**Graded Lab 1**

**Infix to Postfix**

**Code:-**

**Stack.h:**

#pragma once

#include<iostream>

using namespace std;

template<class T>

class Stack {

T\* arr;

int maxSize;

int currentSize;

public:

Stack();

// Returns whether the Stack is empty or not

bool empty();

// Returns whether the Stack is full or not

bool full();

// Returns the current size of the Stack

int size();

// Returns the last element of the Stack

T top();

// Adds the element of type Type at the top of the stac

void push(T);

// Deletes the top most element of the stack and returns it

T pop();

~Stack();

private:

void regrow();

};

template <class T>

Stack<T>::Stack() {

this->maxSize = 1;

arr = new T[maxSize];

currentSize = 0;

}

template <class T>

void Stack<T>::regrow() {

T\* newArray = new T[++maxSize];

for (int a = 0; a < currentSize; a++) {

newArray[a] = arr[a];

}

delete[] arr;

arr = newArray;

}

// Returns whether the Stack is empty or not

template <class T>

bool Stack<T>::empty() {

return currentSize == 0;

}

// Returns whether the Stack is full or not

template <class T>

bool Stack<T>::full() {

return currentSize == maxSize;

}

// Returns the current size of the Stack

template <class T>

int Stack<T>::size() {

return currentSize;

}

// Returns the last element of the Stack

template <class T>

T Stack<T>::top() {

if (!empty()) {

return arr[currentSize - 1];

}

else {

cout << "Stack is Empty!..." << endl;

return NULL;

}

}

template<class T>

void Stack<T>::push(T value) {

if (currentSize < maxSize) {

arr[currentSize++] = value;

}

else {

regrow();

arr[Stack<T>::currentSize++] = value;

}

}

template<class T>

T Stack<T>::pop() {

if (!empty()) {

Stack<T>::currentSize--;

return Stack<T>::arr[Stack<T>::currentSize];

}

else {

cout << "Stack is Empty" << endl;

return NULL;

}

}

template <class T>

Stack<T>::~Stack() {

delete[] arr;

}

**Source.cpp:**

#include <iostream>

#include "Stack.h"

using namespace std;

int checkPrecedence(char c) {

if (c == '^') {

return 3;

}

else if (c == '/' || c == '\*') {

return 2;

}

else if (c == '+' || c == '-') {

return 1;

}

else {

return -1;

}

}

// infix to postfix expression

char\* infixToPostfix(char infix[], int size) {

Stack<char> st;

char\* postfix = new char[size];

int ind = 0; // Index for Result Array

for (int a = 0; a < size; a++) {

char scan = infix[a];

if ((scan >= 'a' && scan <= 'z') || (scan >= 'A' && scan <= 'Z') || (scan >= '0' && scan <= '9')) {

postfix[ind] = scan;

ind++;

}

else if (scan == '(') {

st.push('(');

}

else if (scan == ')') {

while (st.top() != '(') {

postfix[ind] = st.pop();

ind++;

}

st.pop();

}

else {

while ((!st.empty() && checkPrecedence(scan) < checkPrecedence(st.top())) ||

(!st.empty() && checkPrecedence(scan) == checkPrecedence(st.top()))) {

postfix[ind] = st.pop();

ind++;

}

st.push(scan);

}

}

while (!st.empty()) {

postfix[ind] = st.pop();

ind++;

}

postfix[ind] = '\0';

return postfix;

}

int main() {

char inf[] = "((A+B)-C\*(D/E))+F";

int size = 0;

for (int a = 0; inf[a] != '\0'; a++) {

size++;

}

char\* result = infixToPostfix(inf, size);

cout << "---Muhammad Ahmad---" << endl;

cout << "Infix Expression: " << inf << endl;

cout << "Result in PostFix: " << result << endl;

return 0;

}

**Output:**

