CHAPTER 8

Recursion

Objectives

- To know what is a recursive function and the benefits of using recursive functions (§8.1).
- To determine the base cases in a recursive function (§§8.2-8.6).
- To understand how recursive function calls are handled in a call stack (§§8.2-8.6).
- To solve problems using recursion (§§8.2-8.6).
- To use an overloaded helper function to derive a recursive function (§8.5).
- To understand the relationship and difference between recursion and iteration (§8.7).

8.1 Introduction

A recursive function is a function that invokes itself directly or indirectly. Recursion is a useful programming technique. In some cases, using recursion enables you to develop a natural, straightforward, simple solution to a problem that would otherwise be difficult to solve. This chapter introduces the concepts and techniques of recursive programming and demonstrates how to “think recursively” using examples.

8.2 Example: Factorials

Many mathematical functions are defined using recursion. The factorial of a number \( n \) can be recursively defined as follows:

\[
\begin{align*}
0! &= 1; \\
n! &= n \times (n - 1)!; & n > 0
\end{align*}
\]

How do you find \( n! \) for a given \( n \)? It is easy to find 1! because you know 0! and 1! is 1 × 0!. Assuming that you know \( (n-1)! \), \( n! \) can be obtained immediately using \( n \times (n-1)! \). 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.

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

```cpp
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. When it reaches a stopping case, 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 with the result of factorial(n-1).

Listing 8.1 gives a complete program that prompts the user to enter a non-negative integer and displays the factorial for the number.

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

// Return the factorial for a specified number
int factorial(int); 

int main()
{
    // Prompt the user to enter an integer
    cout << "Please enter a non-negative integer: ";
    int n;
    cin >> n;

    // Display factorial
    cout << "Factorial of " << n << " is " << factorial(n);

    return 0;
}
```

Listing 8.1 ComputeFactorial.cpp (Computing Factorial)

***PD: Please add line numbers in the following code***

<Side Remark line 23: base case>
<Side Remark line 26: recursion>
// Return the factorial for a specified number
int factorial(int n)
{
    if (n == 0) // Base case
        return 1;
    else
        return n * factorial(n - 1); // Recursive call
}

<Output>
Please enter a non-negative integer: 5
Factorial of 5 is 120
<End Output>

The factorial function (lines 19–24) 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.

Figure 8.1 illustrates the execution of the recursive calls, starting with n=4. The use of stack space for recursive calls is shown in Figure 8.2.

---

**Figure 8.1**

Invoking factorial(4) spawns recursive calls to factorial.

***Same as Fig4.19 in intro5e p152***
When factorial(4) is being executed, the factorial function is called recursively, causing memory space to dynamically change.

**CAUTION**

*Side Remark: infinite recursion*

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

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

The function runs infinitely and causes the stack overflow.

***End of CAUTION***

**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.

### 8.3 Example: Fibonacci Numbers

The factorial function in the preceding section could easily 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 recursively defined as follows:

\[
\begin{align*}
\text{fib}(0) &= 0; \\
\text{fib}(1) &= 1; \\
\text{fib}(\text{index}) &= \text{fib}(\text{index} - 2) + \text{fib}(\text{index} - 1); \text{ index} > 2
\end{align*}
\]

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

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

\[
\begin{align*}
\text{if} \ (\text{index} == 0) \\
& \quad \text{return} \ 0; \\
\text{else if} \ (\text{index} == 1) \\
& \quad \text{return} \ 1; \\
\text{else} \\
& \quad \text{return} \ \text{fib}(\text{index} - 1) + \text{fib}(\text{index} - 2);
\end{align*}
\]

Listing 8.2 gives a complete program that prompts the user to enter an index and computes the Fibonacci number for the index.

