CHAPTER 19

Recursion

TOPICS

19.1 Introduction to Recursion
19.2 Solving Problems with Recursion
19.3 Focus on Problem Solving and Program Design: The Recursive gcd Function
19.4 Focus on Problem Solving and Program Design: Solving Recursively Defined Problems
19.5 Focus on Problem Solving and Program Design: Recursive Linked List Operations
19.6 Focus on Problem Solving and Program Design: A Recursive Binary Search Function
19.7 The Towers of Hanoi
19.8 Focus on Problem Solving and Program Design: The QuickSort Algorithm
19.9 Exhaustive Algorithms
19.10 Focus on Software Engineering: Recursion vs. Iteration

19.1 Introduction to Recursion

CONCEPT: A recursive function is one that calls itself.

You have seen instances of functions calling other functions. Function A can call function B, which can then call function C. It's also possible for a function to call itself. A function that calls itself is a *recursive function*. Look at this `message` function:

```
void message()
{
     cout << "This is a recursive function.\n";
     message();
}
```

This function displays the string "This is a recursive function.\n", and then calls itself. Each time it calls itself, the cycle is repeated. Can you see a problem with the function? There's no way to stop the recursive calls. This function is like an infinite loop because there is no code to stop it from repeating.

NOTE: The function example `message` will eventually cause the program to crash. Do you remember learning in Chapter 18 that the system stores temporary data on a stack each time a function is called? Eventually, these recursive function calls will use up all available stack memory and cause it to overflow.

Like a loop, a recursive function must have some method to control the number of times it repeats. The following is a modification of the `message` function. It passes an integer argument, that holds the number of times the function is to call itself.

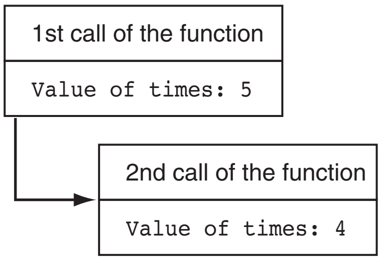
```
void message(int times)
{
    if (times > 0)
    {
        cout << "This is a recursive function.\n";
        message(times - 1);
    }
}
```

This function contains an `if` statement that controls the repetition. As long as the `times` argument is greater than zero, it will display the message and call itself again. Each time it calls itself, it passes `times - 1` as the argument. For example, let's say a program calls the function with the following statement:

```
message(5);
```

The argument, 5, will cause the function to call itself five times. The first time the function is called, the `if` statement will display the message and then call itself with 4 as the argument. Figure 19-1 illustrates this:

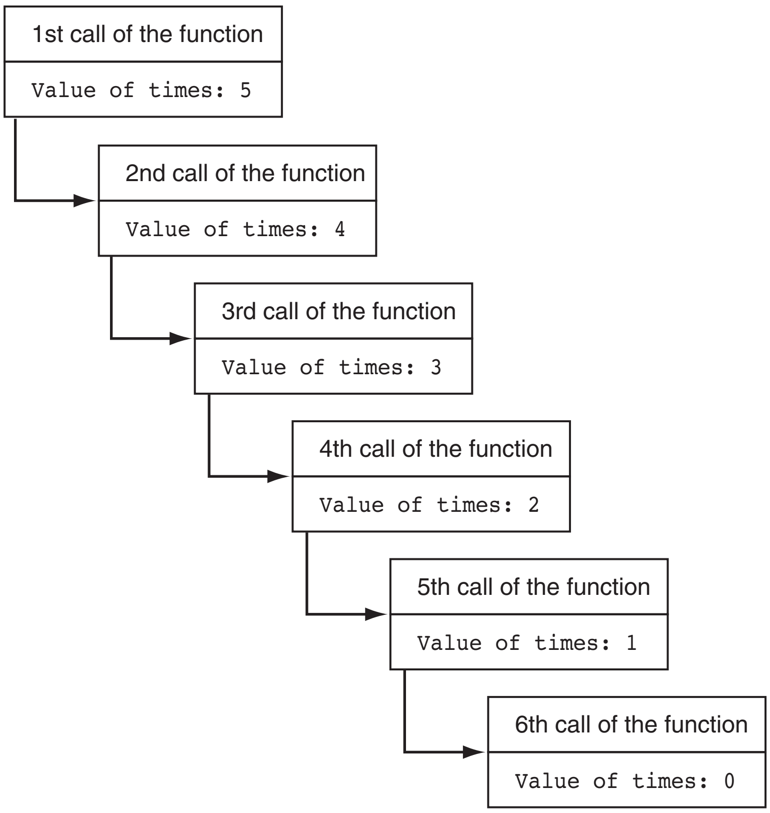
Figure 19-1



The diagram in Figure 19-1 illustrates two separate calls of the `message` function. Each time the function is called, a new instance of the `times` parameter is created in memory. The first time the function is called, the `times` parameter is set to 5. When the function calls itself, a new instance of `times` is created, and the value 4 is passed into it. This cycle repeats until, finally, zero is passed to the function. This is illustrated in Figure 19-2.

As you can see from Figure 19-2 the function is called a total of six times. The first time it is called from `main`, and the other five times it calls itself, so the *depth of recursion* is five. When the function reaches its sixth call, the `times` parameter will be set to 0. At that point, the `if` statement's conditional expression will be false, so the function will return. Control

Figure 19-2



of the program will return from the sixth instance of the function to the point in the fifth instance directly after the recursive function call:

```
void message (int times)
{
     if (times > 0)
     {
          cout << "This is a recursive function.\n"
          message (times - 1);    ←—— Recursive call
     }
}  ←—— Control returns here from the recursive call,
       causing the function to return.
```

Because there are no more statements to be executed after the function call, the fifth instance of the function returns control of the program back to the fourth instance. This repeats until all instances of the function return. Program 19-1 demonstrates the recursive `message` function.

Program 19-1

```
// This program demonstrates a simple recursive function.
#include <iostream>
using namespace std;

```

(program continues)

Program 19-1 *(continued)*

```
// Function prototype
void message(int);

int main()
{
    message(5);
    return 0;
}

//***********************************************************
// Definition of function Message. If the value in times is *
// greater than 0, the message is displayed and the         *
// function is recursively called with the argument         *
// times - 1.                                               *
//***********************************************************

void message(int times)
{
    if (times > 0)
    {
        cout << "This is a recursive function.\n";
        message(times - 1);
    }
}
```

Program Output

```
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
```

To further illustrate the inner workings of this recursive function, let's look at another version of the program. In Program 19-2, a message is displayed each time the function is entered, and another message is displayed just before the function returns.

Program 19-2

```
// This program demonstrates a simple recursive function.
#include <iostream>
using namespace std;

// Function prototype
void message(int);

int main()
{
    message(5);
    return 0;
}

```

```
//*************************************************************
// Definition of function message. If the value in times is  *
// greater than 0, the message is displayed and the function *
// is recursively called with the argument times - 1.        *
//*************************************************************

void message(int times)
{
    cout << "message called with " << times << " in times.\n";

    if (times > 0)
    {
        cout << "This is a recursive function.\n";
        message(times - 1);
    }

    cout << "message returning with " << times;
    cout << " in times.\n";
}
```

Program Output

```
message called with 5 in times.
This is a recursive function.
message called with 4 in times.
This is a recursive function.
message called with 3 in times.
This is a recursive function.
message called with 2 in times.
This is a recursive function.
message called with 1 in times.
This is a recursive function.
message called with 0 in times.
message returning with 0 in times.
message returning with 1 in times.
message returning with 2 in times.
message returning with 3 in times.
message returning with 4 in times.
message returning with 5 in times.
```

19.2 Solving Problems with Recursion

CONCEPT: A problem can be solved with recursion if it can be broken down into successive smaller problems that are identical to the overall problem.

