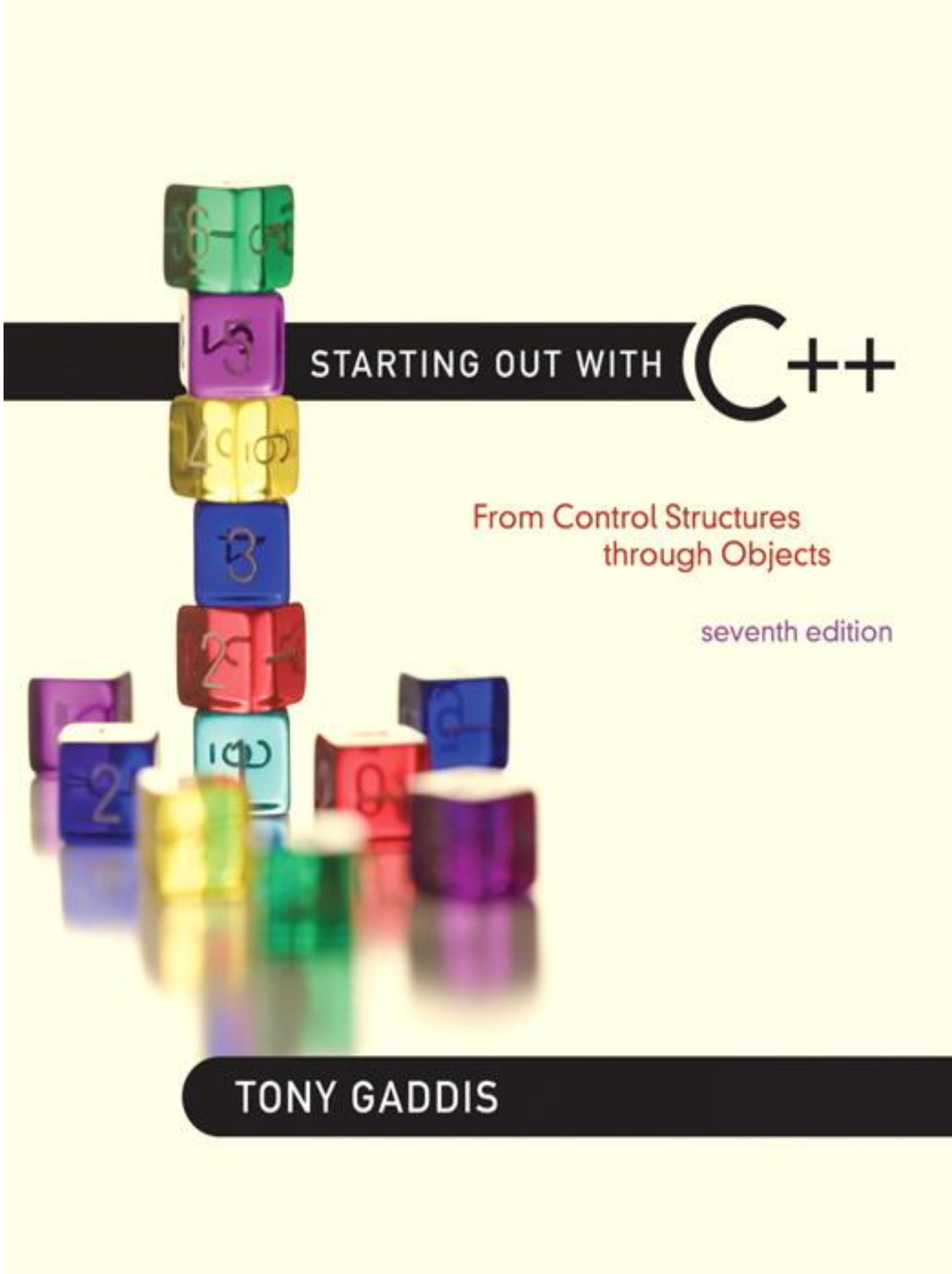
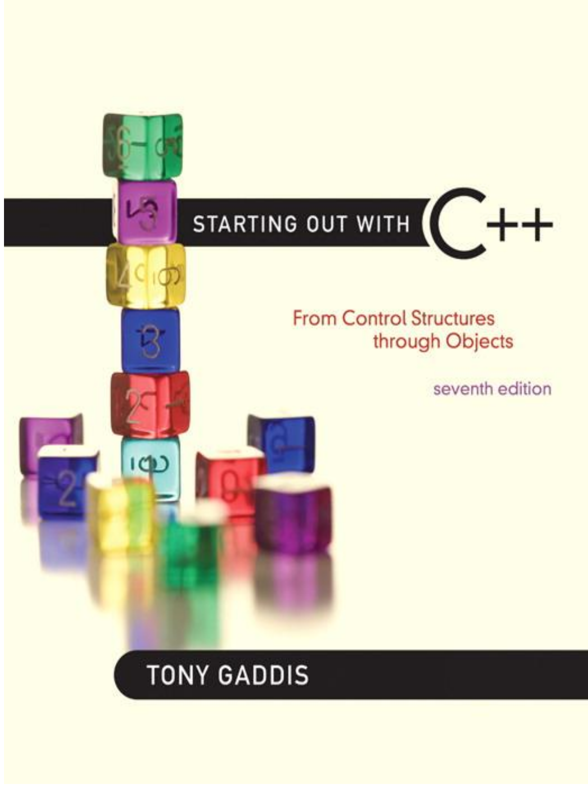
Chapter 19:

Recursion



Addison-Wesley
is an imprint of
PEARSON



19.1

Introduction to Recursion

Copyright © 2012 Pearson Education, Inc.

