



# Data structures

## Recursion

Lecture No. 11

# Today's Lecture

In this lecture we will:

- study recursion. **Recursion** is a programming technique in which procedures and functions call themselves.
- look at the stack — a structure maintained by each program at run time.

# Why Use Recursion

- Recursively defined data structures, like lists, are very well-suited to processing by recursive procedures and functions.
- A recursive procedure is mathematically more elegant than one using loops → Easy to prove its correctness.
- Sometimes procedures that would be tricky to write using a loop are straightforward using recursion.

# Introduction

- In C++, any function can call another function.
- A function can even call itself.
- When a function call itself, it is making a recursive call.
- **Recursive Call**
  - **A function call in which the function being called is the same as the one making the call.**
- Recursion is a powerful technique that can be used in the place of iteration(looping).
- **Recursion**
  - **Recursion is a programming technique in which procedures and functions call themselves.**

# Recursive Definition

- **A definition in which something is defined in terms of smaller versions of itself.**
- To do recursion we should know the followings
  - **Base Case:**
    - The case for which the solution can be stated non-recursively
    - The case for which the answer is explicitly known.
    - Always have at least one case that can be solved without recursion.
  - **General Case:**
    - The case for which the solution is expressed in smaller version of itself. Also known as recursive case.

# Example I

- We can write a function called **power** that calculates the result of raising an integer to a positive power. If  $X$  is an integer and  $N$  is a positive integer, the formula for is

$$X^N = \underbrace{X * X * X * X * X * \dots * X}_{N - \text{times}}$$

- We can also write this formula as

$$X^N = X * \underbrace{X * X * X * X * \dots * X}_{(N - 1) - \text{times}}$$

- Also as

$$X^N = X * X * \underbrace{X * X * X * \dots * X}_{(N - 2) - \text{times}}$$

- In fact we can write this formula as

$$X^N = X * X^{N-1}$$

# Example 1 (Recursive Power Function)

- Now let's suppose that  $X=3$  and  $N=4$

$$X^N = 3^4$$

- Now we can simplify the above equation as

$$3^4 = 3 * 3^3$$

$$3^3 = 3 * 3^2$$

$$3^2 = 3 * 3^1$$

- So the base case in this equation is

$$3^1 = 3$$

```
int Power ( int  x ,int  n )
{
    if ( n == 1 )
        return x;    //Base case
    else
        return x * Power (x, n-1);
                        // recursive call
}
```

# Factorial-- A Case Study

- ❑ Factorial Function: Given a positive integer  $n$ ,  $n$  factorial is defined as the product of all integers between 1 and  $n$ , including  $n$ .
- ❑ Definition:  $n! = n * (n-1) * (n-2) * \dots * 1$
- ❑ Mathematical Definition:

$$f(n) = \begin{cases} 1 & , \text{for } n = 1 \\ n * f(n-1) & , \text{for } n > 1 \end{cases}$$

- ❑ Implementation using loop

```
int FactorialLoop (int N) {  
    int result=1;  
    for (i=1; i<=N; i++)  
        result=result*i;  
    return result;  
}
```



# Factorial Case Study

- Factorial definition

$$n! = n \times n-1 \times n-2 \times n-3 \times \dots \times 3 \times 2 \times 1$$

$$0! = 1$$

- To calculate factorial of n

- Base case

- If  $n = 0$ , return 1

- Recursive step

- Calculate the factorial of  $n-1$
    - Return  $n \times$  (the factorial of  $n-1$ )

# Factorial Case Study

- ❑ Here's a function that computes the factorial of a number  $N$  without using a loop.
- ❑ It checks whether  $N$  is smaller than 1. If so, the function just returns 1.
- ❑ Otherwise, it computes the factorial of  $N - 1$  and multiplies it by  $N$ .

```
int factorial(n)
{
    if(n <= 1) //Base Case
        return 1;
    else
        return n * factorial(n-1);
    //Recursion
}
```

