Data Structures

13. Recursion Review

Recursive Function (1)

- A function is one that calls itself is called recursive function

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

- What is the problem with the above function?
  - No code to stop it from repeating (i.e., calling itself)
  - Function behaves like an infinite loop

Recursive Function – Number of Repetitions

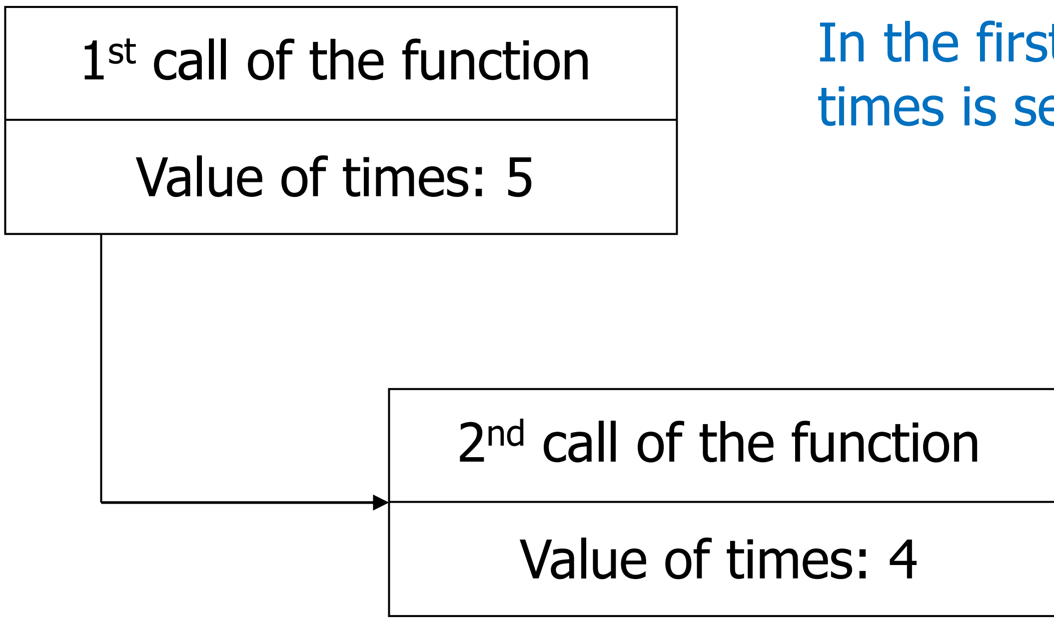
- Recursive function must have some algorithm (i.e., logic) to control the number of times it repeats

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

- Modification to `Message` function
  - Receive an `int` argument to control the number to times to call itself

Recursive Function – Execution (1)

- Each time the function is called, a new instance of the **`times`** parameter is created
  - Suppose program invokes the function as `Message(5)`



In the first call to function, times is set to 5.

When the function calls itself, a new instance of times is created with the value 4.

Recursive Function – Execution (2)