Introduction to Recursion

- A recursive function contains a call to itself:

```
void countDown(int num)
{
   if (num == 0)
      cout << "Blastoff!";
   else
   {
      cout << num << "...\n";
      countDown(num-1); // recursive call
   }
}
```

What Happens When Called?

If a program contains a line like countDown(2);

1. countDown(2) generates the output 2..., then it calls countDown(1)

2. countDown(1) generates the output 1..., then it calls countDown(0)

3. countDown(0) generates the output Blastoff!, then returns to countDown(1)

4. countDown(1) returns to countDown(2)

5. countDown(2)returns to the calling function

What Happens When Called?

first call to countDown
num is 2

countDown(1);

second call to countDown
num is 1

countDown(0);

third call to countDown
num is 0

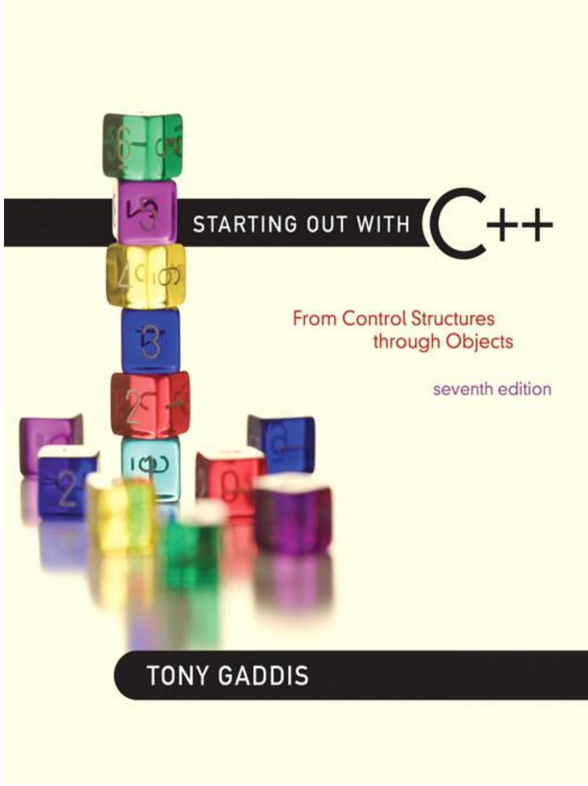
// no
// recursive
// call

output:

2...

1...

Blastoff!



19.2

Solving Problems with Recursion

Copyright © 2012 Pearson Education, Inc.

Recursive Functions – Purpose

- Recursive functions are used to reduce a complex problem to a simpler-to-solve problem.

- The simpler-to-solve problem is known as the base case

- Recursive calls stop when the base case is reached

Stopping the Recursion

- A recursive function must always include a test to determine if another recursive call should be made, or if the recursion should stop with this call

- In the sample program, the test for the base case is:

```
if (num == 0)
```

Stopping the Recursion

```
void countDown(int num)
{
   if (num == 0) //test for base case
      cout << "Blastoff!";
   else
   {
      cout << num << "...\n";
      countDown(num-1); //recursive call
   }
}
```

Stopping the Recursion

- Recursion uses a process of <u>breaking a problem down into smaller problems</u> until the problem can be solved

- In the countDown function, a different value is passed to the function each time it is called

- Eventually, the parameter reaches the value in the test (base case), and the recursion stops

