome(s, 0, s.size() - 1);                 invoke helper function
18 }
19
20 int main()
21 {
22     cout << "Enter a string: ";
23     string s;
24     getline(cin, s);                                         input string
25
26     if (isPalindrome(s))
27         cout << s << " is a palindrome" << endl;
28     else
29         cout << s << " is not a palindrome" << endl;
30
31     return 0;
32 }
```

Enter a string: aba

aba is a palindrome

Enter a string: abab

abab is not a palindrome



Two overloaded `isPalindrome` functions are defined. The function `isPalindrome(const string& s)` (line 15) checks whether a string is a palindrome, and the second function `isPalindrome(const string& s, int low, int high)` (line 5) checks whether a substring `s(low..high)` is a palindrome. The first function passes the string `s` with `low = 0` and `high = s.size() - 1` to the second function. The second function can be invoked recursively to check a palindrome in an ever-shrinking substring. It is a common design technique in recursive programming to define a second function that receives additional parameters. Such a function is known as a *recursive helper function*.

recursive helper function

Helper functions are very useful to design recursive solutions for problems involving strings and arrays. The sections that follow present two more examples.

17.5.1 Selection Sort

Selection sort was introduced in Section 7.10, “Sorting Arrays.” Now we introduce a recursive selection sort for characters in a string. A variation of selection sort works as follows. It finds the largest element in the list and places it last. It then finds the largest element remaining and places it next to last, and so on until the list contains only a single element. The problem can be divided into two subproblems:

- Find the largest element in the list and swap it with the last element.
- Ignore the last element and sort the remaining smaller list recursively.

The base case is that the list contains only one element.

Listing 17.5 gives the recursive sort function.

LISTING 17.5 RecursiveSelectionSort.cpp

helper sort function

```

1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  void sort(string& s, int high)
6  {
7      if (high > 0)
8      {
9          // Find the largest element and its index
10         int indexOfMax = 0;
11         char max = s[0];
12         for (int i = 1; i <= high; i++)
13         {
14             if (s[i] > max)
15             {
16                 max = s[i];
17                 indexOfMax = i;
18             }
19         }
20
21         // Swap the largest with the last element in the list
22         s[indexOfMax] = s[high];
23         s[high] = max;
24
25         // Sort the remaining list
26         sort(s, high - 1);
27     }
28 }
29
30 void sort(string& s)

```

recursive call

sort function


```

31 {
32     sort(s, s.size() - 1);
33 }
34
35 int main()
36 {
37     cout << "Enter a string: ";
38     string s;
39     getline(cin, s);
40
41     sort(s);
42
43     cout << "The sorted string is " << s << endl;
44
45     return 0;
46 }

```

invoke helper function

input string

Enter a string: ghfdacb 

The sorted string is abcd fgh



Two overloaded `sort` functions are defined. The function `sort(string& s)` sorts characters in `s[0..s.size() - 1]` and the second function `sort(string& s, int high)` sorts characters in `s[0..high]`. The helper function can be invoked recursively to sort an ever-shrinking substring.

17.5.2 Binary Search

Binary search was introduced in Section 7.9.2, “The Binary Search Approach.” For binary search to work, the elements in the array must already be ordered. The binary search first compares the key with the element in the middle of the array. Consider the following cases:

- Case 1: If the key is less than the middle element, recursively search the key in the first half of the array.
- Case 2: If the key is equal to the middle element, the search ends with a match.
- Case 3: If the key is greater than the middle element, recursively search the key in the second half of the array.

Case 1 and Case 3 reduce the search to a smaller list. Case 2 is a base case when there is a match. Another base case is that the search is exhausted without a match. Listing 17.6 gives a clear, simple solution for the binary search problem using recursion.



VideoNote
Binary search

LISTING 17.6 RecursiveBinarySearch.cpp

```

1  #include <iostream>
2  using namespace std;
3
4  int binarySearch(const int list[], int key, int low, int high)
5  {
6      if (low > high) // The list has been exhausted without a match
7          return -low - 1; // key not found, return the insertion point
8
9      int mid = (low + high) / 2;
10     if (key < list[mid])
11         return binarySearch(list, key, low, mid - 1);
12     else if (key == list[mid])

```

helper function

base case

recursive call

658 Chapter 17 Recursion

base case	13 return mid;
	14 else
recursive call	15 return binarySearch(list, key, mid + 1, high);
	16 }
	17
binarySearch function	18 int binarySearch(const int list[], int key, int size)
	19 {
	20 int low = 0;
	21 int high = size - 1;
call helper function	22 return binarySearch(list, key, low, high);
	23 }
	24
	25 int main()
	26 {
	27 int list[] = { 2, 4, 7, 10, 11, 45, 50, 59, 60, 66, 69, 70, 79};
	28 int i = binarySearch(list, 2, 13); // Returns 0
	29 int j = binarySearch(list, 11, 13); // Returns 4
	30 int k = binarySearch(list, 12, 13); // Returns -6
	31
	32 cout << "binarySearch(list, 2, 13) returns " << i << endl;
	33 cout << "binarySearch(list, 11, 13) returns " << j << endl;
	34 cout << "binarySearch(list, 12, 13) returns " << k << endl;
	35
	36 return 0;
	37 }



