**Applications of Stack**

In this lecture, the following topics are covered:

* Applications of stack
* **Reversing a List**  problem
* Recursion

1. **Applications of Stack**

In this section we will discuss typical problems where stacks can be easily applied for a simple and efficient solution.

The following are some of the problems where the stack can be applied:

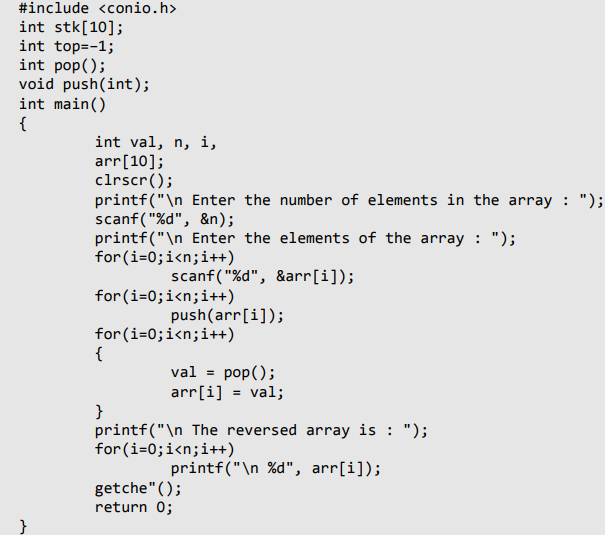
* Reversing a list
* Parentheses checker
* Conversion of an infix expression into a postfix expression
* Evaluation of a postfix expression
* Conversion of an infix expression into a prefix expression
* Evaluation of a prefix expression
* Recursion

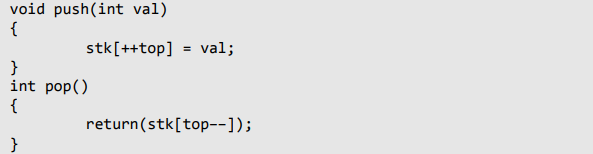
However, only reversing a list and recursion are introduced here.

1. **Reversing a List**

A list of numbers can be reversed by reading each number from an array starting from the first index and pushing it on a stack. Once all the numbers have been read, the numbers can be popped one at a time and then stored in the array starting from the first index. The following program shows how to use the stack data structure to reverse an **n** integer numbers stored in an array. Notice that pushing first these numbers into the stack and then popping them out can accomplish this task.

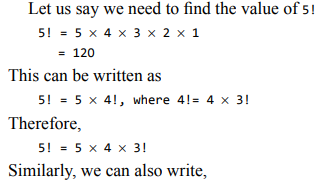


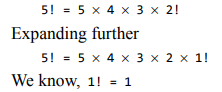




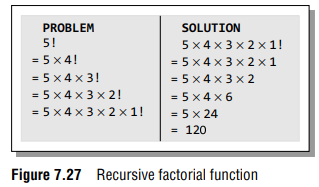
1. **Recursion**

* Recursion is an implicit application of the stack.
* A recursive function is defined as a function that calls itself to solve a smaller version of its task until a final call is made which does not require a call to itself.
* Since a recursive function repeatedly calls itself, it makes use of the system stack to temporarily store the return address and local variables of the calling function.
* Every recursive solution has two major cases. They are:
  + **Base case**, in which the problem is simple enough to be solved directly without making any further calls to the same function.
  + **Recursive case**, in which first the problem at hand is divided into simpler sub-parts. Second, the function calls itself but with sub-parts of the problem obtained in the first step. Third, the result is obtained by combining the solutions of simpler sub-parts.
  + Therefore, recursion is defining large and complex problems in terms of smaller and more easily solvable problems. In recursive functions, a complex problem is defined in terms of simpler problems and the simplest problem is given explicitly.
* To understand recursive functions, let us take an example of calculating factorial of a number. To calculate **n!**, we multiply the number with factorial of the number that is 1 less than that number. In other words, n**!** = n × (n–1)**!.**

****

****

The series of problems and solutions can be given as shown in Fig. 7.27.

.

Now if you look at the problem carefully, you can see that we can write a recursive function to calculate the factorial of a number.

Every recursive function must have a base case and a recursive case. For the **factorial function**:

* **Base case** is when n = 1, because if n = 1, the result will be 1 as 1! = 1.
* **Recursive case** of the factorial function will call itself but with a smaller value of **n**. This case can be given as:



Look at the following program which calculates the factorial of a number recursively