Programs 19-1 and 19-2 in the previous section show simple demonstrations of *how* a recursive function works. But these examples don't show us *why* we would want to write a recursive function. Recursion can be a powerful tool for solving repetitive problems and is an important topic in upper-level computer science courses. What might not be clear to you yet is how to use recursion to solve a problem.

First, it should be noted that recursion is never absolutely required to solve a problem. Any problem that can be solved recursively can also be solved iteratively, with a loop. In fact, recursive algorithms are usually less efficient than iterative algorithms. This is because a function call requires several actions to be performed by the C++ runtime system. These actions include allocating memory for parameters and local variables and storing the address of the program location where control returns after the function terminates. These actions, which are sometimes referred to as *overhead*, take place with each function call. Such overhead is not necessary with a loop.

Some repetitive problems, however, are more easily solved with recursion than with iteration. Where an iterative algorithm might result in faster execution time, the programmer might be able to design a recursive algorithm faster.

In general, a recursive function works like this:

- If the problem can be solved now, without recursion, then the function solves it and returns.
- If the problem cannot be solved now, then the function reduces it to a smaller but similar problem and calls itself to solve the smaller problem.

VideoNote
Reducing a Problem with Recursion

In order to apply this approach, we first identify at least one case in which the problem can be solved without recursion. This is known as the *base case*. Second, we determine a way to solve the problem in all other circumstances using recursion. This is called the *recursive case*. In the recursive case, we must always reduce the problem to a smaller version of the original problem. By reducing the problem with each recursive call, the base case will eventually be reached and the recursion will stop.

Example: Using Recursion to Calculate the Factorial of a Number

Let's take an example from mathematics to examine an application of recursion. In mathematics, the notation $n!$ represents the factorial of the number n. The factorial of a non-negative number can be defined by the following rules:

If $n = 0$ then $\quad n! = 1$

If $n > 0$ then $\quad n! = 1 \times 2 \times 3 \times \ldots \times n$

Let's replace the notation $n!$ with factorial(n), which looks a bit more like computer code, and rewrite these rules as

If $n = 0$ then $\quad \text{factorial}(n) = 1$

If $n > 0$ then $\quad \text{factorial}(n) = 1 \times 2 \times 3 \times \ldots \times n$

These rules state that when n is 0, its factorial is 1. When n is greater than 0, its factorial is the product of all the positive integers from 1 up to n. For instance, factorial(6) is calculated as $1 \times 2 \times 3 \times 4 \times 5 \times 6$.

When designing a recursive algorithm to calculate the factorial of any number, we first identify the base case, which is the part of the calculation that we can solve without recursion. That is the case where n is equal to 0:

If $n = 0$ then $\quad \text{factorial}(n) = 1$

This tells how to solve the problem when n is equal to 0, but what do we do when n is greater than 0? That is the recursive case, or the part of the problem that we use recursion to solve. This is how we express it:

If $n > 0$ then $\quad \text{factorial}(n) = n \times \text{factorial}(n - 1)$

This states that if n is greater than 0, the factorial of n is n times the factorial of $n - 1$. Notice how the recursive call works on a reduced version of the problem, $n - 1$. So, our recursive rule for calculating the factorial of a number might look like this:

If $n = 0$ then $\quad \text{factorial}(n) = 1$

If $n > 0$ then $\quad \text{factorial}(n) = n \times \text{factorial}(n - 1)$

The following pseudocode shows how we might implement the factorial algorithm as a recursive function:

```
factorial(n)
    If n is 0 then
        return 1.
    else
        return n times the factorial of n - 1.
end factorial.
```

Here is the C++ code for such a function:

```
int factorial(int n)
{
    if (n == 0)
        return 1;                      // Base case
    else
        return n * factorial(n - 1);   // Recursive case
}
```

Program 19-3 demonstrates the recursive factorial function.

Program 19-3

```
// This program demonstrates a recursive function to
// calculate the factorial of a number.
#include <iostream>
using namespace std;

// Function prototype
int factorial(int);

int main()
{
    int number;

```

(program continues)

Program 19-3 *(continued)*

```
13      // Get a number from the user.
14      cout << "Enter an integer value and I will display\n";
15      cout << "its factorial: ";
16      cin >> number;
17
18      // Display the factorial of the number.
19      cout << "The factorial of " << number << " is ";
20      cout << factorial(number) << endl;
21      return 0;
22  }
23
24  //*************************************************************
25  // Definition of factorial. A recursive function to calculate *
26  // the factorial of the parameter n.                          *
27  //*************************************************************
28
29  int factorial(int n)
30  {
31      if (n == 0)
32          return 1;                       // Base case
33      else
34          return n * factorial(n - 1); // Recursive case
35  }
```

Program Output with Example Input Shown in Bold

```
Enter an integer value and I will display
its factorial: 4 [Enter]
The factorial of 4 is 24
```

In the example run of the program, the `factorial` function is called with the argument 4 passed into `n`. Because `n` is not equal to 0, the `if` statement's `else` clause executes the following statement, in line 34:

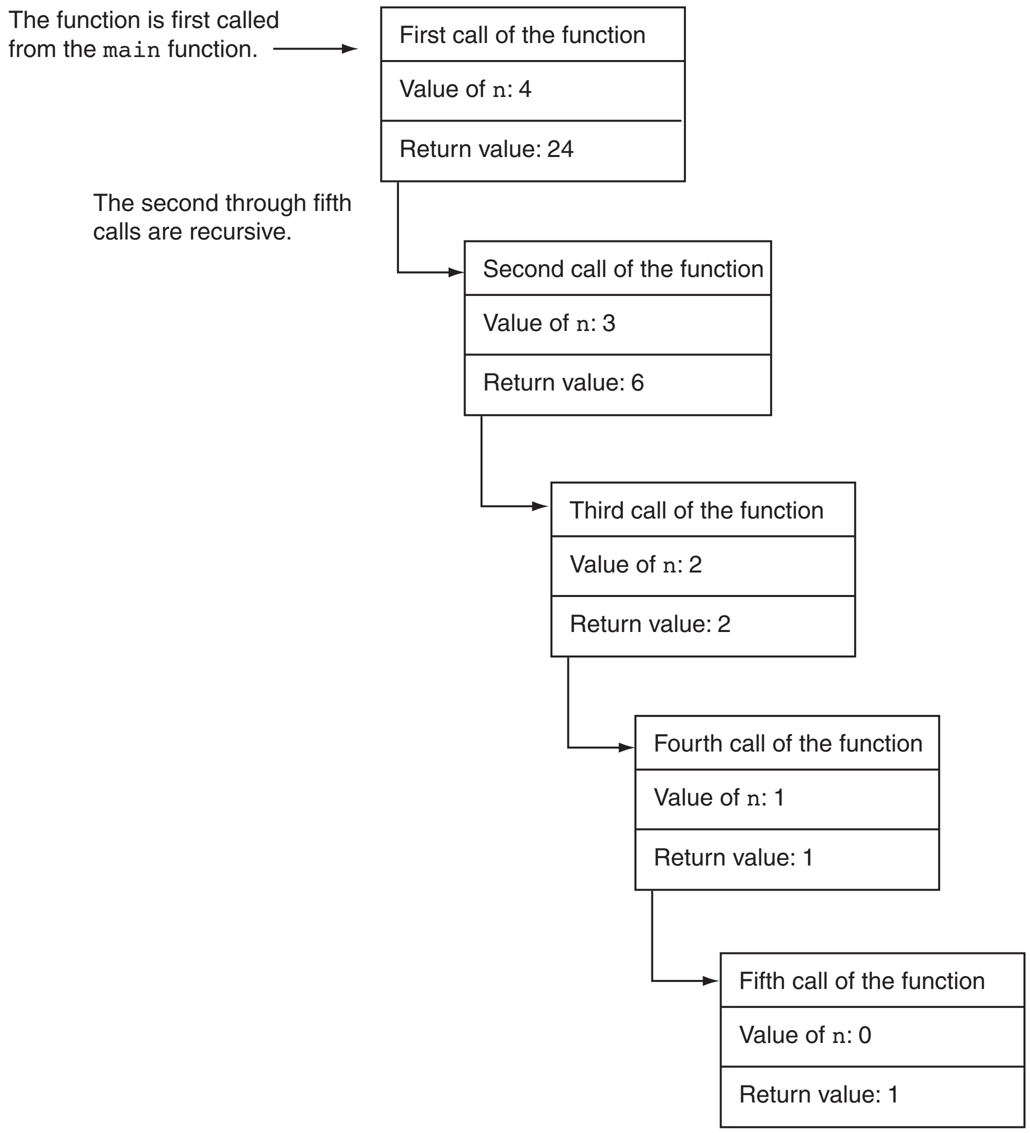
```
return n * factorial(n - 1);
```

Although this is a `return` statement, it does not immediately return. Before the return value can be determined, the value of `factorial(num - 1)` must be determined. The `factorial` function is called recursively until the fifth call, in which the `n` parameter will be set to zero. The diagram in Figure 19-3 illustrates the value of `n` and the return value during each call of the function.

This diagram illustrates why a recursive algorithm must reduce the problem with each recursive call. Eventually the recursion has to stop in order for a solution to be reached. If each recursive call works on a smaller version of the problem, then the recursive calls work toward the base case. The base case does not require recursion, so it stops the chain of recursive calls.

Usually, a problem is reduced by making the value of one or more parameters smaller with each recursive call. In our `factorial` function, the value of the parameter `n` gets closer to 0 with each recursive call. When the parameter reaches 0, the function returns a value without making another recursive call.

Figure 19-3


The function is first called from the main function.
The second through fifth calls are recursive.
First call of the function
Value of n: 4
Return value: 24
Second call of the function
Value of n: 3
Return value: 6
Third call of the function
Value of n: 2
Return value: 2
Fourth call of the function
Value of n: 1
Return value: 1
Fifth call of the function
Value of n: 0
Return value: 1

Example: Using Recursion to Count Characters

Let's look at another simple example of recursion. The following function counts the number of times a specific character appears in a string. The line numbers are from Program 19-4, which we will examine momentarily.

```
29  int numChars(char search, string str, int subscript)
30  {
31      if (subscript >= str.length())
32      {
33          // Base case: The end of the string is reached.
34          return 0;
35      }
36      else if (str[subscript] == search)
37      {
38          // Recursive case: A matching character was found.
39          // Return 1 plus the number of times the search
40          // character appears in the rest of the string.
41          return 1 + numChars(search, str, subscript+1);
42      }
43      else
44      {
45          // Recursive case: A character that does not match the
46          // search character was found. Return the number of times
47          // the search character appears in the rest of the string.
48          return numChars(search, str, subscript+1);
49      }
50  }
```

The function's parameters are

- `search`: The character to be searched for and counted
- `str`: a `string` object containing the string to be searched
- `subscript`: The starting subscript for the search

When this function examines a character in the string, three possibilities exist:

- The end of the string has been reached. This is the base case because there are no more characters to search.
- A character that matches the search character is found. This is a recursive case because we still have to search the rest of the string.
- A character that does not match the search character is found. This is also a recursive case because we still have to search the rest of the string.

Let's take a closer look at the code. The first `if` statement, in line 31, determines whether the end of the string has been reached:

```
if (subscript >= str.length())
```

Reaching the end of the string is the base case of the problem. If the end of the string has been reached, the function returns 0, indicating that 0 matching characters were found. Otherwise, the following `else if` clause, in lines 36 through 42, is executed:

```
else if (str[subscript] == search)
{
    // Recursive case: A matching character was found.
    // Return 1 plus the number of times the search
    // character appears in the rest of the string.
    return 1 + numChars(search, str, subscript+1);
}
```

If `str[subscript]` contains the search character, then we have found one matching character. But because we have not reached the end of the string, we must continue to search the rest of the string for more matching characters. So, at this point the function performs a recursive call. The `return` statement returns 1 plus the number of times the search character appears in the string, starting at `subscript+1`. In essence, this statement returns 1 plus the number of times the search character appears in the rest of the string.

Finally, if `str[subscript]` does not contain the search character, the following `else` clause in lines 43 through 49 is executed:

```
else
{
    // Recursive case: A character that does not match the
    // search character was found. Return the number of times
    // the search character appears in the rest of the string.
    return numChars(search, str, subscript+1);
}
```

The `return` statement in line 48 makes a recursive call to search the remainder of the string. In essence, this code returns the number of times the search character appears in the rest of the string. Program 19-4 demonstrates the function.

Program 19-4

```
 1  // This program demonstrates a recursive function for counting
 2  // the number of times a character appears in a string.
 3  #include <iostream>
 4  #include <string>
 5  using namespace std;
 6
 7  // Function prototype
 8  int numChars(char, string, int);
 9
10  int main()
11  {
12      string str = "abcddddef";
13
14      // Display the number of times the letter
15      // 'd' appears in the string.
16      cout << "The letter d appears "
17           << numChars('d', str, 0) << " times.\n";
18
19      return 0;
20  }
21
22  //*************************************************
23  // Function numChars. This recursive function    *
24  // counts the number of times the character      *
25  // search appears in the string str. The search *
26  // begins at the subscript stored in subscript. *
27  //*************************************************
28
29  int numChars(char search, string str, int subscript)
30  {
31      if (subscript >= str.length())
32      {
33          // Base case: The end of the string is reached.
34          return 0;
35      }
36      else if (str[subscript] == search)
```

(program continues)

Program 19-4 *(continued)*

```
      {
          // Recursive case: A matching character was found.
          // Return 1 plus the number of times the search
          // character appears in the rest of the string.
          return 1 + numChars(search, str, subscript+1);
      }
      else
      {
          // Recursive case: A character that does not match the
          // search character was found. Return the number of times
          // the search character appears in the rest of the string.
          return numChars(search, str, subscript+1);
      }
  }
```

Program Output

```
The letter d appears 4 times.
```

Direct and Indirect Recursion

The examples we have discussed so far show recursive functions that directly call themselves. This is known as *direct recursion*. There is also the possibility of creating *indirect recursion* in a program. This occurs when function A calls function B, which in turn calls function A. There can even be several functions involved in the recursion. For example, function A could call function B, which could call function C, which calls function A.