```
binarySearch(list, 2, 13) returns 0
binarySearch(list, 11, 13) returns 4
binarySearch(list, 12, 13) returns -6
```

The **binarySearch** function in line 18 finds a key in the whole list. The helper **binarySearch** function in line 4 finds a key in the list with index from **low** to **high**.

The **binarySearch** function in line 18 passes the initial array with **low = 0** and **high = size - 1** to the helper **binarySearch** function. The helper function is invoked recursively to find the key in an ever-shrinking subarray.



17.10 Show the call stack for **isPalindrome("abcba")** using the functions defined in Listings 17.3 and 17.4, respectively.

17.11 Show the call stack for **selectionSort("abcba")** using the function defined in Listing 17.5.

17.12 What is a recursive helper function?



VideoNote
Towers of Hanoi



17.6 Towers of Hanoi

The classic Towers of Hanoi problem can be solved easily using recursion, but it is difficult to solve otherwise.

The Towers of Hanoi problem is a classic recursion example. It can be solved easily using recursion but is difficult to solve otherwise.

The problem involves moving a specified number of disks of distinct sizes from one tower to another while observing the following rules:

- There are n disks labeled 1, 2, 3, . . . , n , and three towers labeled A, B, and C.
- No disk can be on top of a smaller disk at any time.

- All the disks are initially placed on tower A.
- Only one disk can be moved at a time, and it must be the top disk on the tower.

The objective is to move all the disks from A to B with the assistance of C. For example, if you have three disks, the steps to move all of the disks from A to B are shown in Figure 17.5.

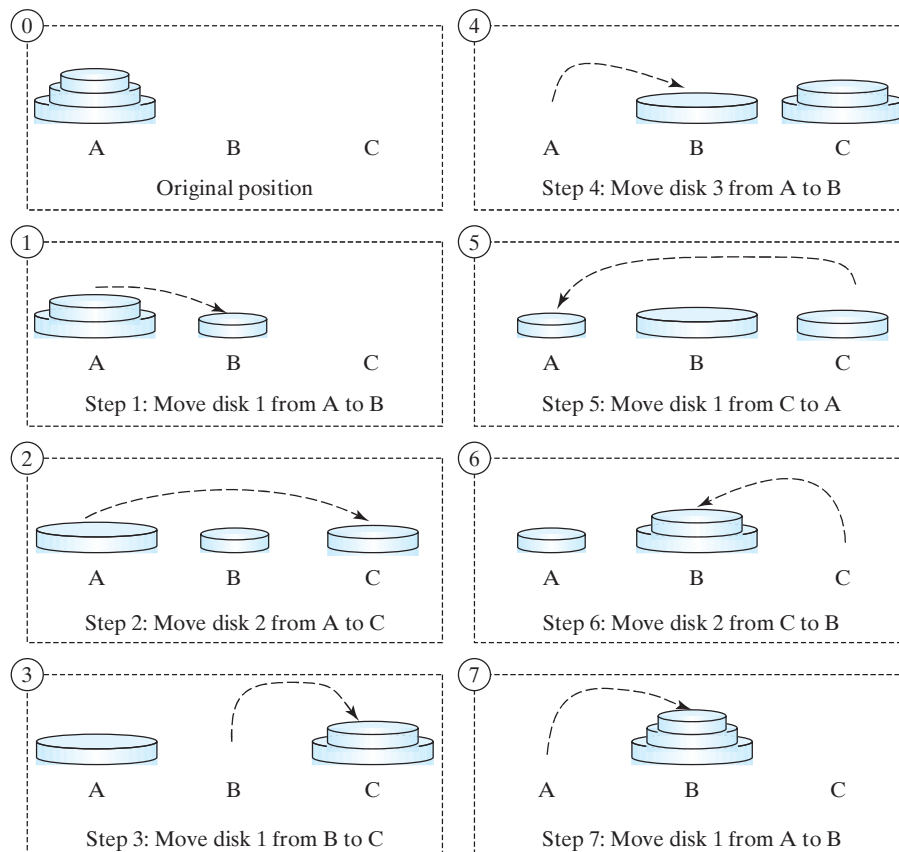


FIGURE 17.5 The goal of the Towers of Hanoi problem is to move disks from tower A to tower B without breaking the rules.



Note

The Towers of Hanoi is a classic computer science problem. Many websites are devoted to this problem. The website www.cut-the-knot.com/recurrence/hanoi.shtml is worth a look.

In the case of three disks, you can find the solution manually. For a larger number of disks, however—even for four—the problem is quite complex. Fortunately, the problem has an inherently recursive nature, which leads to a straightforward recursive solution.

The base case for the problem is $n = 1$. If $n == 1$, you could simply move the disk from A to B. When $n > 1$, you could split the original problem into three subproblems and solve them sequentially.

1. Move the first $n - 1$ disks from A to C recursively with the assistance of tower B, as shown in Step 1 in Figure 17.6.
2. Move disk n from A to B, as shown in Step 2 in Figure 17.6.

3. Move $n - 1$ disks from C to B recursively with the assistance of tower A, as shown in Step 3 in Figure 17.6.

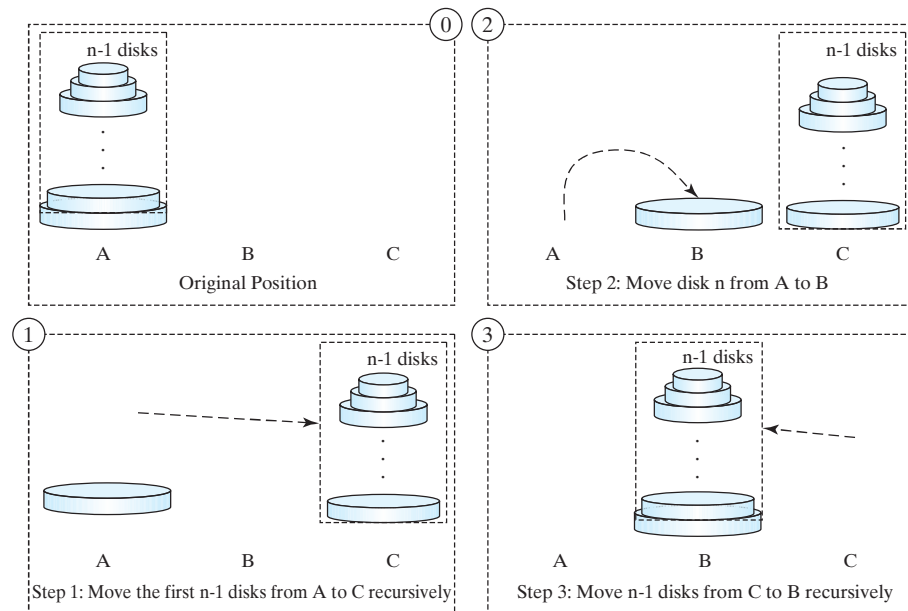


FIGURE 17.6 The Towers of Hanoi problem can be decomposed into three subproblems.

The following function moves n disks from the **fromTower** to the **toTower** with the assistance of the **auxTower**:

```
void moveDisks(int n, char fromTower, char toTower, char auxTower)
```

The algorithm for the function can be described as follows:

```
if (n == 1) // Stopping condition
    Move disk 1 from the fromTower to the toTower;
else
{
    moveDisks(n - 1, fromTower, auxTower, toTower);
    Move disk n from the fromTower to the toTower;
    moveDisks(n - 1, auxTower, toTower, fromTower);
}
```

Listing 17.7 prompts the user to enter the number of disks and invokes the recursive function **moveDisks** to display the solution for moving the disks.

LISTING 17.7 TowersOfHanoi.cpp

```
1 #include <iostream>
2 using namespace std;
3
4 // The function for finding the solution to move n disks
5 // from fromTower to toTower with auxTower
```

```

6 void moveDisks(int n, char fromTower,
7   char toTower, char auxTower)
8 {
9   if (n == 1) // Stopping condition
10    cout << "Move disk " << n << " from " <<
11      fromTower << " to " << toTower << endl;
12   else
13   {
14     moveDisks(n - 1, fromTower, auxTower, toTower);
15     cout << "Move disk " << n << " from " <<
16       fromTower << " to " << toTower << endl;
17     moveDisks(n - 1, auxTower, toTower, fromTower);
18   }
19 }
20
21 int main()
22 {
23   // Read number of disks, n
24   cout << "Enter number of disks: ";
25   int n;
26   cin >> n;
27
28   // Find the solution recursively
29   cout << "The moves are: " << endl;
30   moveDisks(n, 'A', 'B', 'C');
31
32   return 0;
33 }

```

recursive function

recursion

recursion

