**STACK**

Stacks are dynamic data structures that follow the **Last In First Out (LIFO)** principle. The last item to be inserted into a stack is the first one to be deleted from it.

For example, you have a stack of trays on a table. The tray at the top of the stack is the first item to be moved if you require a tray from that stack.

Stack mainly consist operations that is:-

1. PUSH() for inserting
2. POP() for deleting.
3. isfull() for checking that stack is full or not.
4. isempty() for checking that stack is empty or not.
5. peek() to get top most element of the stack.
6. size() is to get the number of elements present on the stack.

All Operating take o(1) time complexity.

Features of stack-:

1. Dyanamic Data structure
2. no fixed size
3. do not consume the fixed amount of memory.
4. size of stack increases by push() and pop() operation .

Implementation

#include<iostream>

#include<stack>

using namespace std;

main(){

   stack<int> stk;

   if(stk.empty()){

      cout << "Stack is empty" << endl;

   } else {

      cout << "Stack is not empty" << endl;

   }

   //insert elements into stack

   stk.push(10);

   stk.push(20);

   stk.push(30);

   stk.push(40);

   stk.push(50);

   cout << "Size of the stack: " << stk.size() << endl;

   //pop and dispay elements

   while(!stk.empty()) {

      int item = stk.top(); // same as peek operation

      stk.pop();

      cout << item << " ";

   }

}

output

Stack is empty

Size of the stack: 5

50 40 30 20 10

Applications

1. conversion of infix to postfix expression
2. balancing of parentheses.
3. in graph algorithms like topological sorting.
4. in algorithms like tower of Hanoi ,stock span problem,etc

**CONVERSION OF INFIX TO POSTFIX EXPRESSION**

**Infix expression:**The expression of the form a op b. When an operator is in-between every pair of operands.

**Postfix expression:**The expression of the form a b op. When an operator is followed for every pair of operands.

The compiler scans the expression either from left to right or from right to left.

**Algorithm**  
**1.** Scan the infix expression from left to right.  
**2.** If the scanned character is an operand, output it.  
**3.**Else,  
…..**3.1** If the precedence of the scanned operator is greater than the precedence of the operator in the stack(or the stack is empty or the stack contains a ‘(‘ ), push it.  
…..**3.2** Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)  
**4.** If the scanned character is an ‘(‘, push it to the stack.  
**5.** If the scanned character is an ‘)’, pop the stack and and output it until a ‘(‘ is encountered, and discard both the parenthesis.  
**6.** Repeat steps 2-6 until infix expression is scanned.  
**7.** Print the output  
**8.**Pop and output from the stack until it is not empty.

**EVALUATION OF POSTFIX EXPRESSION**

Following is algorithm for evaluation postfix expressions.  
1) Create a stack to store operands (or values).  
2) Scan the given expression and do following for every scanned element.  
…..a) If the element is a number, push it into the stack  
…..b) If the element is a operator, pop operands for the operator from stack. Evaluate the operator and push the result back to the stack  
3) When the expression is ended, the number in the stack is the final answer

**RECURSION AND STACKS**

Recursion is a technique in which we try to express a given problem in terms of itself.

Like suppose we had done a program of factorial using recusion i.e. n\*fact(n-1).

Every time the function calls itself, a copy of it is created and pushed onto the stack and this keeps on going until the condition for breaking out of recursion is met or the stack memory is full.

**why we require recursion?**

solution to the problem more intuitive to solve and understand .

If each function call of recursive algorithm takes O(m) space and if the maximum depth of recursion tree is 'n' then space complexity of recursive algorithm would be O(nm) and time complexity will be 2^n which is very high and instead of this we use memorization method.