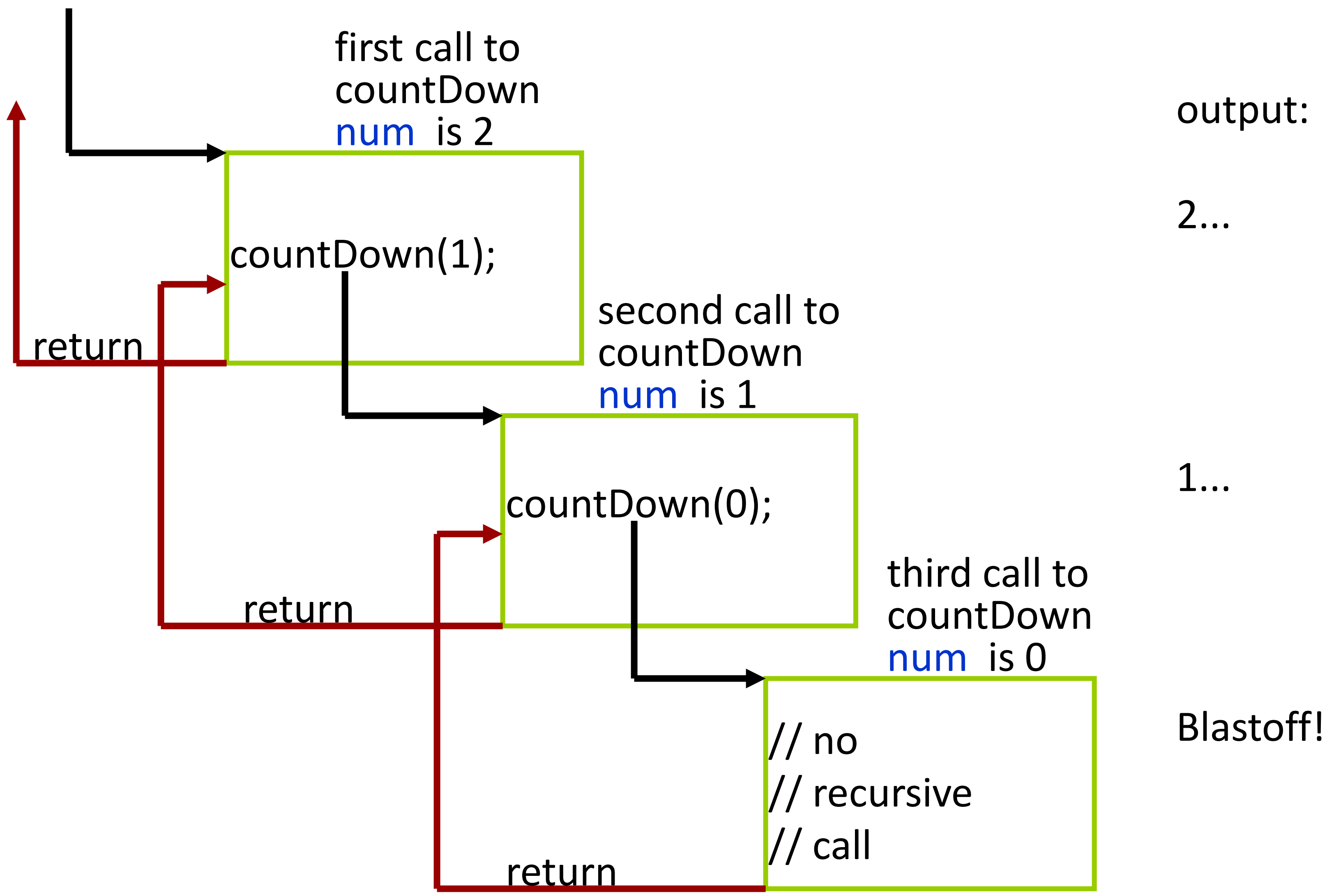
Stopping the Recursion

```
void countDown(int num)
{
   if (num == 0) //test for base case
      cout << "Blastoff!";
   else
   {
      cout << num << "...\n";
      countDown(num-1); //recursive call
   }
}
```

What Happens When Called?

- Each time a recursive function is called, <u>a new copy of the function runs</u>, with new instances of parameters and local variables created
- As each copy finishes executing, it returns to the copy of the function that called it
- When the initial copy finishes executing, it returns to the part of the program that made the initial call to the function

What Happens When Called?



Types of Recursion

- Direct
  - a function calls itself
- Indirect
  - function A calls function B, and function B calls function A
  - function A calls function B, which calls …, which calls function A

The Recursive Factorial Function

- The factorial function:

```
n! = n*(n-1)*(n-2)*...*3*2*1 if n > 0
n! = 1 if n = 0
```

- Can compute factorial of `n` if the factorial of `(n-1)` is known: `n! = n * (n-1)!`
- `n = 0` is the base case

The Recursive Factorial Function

```
int factorial (int num)
{
  if (num > 0)
    return num * factorial(num - 1);
 else
    return 1;
}
```

Program 19-3

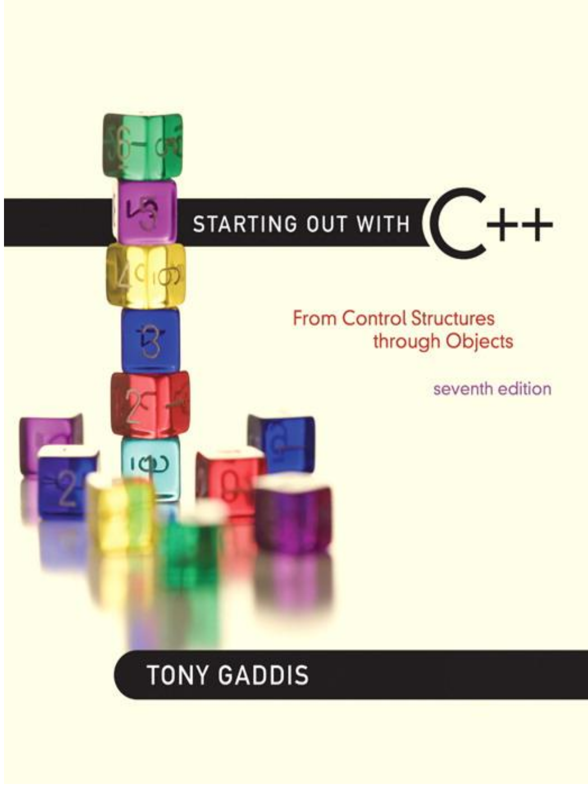
```
// This program demonstrates a recursive function to
// calculate the factorial of a number.
#include <iostream>
using namespace std;

// Function prototype
int factorial(int);

int main()
{
   int number;

   // Get a number from the user.
   cout << "Enter an integer value and I will display\n";
   cout << "its factorial: ";
   cin >> number;

   // Display the factorial of the number.
   cout << "The factorial of " << number << " is ";
   cout << factorial(number) << endl;
   return 0;
}

```

Program 19-3 (Continued)

```
//*****************************************************************
// Definition of factorial. A recursive function to calculate *
// the factorial of the parameter n.                          *
//*****************************************************************

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

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
```



19.3

The Recursive gcd Function

Copyright © 2012 Pearson Education, Inc.