Listing 8.2 ComputeFibonacci.cpp (Finding Fibonacci Number)

***PD: Please add line numbers in the following code***

#include <iostream>
using namespace std;

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

Listing 8.2 ComputeFibonacci.cpp (Finding Fibonacci Number)

***PD: Please add line numbers in the following code***

#include <iostream>
using namespace std;
// The function for finding the Fibonacci number
int fib(int);

int main()
{
    // Prompt the user to enter an integer
    cout << "Enter an index for the Fibonacci number: ";
    int index;
    cin >> index;
    // Display factorial
    cout << "Fibonacci number at index " << index << " is 
        " << fib(index) << endl;
    return 0;
}

// The function for finding the Fibonacci number
int fib(int index)
{
    if (index == 0) // Base case
        return 0;
    else if (index == 1) // Base case
        return 1;
    else // Reduction and recursive calls
        return fib(index - 1) + fib(index - 2);
}

<Output>
Enter an index for the Fibonacci number: 7
Fibonacci number at index 7 is 13

<End Output>

The program does not show the considerable amount of work done behind the scenes by the computer. Figure 8.3, 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? Operands for the binary + operator are evaluated from left to right. The labels in Figure 8.3 show the order in which functions are called.

Figure 8.3

Invoking fib(4) spawns recursive calls to fib.

As shown in Figure 8.3, there are many duplicated recursive calls. For instance, fib(2) is called twice, fib(1) is
called three times, and \( \text{fib}(0) \) is called twice. In general, computing \( \text{fib}(\text{index}) \) requires twice as many recursive calls as are needed for computing \( \text{fib}(\text{index} - 1) \). As you try larger index values, the number of calls substantially increases.

Besides the large number of recursive calls, the computer requires more time and space to run recursive functions.

**Pedagogical NOTE**

The recursive implementation of the \( \text{fib} \) function is very simple and straightforward, but not efficient. See Exercise 8.2 for an efficient solution using loops. The recursive \( \text{fib} \) function is a good example to demonstrate how to write recursive functions, though it is not practical.

### 8.4 Problem Solving Using Recursion

**<Side Remark: recursion characteristics>**

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

**<Side Remark: if-else>**

- The function is implemented using an `if-else` or a `switch` statement that leads to different cases.

**<Side Remark: base cases>**

- One or more base cases (the simplest case) are used to stop recursion.

**<Side Remark: reduction>**

- Every recursive call reduces the original problem, bringing it increasingly closer to a base case until it becomes that case.

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.

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 for \( n-1 \) 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(char * message, int times)
{
    if (times >= 1) {
        cout << message << endl;
        nPrintln(message, times - 1);
    }
}

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

Many of the problems presented in the early chapters can be solved using recursion if you think recursively. Consider the palindrome problem in Listing 7.16. Recall that a string is palindrome if it reads the same from left and from the right. For example, mom and dad are palindrome, but uncle and aunt are not. The problem to check whether a string is 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 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 8.3.

Listing 8.3 RecursivePalindrome1.cpp (Recursive Palindrome Function)

***PD: Please add line numbers in the following code***

bool isPalindrome(const char * const s)
{
    if (strlen(s) <= 1) // Base case
        return true;
    else if (s[0] != s[strlen(s) - 1]) // Base case
        return false;
    else
        return isPalindrome(substring(s, 1, strlen(s) - 2));
int main()
{
    cout << "Enter a string: ";
    char s[80];
    cin.getline(s, 80);

    if (isPalindrome(s))
        cout << s << " is a palindrome" << endl;
    else
        cout << s << " is not a palindrome" << endl;

    return 0;
}

Enter a string: aba
aba is a palindrome

Enter a string: abab
abab is not a palindrome

Line 3 includes the header file for the substring function, introduced in Listing 7.14. The 
strlen(s) function returns the length of string. This function was introduced in §7.9.4, “String Functions.”

Invoking substring(s, 1, strlen(s) – 2) returns a new string that is a substring of s from index 1 to index strlen(s) – 2.

8.5 Recursive Helper Functions

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 char * const s), you have to create a new function isPalindrome(const char * const s, int low, int high) to accept additional information on the string, as shown in Listing 8.4.