```

Enter number of disks: 4
The moves are:
Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B
Move disk 3 from A to C
Move disk 1 from B to A
Move disk 2 from B to C
Move disk 1 from A to C
Move disk 4 from A to B
Move disk 1 from C to B
Move disk 2 from C to A
Move disk 1 from B to A
Move disk 3 from C to B
Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B

```



This problem is inherently recursive. Using recursion makes it possible to find a natural, simple solution. It would be difficult to solve the problem without using recursion.

Consider tracing the program for $n = 3$. The successive recursive calls are shown in Figure 17.7. As you can see, writing the program is easier than tracing the recursive calls. The system uses stacks to trace the calls behind the scenes. To some extent, recursion provides a level of abstraction that hides iterations and other details from the user.

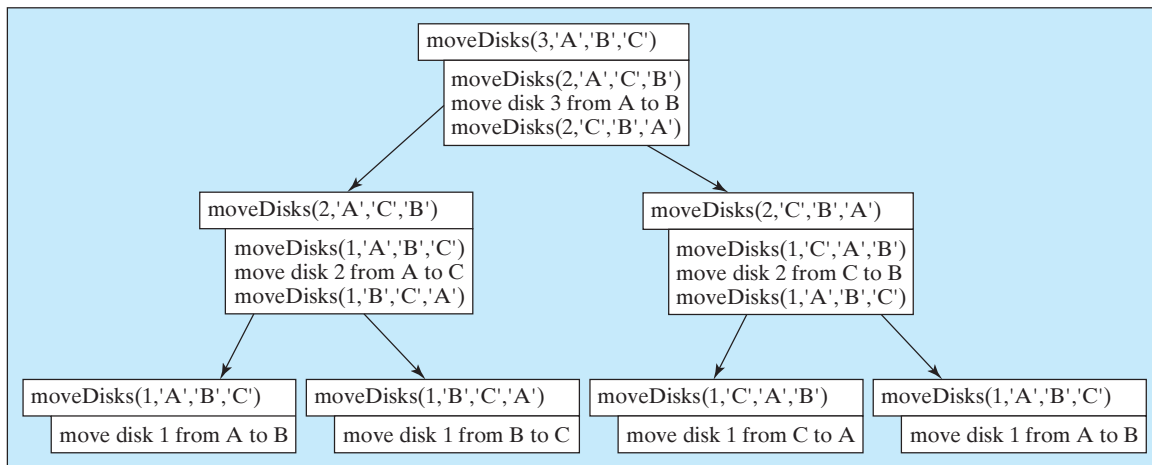


FIGURE 17.7 Invoking `moveDisks(3, 'A', 'B', 'C')` spawns calls to `moveDisks` recursively.



17.13 How many times is the `moveDisks` function in Listing 17.7 invoked for `moveDisks(5, 'A', 'B', 'C')`?



17.7 Eight Queens

The Eight Queens problem can be solved using recursion.

This section gives a recursive solution to the Eight Queens problem presented earlier. The task is to place a queen in each row on a chessboard in such a way that no two queens can attack each other. You may use a two-dimensional array to represent a chessboard. However, since each row can have only one queen, it is sufficient to use a one-dimensional array to denote the position of the queen in the row. So, let us declare array `queens` as follows:

```
int queens[8];
```

Assign `j` to `queens[i]` to denote that a queen is placed in row `i` and column `j`. Figure 17.8a shows the contents of array `queens` for the chessboard in Figure 17.8b.

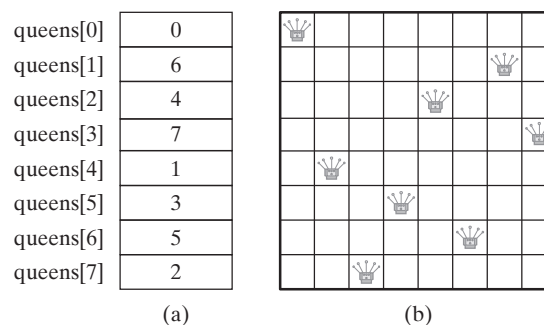


FIGURE 17.8 `queens[i]` denotes the position of the queen in row `i`.

Listing 17.8 is a program that finds a solution for the Eight Queens problem.

LISTING 17.8 EightQueen.cpp

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