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

1st call of the function
Value of times: 5

2nd call of the function
Value of times: 4

3rd call of the function
Value of times: 3

4th call of the function
Value of times: 2

5th call of the function
Value of times: 1

6th call of the function
Value of times: 0

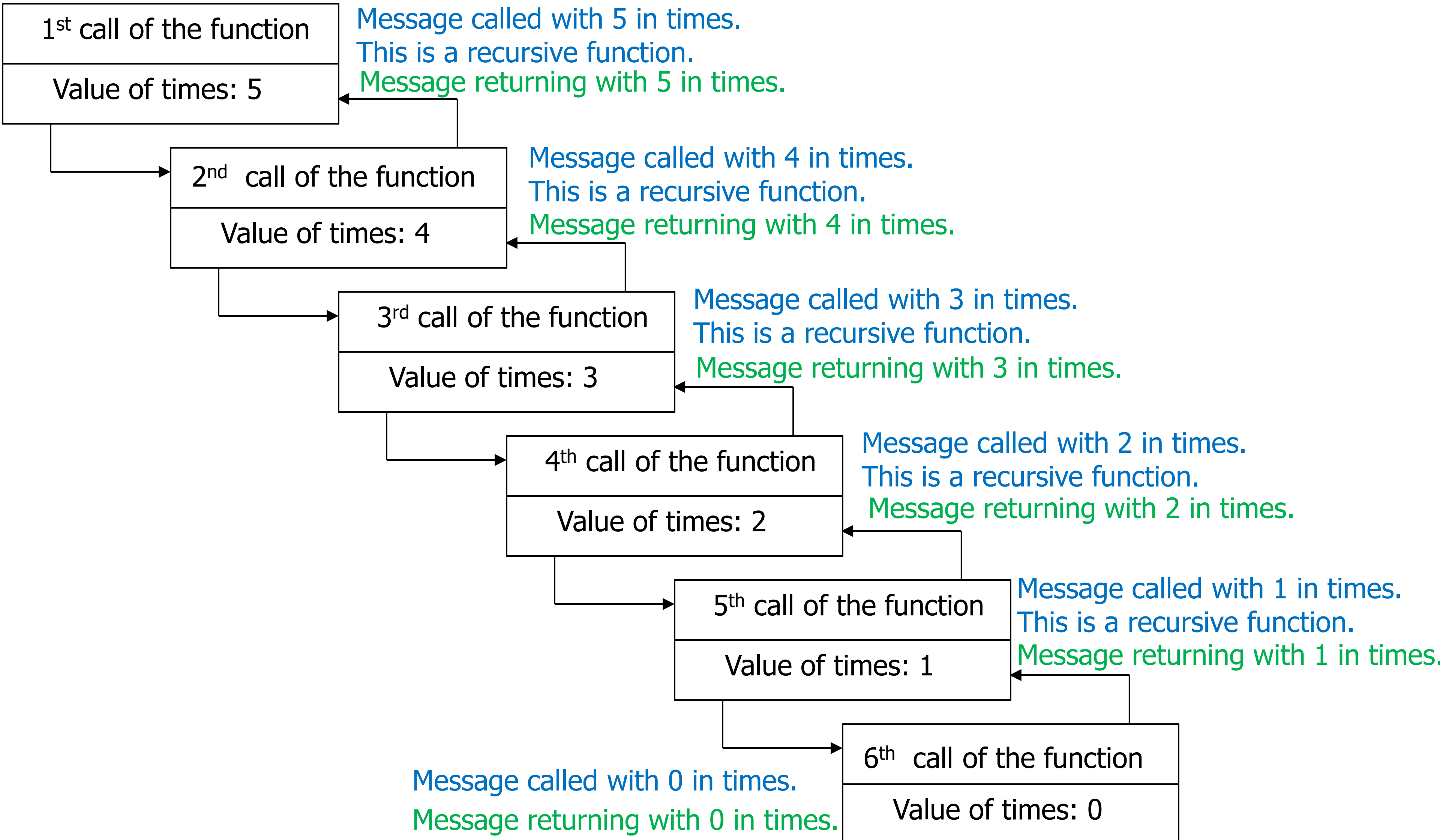
- This cycle repeats itself until 0 is passed to the function
- Depth of recursion: 6

Recursive Function – Modification

- Statements after the recursive invocation of the function

```
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";
}
```

Recursive Function – Execution (3)



Recursion (1)

- Solving a problem by reducing it to a smaller version of itself

- A properly written recursive function must
  - Handle the base case, and
  - Convergence to the base case

- Failure to properly handle the base case or converge to the base case (divergence) may result in infinite recursion

Recursion (2)

- To solve problem recursively

```
1. Define the base case(s)
2. Define the recursive case(s)
    a) Divide the problem into smaller sub-problems
    b) Solve the sub-problems
    c) Combine results to get answer
```

Sub-problems solved as a recursive call to the same function

- Sub-problem must be smaller than the original problem
  - Otherwise recursion never terminates

Example: Binary Search

```
int binarySearch(int array[], int value, int first, int last) {
    int mid = (first + last)/2;

    if (first > last ) return -1;  // Base case
    if (array[mid] == value) {
        return mid;
    }
    else if (array [mid] < value){
        return binarySearch(array, value, mid+1, last);
    }
    else { // last possibility: arrray[mid] > value
        return binarySearch(array, value, first, mid-1);
    }
}
```

Example: Factorial Function (1)

- A mathematical definition: For a non-negative integer n

$$fac(n) = \begin{cases} 1 & if\ n \le 1 \\ n \times fac(n-1) & otherwise \end{cases}$$

- Factorial is defined in terms of itself
  - Defined in cases: a base case and a recursive case

```
public static int fac(int n) {
    if (n <= 1) {
        return 1;
    }
    else {
        return n * fac(n - 1);
    }
}
```