Listing 8.4 RecursivePalindrome2.cpp (Using a Helper Function)

***PD: Please add line numbers in the following code***
```cpp
#include <iostream>
#include <cstring>
using namespace std;

bool isPalindrome(const char * const s, int low, int high)
{
    if (high <= low) // Base case
        return true;
    else if (s[low] != s[high]) // Base case
        return false;
    else
        return isPalindrome(s, low + 1, high - 1);
}

bool isPalindrome(const char * const s)
{
    return isPalindrome(s, 0, strlen(s) - 1);
}

int main()
{
    cout << "Enter a string: ";
    char s[80];
    cin.getline(s, 80);
    if (isPalindrome(s))
        cout << s << " is a palindrome" << endl;
    else
        cout << s << " is not a palindrome" << endl;
    return 0;
}
```

Enter a string: aba
aba is a palindrome

Enter a string: abab
abab is not a palindrome

Two overloaded isPalindromes functions are declared. The function isPalindromes(char * s) (line 15) checks whether a string is a palindrome and the second function isPalindromes(char * s, int low, int high) (line 5) checks whether a substring \texttt{s(low..high)} is a palindrome. The first function passes the string \texttt{s with low = 0 and high = strlen(s) - 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 declare a second function that
receives additional parameters. Such a function is known as a recursive helper function.

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

8.5.1 Selection Sort

Selection sort was introduced in §6.6.1. This section introduces a recursive selection sort for characters in a string. Recall that selection sort 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 swaps 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 8.5 gives the recursive sort function.

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

void sort(char list[], int high)
{
    if (high > 1)
    {
        // Find the largest element and its index
        int indexOfMax = 0;
        char max = list[0];
        for (int i = 1; i <= high; i++)
        {
            if (list[i] > max)
                {
                    max = list[i];
                    indexOfMax = i;
                }
        }
        // Swap the largest with the last element in the list
    }
}
```

list[indexOfMax] = list[high];
list[high] = max;

// Sort the remaining list
sort(list, high - 1);
}

void sort(char list[], int high)
{
    if (high > 0)
    {
        int indexMax = indexOfMax(list, 0, high);
        list[indexOfMax] = list[high];
        list[high] = max;

        // Sort the remaining list
        sort(list, high - 1);
    }
}

int main()
{
    cout << "Enter a string: ";
    char s[80];
    cin.getline(s, 80);

    sort(s);

    cout << "The sorted string is " << s << endl;

    return 0;
}

<Output>
Enter a string: ghfdacb
The sorted string is abcdfgh
<End Output>

Two overloaded sort functions are declared. The function sort(char list[]) sorts characters in list[0..strlen(list)-1] and the second function sort(double list[], int high) sorts characters in list[0..high]. The helper function can be invoked recursively to sort an ever-shrinking subarray.

8.5.2 Binary Search

Binary search was introduced in §6.5.2. 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 three 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 in 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 8.6 gives a clear, simple solution for the binary search problem using recursion.

Listing 8.6 RecursiveBinarySearch.cpp (Recursive Binary Search Function)

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

int binarySearch(const int list[], int key, int low, int high)
{
    if (low > high)  // The list has been exhausted without a match
        return -low - 1; // Return -insertion point - 1

    int mid = (low + high) / 2;
    if (key < list[mid])
        return binarySearch(list, key, low, mid - 1);
    else if (key == list[mid])
        return mid;
    else
        return binarySearch(list, key, mid + 1, high);
}

int binarySearch(const int list[], int key, int size)
{
    int low = 0;
    int high = size - 1;
    return binarySearch(list, key, low, high);
}

int main()
{
    int list[] = {2, 4, 7, 10, 11, 45, 50, 59, 60, 66, 69, 70, 79};
    int i = binarySearch(list, 2, 13); // returns 0
    int j = binarySearch(list, 11, 13); // returns 4
    int k = binarySearch(list, 12, 13); // returns -6

    cout << "binarySearch(list, 2, 13) returns " << i << endl;
    cout << "binarySearch(list, 11, 13) returns " << j << endl;
    cout << "binarySearch(list, 12, 13) returns " << k << endl;
}
```

return 0;

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.

8.6 Tower of Hanoi

The Tower of Hanoi problem is another classic recursion example. The problem can be solved easily using recursion, but is difficult to solve without using recursion.

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, \ldots, \( n \), 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 of the problem is to move all the disks from A to B with the assistance of C. For example, if you have three disks, as shown in Figure 8.4, the following steps will move all of the disks from A to B:

1. Move disk 1 from A to B.
2. Move disk 2 from A to C.
3. Move disk 1 from B to C.
4. Move disk 3 from A to B.
5. Move disk 1 from C to A.
6. Move disk 2 from C to B.
7. Move disk 1 from A to B.

