Data Structures
Lab Manual 7

Topic: Recursion

Session: Spring 2023

Faculty of Information Technology

UCP Lahore Pakistan

Objectives:

The objective of learning recursion is to utilize a powerful problem-solving technique. Recursion involves solving a problem by breaking it down into smaller, similar subproblems.

The key objectives of learning recursion are:

1. Problem decomposition: Recursion allows you to break down complex problems into smaller, more manageable subproblems. This can help simplify the overall problem-solving process.
2. Clearer and concise code: Recursion can often lead to more elegant and concise code compared to iterative approaches. It allows you to express the problem-solving logic in a more natural and intuitive way.
3. Solving recursive data structures: Many data structures, such as trees and linked lists, naturally lend themselves to recursive solutions. Understanding recursion enables you to navigate and manipulate these data structures effectively.
4. Understanding divide-and-conquer algorithms: Recursion is a fundamental technique in divide-and-conquer algorithms, where a problem is divided into smaller subproblems that are solved independently and then combined to obtain the final solution.
5. Efficiency and time complexity analysis: Recursion can be used to solve problems with significantly lower time complexity than iterative approaches in certain cases. By understanding recursion, you can analyze the time and space complexity of recursive algorithms and optimize them when needed.

Overall, learning recursion equips you with a valuable problem-solving tool and helps you develop a deeper understanding of various data structures and algorithms. It enables you to solve complex problems efficiently and elegantly.

Recursion:

Recursion is a programming technique where a function calls itself to solve a problem by breaking it down into smaller, similar subproblems. In other words, a function is defined in terms of itself. Here's a step-by-step explanation of how recursion works:

1. Base Case: Recursive functions have a base case, which is the simplest form of the problem that can be solved directly without further recursion. When the base case is reached, the recursion stops, and the function starts returning values back to the previous calls.
2. Recursive Call: In the recursive function, there is a call to itself, typically with a smaller or simpler input. This recursive call breaks down the original problem into smaller subproblems. Each recursive call operates on a smaller portion of the problem until the base case is reached.
3. Subproblem Solving: Each recursive call works on a smaller subproblem. The recursive function assumes that it can solve these subproblems correctly by making the recursive

calls. The solutions to the subproblems are combined or processed further to obtain the solution to the original problem.

4. Backtracking and Return: Once the base case is reached, the recursion starts unwinding. The function returns the results of each recursive call back to the previous call. This process continues until the original call is reached, and the final result is obtained.

It's important to ensure that recursive functions are designed to converge towards the base case, meaning that each recursive call brings the problem closer to the base case. Without proper termination conditions or convergence, recursion can lead to infinite loops or stack overflow errors.

Recursion can be a powerful and expressive technique for solving problems, especially those that exhibit self-similarity or can be divided into smaller subproblems. However, it's important to use recursion judiciously and understand its limitations, such as potential performance issues when dealing with large problem sizes.

Types of Recursion:

Direct recursion: When a function is called within itself directly it is called direct recursion. This can be further categorized into four types:

- Tail recursion
- Head recursion
- Tree recursion
- Nested recursion

Indirect recursion: Indirect recursion occurs when a function calls another function that eventually calls the original function and it forms a cycle.

Applications of Recursion:

Recursion is used in many fields of computer science and mathematics, which includes:

- Searching and sorting algorithms: Recursive algorithms are used to search and sort data structures like trees and graphs.
- Mathematical calculations: Recursive algorithms are used to solve problems such as factorial, Fibonacci sequence, etc.
- Compiler design: Recursion is used in the design of compilers to parse and analyze programming languages.
- Graphics: many computer graphics algorithms, such as fractals and the Mandelbrot set, use recursion to generate complex patterns.
- Artificial intelligence: recursive neural networks are used in natural language processing, computer vision, and other AI applications.

Advantages of Recursion:

- Recursion can simplify complex problems by breaking them down into smaller, more manageable pieces.
- Recursive code can be more readable and easier to understand than iterative code.
- Recursion is essential for some algorithms and data structures.
- Also with recursion, we can reduce the length of code and become more readable and understandable to the user/ programmer.

Disadvantages of Recursion:

- Recursion can be less efficient than iterative solutions in terms of memory and performance.
- Recursive functions can be more challenging to debug and understand than iterative solutions.
- Recursion can lead to stack overflow errors if the recursion depth is too high.