Example: Factorial Function (1)

- Suppose the factorial function is invoked as `fac(5)`

```
fac(5)
5 * fac(4)
5 * 4 * fac(3)
5 * 4 * 3 * fac(2)
5 * 4 * 3 * 2 * fac(1)
5 * 4 * 3 * 2 * 1
5 * 4 * 3 * 2
5 * 4 * 6
5 * 24
120
```

Stack Overflow (1)

- Recursive functions cannot use statically allocated local variables
  - Each instance of the function needs its own copies of local variables

- Most modern languages allocate local variables for functions on the run-time stack

- Calling a recursive function many times or with large arguments may result in stack overflow

```
$ java Fac 10000
Exception in thread "main" java.lang.StackOverflowError
at Fac.facIter(Fac.java:35)
at Fac.facIter(Fac.java:38)
at Fac.facIter(Fac.java:38)
...
```

Stack Overflow (2)

- Three ways to deal with stack overflow:
  - Limit input size (Brittle: How to know limit on a particular machine?)
  - Increase stack size (brittle – how to know how big?)
  - Replace recursion with iteration

```
public static int facIterative(int n) {
    int factorialAccumulator = 1;
    for (int x = n; x > 0; x--) {
        factorialAccumulator *= x;
    }
    return factorialAccumulator;
}
```

- Recursive definitions are often more natural
  - Imperative/iterative definitions often perform better

Example: Linked List Operations

- Many operations on linked lists may be implemented by using recursion

- We will discuss two functions:
  - Counting the number of nodes in a list
  - Displaying the value of the list nodes in reverse order

Counting Nodes in The List

- The function's recursive logic can be expressed as

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

- What is the base case?
  - `nodePtr` being equal to `NULL`

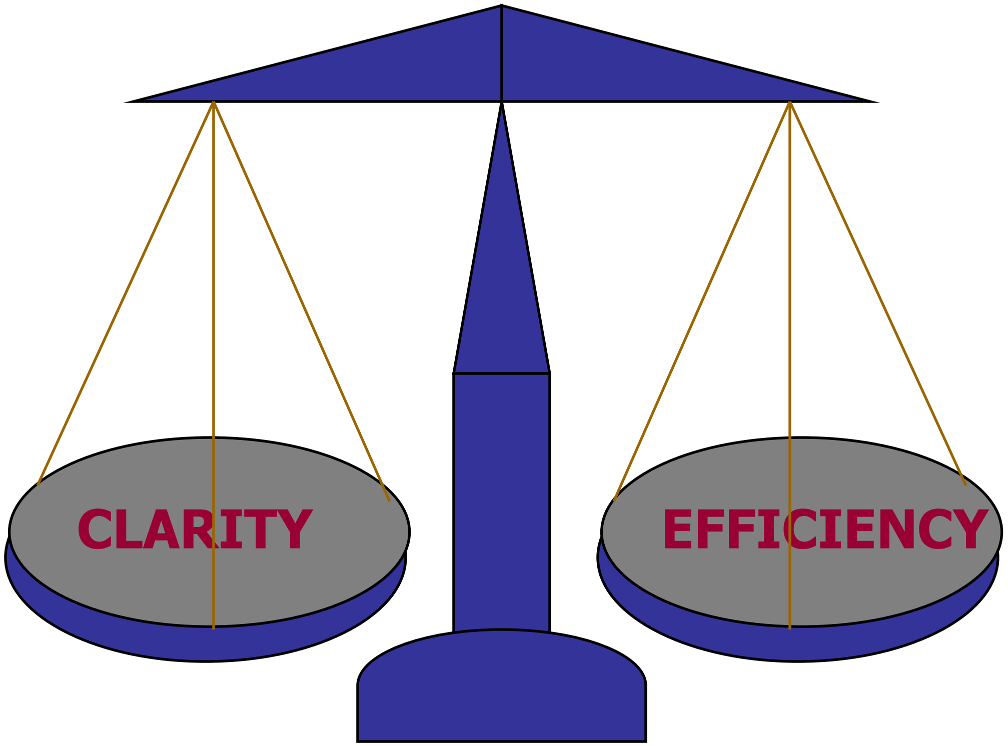
Displaying List Nodes in Reverse Order

- The function's recursive logic can be expressed as

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

- The base case for the function is `nodePtr` being equal to `NULL`

Recursion or Iteration?



Any Question So Far?