

Rupinder Kaur

Nov 08, 2020

Part 1- Comment ALL of the code in the file “stack linked list.txt” and “postfix.txt”

```

//*****
// Author: D.S. Malik
//
// Program: Postfix Calculator
// This program evaluates postfix expressions.
//*****

#include <iostream>
#include <iomanip>
#include <fstream>
#include <assert.h>
//#include "mystack.h"

using namespace std;

//function to evaluates each postfix expression
void evaluateExpression(ifstream& inpF, ofstream& outF,
                      stackType<double>& stack,
                      char& ch, bool& isExpOk);

// function to evaluate an expression
void evaluateOpr(ofstream& out, stackType<double>& stack,
                char& ch, bool& isExpOk);

// function to discover error in the expression
void discardExp(ifstream& in, ofstream& out, char& ch);

// function to print result
void printResult(ofstream& outF, stackType<double>& stack,
                bool isExpOk);

// main function

```

```

int main()
{
    bool expressionOk;
    char ch;
    stackType<double> stack(100);
    ifstream infile;
    ofstream outfile;

    infile.open("RpnData.txt");

    if (!infile)
    {
        cout << "Cannot open the input file. "    // print statement
             << "Program terminates!" << endl;
        return 1;
    }

    outfile.open("RpnOutput.txt");

    outfile << fixed << showpoint;
    outfile << setprecision(2);

    infile >> ch;
    while (infile)
    {
        stack.initializeStack();    //initialize the stack
        expressionOk = true;        //process the expression
        outfile << ch;              //output result

        evaluateExpression(infile, outfile, stack, ch,
                           expressionOk);
        printResult(outfile, stack, expressionOk);
        infile >> ch;    //begin processing the next expression
    }    //end while

    infile.close();
    outfile.close();

    return 0;
}

```

```
} //end main
```

```
//function to evaluates each postfix expression
```

```
void evaluateExpression(ifstream& inpF, ofstream& outF, stackType<double>& stack, char&
ch, bool& isExpOk)
```

```
{
```

```
    double num;
```

```
    while (ch != '#')    //process each expression, '#' marks the end of an expression
```

```
    {
```

```
        switch (ch)
```

```
        {
```

```
        case '#':
```

```
            inpF >> num;                // read a number
```

```
            outF << num << " ";        // output the number
```

```
            if (!stack.isFullStack())
```

```
                stack.push(num);        // push the number onto the stack
```

```
            else
```

```
            {
```

```
                cout << "Stack overflow. "
```

```
                << "Program terminates!" << endl;
```

```
                exit(0);    //terminate the program
```

```
            }
```

```
        break;
```

```
        default:
```

```
            // assume that ch is an operation
```

```
            evaluateOpr(outF, stack, ch, isExpOk);    // evaluate the operation
```

```
    } //end switch
```

```
    if (isExpOk) //if no error
```

```
    {
```

```
        inpF >> ch;    // read next ch
```

```
        outF << ch;    // output
```

```
        if (ch != '#')
```

```
            outF << " ";
```

```
    }
```

```
    else
```

```

        discardExp(inpF, outF, ch);           // discard the expression
    }    //end while (!= '=')
}    //end evaluateExpression

```

// function to evaluate an expression

```

void evaluateOpr(ofstream& out, stackType<double>& stack,
    char& ch, bool& isExpOk)
{
    double op1, op2;

    if (stack.isEmptyStack())
    {
        out << " (Not enough operands)";    //error in the expression
        isExpOk = false;                    //set expressionOk to false
    }
    else
    {
        op2 = stack.top();                  //retrieve top element into op2
        stack.pop();                        //pop stack

        if (stack.isEmptyStack())
        {
            out << " (Not enough operands)";    //error output
            isExpOk = false;                    //set expressionOk to false
        }
        else
        {
            op1 = stack.top();              //retrieve top element into op1
            stack.pop();                    //pop stack

            switch (ch)
            {
                //perform the operation and push result onto the stack
                case '+':
                    stack.push(op1 + op2);
                    break;

                //perform the operation and push result onto the stack
                case '-':
                    stack.push(op1 - op2);

```

```

        break;
    case '*':
        stack.push(op1 * op2);
        break;
//if (op2 != 0), perform the operation and push the result
    case '/':
        if (op2 != 0)
            stack.push(op1 / op2);
        else
            //otherwise report the error and set expressionOk to false
            {
                out << " (Division by 0)";
                isExpOk = false;
            }
        break;
    default:
        out << " (Illegal operator)";
        isExpOk = false;
    } //end switch
} //end else
} //end else
} //end evaluateOpr

```

```

//function to discover the error in the expression
void discardExp(ifstream& in, ofstream& out, char& ch)
{
    while (ch != '=')
        //process each expression, '=' marks the end of an expression
        {
            in.get(ch);
            out << ch;
        }
}
//end discardExp

```

```

//function to print the result
void printResult(ofstream& outF, stackType<double>& stack,
                bool isExpOk)
{
    double result;

```

```

if (isExpOk)           //if no error, print the result
{
    if (!stack.isEmptyStack())
    {
        result = stack.top();           //retrieve top element into result
        stack.pop();                   //pop stack

        if (stack.isEmptyStack())
            outF << result << endl;
        else
            outF << " (Error: Too many operands)" << endl;
    } //end if
    else
        outF << " (Error in the expression)" << endl;
}
else
    outF << " (Error in the expression)" << endl;

outF << " _____ "
    << endl << endl;
} //end printResult