# Evaluation of Factorial Example

To evaluate Factorial(3)

evaluate  $3 * \text{Factorial}(2)$

To evaluate Factorial(2)

evaluate  $2 * \text{Factorial}(1)$

To evaluate Factorial(1)

evaluate  $1 * \text{Factorial}(0)$

Factorial(0) is 1

Return 1

Evaluate  $1 * 1$

Return 1

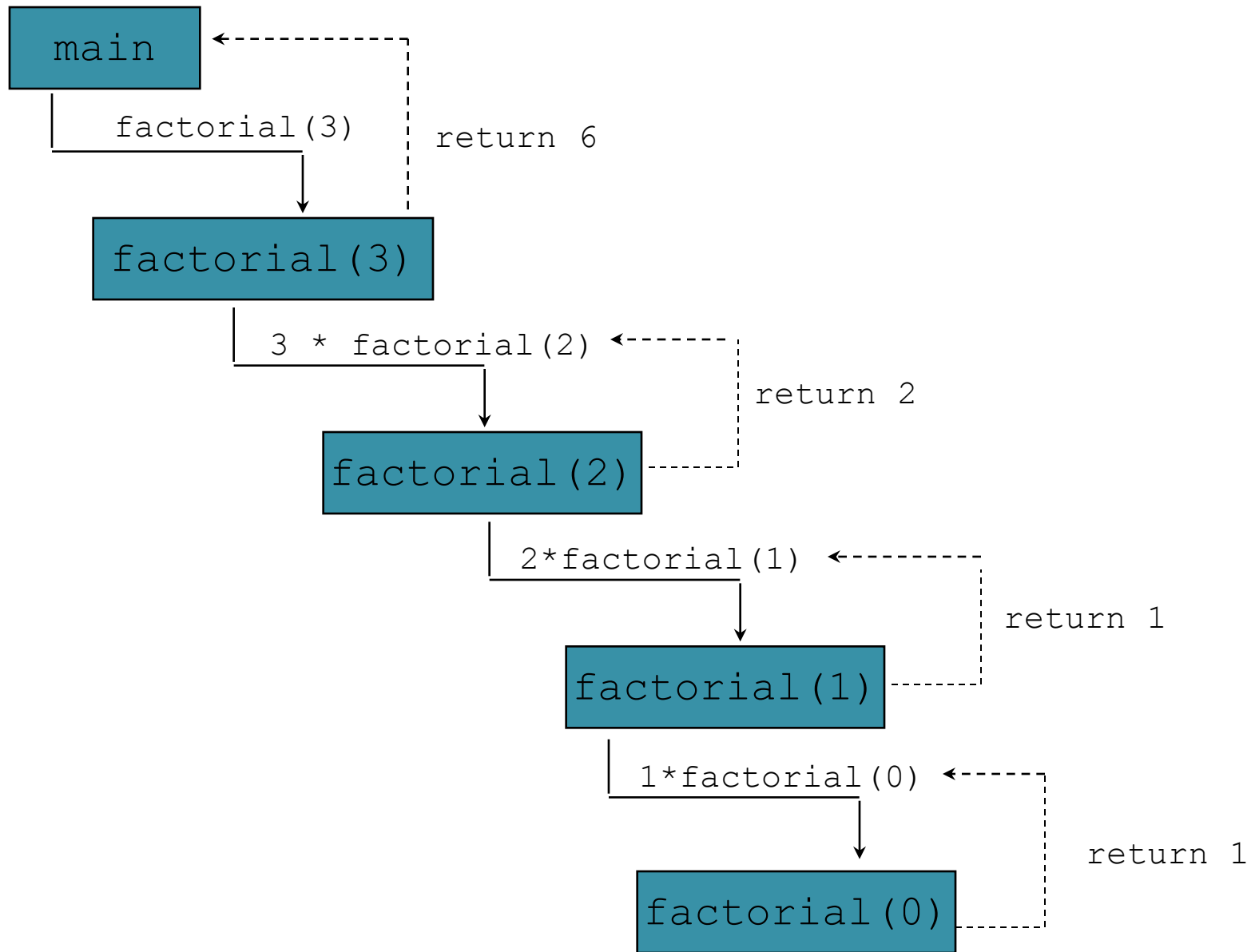
Evaluate  $2 * 1$

Return 2

Evaluate  $3 * 2$

Return 6

# Recursive Programming



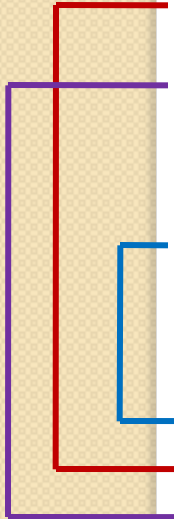
# Another Example-- I

- Here is a function to print all the elements of a linked list.
- `getFromList()` extracts one item from the head of the list and then return its value. In addition, it also return a new pointer pointing to rest of the list NOT include the returned item.
- `printAll()` displays on the console ALL the items stored in the entire linked list. It terminates when the next pointer is pointing to NULL (i.e. empty).

```
int getFromList (Nodeptr &hdList)  
{  
    int number;  
    number = hdList→data;  
    hdList = List→next;  
    return (number)  
}  
Void printAll (Nodeptr hdList)  
{  
    cout<<"List of numbers  
are:";  
    while (hdList != NULL)  
    {  
        cout<<getFromList(hdList);  
    }  
}
```

# Another Example--2

- The function below prints out the numbers in a list of numbers. Unlike the version of printAll in previous slide, it doesn't use a loop. This is how it works.
  - It prints out the first number in the list.
  - Then it calls itself to print out the rest of the list.
  - Each time it calls itself, the list is a bit shorter.
  - Eventually we reach an empty list, and the whole process terminates.



```
void printAll (NodePtr hdList) {  
    if (hdList != NULL) {  
        cout << getFromList(hdList) << endl;  
        printAll(hdList);  
    }  
}
```

# Visualization

```
void printAll (NodePtr hdList) {  
    if (hdList != NULL) {  
        cout << getFromList(hdList) << endl;  
        printAll(hdList);  
    }  
}  
  
void printAll (NodePtr hdList) {  
    if (hdList != NULL) {  
        cout << getFromList(hdList) << endl;  
        printAll(hdList);  
    }  
}
```

The diagram illustrates the recursive calls of the `printAll` function. Red arrows show the flow of execution: from the `printAll(hdList);` line in the first function call to the start of the second function call, and from the `printAll(hdList);` line in the second function call to the start of a third function call. This visualizes the call stack and the return path of the recursive calls.

# Rules For Recursive Function

1. In recursion, it is essential for a function to call itself, otherwise recursion will not take place.
