C++ program for expression conversion as infix to postfix and its evaluation using stack

based on given conditions

i. Operands and operator,both must be single character.

ii. Input Postfix expression must be in a desired format.

iii. Only '+', '-', '\*' and '/ ' operators are expected.

InfixToPostfix.cpp

#include<iostream>

#include<cstring>

#include<stack>

using namespace std;

// get weight of operators as per precedence

// higher weight given to operators with higher precedence

// for non operators, return 0

int getWeight(char ch) {

switch (ch) {

case '/':

case '\*': return 2;

case '+':

case '-': return 1;

default : return 0;

}

}

// convert infix expression to postfix using a stack

void infix2postfix(char infix[], char postfix[], int size) {

stack<char> s;

int weight;

int i = 0;

int k = 0;

char ch;

// iterate over the infix expression

while (i < size) {

ch = infix[i];

if (ch == '(') {

// simply push the opening parenthesis

s.push(ch);

i++;

continue;

}

if (ch == ')') {

// if we see a closing parenthesis,

// pop of all the elements and append it to

// the postfix expression till we encounter

// a opening parenthesis

while (!s.empty() && s.top() != '(') {

postfix[k++] = s.top();

s.pop();

}

// pop off the opening parenthesis also

if (!s.empty()) {

s.pop();

}

i++;

continue;

}

weight = getWeight(ch);

if (weight == 0) {

// we saw an operand

// simply append it to postfix expression

postfix[k++] = ch;

}

else {

// we saw an operator

if (s.empty()) {

// simply push the operator onto stack if

// stack is empty

s.push(ch);

}

else {

// pop of all the operators from the stack and

// append it to the postfix expression till we

// see an operator with a lower precedence that

// the current operator

while (!s.empty() && s.top() != '(' &&

weight <= getWeight(s.top())) {

postfix[k++] = s.top();

s.pop();

}

// push the current operator onto stack

s.push(ch);

}

}

i++;

}

// pop of the remaining operators present in the stack

// and append it to postfix expression

while (!s.empty()) {

postfix[k++] = s.top();

s.pop();

}

postfix[k] = 0; // null terminate the postfix expression

}

// main

int main() {

char infix[100];//"A\*(B+C)/D";

cout<<"\nENter Infix Operation:";

cin>>infix;

int size = strlen(infix);

char postfix[size];

infix2postfix(infix,postfix,size);

cout<<"\nInfix Expression :: "<<infix;

cout<<"\nPostfix Expression :: "<<postfix;

cout<<endl;

return 0;

}

ENter Infix Operation:(c+d)\*bc

Infix Expression :: (c+d)\*bc

Postfix Expression :: cd+bc\*

**Title**: Infix to Postfix Expression conversion and its evaluation using stack.

**Objectives:**

* To study Stack as a Data Structure.
* To understand implementation of Stack and perform various operations on it.
* To convert Infix Expression into Postfix using stack.

**Problem Statement:**

Implement C++ program for expression conversion as infix to postfix and its evaluation using stack based on given conditions

1. Operands and operator, both must be single character.
2. Input Postfix expression must be in a desired format.

* Only ‘+’, ‘-‘, ‘\*’ and ‘/ ‘ operators are expected.

**Outcomes:**

On completion of this assignment students will be able to-

* Implement Stack and perform various basic operations on it.
* Solve the real world problem of Infix to Postfix expression conversion and evaluation of postfix expression using stack.

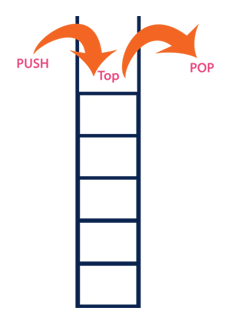
**Software & Hardware requirements:**

* Open Source C Programming tools like G++/GCC or Eclipse.
* 64-bit Open source Linux or its derivative.

**Theory- Concept:**

* **What is Stack?**

Stack is a linear data structure in which the insertion and deletion operations are performed at only one end. In a stack, adding and removing of elements are performed at single position which is known as “top”. That means, new element is added at top of the stack and an element is removed from the top of the stack. In stack, the insertion and deletion operations are performed based on LIFO (Last In First Out) principle.



In a stack, the insertion operation is performed using a function called “push” and deletion operation is performed using a function called “pop”.

In the figure, PUSH and POP operations are performed at top position in the stack. That means, both the insertion and deletion operations are performed at one end (i.e., at Top)

**A stack data structure can be defined as follows…**

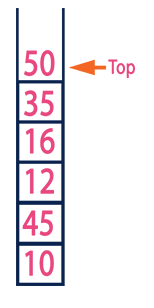
**Stack is a linear data structure in which the operations are performed based on LIFO principle.**

**Stack can also be defined as**

**“A Collection of similar data items in which both insertion and deletion operations are performed based on LIFO principle”.**

**Example**

If we want to create a stack by inserting 10, 45, 12, 16, 35 and 50. Then 10 becomes the bottom most element and 50 is the top most element. Top is at 50 as shown in the image below…

****

**Operations on a Stack:**

The following operations are performed on the stack…

1. Push (To insert an element on to the stack)
2. Pop (To delete an element from the stack)
3. Display (To display elements of the stack)

**Stack data structure can be implementing in two ways. They are as follows…**

1. Using Array
2. Using Linked List

When stack is implemented using array, that stack can organize only limited number of elements. When stack is implemented using linked list, that stack can organize unlimited number of elements.