The Recursive `gcd` Function

- Greatest common divisor (gcd) is the <u>largest factor that two integers have in common</u>

- Computed using Euclid's algorithm:

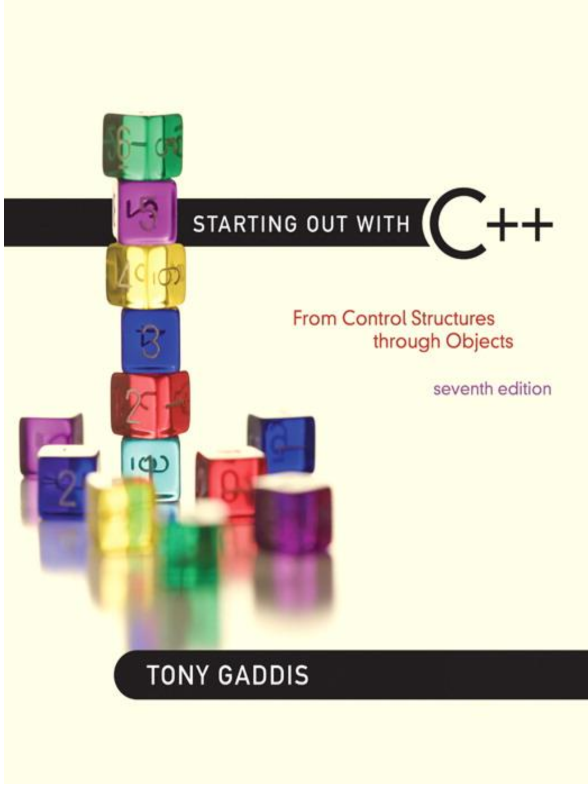
  `gcd(x, y) = y` if `y` divides `x` evenly (x%y == 0)

  `gcd(x, y) =` **`gcd(y, x % y)`** otherwise

- `gcd(x, y) = y` is the base case

The Recursive gcd Function

```
int gcd(int x, int y)
{
    if (x % y == 0)
        return y;
    else
        return gcd(y, x % y);
}
```



19.4

Solving Recursively Defined Problems

Copyright © 2012 Pearson Education, Inc.

Solving Recursively Defined Problems

- The natural definition of some problems leads to a recursive solution

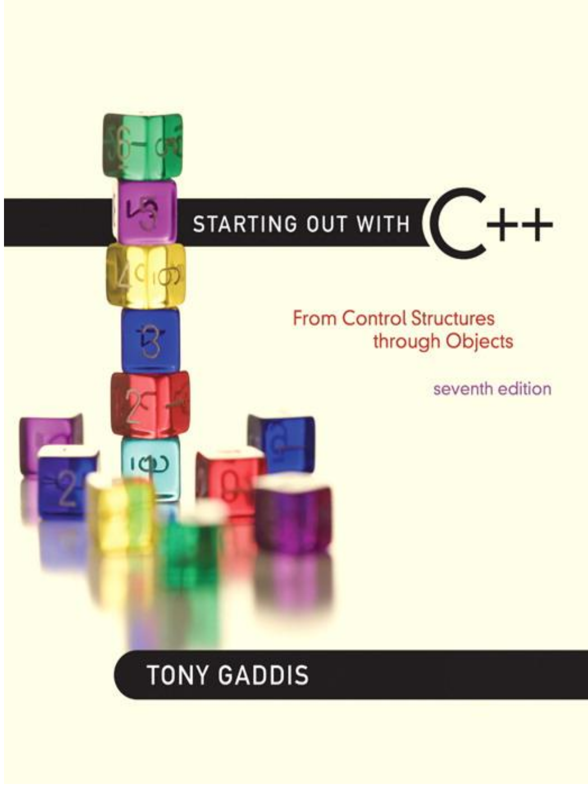
- Example: Fibonacci numbers:

  `0, 1, 1, 2, 3, 5, 8, 13, 21, ...`

- After the starting `0, 1`, each number is the sum of the two preceding numbers
- Recursive solution: `fib(n) = fib(n – 1) + fib(n – 2);`
- Base cases: `n <= 0, n == 1`

Solving Recursively Defined Problems

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



19.5

Recursive Linked List Operations

Copyright © 2012 Pearson Education, Inc.