Checkpoint

19.1 What happens if a recursive function never returns?

19.2 What is a recursive function's base case?

19.3 What will the following program display?

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

// Function prototype
void showMe(int arg);

int main()
{
    int num = 0;
    showMe(num);
    return 0;
}

void showMe(int arg)
{
    if (arg < 10)
        showMe(++arg);
    else
        cout << arg << endl;
}
```

19.4 What is the difference between direct and indirect recursion?

19.3 Focus on Problem Solving and Program Design: The Recursive gcd Function

CONCEPT: The gcd function uses recursion to find the greatest common divisor (GCD) of two numbers.

Our next example of recursion is the calculation of the greatest common divisor, or GCD, of two numbers. Using Euclid's algorithm, the GCD of two positive integers, x and y, is:

gcd(x, y) = y;	if y divides x evenly
gcd(y, remainder of x/y);	otherwise

The definition above states that the GCD of x and y is y if x/y has no remainder. Otherwise, the answer is the GCD of y and the remainder of x/y. Program 19-5 shows the recursive C++ implementation:

Program 19-5

```
// This program demonstrates a recursive function to calculate
// the greatest common divisor (gcd) of two numbers.
#include <iostream>
using namespace std;

// Function prototype
int gcd(int, int);

int main()
{
    int num1, num2;

    // Get two numbers.
    cout << "Enter two integers: ";
    cin >> num1 >> num2;

    // Display the GCD of the numbers.
    cout << "The greatest common divisor of " << num1;
    cout << " and " << num2 << " is ";
    cout << gcd(num1, num2) << endl;
    return 0;
}

//***********************************************************
// Definition of gcd. This function uses recursion to      *
// calculate the greatest common divisor of two integers, *
// passed into the parameters x and y.                     *
//***********************************************************

int gcd(int x, int y)
```

(program continues)

Program 19-5 *(continued)*

```
{
    if (x % y == 0)
        return y;              // Base case
    else
        return gcd(y, x % y);  // Recursive case
}
```

Program Output with Example Input Shown in Bold

```
Enter two integers: 49 28 [Enter]
The greatest common divisor of 49 and 28 is 7
```

19.4 Focus on Problem Solving and Program Design: Solving Recursively Defined Problems

CONCEPT: Some mathematical problems are designed for a recursive solution.

Some mathematical problems are designed to be solved recursively. One well-known example is the calculation of *Fibonacci numbers*. The Fibonacci numbers, named after the Italian mathematician Leonardo Fibonacci (born circa 1170), are the following sequence:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, . . .

Notice that after the second number, each number in the series is the sum of the two previous numbers. The Fibonacci series can be defined as

$F_0 = 0$

$F_1 = 1$

$F_N = F_{N-1} + F_{N-2}$ *for* $N \geq 2$.

A recursive C++ function to calculate the *n*th number in the Fibonacci series is shown here:

```
int fib(int n)
{
    if (n <= 0)
        return 0;
    else if (n == 1)
        return 1;
    else
        return fib(n - 1) + fib(n - 2);
}
```

The function is demonstrated in Program 19-6, which displays the first 10 numbers in the Fibonacci series.

Program 19-6

```
// This program demonstrates a recursive function
// that calculates Fibonacci numbers.
#include <iostream>
using namespace std;

```

```
// Function prototype
int fib(int);

int main()
{
    cout << "The first 10 Fibonacci numbers are:\n";
    for (int x = 0; x < 10; x++)
        cout << fib(x) << " ";
    cout << endl;
    return 0;
}

//*****************************************
// Function fib. Accepts an int argument *
// in n. This function returns the nth   *
// Fibonacci number.                     *
//*****************************************

int fib(int n)
{
    if (n <= 0)
        return 0;                       // Base case
    else if (n == 1)
        return 1;                       // Base case
    else
        return fib(n - 1) + fib(n - 2);   // Recursive case
}
```

Program Output

```
The first 10 Fibonacci numbers are:
0 1 1 2 3 5 8 13 21 34
```

Another such example is Ackermann's function. A Programming Challenge at the end of this chapter asks you to write a recursive function that calculates Ackermann's function.

19.5 Focus on Problem Solving and Program Design: Recursive Linked List Operations

CONCEPT: Recursion can be used to traverse the nodes in a linked list.

Recall that in Chapter 17 we discussed a class named `NumberList` that holds a linked list of `double` values. In this section we will modify the class by adding recursive member functions. The functions will use recursion to traverse the linked list and perform the following operations:

- Count the number of nodes in the list.

  To count the number of nodes in the list by recursion, we introduce two new member functions: `numNodes` and `countNodes`. `countNodes` is a private member function that uses recursion, and `numNodes` is the public interface that calls it.

- Display the value of the list nodes in reverse order.

  To display the nodes in the list in reverse order, we introduce two new member functions: `displayBackwards` and `showReverse`. `showReverse` is a private member function that uses recursion, and `displayBackwards` is the public interface that calls it.

The class declaration, which is saved in `NumberList.h`, is shown here:

```
// Specification file for the NumberList class
#ifndef NUMBERLIST_H
#define NUMBERLIST_H

class NumberList
{
private:
     // Declare a structure for the list
     struct ListNode
     {
          double value;
          struct ListNode *next;
     };

     ListNode *head;     // List head pointer

     // Private member functions
     int countNodes(ListNode *) const;
     void showReverse(ListNode *) const;

public:
     // Constructor
     NumberList()
          { head = nullptr; }

     // Destructor
     ~NumberList();

     // Linked List Operations
     void appendNode(double);
     void insertNode(double);
     void deleteNode(double);
     void displayList() const;
     int numNodes() const
          { return countNodes(head); }
     void displayBackwards() const
          { showReverse(head); }
};
#endif
```

Counting the Nodes in the List

The `numNodes` function is declared inline. It simply calls the `countNodes` function and passes the `head` pointer as an argument. (Because the `head` pointer, which is private, must be passed to `countNodes`, the `numNodes` function is needed as an interface.)

The function definition for `countNodes` is shown here:

```
int NumberList::countNodes(ListNode *nodePtr) const
{
    if (nodePtr != nullptr)
        return 1 + countNodes(nodePtr->next);
    else
        return 0;
}
```

The function's recursive logic can be expressed as:

```
If the current node has a value
   Return 1 + the number of the remaining nodes.
Else
   Return 0.
End If.
```

Program 19-7 demonstrates the function.

Program 19-7

```
// This program counts the nodes in a list.
#include <iostream>
#include "NumberList.h"
using namespace std;

int main()
{
    const int MAX = 10; // Maximum number of values

    // Define a NumberList object.
    NumberList list;

    // Build the list with a series of numbers.
    for (int x = 0; x < MAX; x++)
        list.insertNode(x);

    // Display the number of nodes in the list.
    cout << "The number of nodes is "
         << list.numNodes() << endl;
    return 0;
}
```

Program Output

```
The number of nodes is 10
```

Displaying List Nodes in Reverse Order

The technique for displaying the list nodes in reverse order is designed like the node counting procedure: A public member function, which serves as an interface, passes the head pointer to a private member function. The public `displayBackwards` function, declared inline, is the interface. It calls the showReverse function and passes the head pointer as an argument. The function definition for showReverse is shown here:

```
void NumberList::showReverse(ListNode *nodePtr) const
{
    if (nodePtr != nullptr)
    {
        showReverse(nodePtr->next);
        cout << nodePtr->value << " ";
    }
}
```

The base case for the function is nodePtr being set to nullptr. When this is true, the function has reached the last node in the list, so it returns. It is not until this happens that any instances of the cout statement execute. The instance of the function whose nodePtr variable points to the last node in the list will be the first to execute the cout statement. It will then return, and the previous instance of the function will execute its cout statement. This repeats until all the instances of the function have returned.

The modified class declaration is stored in NumberList.h, and its member function implementation is in NumberList.cpp. The remainder of the class implementation is unchanged from Chapter 17, so it is not shown here. Program 19-8 demonstrates the function.

Program 19-8

