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!";  // recursive call
    else
    {
        cout << num << "...
        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

Blastoff!

output:

2...

1...

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

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

Second call to `countDown`:
- `num` is 1

Third call to `countDown`:
- `num` is 0
  - // no recursive call

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:
  \[ n! = \begin{cases} 
  n*(n-1)*(n-2)*...*3*2*1 & \text{if } n > 0 \\
  1 & \text{if } n = 0 
  \end{cases} \]

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

• \( n = 0 \) is the base case

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

```cpp
24  // Definition of factorial. A recursive function to calculate *
25  // the factorial of the parameter n. *
26  // ***************************************************************************
27  int factorial(int n)  
28  {  
29      if (n == 0) 
30          return 1;  // Base case
31      else 
32          return n * factorial(n - 1);  // Recursive case
33  }
```

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

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

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

• \[ \text{gcd}(x, y) = y \] is the base case
The Recursive gcd Function

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

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: \( n \leq 0, n == 1 \)

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

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 (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!
The QuickSort Algorithm

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

```
  pivot

  sublist 1      sublist 2
```
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