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

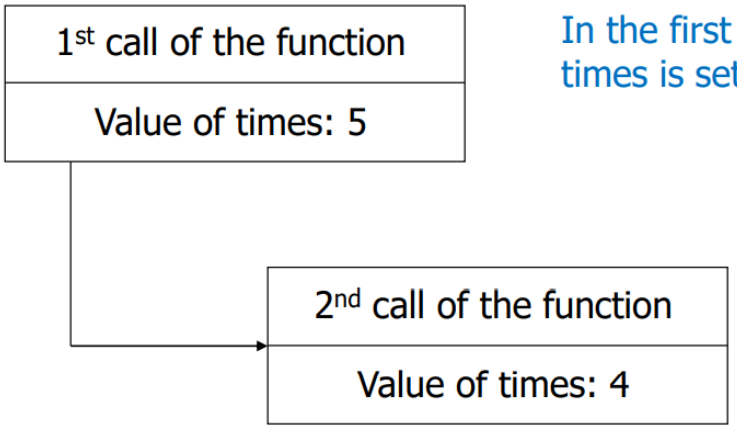
- Recursive function must have some algorithm (i.e., logic) to control the number of times it repeats

```
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.

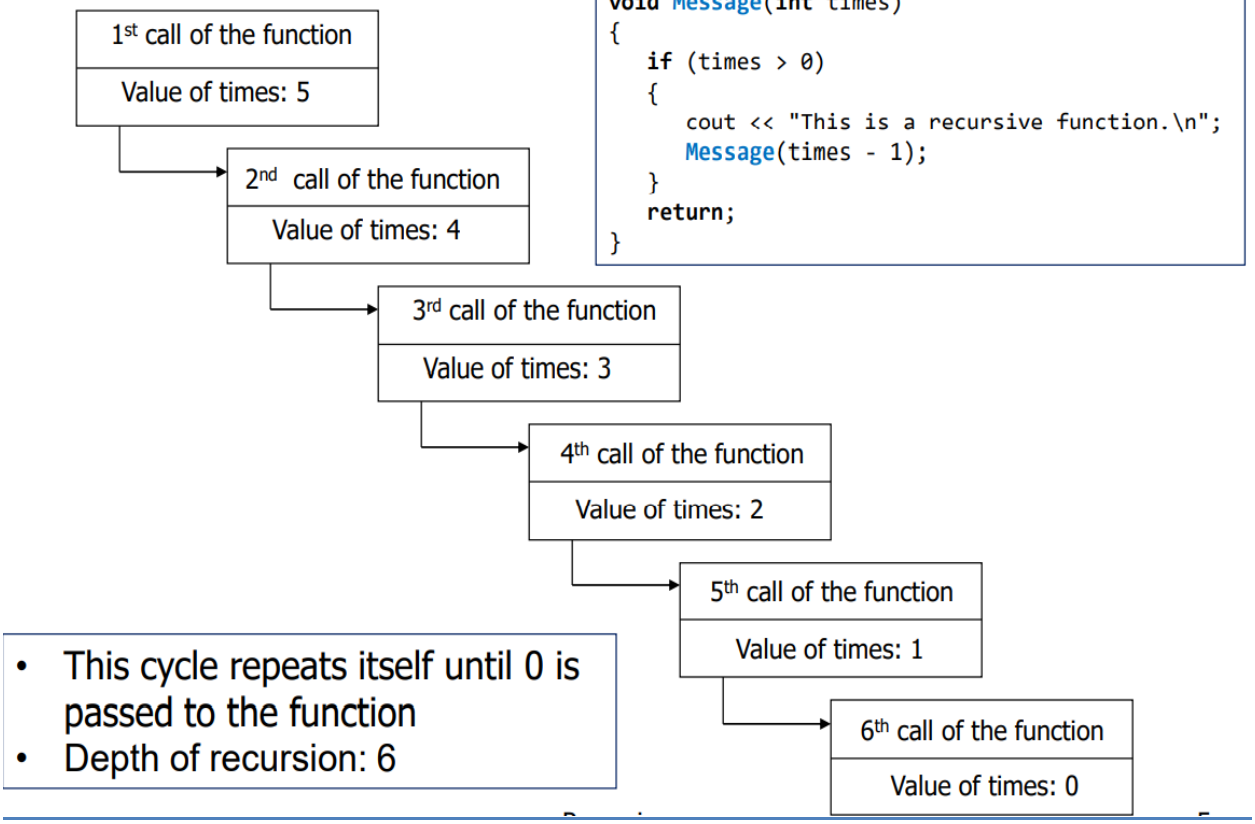
When the function calls itself, a new instance of times is created with the value 4.

Recursive Function – 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";
}
```