Recursive Linked List Operations

- Recursive functions can be members of a linked list class
- Some applications:
  - Compute the size of the list (number of nodes)
  - Traverse the list in reverse order

Counting the Nodes in a Linked List

- Uses a pointer to visit each node

- Algorithm:
  - pointer starts at head of list
  - If pointer is NULL
    return 0 (base case)
    else
    return 1 + (number of nodes in the list
    pointed to by current node)

- See the NumberList class

The countNodes function (a private member function)

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

The countNodes function is executed by the public numNodes function:

```
int numNodes() const
   { return countNodes(head); }
```

Contents of a List in Reverse Order

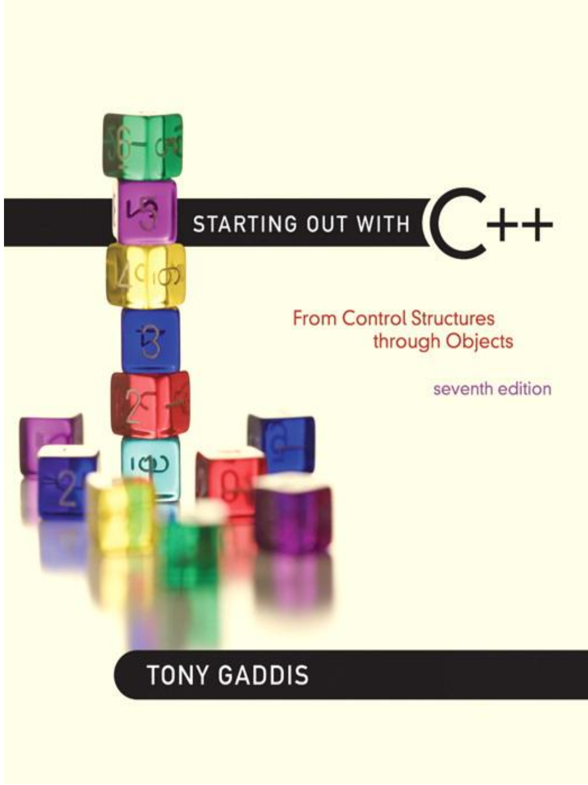
- Algorithm:
  - pointer starts at head of list
  - If the pointer is NULL
    return (base case)
  - If the pointer is not NULL
    advance to next node
  - Upon returning from recursive call, display contents of current node

The showReverse function (a private member function)

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

The **showReverse** function is executed by the public **displayBackwards** function:

```
void displayBackwards() const
   { showReverse(head); }
```



19.6

A Recursive Binary Search Function

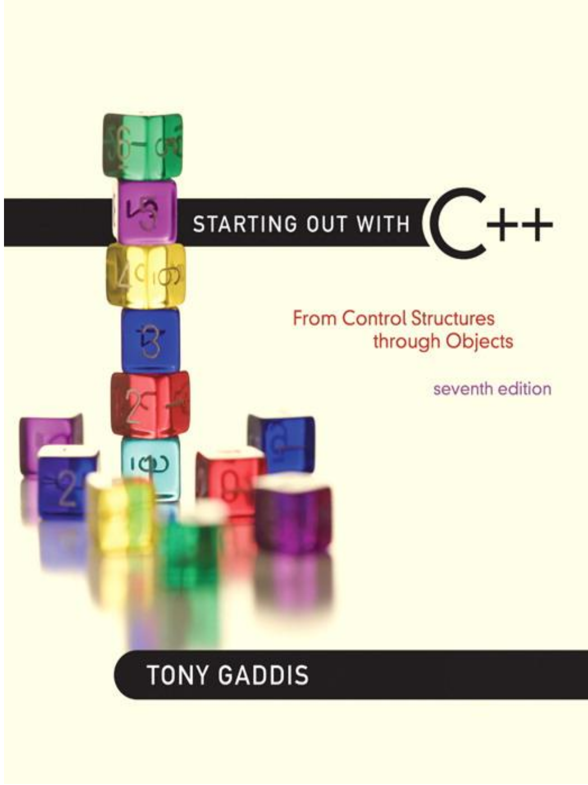
Copyright © 2012 Pearson Education, Inc.

A Recursive Binary Search Function

- Binary search algorithm can easily be written to use recursion
- Base cases:
  - desired value is found
  - there are no more array elements to search
- Algorithm (array in ascending order):
  - If middle element of array segment is desired value
    then done
    Else
    if the middle element is too large, repeat binary search in first half of array segment
    Else
    if the middle element is too small, repeat binary search on the second half of array segment

A Recursive Binary Search Function (Continued)

```
int binarySearch(int array[], int first, int last, int value)
{
   int middle;    // Mid point 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);
}
```



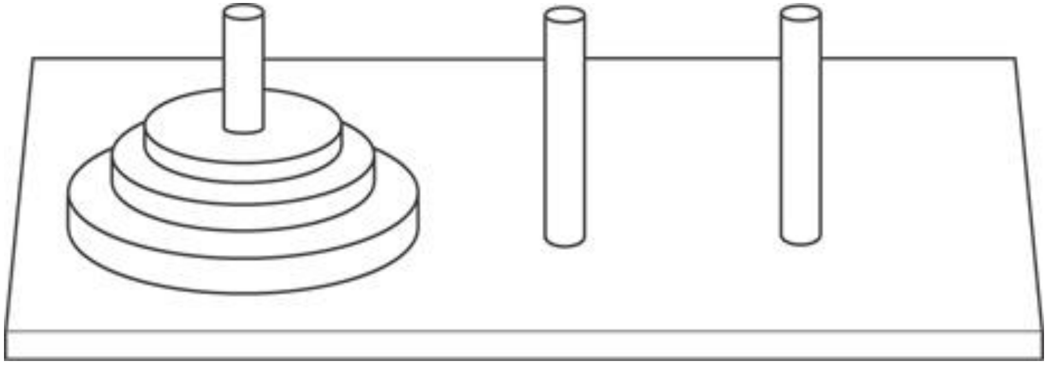
19.7

The Towers of Hanoi

Copyright © 2012 Pearson Education, Inc.