```

//Header File: linkedStack.h

```

#ifndef H_StackType
#define H_StackType

#include <iostream>
#include <cassert>

#include "stackADT.h"

using namespace std;

```

```

//Definition of the node
template <class Type>
struct nodeType
{
    Type info;
    nodeType<Type> *link;
};

template <class Type>
class linkedStackType: public stackADT<Type>
{
public:
    const linkedStackType<Type>& operator=
        (const linkedStackType<Type>&);
    //Overload the assignment operator.

    bool isEmptyStack() const;
    //Function to determine whether the stack is empty.
    //Postcondition: Returns true if the stack is empty;
    //               otherwise returns false.

    bool isFullStack() const;
    //Function to determine whether the stack is full.
    //Postcondition: Returns false.

    void initializeStack();
    //Function to initialize the stack to an empty state.
    //Postcondition: The stack elements are removed;
    //               stackTop = nullptr;

    void push(const Type& newItem);
    //Function to add newItem to the stack.
    //Precondition: The stack exists and is not full.
    //Postcondition: The stack is changed and newItem
    //               is added to the top of the stack.

    Type top() const;
    //Function to return the top element of the stack.
    //Precondition: The stack exists and is not empty.

```

```

//Postcondition: If the stack is empty, the program
//               terminates; otherwise, the top
//               element of the stack is returned.

void pop();
//Function to remove the top element of the stack.
//Precondition: The stack exists and is not empty.
//Postcondition: The stack is changed and the top
//               element is removed from the stack.

linkedStackType();
//Default constructor
//Postcondition: stackTop = nullptr;

linkedStackType(const linkedStackType<Type>& otherStack);
//Copy constructor

~linkedStackType();
//Destructor
//Postcondition: All the elements of the stack are
//               removed from the stack.

private:
nodeType<Type> *stackTop;           //pointer to the stack

void copyStack(const linkedStackType<Type>& otherStack);
//Function to make a copy of otherStack.
//Postcondition: A copy of otherStack is created and
//               assigned to this stack.
};

//Default constructor
template <class Type>
linkedStackType<Type>::linkedStackType()
{
    stackTop = nullptr;
}

template <class Type>

```



```

bool linkedStackType<Type>::isEmptyStack() const
{
    return(stackTop == nullptr);
} //end isEmptyStack

template <class Type>
bool linkedStackType<Type>:: isFullStack() const
{
    return false;
} //end isFullStack

template <class Type>
void linkedStackType<Type>:: initializeStack()
{
    nodeType<Type> *temp; //pointer to delete the node

    while (stackTop != nullptr) //while there are elements in
        //the stack
    {
        temp = stackTop; //set temp to point to the
            //current node
        stackTop = stackTop->link; //advance stackTop to the
            //next node
        delete temp; //deallocate memory occupied by temp
    }
} //end initializeStack

template <class Type>
void linkedStackType<Type>::push(const Type& newElement)
{
    nodeType<Type> *newNode; //pointer to create the new node

    newNode = new nodeType<Type>; //create the node

    newNode->info = newElement; //store newElement in the node
    newNode->link = stackTop; //insert newNode before stackTop
    stackTop = newNode; //set stackTop to point to the
        //top node
} //end push

```

```

template <class Type>
Type linkedStackType<Type>::top() const
{
    assert(stackTop != nullptr); //if stack is empty,
                                //terminate the program
    return stackTop->info; //return the top element
} //end top

```

```

template <class Type>
void linkedStackType<Type>::pop()
{
    nodeType<Type> *temp; //pointer to deallocate memory

    if (stackTop != nullptr)
    {
        temp = stackTop; //set temp to point to the top node

        stackTop = stackTop->link; //advance stackTop to the
                                //next node
        delete temp; //delete the top node
    }
    else
        cout << "Cannot remove from an empty stack." << endl;
} //end pop

```

```

template <class Type>
void linkedStackType<Type>::copyStack
    (const linkedStackType<Type>& otherStack)
{
    nodeType<Type> *newNode, *current, *last;

    if (stackTop != nullptr) //if stack is nonempty, make it empty
        initializeStack();

    if (otherStack.stackTop == nullptr)
        stackTop = nullptr;
    else
    {

```

```

current = otherStack.stackTop; //set current to point
                               //to the stack to be copied

    //copy the stackTop element of the stack
stackTop = new nodeType<Type>; //create the node

stackTop->info = current->info; //copy the info
stackTop->link = nullptr; //set the link field of the
                        //node to nullptr
last = stackTop; //set last to point to the node
current = current->link; //set current to point to
                        //the next node

    //copy the remaining stack
while (current != nullptr)
{
    newNode = new nodeType<Type>;

    newNode->info = current->info;
    newNode->link = nullptr;
    last->link = newNode;
    last = newNode;
    current = current->link;
} //end while
} //end else
} //end copyStack

//copy constructor
template <class Type>
linkedStackType<Type>::linkedStackType(
    const linkedStackType<Type>& otherStack)
{
    stackTop = nullptr;
    copyStack(otherStack);
} //end copy constructor

//destructor
template <class Type>
linkedStackType<Type>::~~linkedStackType()
{

```

```
    initializeStack();  
} //end destructor  
  
    //overloading the assignment operator  
template <class Type>  
const linkedStackType<Type>& linkedStackType<Type>::operator=  
    (const linkedStackType<Type>& otherStack)  
{  
    if (this != &otherStack) //avoid self-copy  
        copyStack(otherStack);  
  
    return *this;  
} //end operator=  
  
#endif
```