Chapter 19:

Recursion
19.1

Introduction to Recursion
Introduction to Recursion

• A **recursive function** contains a call to itself:

```cpp
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
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 is:

```python
if (num == 0)
```
Stopping the Recursion

void countDown(int num)
{
    if (num == 0) // test
        cout << "Blastoff!";
    else
    {
        cout << num << "...
";
        countDown(num-1); // recursive
    }                    // call
}
Stopping the Recursion

• Recursion uses a process of breaking a problem down into smaller problems 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, and the recursion stops
Stopping the Recursion

```cpp
void countDown(int num)
{
    if (num == 0)
        cout << "Blastoff!";
    else
    {
        cout << num << "...
";
        countDown(num - 1); // note that the value
                           // passed to recursive
                           // calls decreases by
                           // one for each call
    }
}
```

Copyright © 2012 Pearson Education, Inc.
What Happens When Called?

• Each time a recursive function is called, a new copy of the function runs, 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?

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

return

output:

2...

1...

Blastoff!
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:
  \[ n! = n \times (n-1) \times (n-2) \times \ldots \times 3 \times 2 \times 1 \text{ if } n > 0 \]
  \[ n! = 1 \text{ if } n = 0 \]

• Can compute factorial of \( n \) if the factorial of \( (n-1) \) is known:
  \[ n! = n \times (n-1)! \]

• \( n = 0 \) is the base case
The Recursive Factorial Function

```c
int factorial (int num)
{
    if (num > 0)
        return num * factorial(num - 1);
    else
        return 1;
}
```
Program 19-3

1 // This program demonstrates a recursive function to
2 // calculate the factorial of a number.
3 #include <iostream>
4 using namespace std;
5
6 // Function prototype
7 int factorial(int);
8
9 int main()
10 {
11    int number;
12
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
Program 19-3 (Continued)

```c
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
The Recursive gcd Function

• Greatest common divisor (gcd) is the largest factor that two integers have in common

• Computed using Euclid's algorithm:
  \[ \text{gcd}(x, y) = y \text{ if } y \text{ divides } x \text{ evenly} \]
  \[ \text{gcd}(x, y) = \text{gcd}(y, x \mod y) \text{ otherwise} \]

• \( \text{gcd}(x, y) = y \) is the base case
int gcd(int x, int y)
{
    if (x % y == 0)
        return y;
    else
        return gcd(y, x % y);
}
19.4

Solving Recursively Defined Problems
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:
  \[ \text{fib}(n) = \text{fib}(n - 1) + \text{fib}(n - 2); \]
• Base cases: \( n \leq 0, \ n == 1 \)
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
Recursive Linked List

Operations

- Recursive functions can be members of a linked list class
- Some applications:
  - Compute the size of (number of nodes in) a list
  - 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 in Chapter 19
The `countNodes` function, a private member function:

```cpp
173  int NumberList::countNodes(ListNode *nodePtr) const
174  {  
175    if (nodePtr != NULL)
176      return 1 + countNodes(nodePtr->next);
177    else
178      return 0;
179  }
```

The `countNodes` function is executed by the public `numNodes` function:

```cpp
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

```cpp
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:

```cpp
void displayBackwards() const
{
    showReverse(head);
}
19.6

A Recursive Binary Search Function
A Recursive Binary Search Function

- Binary search algorithm can easily be written to use recursion
- Base cases: desired value is found, or 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)

```c
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
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.
Moving Three Discs

Original setup.

First move: Move disc 1 to peg 3.

Second move: Move disc 2 to peg 2.

Third move: Move disc 1 to peg 2.

Fourth move: Move disc 3 to peg 3.

Fifth move: Move disc 1 to peg 1.

Sixth move: Move disc 2 to peg 3.

Seventh move: Move disc 1 to peg 3.
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
// 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);

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)

16    // Play the game.
17    moveDiscs(NUM_DISCS, FROM_PEG, TO_PEG, TEMP_PEG);
18    cout << "All the pegs are moved!\n";
19    return 0;
20 }

21
22 /******************************************************************************
23 // The moveDiscs function displays a disc move in the Towers of Hanoi game.
24 // The parameters are:
25 // num: The number of discs to move.
26 // fromPeg: The peg to move from.
27 // toPeg: The peg to move to.
28 // tempPeg: The temporary peg.
29 /******************************************************************************

31 void moveDiscs(int num, int fromPeg, int toPeg, int tempPeg)
32 {
33     if (num > 0)
34     {
35         moveDiscs(num - 1, fromPeg, tempPeg, toPeg);
36         cout << "Move a disc from peg " << fromPeg
37             << " to peg " << toPeg << endl;
38         moveDiscs(num - 1, tempPeg, toPeg, fromPeg);
39     }
40 }

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
The QuickSort Algorithm

- Recursive algorithm that can sort an array or a linear linked list
- Determines an element/node to use as 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
19.9

Exhaustive and Enumeration Algorithms
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
Recursion vs. 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