```
// This program demonstrates the recursive function
// for displaying the list's nodes in reverse.
#include <iostream>
#include "NumberList.h"
using namespace std;

int main()
{
    const double MAX = 10.0; // Upper limit of values

    // Create a NumberList object.
    NumberList list;

    // Add a series of numbers to the list.
    for (double x = 1.5; x < MAX; x += 1.1)
        list.appendNode(x);

    // Display the values in the list.
    cout << "Here are the values in the list:\n";
    list.displayList();

    // Display the values in reverse order.
    cout << "Here are the values in reverse order:\n";
    list.displayBackwards();
    return 0;
}
```

Program Output

```
Here are the values in the list:
1.5
2.6
3.7
4.8
5.9
7
8.1
9.2
Here are the values in reverse order:
9.2 8.1 7 5.9 4.8 3.7 2.6 1.5
```

19.6 Focus on Problem Solving and Program Design: A Recursive Binary Search Function

CONCEPT: The binary search algorithm can be defined as a recursive function.

In Chapter 8 you learned about the binary search algorithm and saw an iterative example written in C++. The binary search algorithm can also be implemented recursively. For example, the procedure can be expressed as

```
If array[middle] equals the search value, then the value is found.

Else, if array[middle] is less than the search value, perform a
binary search on the upper half of the array.

Else, if array[middle] is greater than the search value, perform a
binary search on the lower half of the array.
```

The recursive binary search algorithm is an example of breaking a problem down into smaller pieces until it is solved. A recursive binary search function is shown here:

```
int binarySearch(int array[], int first, int last, int value)
{
   int middle;    // Midpoint of search

   if (first > last)
       return -1;
   middle = (first + last) / 2;
   if (array[middle] == value)
       return middle;
   if (array[middle] < value)
       return binarySearch(array, middle+1,last,value);
   else
       return binarySearch(array, first,middle-1,value);
}
```

The first parameter, `array`, is the array to be searched. The next parameter, `first`, holds the subscript of the first element in the search range (the portion of the array to be searched). The next parameter, `last`, holds the subscript of the last element in the search range. The last parameter, `value`, holds the value to be searched for. Like the iterative version, this function returns the subscript of the value if it is found, or -1 if the value is not found. Program 19-9 demonstrates the function.

Program 19-9

```
// This program demonstrates the recursive binarySearch function.
#include <iostream>
using namespace std;

// Function prototype
int binarySearch(int [], int, int, int);

const int SIZE = 20; // Array size

int main()
{
    // Define an array of employee ID numbers
    int tests[SIZE] = {101, 142, 147, 189, 199, 207, 222,
                       234, 289, 296, 310, 319, 388, 394,
                       417, 429, 447, 521, 536, 600};
    int empID;    // To hold an ID number
    int results;  // To hold the search results

    // Get an employee ID number to search for.
    cout << "Enter the Employee ID you wish to search for: ";
    cin >> empID;

    // Search for the ID number in the array.
    results = binarySearch(tests, 0, SIZE - 1, empID);

    // Display the results of the search.
    if (results == -1)
        cout << "That number does not exist in the array.\n";
    else
    {
        cout << "That ID is found at element " << results;
        cout << " in the array\n";
    }
    return 0;
}

//*****************************************************************
// The binarySearch function performs a recursive binary search *
// on a range of elements of an integer array passed into the   *
// parameter array. The parameter first holds the subscript of  *
// the range's starting element, and last holds the subscript   *
// of the range's last element. The parameter value holds the   *
// search value. If the search value is found, its array        *
// subscript is returned. Otherwise, -1 is returned indicating  *
// the value was not in the array.                              *
//*****************************************************************

int binarySearch(int array[], int first, int last, int value)
{
    int middle; // Midpoint of search

    if (first > last)
        return -1;
    middle = (first + last)/2;
    if (array[middle]==value)
        return middle;
    if (array[middle]<value)
        return binarySearch(array, middle+1,last,value);
    else
        return binarySearch(array, first,middle-1,value);
}
```

Program Output with Example Input Shown in Bold

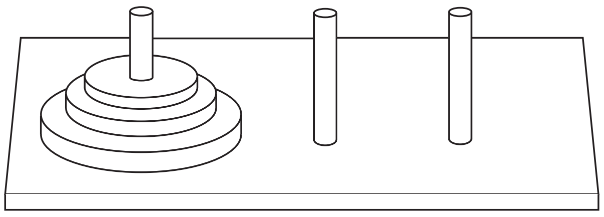
```
Enter the Employee ID you wish to search for: 521 [Enter]
That ID is found at element 17 in the array
```

19.7 The Towers of Hanoi

CONCEPT: The repetitive steps involved in solving the Towers of Hanoi game can be easily implemented in a recursive algorithm.

The Towers of Hanoi is a mathematical game that is often used in computer science textbooks to illustrate the power of recursion. The game uses three pegs and a set of discs with holes through their centers. The discs are stacked on one of the pegs as shown in Figure 19-4.

Figure 19-4 The pegs and discs in the Towers of Hanoi game



Notice that the discs are stacked on the leftmost peg, in order of size with the largest disc at the bottom. The game is based on a legend in which a group of monks in a temple in Hanoi have a similar set of pegs with 64 discs. The job of the monks is to move the discs from the first peg to the third peg. The middle peg can be used as a temporary holder. Furthermore, the monks must follow these rules while moving the discs:

- Only one disc may be moved at a time.
- A disc cannot be placed on top of a smaller disc.
- All discs must be stored on a peg except while being moved.

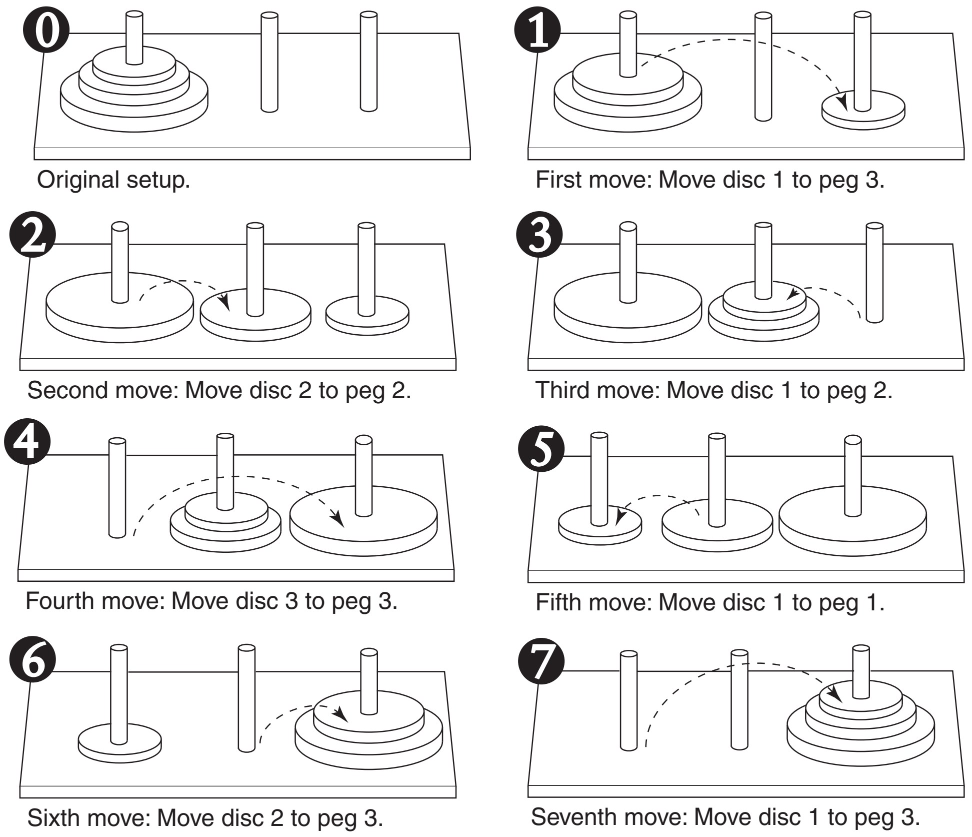
According to the legend, when the monks have moved all of the discs from the first peg to the last peg, the world will come to an end.

To play the game, you must move all of the discs from the first peg to the third peg, following the same rules as the monks. Let's look at some example solutions to this game, for different numbers of discs. If you only have one disc, the solution to the game is simple: move the disc from peg 1 to peg 3. If you have two discs, the solution requires three moves:

- Move disc 1 to peg 2.
- Move disc 2 to peg 3.
- Move disc 1 to peg 3.

Notice that this approach uses peg 2 as a temporary location. The complexity of the moves continues to increase as the number of discs increases. To move three discs requires the seven moves shown in Figure 19-5.

Figure 19-5


0
Original setup.
1
First move: Move disc 1 to peg 3.
2
Second move: Move disc 2 to peg 2.
3
Third move: Move disc 1 to peg 2.
4
Fourth move: Move disc 3 to peg 3.
5
Fifth move: Move disc 1 to peg 1.
6
Sixth move: Move disc 2 to peg 3.
7
Seventh move: Move disc 1 to peg 3.

The following statement describes the overall solution to the problem:

Move n discs from peg 1 to peg 3 using peg 2 as a temporary peg.

The following algorithm can be used as the basis of a recursive function that simulates the solution to the game. Notice that in this algorithm we use the variables *A*, *B*, and *C* to hold peg numbers.