The Towers of Hanoi

- The Towers of Hanoi is a mathematical game that is often used to demonstrate the power of recursion.
- The game uses three pegs and a set of discs, stacked on one of the pegs.



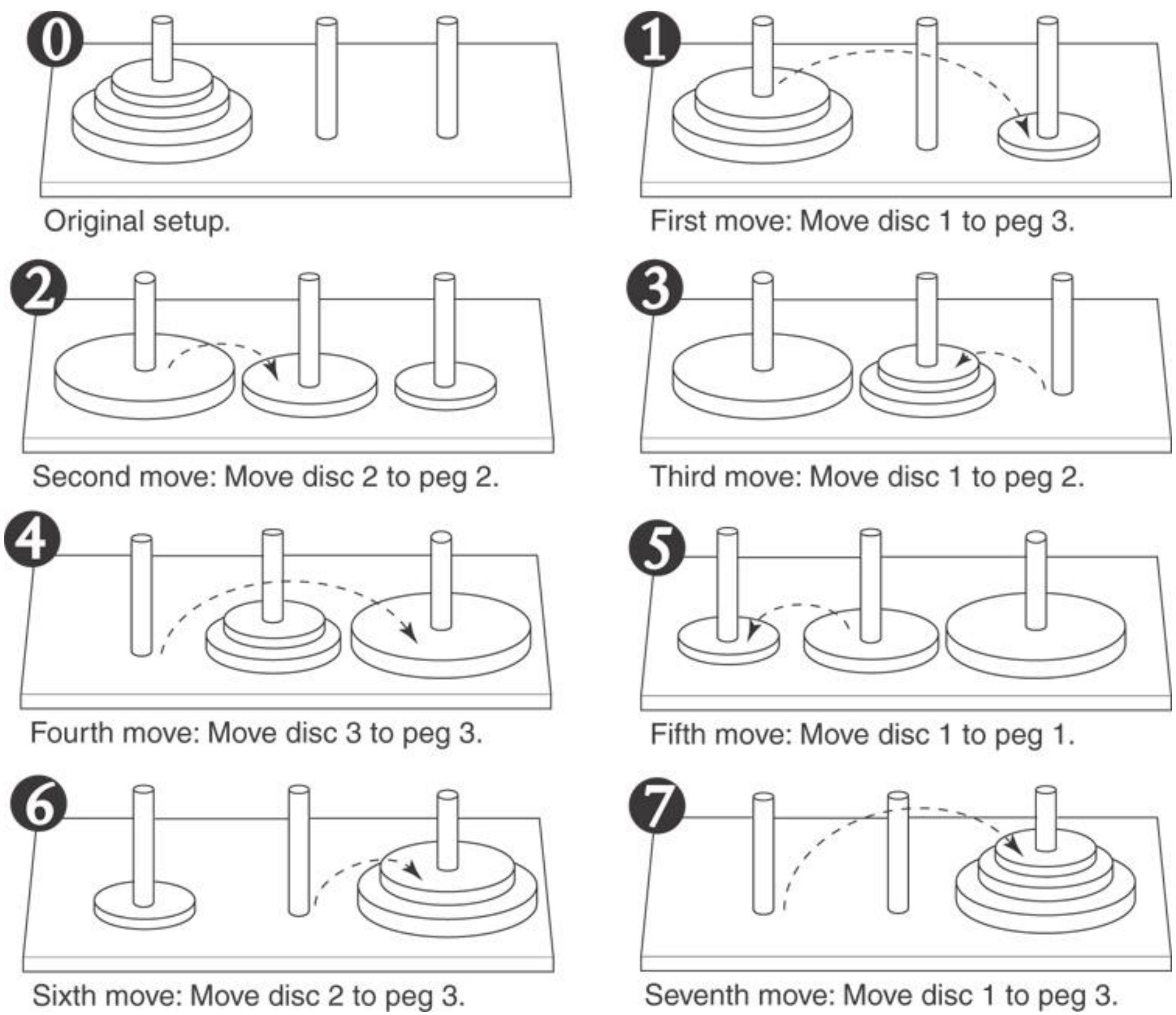
The Towers of Hanoi

- The object of the game is to move the discs from the first peg to the third peg.

  Here are the rules:

  - 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.
- Towers of Hanoi Legend:
  - Monks must move 64 discs. When all the discs have been moved, the world will end.