**Figure 8.4**

*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. There are many Web sites devoted to this problem. The Web site [www.cut-the-knot.com/recurrence/hanoi.html](http://www.cut-the-knot.com/recurrence/hanoi.html) is worth seeing.

In the case of three disks, you can find the solution manually. However, the problem is quite complex for a larger number of disks—even for four. 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 with the assistance of tower B.

2. Move disk \( n \) from A to B.

3. Move \( n - 1 \) disks from C to B with the assistance of tower A.

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

```cpp
void moveDisks(int n, char fromTower, char toTower, char auxTower)
{
    if (n == 1) // Stopping condition
        cout << "Move disk " << n << " from " << fromTower << " to " << toTower << endl;
    else
    {
        moveDisks(n - 1, fromTower, auxTower, toTower);
        cout << "Move disk " << n << " from " << fromTower << " to " << toTower << endl;
        moveDisks(n - 1, auxTower, toTower, fromTower);
    }
}
```

Listing 8.7 gives a program that prompts the user to enter the number of disks and invokes the recursive function moveDisks to display the solution for moving the disks.
// Read number of disks, n
cout << "Enter number of disks: ";
int n;
 cin >> n;

// Find the solution recursively
cout << "The moves are: " << endl;
movesDisks(n, 'A', 'B', 'C');
return 0;

<Output>
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
<End Output>

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 8.5. 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 8.5
8.7 Recursion versus Iteration

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 many negative aspects: it uses up 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 that would otherwise be difficult to obtain.

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 the problem. 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 will generally be more efficient than the recursive option.

**NOTE**

<i>Side Remark: stack overflow</i>
Your recursive program could run out of memory, causing a stack overflow runtime error.

**TIP**

<i>Side Remark: performance concern</i>
If you are concerned about your program’s performance, avoid using recursion, because it takes more time and consumes more memory than iteration.

**Key Terms**
• **base case** 149
• **infinite recursion** 126
• **recursive function** 149
• **recursive helper function** 126
• **stopping condition** 149

**Chapter Summary**

• A recursive function is a function that invokes itself directly or indirectly. For a recursive function to terminate, there must be one or more base cases.

• Recursion is an alternative form of program control. It is essentially repetition without a loop control. It can be used to specify simple, clear solutions for inherently recursive problems that would otherwise be difficult to solve.

• Sometimes the original function needs to be modified to receive additional parameters in order to be invoked recursively. A recursive helper function can be declared for this purpose.

• 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.

**Review Questions**

*Sections 8.1-8.3*

8.1
What is a recursive function? Describe the characteristics of recursive functions. What is an infinite recursion?

8.2
Write a recursive mathematical definition for computing \(2^n\) for a positive integer \(n\).

8.3
Write a recursive mathematical definition for computing \(x^n\) for a positive integer \(n\) and a real number \(x\).

8.4
Write a recursive mathematical definition for computing \(1+2+3+\ldots+n\) for a positive integer.

8.5
How many times the factorial function in Listing 8.1 is invoked for \texttt{factorial(6)}?

8.6
How many times the \texttt{fib} function in Listing 8.2 is invoked for \texttt{fib(6)}?

8.7
Show the output of the following program:

```cpp
#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;
|
```

8.8
Show the output of the following two programs:

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

void f(int n)
|
|   if (n > 0)
|       cout << n % 10;
|   f(n / 10);
|
int main()
|
|   f(1234567);
|
```

8.9
What is wrong in the following function?

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

void f(double n)
|
|   if (n != 0)
|       cout << n;
|   f(n / 10);
|
int main()
|
|   f(1234567);
|
```
Show the call stack for \texttt{isPalindrome("abcba")} using the functions declared in Listing 8.3 and Listing 8.4, respectively.

8.11
Show the call stack for \texttt{selectionSort("abcba")} using the function declared in Listing 8.5.

8.12
What is a recursive helper function?

Section 8.6 Towers of Hanoi
8.13
How many times the \texttt{moveDisks} function in Listing 8.7 is invoked for \texttt{moveDisks(5, 'A', 'B', 'C')}?