```
To move n discs from peg A to peg C, using peg B as a temporary peg:
If n > 0 Then
    Move n – 1 discs from peg A to peg B, using peg C as a temporary peg.
    Move the remaining disc from the peg A to peg C.
    Move n – 1 discs from peg B to peg C, using peg A as a temporary peg.
End If
```

The base case for the algorithm is reached when there are no more discs to move. The following code is for a function that implements this algorithm. Note that the function does not actually move anything, but displays instructions indicating all of the disc moves to make.

```
void moveDiscs(int num, int fromPeg, int toPeg, int tempPeg)
{
   if (num > 0)
   {
       moveDiscs(num - 1, fromPeg, tempPeg, toPeg);
       cout << "Move a disc from peg " << fromPeg
            << " to peg " << toPeg << endl;
       moveDiscs(num - 1, tempPeg, toPeg, fromPeg);
   }
}
```

This function accepts arguments into the following three parameters:

`num`	The number of discs to move.
`fromPeg`	The peg to move the discs from.
`toPeg`	The peg to move the discs to.
`tempPeg`	The peg to use as a temporary peg.

If `num` is greater than 0, then there are discs to move. The first recursive call is

```
moveDiscs(num - 1, fromPeg, tempPeg, toPeg);
```

This statement is an instruction to move all but one disc from `fromPeg` to `tempPeg`, using `toPeg` as a temporary peg. The next statement is

```
cout << "Move a disc from peg " << fromPeg
     << " to peg " << toPeg << endl;
```

This simply displays a message indicating that a disc should be moved from `fromPeg` to `toPeg`. Next, another recursive call is executed:

```
moveDiscs(num - 1, tempPeg, toPeg, fromPeg);
```

This statement is an instruction to move all but one disc from `tempPeg` to `toPeg`, using `fromPeg` as a temporary peg. Program 19-10 demonstrates this function.

Program 19-10

```
// This program displays a solution to the Towers of
// Hanoi game.
#include <iostream>
using namespace std;

// Function prototype
void moveDiscs(int, int, int, int);

int main()
{
    const int NUM_DISCS = 3;  // Number of discs to move
    const int FROM_PEG = 1;   // Initial "from" peg
    const int TO_PEG = 3;     // Initial "to" peg
    const int TEMP_PEG = 2;   // Initial "temp" peg

    // Play the game.
    moveDiscs(NUM_DISCS, FROM_PEG, TO_PEG, TEMP_PEG);
    cout << "All the pegs are moved!\n";
```

(program continues)

Program 19-10 *(continued)*

```
    return 0;
}

//**************************************************
// The moveDiscs function displays a disc move in  *
// the Towers of Hanoi game.                       *
// The parameters are:                             *
// num: The number of discs to move.               *
// fromPeg: The peg to move from.                  *
// toPeg: The peg to move to.                      *
// tempPeg: The temporary peg.                     *
//**************************************************

void moveDiscs(int num, int fromPeg, int toPeg, int tempPeg)
{
    if (num > 0)
    {
        moveDiscs(num - 1, fromPeg, tempPeg, toPeg);
        cout << "Move a disc from peg " << fromPeg
             << " to peg " << toPeg << endl;
        moveDiscs(num - 1, tempPeg, toPeg, fromPeg);
    }
}
```

Program Output

```
Move a disc from peg 1 to peg 3
Move a disc from peg 1 to peg 2
Move a disc from peg 3 to peg 2
Move a disc from peg 1 to peg 3
Move a disc from peg 2 to peg 1
Move a disc from peg 2 to peg 3
Move a disc from peg 1 to peg 3
All the pegs are moved!
```

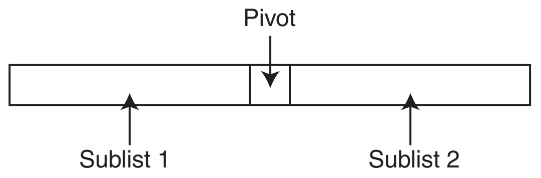
19.8 Focus on Problem Solving and Program Design: The QuickSort Algorithm

CONCEPT: The QuickSort algorithm uses recursion to efficiently sort a list.

The QuickSort algorithm is a popular general-purpose sorting routine developed in 1960 by C.A.R. Hoare. It can be used to sort lists stored in arrays or linear linked lists. It sorts a list by dividing it into two sublists. Between the sublists is a selected value known as the *pivot*. This is illustrated in Figure 19-6.

Notice in the figure that sublist 1 is positioned to the left of (before) the pivot, and sublist 2 is positioned to the right of (after) the pivot. Once a pivot value has been selected, the algorithm exchanges the other values in the list until all the elements in sublist 1 are less than the pivot, and all the elements in sublist 2 are greater than the pivot.

Figure 19-6



Once this is done, the algorithm repeats the procedure on sublist 1, and then on sublist 2. The recursion stops when there is only one element in a sublist. At that point the original list is completely sorted.

The algorithm is coded primarily in two functions: `quickSort` and `partition`. `quickSort` is a recursive function. Its pseudocode is shown here:

```
quickSort:
If Starting Index < Ending Index
   Partition the List around a Pivot.
   quickSort Sublist 1.
   quickSort Sublist 2.
End If.
```

Here is the C++ code for the `quickSort` function:

```
void quickSort(int set[], int start, int end)
{
   int pivotPoint;

   if (start < end)
   {
      // Get the pivot point.
      pivotPoint = partition(set, start, end);
      // Sort the first sublist.
      quickSort(set, start, pivotPoint - 1);
      // Sort the second sublist.
      quickSort(set, pivotPoint + 1, end);
   }
}
```

This version of `quickSort` works with an array of integers. Its first argument is the array holding the list that is to be sorted. The second and third arguments are the starting and ending subscripts of the list.

The subscript of the pivot element is returned by the `partition` function. `partition` not only determines which element will be the pivot, but also controls the rearranging of the other values in the list. Our version of this function selects the element in the middle of the list as the pivot, then scans the remainder of the list searching for values less than the pivot.

The code for the `partition` function is shown here:

```
int partition(int set[], int start, int end)
{
   int pivotValue, pivotIndex, mid;

   mid = (start + end) / 2;
   swap(set[start], set[mid]);
   pivotIndex = start;
   pivotValue = set[start];
```

```
        for (int scan = start + 1; scan <= end; scan++)
        {
            if (set[scan] < pivotValue)
            {
                 pivotIndex++;
                      swap(set[pivotIndex], set[scan]);
            }
        }
        swap(set[start], set[pivotIndex]);
        return pivotIndex;
    }
```

NOTE: The `partition` function does not initially sort the values into their final order. Its job is only to move the values that are less than the pivot to the pivot's left, and move the values that are greater than the pivot to the pivot's right. As long as that condition is met, they may appear in any order. The ultimate sorting order of the entire list is achieved cumulatively, though the recursive calls to `quickSort`.

There are many different ways of partitioning the list. As previously stated, the method shown in the function above selects the middle value as the pivot. That value is then moved to the beginning of the list (by exchanging it with the value stored there). This simplifies the next step, which is to scan the list.

A `for` loop scans the remainder of the list, and when an element is found whose value is less than the pivot, that value is moved to a location left of the pivot point.

A third function, `swap`, is used to swap the values found in any two elements of the list. The function is shown below.

```
void swap(int &value1, int &value2)
{
     int temp = value1;
     value1 = value2;
     value2 = temp;
}
```

Program 19-11 demonstrates the QuickSort algorithm shown here.

Program 19-11

```
// This program demonstrates the QuickSort Algorithm.
#include <iostream>
using namespace std;

// Function prototypes
void quickSort(int [], int, int);
int partition(int [], int, int);
void swap(int &, int &);

int main()
{
     const int SIZE = 10; // Array size
     int count;           // Loop counter
     int array[SIZE] = {7, 3, 9, 2, 0, 1, 8, 4, 6, 5};
```

```

    // Display the array contents.
    for (count = 0; count < SIZE; count++)
        cout << array[count] << " ";
    cout << endl;

    // Sort the array.
    quickSort(array, 0, SIZE - 1);

    // Display the array contents.
    for (count = 0; count < SIZE; count++)
        cout << array[count] << " ";
    cout << endl;
    return 0;
}

//************************************************
// quickSort uses the quicksort algorithm to    *
// sort set, from set[start] through set[end].  *
//************************************************

void quickSort(int set[], int start, int end)
{
    int pivotPoint;

    if (start < end)
    {
        // Get the pivot point.
        pivotPoint = partition(set, start, end);
        // Sort the first sublist.
        quickSort(set, start, pivotPoint - 1);
        // Sort the second sublist.
        quickSort(set, pivotPoint + 1, end);
    }
}

//**********************************************************
// partition selects the value in the middle of the       *
// array set as the pivot. The list is rearranged so      *
// all the values less than the pivot are on its left     *
// and all the values greater than pivot are on its right. *
//**********************************************************

int partition(int set[], int start, int end)
{
    int pivotValue, pivotIndex, mid;

    mid = (start + end) / 2;
    swap(set[start], set[mid]);
    pivotIndex = start;
    pivotValue = set[start];
    for (int scan = start + 1; scan <= end; scan++)
    {
        if (set[scan] < pivotValue)
        {
```

(program continues)

Program 19-11 *(continued)*

```
                 pivotIndex++;
                 swap(set[pivotIndex], set[scan]);
            }
       }
       swap(set[start], set[pivotIndex]);
       return pivotIndex;
}

//*******************************************
// swap simply exchanges the contents of   *
// value1 and value2.                       *
//*******************************************

void swap(int &value1, int &value2)
{
     int temp = value1;

     value1 = value2;
     value2 = temp;
}
```

Program Output

```
7 3 9 2 0 1 8 4 6 5
0 1 2 3 4 5 6 7 8 9
```

19.9 Exhaustive Algorithms

CONCEPT: An exhaustive algorithm is one that finds a best combination of items by looking at all the possible combinations.

Recursion is helpful if you need to examine many possible combinations and identify the best combination. For example, consider all the different ways you can make change for $1.00 using our system of coins:

1 dollar piece, or
2 fifty-cent pieces, or
4 quarters, or
1 fifty-cent piece and 2 quarters, or
3 quarters, 2 dimes, and 1 nickel, or
… *there are many more possibilities.*

Although there are many ways to make change for $1.00, some ways are better than others. For example, you would probably rather give a single dollar piece instead of 100 pennies.

An algorithm that looks at all the possible combinations of items in order to find the best combination of items is called an exhaustive algorithm. Program 19-12 presents a recursive function that exhaustively tries all the possible combinations of coins. The program then displays the total number of combinations that can be used to make the specified change and the best combination of coins.

Program 19-12

```
// This program demonstrates a recursive function that exhaustively
// searches through all possible combinations of coin values to find
// the best way to make change for a specified amount.
#include <iostream>
using namespace std;

// Constants
const int MAX_COINS_CHANGE = 100; // Max number of coins to give in change
const int MAX_COIN_VALUES = 6;   // Max number of coin values
const int NO_SOLUTION = INT_MAX; // Indicates no solution

// Function prototype
void makeChange(int, int, int[], int);

// coinValues - global array of coin values to choose from
int coinValues[MAX_COIN_VALUES] = {100, 50, 25, 10, 5, 1 };

// bestCoins - global array of best coins to make change with
 int bestCoins[MAX_COINS_CHANGE];

// Global variables
int numBestCoins = NO_SOLUTION,   // Number of coins in bestCoins
    numSolutions = 0,             // Number of ways to make change
    numCoins;                     // Number of allowable coins


int main()
{
    int coinsUsed[MAX_COINS_CHANGE],   // List of coins used
    numCoinsUsed = 0,                  // The number of coins used
    amount;                            // The amount to make change for

    // Display the possible coin values.
    cout << "Here are the valid coin values, in cents: ";
    for (int index = 0; index < 5; index++)
        cout << coinValues[index] << " ";
    cout << endl;

    // Get input from the user.
    cout << "Enter the amount of cents (as an integer) "
         << "to make change for: ";
    cin >> amount;
    cout << "What is the maximum number of coins to give as change? ";
    cin >> numCoins;

    // Call the recursive function.
    makeChange(numCoins, amount, coinsUsed, numCoinsUsed);

    // Display the results.
    cout << "Number of possible combinations: " << numSolutions << endl;
    cout << "Best combination of coins:\n";
    if (numBestCoins == NO_SOLUTION)
        cout << "\tNo solution\n";
```

(program continues)

Program 19-12 *(continued)*

```
    else
    {
        for (int count = 0; count < numBestCoins; count++)
            cout << bestCoins[count] << " ";
    }
    cout << endl;
    return 0;
}