Moving Three Discs



The Towers of Hanoi

- 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 Towers of Hanoi

- Algorithm:
  - 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

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

```

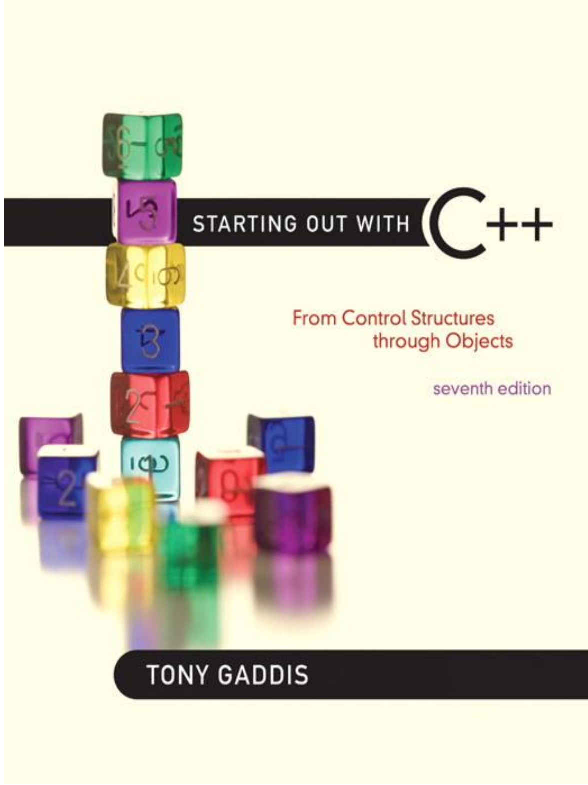
Program 19-10 *(continued)*

```
   // Play the game.
   moveDiscs(NUM_DISCS, FROM_PEG, TO_PEG, TEMP_PEG);
   cout << "All the pegs are moved!\n";
   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 19-10 (Continued)

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!
```



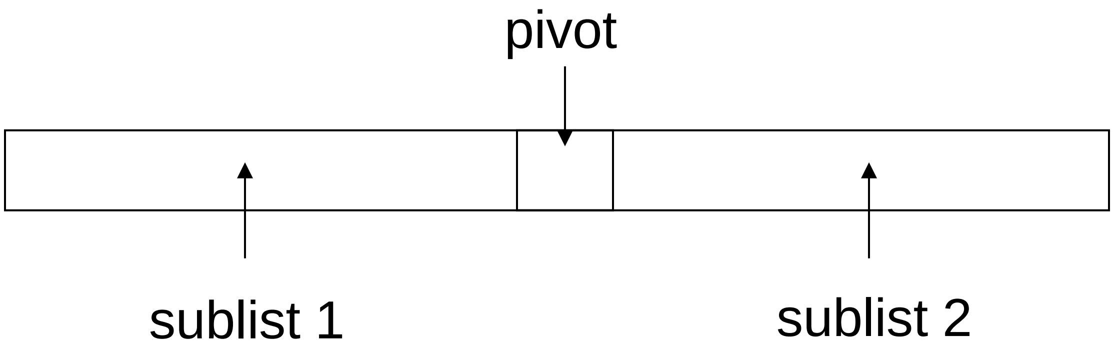
19.8

The QuickSort Algorithm

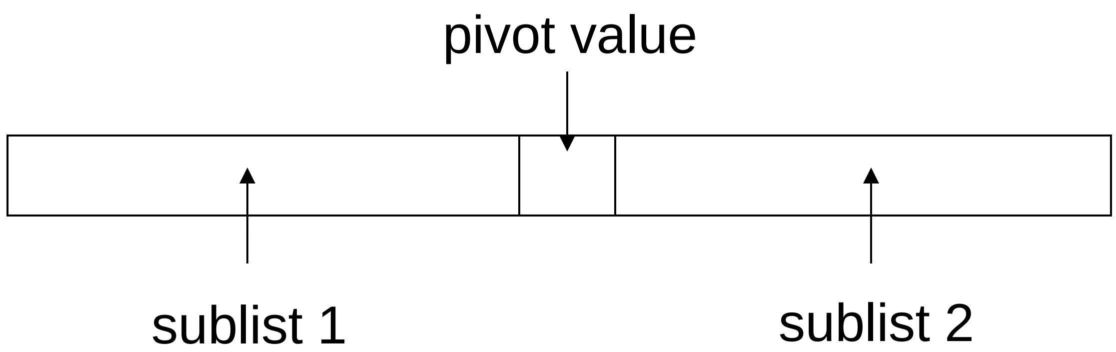
Copyright © 2012 Pearson Education, Inc.

The QuickSort Algorithm

- Recursive algorithm that can sort an array or a linear linked list
- Determines an element/node to use as a pivot value:



The QuickSort Algorithm



- Once pivot value is determined, values are shifted so that elements in sublist1 are < pivot and elements in sublist2 are > pivot
- Algorithm then sorts sublist1 and sublist2
- Base case: sublist has size 1

QuickSort Program

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

const int INPUT_SIZE = 10;

// A simple print function
void print(int *input)
{
    for ( int i = 0; i < INPUT_SIZE; i++ )
        cout << input[i] << " ";
    cout << endl;
}
```

QuickSort Program

```
// The partition function
int partition(int* input, int p, int r)
{
    int pivot = input[r];

    while ( p < r )
    {
        while ( input[p] < pivot )  p++;
        while ( input[r] > pivot )  r--;

        if ( input[p] == input[r] ) p++;
        else if ( p < r )
        {
            int tmp = input[p]; input[p] = input[r]; input[r] = tmp;
        }
    }
    return r;
}
```

QuickSort Program

```
// The quicksort recursive function
void quicksort(int* input, int p, int r)
{
    if ( p < r )
    {
        int j = partition(input, p, r);
        quicksort(input, p, j-1);
        quicksort(input, j+1, r);
    }
}
```