```

3
4  const int NUMBER_OF_QUEENS = 8; // Constant: eight queens
5  int queens[NUMBER_OF_QUEENS];
6
7  // Check whether a queen can be placed at row i and column j
8  bool isValid(int row, int column)
9  {
10     for (int i = 1; i <= row; i++)
11         if (queens[row - i] == column // Check column
12             || queens[row - i] == column - i // Check upper left diagonal
13             || queens[row - i] == column + i) // Check upper right diagonal
14             return false; // There is a conflict
15     return true; // No conflict
16 }
17
18 // Display the chessboard with eight queens
19 void printResult()
20 {
21     cout << "\n-----\n";
22     for (int row = 0; row < NUMBER_OF_QUEENS; row++)
23     {
24         for (int column = 0; column < NUMBER_OF_QUEENS; column++)
25             printf(column == queens[row] ? "| Q " : "| ");
26         cout << "\n-----\n";
27     }
28 }
29
30 // Search to place a queen at the specified row
31 bool search(int row)
32 {
33     if (row == NUMBER_OF_QUEENS) // Stopping condition
34         return true; // A solution found to place 8 queens in 8 rows
35
36     for (int column = 0; column < NUMBER_OF_QUEENS; column++)
37     {
38         queens[row] = column; // Place a queen at (row, column)
39         if (isValid(row, column) && search(row + 1))
40             return true; // Found, thus return true to exit for loop
41     }
42
43     // No solution for a queen placed at any column of this row
44     return false;
45 }
46
47 int main()
48 {
49     search(0); // Start search from row 0. Note row indices are 0 to 7
50     printResult(); // Display result
51
52     return 0;
53 }

```

check if valid

search this row

search columns

search next row found

not found

Q								
				Q				
							Q	



(continued)

						Q			

			Q						

							Q		

		Q							

				Q					

The program invokes `search(0)` (line 49) to start a search for a solution at row `0`, which recursively invokes `search(1)`, `search(2)`, . . . , and `search(7)` (line 39).

The recursive `search(row)` function returns `true` if all rows are filled (lines 39–40). The function checks whether a queen can be placed in column `0`, `1`, `2`, . . . , and `7` in a `for` loop (line 36). Place a queen in the column (line 38). If the placement is valid, recursively search for the next row by invoking `search(row + 1)` (line 39). If search is successful, return `true` (line 40) to exit the `for` loop. In this case, there is no need to look for the next column in the row. If there is no solution for a queen to be placed on any column of this row, the function returns `false` (line 44).

Suppose you invoke `search(row)` for `row` is `3`, as shown in Figure 17.9a. The function tries to fill in a queen in column `0`, `1`, `2`, and so on in this order. For each trial, the `isValid(row, column)` function (line 39) is called to check whether placing a queen at the specified position causes a conflict with the queens placed before this row. It ensures that no queen is placed in the same column (line 11), no queen is placed in the upper left diagonal (line 12), and no queen is placed in the upper right diagonal (line 13), as shown in Figure 17.9a. If `isValid(row, column)` returns `false`, check the next column, as shown Figure 17.9b. If `isValid(row, column)` returns `true`, recursively invoke `search(row + 1)`, as shown in Figure 17.9d. If `search(row + 1)` returns `false`, check the next column on the preceding row, as shown Figure 17.9c.

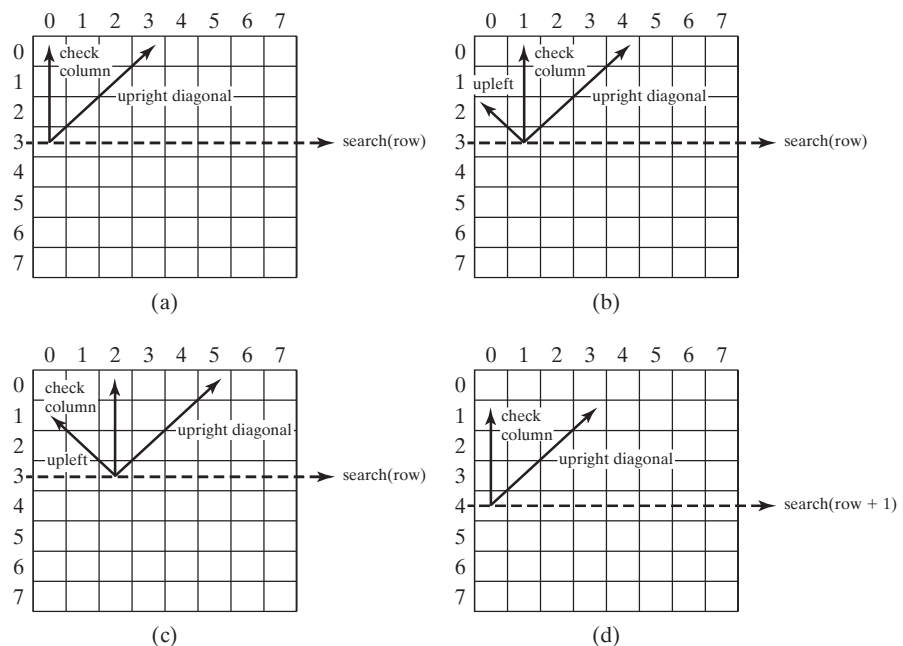


FIGURE 17.9 Invoking `search(row)` fills in a queen in a column on the row.

17.8 Recursion versus Iteration

Recursion is an alternative form of program control. It is essentially repetition without a loop.



Recursion is an alternative form of program control. It is essentially repetition without a loop control. When you use loops, you specify a loop body. The repetition of the loop body is controlled by the loop-control structure. In recursion, the function itself is called repeatedly. A selection statement must be used to control whether to call the function recursively or not.

Recursion bears substantial overhead. Each time the program calls a function, the system must assign space for all of the function’s local variables and parameters. This can consume considerable memory and requires extra time to manage the additional space.

Any problem that can be solved recursively can be solved nonrecursively with iterations. Recursion has some negative aspects: It uses too much time and too much memory. Why, then, should you use it? In some cases, using recursion enables you to specify a clear, simple solution for an inherent recursive problem that would otherwise be difficult to obtain. The Towers of Hanoi problem is such an example, which is rather difficult to solve without using recursion.

The decision whether to use recursion or iteration should be based on the nature of the problem you are trying to solve and your understanding of it. The rule of thumb is to use whichever of the two approaches can best develop an intuitive solution that naturally mirrors the problem. If an iterative solution is obvious, use it. It generally will be more efficient than the recursive option.



Note