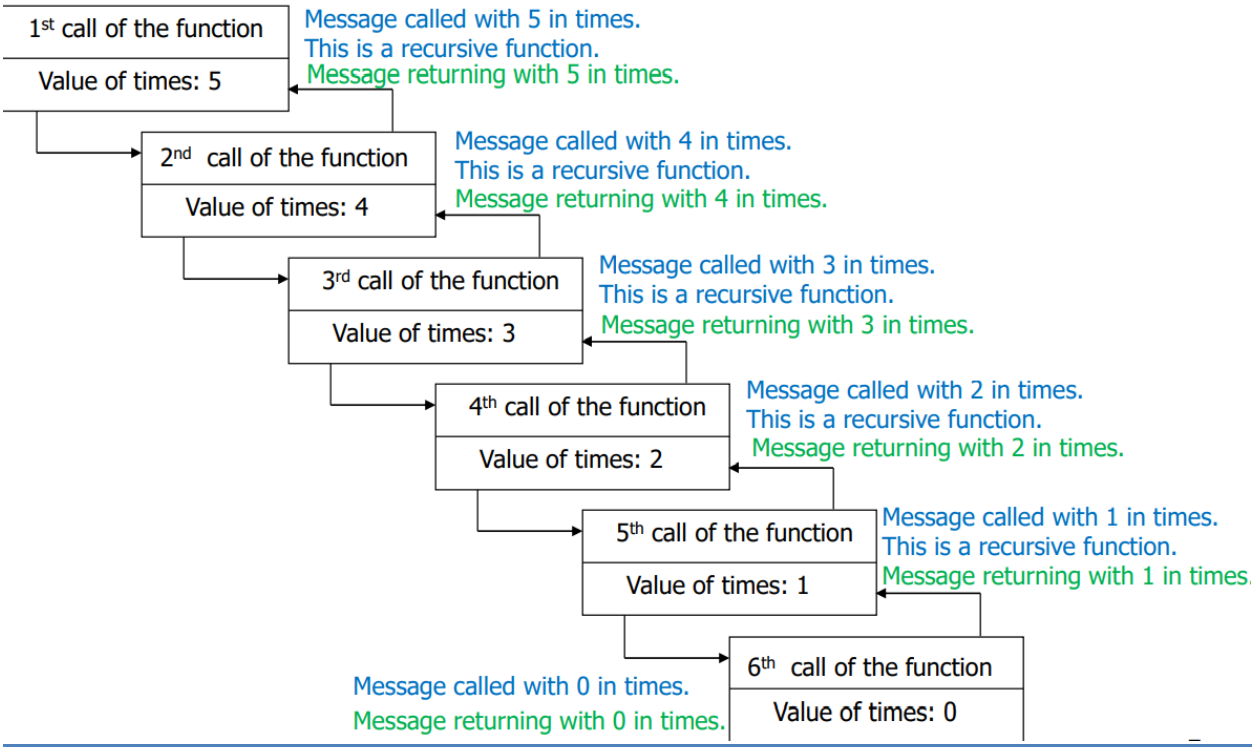
Recursive Function – Execution (2)




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

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

Recursive Function – Execution (3)



Instructions:

- Indent your code.
- Comment your code.
- Use meaningful variable names.
- Plan your code carefully on a piece of paper before you implement it.
- Name of the program should be same as the task name. i.e. the first program should be Task_1.cpp
- **void main() is not allowed. Use int main()**
- **You are not allowed to use any built-in functions**
- **You are required to follow the naming conventions as follow:**
    - **Variables:** firstName; (no underscores allowed)
    - **Function:** getName(); (no underscores allowed)
    - **ClassName:** BankAccount (no underscores allowed)

Students are required to complete the following tasks in lab timings.

Task 1

Write a recursive function which takes head pointer of singly linked list as parameter and prints the singly linked list in reverse order. Restriction: You are not allowed to use any other data structure in the function.

Task 2

Write a program that computes the sum of first N natural numbers. For example, if a user enters 5 as input, the program should display 15 as output (since 5+4+3+2+1 = 15). Your function should be recursive.

Task 3

The least common multiple (LCM) of two numbers is the smallest number that is a multiple of both. Write and test a recursive method LCM with the following specification.

PARAMETERS: positive integers j and k

RETURN VALUE: the least common multiple (LCM) of j and k

EXAMPLES:
LCM (3, 5) is 15
LCM (6, 8) is 24