**Stack Using Array:**

A stack data structure can be implemented using one dimensional array. But stack implemented using array, can store only fixed number of data values. This implementation is very simple, just define a one dimensional array of specific size and insert or delete the values into that array by using LIFO principle with the help of a variable ‘top’. Initially top is set to -1. Whenever we want to insert a value into the stack, increment the top value by one and then insert. Whenever we want to delete a value from the stack, then delete the top value and decrement the top value by one.

**Stack Operations using Array:**

A stack can be implemented using array as follows…

Before implementing actual operations, first follow the below steps to create an empty stack.

**Step 1**: Include all the header files which are used in the program and define a constant ‘SIZE’ with specific value.

**Step 2**: Declare all the functions used in stack implementation.

**Step 3**: Create a one dimensional array with fixed size (int stack[SIZE])

**Step 4**: Define a integer variable ‘top’ and initialize with ‘-1’. (int top = -1)

**Step 5**: In main method display menu with list of operations and make suitable function calls to perform operation selected by the user on the stack.

**push(value) – Inserting value into the stack:**

In a stack, push() is a function used to insert an element into the stack. In a stack, the new element is always inserted at top position. Push function takes one integer value as parameter and inserts that value into the stack. We can use the following steps to push an element on to the stack…

**Step 1**: Check whether stack is FULL. (top == SIZE-1)

**Step 2**: If it is FULL, then display “Stack is FULL!!! Insertion is not possible!!!” and terminate the function.

**Step 3**: If it is NOT FULL, then increment top value by one (top++) and set stack[top] to value (stack[top] = value).

**pop() – Delete a value from the Stack:**

In a stack, pop() is a function used to delete an element from the stack. In a stack, the element is always deleted from top position. Pop function does not take any value as parameter. We can use the following steps to pop an element from the stack…

**Step 1**: Check whether stack is EMPTY. (top == -1)

**Step 2**: If it is EMPTY, then display “Stack is EMPTY!!! Deletion is not possible!!!” and terminate the function.

**Step 3**: If it is NOT EMPTY, then delete stack[top] and decrement top value by one (top–).

**Display () – Displays the elements of a Stack**

We can use the following steps to display the elements of a stack…

**Step 1**: Check whether stack is EMPTY. (top == -1)

**Step 2**: If it is EMPTY, then display “Stack is EMPTY!!!” and terminate the function.

**Step 3**: If it is NOT EMPTY, then define a variable ‘i’ and initialize with top. Display stack[i] value and decrement i value by one (i–).

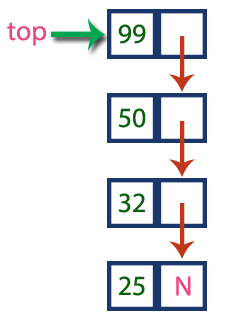
**Step 3**: Repeat above step until i value becomes ‘0’.

**Stack using Linked List:**

The major problem with the stack implemented using array is, it works only for fixed number of data values. That means the amount of data must be specified at the beginning of the implementation itself. Stack implemented using array is not suitable, when we don’t know the size of data which we are going to use. A stack data structure can be implemented by using linked list data structure. The stack implemented using linked list can work for unlimited number of values. That means, stack implemented using linked list works for variable size of data. So, there is no need to fix the size at the beginning of the implementation. The Stack implemented using linked list can organize as many data values as we want.

In linked list implementation of a stack, every new element is inserted as ‘top’ element. That means every newly inserted element is pointed by ‘top’. Whenever we want to remove an element from the stack, simply remove the node which is pointed by ‘top’ by moving ‘top’ to its next node in the list. The next field of the first element must be always NULL.

**Example**

****

In above example, the last inserted node is 99 and the first inserted node is 25. The order of elements inserted is 25, 32, 50 and 99.

**Infix to Postfix Conversion using Stack Data Structure:**

To convert Infix Expression into Postfix Expression using a stack data structure, We can use the following steps…

1. Read all the symbols one by one from left to right in the given Infix Expression.
2. If the reading symbol is operand, then directly print it to the result (Output).
3. If the reading symbol is left parenthesis ‘(‘, then Push it on to the Stack.
4. If the reading symbol is right parenthesis ‘)’, then Pop all the contents of stack until respective left parenthesis is poped and print each poped symbol to the result.
5. If the reading symbol is operator (+ , – , \* , / etc.,), then Push it on to the Stack. However, first pop the operators which are already on the stack that have higher or equal precedence than current operator and print them to the result.

**Postfix Expression Evaluation using Stack Data Structure:**

A postfix expression can be evaluated using the Stack data structure. To evaluate a postfix expression using Stack data structure we can use the following steps…

1. Read all the symbols one by one from left to right in the given Postfix Expression
2. If the reading symbol is operand, then push it on to the Stack.
3. If the reading symbol is operator (+ , – , \* , / etc.,), then perform TWO pop operations and store the two popped operands in two different variables (operand1 and operand2). Then perform reading symbol operation using operand1 and operand2 and push result back on to the Stack.
4. Finally! Perform a pop operation and display the popped value as final result.

**Conclusion:**

Hence we have learned how to implement Stack and perform various operations on it.