Your recursive program could run out of memory, causing a *stack overflow* runtime error.



Tip
If you are concerned about your program’s performance, avoid using recursion, because it takes more time and consumes more memory than iteration.

recursion overhead

recursion advantages

recursion or iteration?

stack overflow

performance concern

17.9 Tail Recursion

A tail recursive function is efficient for reducing stack space.



A recursive function is said to be *tail recursive* if there are no pending operations to be performed on return from a recursive call, as illustrated in Figure 17.10a. However, function B in Figure 17.10b is not tail recursive because there are pending operations after a function call is returned.

tail recursion

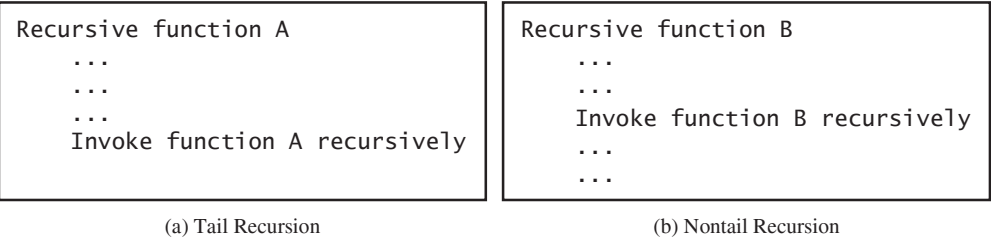


FIGURE 17.10 A tail-recursive function has no pending operations after a recursive call.

For example, the recursive **isPalindrome** function (lines 5–13) in Listing 17.4 is tail recursive because there are no pending operations after recursively invoking **isPalindrome** in

line 12. However, the recursive **factorial** function (lines 21–27) in Listing 17.1 is not tail recursive, because there is a pending operation, namely multiplication, to be performed on return from each recursive call.

Tail recursion is desirable, because the function ends when the last recursive call ends. So there is no need to store the intermediate calls in the stack. Some compilers can optimize tail recursion to reduce stack space.

A nontail-recursive function can often be converted to a tail-recursive function by using auxiliary parameters. These parameters are used to contain the result. The idea is to incorporate the pending operations into the auxiliary parameters in such a way that the recursive call no longer has a pending operation. You may define a new auxiliary recursive function with the auxiliary parameters. This function may overload the original function with the same name but a different signature. For example, the **factorial** function in Listing 17.1 can be written in a tail-recursive way as follows:

original function	1 // Return the factorial for a specified number 2 int factorial(int n) 3 {
invoke auxiliary function	4 return factorial(n, 1); // Call auxiliary function 5 } 6
auxiliary function	7 // Auxiliary tail-recursive function for factorial 8 int factorial(int n, int result) 9 { 10 if (n == 1) 11 return result; 12 else
recursive call	13 return factorial(n - 1, n * result); // Recursive call 14 }

The first **factorial** function simply invokes the second auxiliary function (line 4). The second function contains an auxiliary parameter **result** that stores the result for factorial of **n**. This function is invoked recursively in line 13. There is no pending operation after a call is returned. The final result is returned in line 11, which is also the return value from invoking **factorial(n, 1)** in line 4.



17.14 Which of the following statements are true?

- Any recursive function can be converted into a nonrecursive function.
- A recursive function takes more time and memory to execute than a nonrecursive function.
- Recursive functions are *always* simpler than nonrecursive functions.
- There is always a selection statement in a recursive function to check whether a base case is reached.

17.15 What is the cause for the stack overflow exception?

17.16 Identify tail-recursive functions in this chapter.

17.17 Rewrite the **fib** function in Listing 17.2 using tail recursion.

KEY TERMS

base case	646	recursive helper function	656
infinite recursion	649	stopping condition	646
recursive function	646	tail recursion	665

CHAPTER SUMMARY

1. A recursive function is one that invokes itself directly or indirectly. For a recursive function to terminate, there must be one or more base cases.
2. Recursion is an alternative form of program control. It is essentially repetition without a loop control. It can be used to write simple, clear solutions for inherently recursive problems that would otherwise be difficult to solve.
3. Sometimes the original function needs to be modified to receive additional parameters in order to be invoked recursively. A recursive helper function can be defined for this purpose.
4. Recursion bears substantial overhead. Each time the program calls a function, the system must assign space for all of the function's local variables and parameters. This can consume considerable memory and requires extra time to manage the additional space.
5. A recursive function is said to be *tail recursive* if there are no pending operations to be performed on return from a recursive call. Some compilers can optimize tail recursion to reduce stack space.

Quiz

Answer the quiz for this chapter online at www.cs.armstrong.edu/liang/cpp3e/quiz.html.

PROGRAMMING EXERCISES

Sections 17.2–17.3

17.1 (*Linear search*) Rewrite the linear search function in Listing 7.9 using recursion.

***17.2** (*Fibonacci numbers*) Rewrite the `fib` function in Listing 17.2 using iterations.

Hint: To compute `fib(n)` without recursion, you need to obtain `fib(n - 2)` and `fib(n - 1)` first. Let `f0` and `f1` denote the two previous Fibonacci numbers. The current Fibonacci number would then be `f0 + f1`. The algorithm can be described as follows:

```
f0 = 0; // For fib(0)
f1 = 1; // For fib(1)

for (int i = 2; i <= n; i++)
{
    currentFib = f0 + f1;
    f0 = f1;
    f1 = currentFib;
}

// After the loop, currentFib is fib(n)
```

Write a test program that prompts the user to enter an index and displays its Fibonacci number.

***17.3** (*Compute greatest common divisor using recursion*) The `gcd(m, n)` can also be defined recursively as follows:

- If `m % n` is 0, `gcd(m, n)` is `n`.
- Otherwise, `gcd(m, n)` is `gcd(n, m % n)`.



VideoNote
The GCD problem

Write a recursive function to find the GCD. Write a test program that prompts the user to enter two integers and displays their GCD.

- 17.4** (*Sum series*) Write a recursive function to compute the following series:

$$f(n) = 1 + \frac{1}{4} + \frac{1}{9} + \dots + \frac{1}{n^2}$$

Write a test program that displays **f(n)** for **n = 1, 2, ..., 15**.

- 17.5** (*Sum series*) Write a recursive function to compute the following series:

$$f(n) = \frac{1}{3} + \frac{1}{8} + \frac{1}{15} + \dots + \frac{1}{n(n+2)}$$

Write a test program that displays **f(n)** for **n = 1, 2, ..., 15**.

- **17.6** (*Sum series*) Write a recursive function to compute the following series:

$$f(n) = 1 + \frac{3}{4} + \frac{3}{5} + \frac{1}{2} + \dots + \frac{3}{n+2}$$

Write a test program that displays **f(n)** for **n = 1, 2, ..., 15**.

- *17.7** (*Palindrome string*) Modify Listing 17.3, **RecursivePalindrome.cpp**, so that the program finds the number of times the **isPalindrome** function is called. (*Hint*: Use a global variable and increment it every time the function is called.)

Section 17.4

- **17.8** (*Count even and odd digits*) Write a recursive function that displays the number of even and odd digits in an integer using the following header:

```
void evenAndOddCount(int value)
```

Write a test program that prompts the user to enter an integer and displays the number of even and odd digits in it.

- **17.9** (*Print the characters in a string reversely*) Write a recursive function that displays a string reversely on the console using the following header:

```
void reverseDisplay(const string& s)
```

For example, **reverseDisplay("abcd")** displays **dcba**. Write a test program that prompts the user to enter a string and displays its reversal.

- *17.10** (*Occurrences of a specified character in a string*) Write a recursive function that finds the number of occurrences of a specified letter in a string using the following function header.