Section 8.8 Recursion versus Iteration
8.14
Which of the following statements are true?
- Any recursive functions can be converted into a non-recursive function.
- Recursive function takes more time and memory to execute than non-recursive functions.
- Recursive functions are always simpler than non-recursive functions.
- There is always a condition statement in a recursive function to check whether a base case is reached.

8.15
What is the cause for the stack overflow exception?

Programming Exercises
Sections 8.2-8.3
8.1
\textit{(Computing factorials)} Rewrite the \texttt{factorial} function in Listing 8.1 using iterations.

8.2*
\textit{(Fibonacci numbers)} Rewrite the \texttt{fib} function in Listing 8.2 using iterations.

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

\begin{verbatim}
f0 = 0; // For fib(0)
f1 = 1; // For fib(1)
for (int i = 1; i <= n; i++)
\end{verbatim}
8.3*  
(Computing 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 \( \gcd(n, m \% n) \).

Write a recursive function to find the GCD. Write a test program that computes \( \gcd(24, 16) \) and \( \gcd(255, 25) \).

8.4  
(Summing series) Write a recursive function to compute the following series:

\[
m(i) = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{i}
\]

8.5  
(Summing series) Write a recursive function to compute the following series:

\[
m(i) = \frac{1}{3} + \frac{2}{5} + \frac{3}{7} + \frac{4}{9} + \frac{5}{11} + \frac{6}{13} + \ldots + \frac{i}{2i+1}
\]

8.6**  
(Summing the series) Write a recursive function to compute the following series:

\[
m(i) = \frac{1}{2} + \frac{2}{3} + \ldots + \frac{i}{i+1}
\]

8.7*  
(Fibonacci series) Modify Listing 8.2, ComputeFibonacci.cpp, so that the program finds the number of times the \( \text{fib} \) function is called. (Hint: Use a global variable and increment it every time the function is called.)

Section 8.4 Problem Solving Using Recursion

8.8**  
(Printing the digits in an integer reversely) Write a recursive function that displays an int value reversely on the console using the following header:
void reverseDisplay(int value)

For example, reverseDisplay(12345) displays 54321.

8.9**
(Printing 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 char * const str)

For example, reverseDisplay("abcd") displays dcba.

8.10*
(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 char * const str, char a)

For example, count("Welcome", 'e') returns 2.

8.11**
(Summing the digits in an integer using recursion) Write a recursive function that computes the sum of the digits in an integer. Use the following function header:

int sumDigits(long n)

For example, sumDigits(234) returns 2 + 3 + 4 = 9.

Section 8.5 Recursion Helper Functions

8.12**
(Printing the characters in a string reversely) Rewrite Exercise 8.9 using a helper function to pass the substring high index to the function. The helper function header is:

void reverseDisplay(const char * const str, int high)

8.13**
(Finding the largest number in an array) Write a recursive function that returns the largest integer in an array.

8.14*
(Finding the number of uppercase letters in a string) Write a recursive function to return the number of uppercase letters in a string.

8.15*
(Occurrences of a specified character in a string) Rewrite Exercise 8.10 using a helper function to pass the substring high index to the function. The helper function header is:
8.16*
(Finding the number of uppercase letters in a string) Write a recursive function to return the number of uppercase letters in an array of characters. You need to declare the following two functions. The second one is a recursive helper function.

```c
int count(const char * const str, char a, int high)
```

8.17*
(Occurrences of a specified character in a string) Write a recursive function that finds the number of occurrences of a specified character in an array. You need to declare the following two functions. The second one is a recursive helper function.

```c
int count(const char * const str, char ch)
int count(const char * const str, char ch, int high)
```

Sections 8.6 Tower of Hanoi
8.18*
(Towers of Hanoi) Modify Listing 8.7, 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
8.21***
(String permutation) Write a recursive function to print all permutations of a string. For example, for a string \( \text{abc} \), the printout is

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

Hint: Declare the following two functions. The second is a helper function.

```c
void displayPermuatation(const char * const s)
void displayPermuatation(const char * const s1, const char * const s2)
```

The first function simply invokes `displayPermuatation("", 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.