//***********************************************************************
// Function makeChange. This function uses the following parameters:  *
// coinsLeft - The number of coins left to choose from.                *
// amount - The amount to make change for.                             *
// coinsUsed - An array that contains the coin values used so far.     *
// numCoinsUsed - The number of values in the coinsUsed array.         *
//                                                                     *
// This recursive function finds all the possible ways to make change  *
// for the value in amount. The best combination of coins is stored in *
// the array bestCoins.                                                *
//***********************************************************************

void makeChange(int coinsLeft, int amount, int coinsUsed[],
                int numCoinsUsed)
{
    int coinPos, // To calculate array position of coin being used
        count;   // Loop counter

    if (coinsLeft == 0)   // If no more coins are left
        return;
    else if (amount < 0)  // If amount to make change for is negative
        return;
    else if (amount == 0) // If solution is found
    {
        // Store as bestCoins if best
        if (numCoinsUsed < numBestCoins)
        {
            for (count = 0; count < numCoinsUsed; count++)
                bestCoins[count] = coinsUsed[count];
            numBestCoins = numCoinsUsed;
        }
        numSolutions++;
        return;
    }

    // Find the other combinations using the coin
    coinPos = numCoins - coinsLeft;
    coinsUsed[numCoinsUsed] = coinValues[coinPos];
    numCoinsUsed++;
    makeChange(coinsLeft, amount - coinValues[coinPos],
               coinsUsed, numCoinsUsed);

    // Find the other combinations not using the coin.
    numCoinsUsed--;
    makeChange(coinsLeft - 1, amount, coinsUsed, numCoinsUsed);
}
```

Program Output with Example Input Shown in Bold

```
Here are the valid coin values, in cents: 100 50 25 10 5 1
Enter the amount of cents (as an integer) to make change for: 62 [Enter]
What is the maximum number of coins to give as change? 6 [Enter]
Number of possible combinations: 77
Best combination of coins:
50 10 1 1
```

19.10 Focus on Software Engineering: Recursion vs. Iteration

CONCEPT: Recursive algorithms can also be coded with iterative control structures. There are advantages and disadvantages to each approach.

Any algorithm that can be coded with recursion can also be coded with an iterative control structure, such as a `while` loop. Both approaches achieve repetition, but which is best to use?

There are several reasons not to use recursion. Recursive algorithms are certainly less efficient than iterative algorithms. Each time a function is called, the system incurs overhead that is not necessary with a loop. Also, in many cases an iterative solution may be more evident than a recursive one. In fact, the majority of repetitive programming tasks are best done with loops.

Some problems, however, are more easily solved with recursion than with iteration. For example, the mathematical definition of the GCD formula is well-suited for a recursive approach. The QuickSort algorithm is also an example of a function that is easier to code with recursion than iteration.

The speed and amount of memory available to modern computers diminishes the performance impact of recursion so much that inefficiency is no longer a strong argument against it. Today, the choice of recursion or iteration is primarily a design decision. If a problem is more easily solved with a loop, that should be the approach you take. If recursion results in a better design, that is the choice you should make.

Review Questions and Exercises

Short Answer

1. What is the base case of each of the recursive functions listed in questions 12, 13, and 14?
2. What type of recursive function do you think would be more difficult to debug, one that uses direct recursion, or one that uses indirect recursion? Why?
3. Which repetition approach is less efficient, a loop or a recursive function? Why?
4. When should you choose a recursive algorithm over an iterative algorithm?
5. Explain what is likely to happen when a recursive function that has no way of stopping executes.

Fill-in-the-Blank

6. The __________ of recursion is the number of times a function calls itself.
7. A recursive function's solvable problem is known as its __________. This causes the recursion to stop.

8. __________ recursion is when a function explicitly calls itself.
9. __________ recursion is when function A calls function B, which in turns calls function A.

Algorithm Workbench

10. Write a recursive function to return the number of times a specified number occurs in an array.
11. Write a recursive function to return the largest value in an array.

Predict the Output

What is the output of the following programs?

12.
```
#include <iostream>
using namespace std;

int function(int);

int main()
{
    int x = 10;

    cout << function(x) << endl;
    return 0;
}

int function(int num)
{
    if (num <= 0)
        return 0;
    else
        return function(num - 1) + num;
}
```

13.
```
#include <iostream>
using namespace std;

void function(int);

int main()
{
    int x = 10;

    function(x);
    return 0;
}

void function(int num)
{
    if (num > 0)
    {
        for (int x = 0; x < num; x++)
            cout << '*';
        cout << endl;
        function(num - 1);
    }
}
```

14.
```
#include <iostream>
#include <string>
using namespace std;

void function(string, int, int);

int main()
{
     string mystr = "Hello";
     cout << mystr << endl;
     function(mystr, 0, mystr.size());
     return 0;
}
void function(string str, int pos, int size)
{
     if (pos < size)
     {
          function(str, pos + 1, size);
          cout << str[pos];
     }
}
```

Programming Challenges

1. **Iterative Factorial**

   Write an iterative version (using a loop instead of recursion) of the factorial function shown in this chapter. Test it with a driver program.

2. **Recursive Conversion**

   Convert the following function to one that uses recursion.

```
void sign(int n)
{
     while (n > 0)
          cout << "No Parking\n";
     n--;
}
```

   Demonstrate the function with a driver program.

3. **QuickSort Template**

   Create a template version of the QuickSort algorithm that will work with any data type. Demonstrate the template with a driver function.

4. **Recursive Array Sum**

   Write a function that accepts an array of integers and a number indicating the number of elements as arguments. The function should recursively calculate the sum of all the numbers in the array. Demonstrate the function in a driver program.

VideoNote
Solving the Recursive Multiplication Problem

5. **Recursive Multiplication**

   Write a recursive function that accepts two arguments into the parameters `x` and `y`. The function should return the value of `x` times `y`. Remember, multiplication can be performed as repeated addition:

```
7 * 4 = 4 + 4 + 4 + 4 + 4 + 4 + 4
```

6. **Recursive Power Function**

   Write a function that uses recursion to raise a number to a power. The function should accept two arguments: the number to be raised and the exponent. Assume that the exponent is a nonnegative integer. Demonstrate the function in a program.

7. **Sum of Numbers**

   Write a function that accepts an integer argument and returns the sum of all the integers from 1 up to the number passed as an argument. For example, if 50 is passed as an argument, the function will return the sum of 1, 2, 3, 4, … 50. Use recursion to calculate the sum. Demonstrate the function in a program.

8. **`isMember` Function**

   Write a recursive Boolean function named `isMember`. The function should accept two arguments: an array and a value. The function should return true if the value is found in the array, or false if the value is not found in the array. Demonstrate the function in a driver program.

9. **String Reverser**

   Write a recursive function that accepts a `string` object as its argument and prints the string in reverse order. Demonstrate the function in a driver program.

10. **`maxNode` Function**

    Add a member function named `maxNode` to the `NumberList` class discussed in this chapter. The function should return the largest value stored in the list. Use recursion in the function to traverse the list. Demonstrate the function in a driver program.

11. **Palindrome Detector**

    A palindrome is any word, phrase, or sentence that reads the same forward and backward. Here are some well-known palindromes:

    Able was I, ere I saw Elba
    A man, a plan, a canal, Panama
    Desserts, I stressed
    Kayak

    Write a `bool` function that uses recursion to determine if a string argument is a palindrome. The function should return `true` if the argument reads the same forward and backward. Demonstrate the function in a program.

12. **Ackermann's Function**

    Ackermann's Function is a recursive mathematical algorithm that can be used to test how well a computer performs recursion. Write a function `A(m, n)` that solves Ackermann's Function. Use the following logic in your function:

    ```
    If m = 0 then return n + 1
    If n = 0 then return A(m-1, 1)
    Otherwise,    return A(m-1, A(m, n-1))
    ```

    Test your function in a driver program that displays the following values:

    ```
    A(0, 0) A(0, 1) A(1, 1) A(1, 2) A(1, 3) A(2, 2) A(3, 2)
    ```