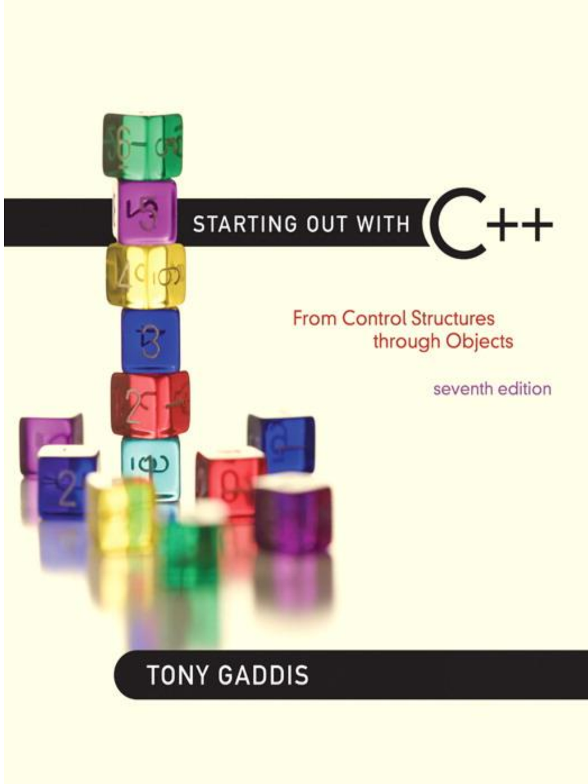
QuickSort Program

```
int main()
{
    int input[INPUT_SIZE] = {500, 700, 800, 100, 300,
                             200, 900, 400, 1000, 600};
    cout << "Input: ";
    print(input);

    quicksort(input, 0, 9);

    cout << "Output: ";
    print(input);
    return 0;
}
```

```
OUTPUT:-
Input: 500 700 800 100 300 200 900 400 1000 600
Output: 100 200 300 400 500 600 700 800 900 1000
```

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

const int INPUT_SIZE = 10;

// A simple print function
void print(int *input)
{
    for ( int i = 0; i < INPUT_SIZE; i++ )
        cout << input[i] << " ";
    cout << endl;
}

// The partition function
int partition(int* input, int p, int r)
{
    int pivot = input[r];

    while ( p < r )
    {
        while ( input[p] < pivot )
            p++;

        while ( input[r] > pivot )
            r--;

        if ( input[p] == input[r] )
            p++;
        else if ( p < r )
        {
            int tmp = input[p];
            input[p] = input[r];
            input[r] = tmp;
        }
    }
    return r;
}

// The quicksort recursive function
void quicksort(int* input, int p, int r)
{
    if ( p < r )
    {
        int j = partition(input, p, r);
        quicksort(input, p, j-1);
        quicksort(input, j+1, r);
    }
}

int main()
{
    int input[INPUT_SIZE] = {500, 700, 800, 100, 300, 200, 900,
400, 1000, 600};
    cout << "Input: ";
    print(input);
    quicksort(input, 0, 9);
    cout << "Output: ";
    print(input);
    return 0;
}
```

```
OUTPUT:-
Input: 500 700 800 100 300 200 900 400 1000 600
Output: 100 200 300 400 500 600 700 800 900 1000
```



19.9

Exhaustive and Enumeration Algorithms

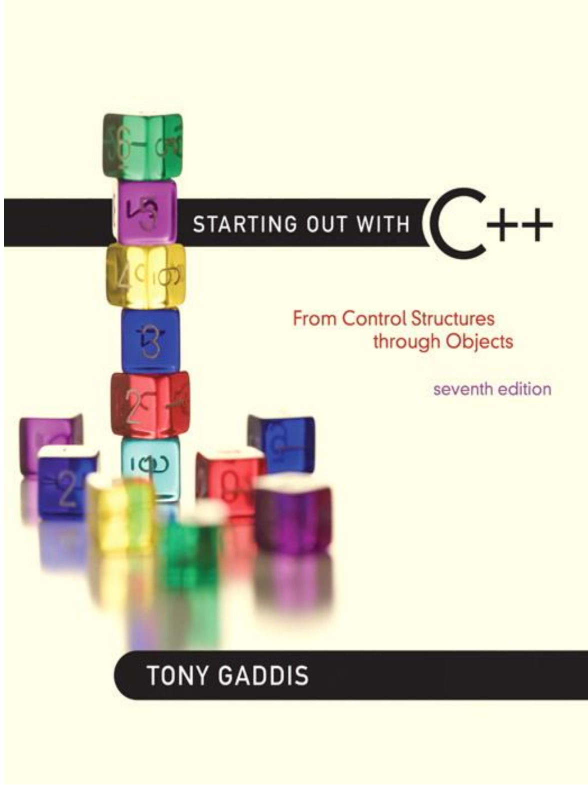
Copyright © 2012 Pearson Education, Inc.

Exhaustive and Enumeration Algorithms

- Exhaustive algorithm: search a set of combinations to find an optimal one

  Example: change for a certain amount of money that uses the fewest coins

- Uses the generation of all possible combinations when determining the optimal one.



19.10

Recursion vs. Iteration

Copyright © 2012 Pearson Education, Inc.

Recursion versus Iteration

- Benefits (+), disadvantages(-) for **recursion**:
  - \+ Models certain algorithms most accurately
  - \+ Results in shorter, simpler functions
  - – May not execute very efficiently
- Benefits (+), disadvantages(-) for **iteration**:
  - \+ Executes more efficiently than recursion
  - – Often is harder to code or understand