2. Only user define function can be involved in recursion.
3. To stop the recursive function it is necessary to base the recursion on test condition and proper terminating statement such as *exit()* or *return* must be written using *if()* statement.
4. When a recursive function is executed, the recursive calls are not implemented instantly. All the recursive calls are pushed onto the stack until the terminating condition is not detected, the recursive calls stored in the stack are popped and executed.
5. During recursion, at each recursive call new memory is allocated to all the local variables of the recursive functions with the same name.



# The Runtime Stack during Recursion

- To understand how recursion works at run time, we need to understand what happens when a function is called.
- Whenever a function is called, a block of memory is allocated to it in a run-time structure called the **stack**.
- This block of memory will contain
  - the function's **local variables**,
  - local copies of the function's **call-by-value parameters**,
  - pointers to its **call-by-reference parameters**, and
  - a **return address**, in other words where in the program the function was called from. When the function finishes, the program will continue to execute from that point.

# Brain Storming!!!

- Printing the list out backwards using a loop is much harder.
- It is possible to print the list from tail first with recursion???
- **It is identical to the printAll program, except that the two lines inside the if statement are the other way around.**

```
void printAll (NodePtr hdList) {  
    if (hdList != NULL) {  
        printAll(hdList->next);  
        cout << getFromList(hdList) << endl;  
    }  
}
```

# Recursion: Final Remarks

- The trick with recursion is to ensure that each recursive call gets closer to a base case. In most of the examples we've looked at, the base case is the empty list, and the list gets shorter with each successive call.
- Recursion can always be used instead of a loop. (This is a mathematical fact.) In declarative programming languages, like Prolog, there are no loops. There is only recursion.
- Recursion is elegant and sometimes very handy, but it is marginally less efficient than a loop, because of the overhead associated with maintaining the stack.

# Exercise

The problem of computing the sum of all the numbers between 1 and any positive integer N can be recursively defined as:

$$\sum_{i=1}^N = N + \sum_{i=1}^{N-1} = N + (N-1) + \sum_{i=1}^{N-2}$$

= etc.

# Exercise

```
int sum(int n)
{
    if(n==1)
        return n;
    else
        return n + sum(n-1);
}
```