```
int count(const string& s, char a)
```

For example, **count("Welcome", 'e')** returns **2**. Write a test program that prompts the user to enter a string and a character, and displays the number of occurrences for the character in the string.

- **17.11** (*Product of digits in an integer using recursion*) Write a recursive function that computes the product of the digits in an integer. Use the following function header:

```
int productDigits(int n)
```

For example, **productDigits(912)** returns **9 * 1 * 2 = 18**. Write a test program that prompts the user to enter an integer and displays the product of digits.



VideoNote
Count occurrence

Section 17.5

- **17.12** (*Print the characters in a string reversely*) Rewrite Programming Exercise 17.9 using a helper function to pass the substring high index to the function. The helper function header is as follows:

```
void reverseDisplay(const string& s, int high)
```

- **17.13** (*Find the smallest number in an array*) Write a recursive function that returns the smallest integer in an array. Write a test program that prompts the user to enter a list of five integers and displays the smallest integer.

- *17.14** (*Find the number of lowercase letters in a string*) Write a recursive function to return the number of lowercase letters in a string. You need to define the following two functions. The second one is a recursive helper function.

```
int getNumberOfLowercaseLetters(const string& s)
int getNumberOfLowercaseLetters(const string& s, int low)
```

Write a test program that prompts the user to enter a string and displays the number of lowercase letters in the string.

- *17.15** (*Occurrences of the space character in a string*) Write a recursive function to return the total number of space characters in a string. You need to define the following two functions. The second one is a recursive helper function.

```
int numberOfSpaces(const string& s)
int numberOfSpaces(const string& s, int i)
```

Write a test program that prompts the user to enter a string, invokes the function, and displays the number of spaces in the string.

Section 17.6

- *17.16** (*Towers of Hanoi*) Modify Listing 17.7, TowersOfHanoi.cpp, so that the program finds the number of moves needed to move n disks from tower A to tower B. (*Hint*: Use a global variable and increment it every time the function is called.)

Comprehensive

- ***17.17** (*String permutations*) Write a recursive function to print all permutations of a string. For example, for a string `abc`, the permutation is

```
abc
acb
bac
bca
cab
cba
```

(*Hint*: Define the following two functions. The second is a helper function.)

```
void displayPermutation(const string& s)
void displayPermutation(const string& s1, const string& s2)
```

The first function simply invokes `displayPermutation("", s)`. The second function uses a loop to move a character from `s2` to `s1` and recursively invoke

it with a new `s1` and `s2`. The base case is that `s2` is empty and prints `s1` to the console.

Write a test program that prompts the user to enter a string and displays all its permutations.

*****17.18** (*Game: Sudoku*) Supplement VI.A gives a program to find a solution for a Sudoku problem. Rewrite it using recursion.

*****17.19** (*Game: multiple Eight Queens solutions*) Rewrite Listing 17.8 using recursion.

*****17.20** (*Game: multiple Sudoku solutions*) Modify Programming Exercise 17.18 to display all possible solutions for a Sudoku puzzle.

***17.21** (*Decimal to binary*) Write a recursive function that converts a decimal number into a binary number as a string. The function header is:

```
string decimalToBinary(int value)
```

Write a test program that prompts the user to enter a decimal number and displays its binary equivalent.

***17.22** (*Decimal to hex*) Write a recursive function that converts a decimal number into a hex number as a string. The function header is:

```
string decimalToHex(int value)
```

Write a test program that prompts the user to enter a decimal number and displays its hex equivalent.

***17.23** (*Binary to decimal*) Write a recursive function that parses a binary number as a string into a decimal integer. The function header is:

```
int binaryToDecimal(const string& binaryString)
```

Write a test program that prompts the user to enter a binary string and displays its decimal equivalent.

***17.24** (*Hex to decimal*) Write a recursive function that parses a hex number as a string into a decimal integer. The function header is:

```
int hexToDecimal(const string& hexString)
```

Write a test program that prompts the user to enter a hex string and displays its decimal equivalent.