**Infix Calculator**

1. **Design Document**

**Introduction**

The purpose of this program is to understand how Linked-based stack with exceptions operate. Linked List is an ordered set of data elements, each containing a link to its successor (and sometimes its predecessor). It is another way of implementing repetitive computations, which we can associate with loops.

This program is a simple post expression calculator that takes an infix expression from the user checks and converts it to postfix for latter calculation.

**Data structure**

The program uses a template class that can adapt to the users need. Linked-based List definitions are often particularly useful when the program is manipulating long and different kind of data types under one list. This program uses linked list as a stack in a form of FIFO in a class with two data structures, multiple string and char data types. The char data type is for storing what the user entered in one single variable while the char will be used in the class linked list to store the individual characters in one single box aside with two other pointers.

**Functions**

The Program has three classes and an interface class. The first ‘Node’ class, uses 7 functions including the constructor while ‘LinkedStack’ class uses 7 functions including 2 constructors with different parameters. The last class ‘PrecondViolatedExcep’ consists of definitions of logic errors and what happens if a run time error occurs. ‘StackInterface’ is the interface I used to primarily define a protocol.

Among this class functions one is a constructor function where it calls itself when the class is initialized in the main function. I used this constructor to initialize my structure data variables head, curr and temp to null. ‘setItem()’ and ‘setNext()’ save a data and a pointer in a linked list.

I have also used four nonmember functions namely ‘isoperator()’ that takes character and returns true if the character is an operator; ‘isoperad()’ that takes character and returns true if the character is an operand; ‘resetS()’ that takes an object and removes all the values that are in the stack and returns true after all data’s are wiped out, and ‘compareOperators()’ that takes 2 chars and compares operators precedence.

**The main program**

The program starts on the main function initializing 3 different objects with 3 different names ‘stackPtrT’, ‘stackPtr’ and ‘stackPtrE.’. ‘stackPtrT’ is for testing whether the string is well formed while ‘stackPtr’ is for converting infix to postfix, and the last object name ‘stackPtrE’ is for evaluating the postfix. After initializing three object names, I created character array ‘infix’ with 100 slots to store the user infix expression and assign a pointer ‘cPtr’ to find out the ending of expression. I have also introduced 3 double data variables ‘op1’, ’op2’, and ’resl’ to do calculation.

In addition to the above datavariables, I have also introduced 2 Boolean variables ‘correctEx’ and ‘redo’ initiaised to false. ‘postFixS’ is used to save the corrected string postfix expression.The program asks for a rerun If the user wants to calculate a different infix expression.

1. **Code list**

**calculator.cpp: main source**

#include <iostream>

#include <string>

#include "LinkedStack.cpp"

using namespace std;

// Simply determine if character is one of the four standard operators.

bool isOperator(char character) {

if (character == '+' || character == '-' || character == '\*' || character == '/') {

return true;

}

return false;

}

// If the character is not an operator or a parenthesis, then it is assumed to be an operand.

bool isOperand(char character) {

if (!isOperator(character) && character != '(' && character != ')') {

return true;

}

return false;

}

// Compare operator precedence of main operators.

// Return 0 if equal, -1 if op2 is less than op1, and 1 if op2 is greater than op1.

int compareOperators(char op1, char op2) {

if ((op1 == '\*' || op1 == '/') && (op2 == '+' || op2 == '-')) { return -1; }

else if ((op1 == '+' || op1 == '-') && (op2 == '\*' || op2 == '/')) { return 1; }

return 0;

}

//resset stack

bool resetS(StackInterface<char>\* stackPtrT){

while (!stackPtrT->isEmpty()) { //loop through the array

stackPtrT->pop();

}

return true;

}

int main()

{

cout << "\tThis program does infix calculation\n" << endl;

StackInterface<char>\* stackPtrT = new LinkedStack<char>(); //testing for well formed infix

StackInterface<char>\* stackPtr = new LinkedStack<char>(); // for converting infix to postfix

StackInterface<char>\* stackPtrE = new LinkedStack<char>(); //evaluating postfix

char infix[100], repeat;

// Get a pointer to our character array.

char \*cPtr = infix;

double op1, op2, resl;

bool correctEx = false, redo = false, difC = true;

string postFixS = "";

do{

// reseting all stacks

resetS(stackPtrT);

resetS(stackPtr);

resetS(stackPtrE);

postFixS = "";

do{

difC = true;

cPtr = &infix[0];// resetting the array

cout << "enter your Expression: ";

cin >> infix;

// checking if the parenthesis in the expression are balanced

while (\*cPtr != '\0') { //loop through the array

if (\*cPtr == '(')

stackPtrT->push(\*cPtr);

else if (\*cPtr == ')')

stackPtrT->pop();

cPtr++;

}

if (!stackPtrT->isEmpty())

{

cout << "please re";

resetS(stackPtrT);

correctEx = false;

continue;

}

cPtr = &infix[0];// resetting the array

// check for characters specified other than requested

while (\*cPtr != '\0') {

if (!isdigit(\*cPtr) != 0 && !isOperator(\*cPtr))

{

difC = false;

break;

}

cPtr++;

}

if (!difC)

{

cout << "please re";

resetS(stackPtrT);

correctEx = false;

continue;

}

cPtr = &infix[0];// resetting the array

// check for well-formed expression

while (\*cPtr != '\0') {

if (isOperand(\*cPtr))

stackPtrT->push(\*cPtr);

else if (isOperator(\*cPtr))

stackPtrT->pop();

cPtr++;

}

if (!stackPtrT->isEmpty())

{

if (isOperand(stackPtrT->peek()))

correctEx = true;

}

else

{

cout << "please re";

resetS(stackPtrT);

}

} while (!correctEx);

cPtr = &infix[0];// resetting the array

// Loop through the array (one character at a time) until we reach the end of the string.

while (\*cPtr != '\0') {

// If operand, simply add it to our postfix string.

// If it is an operator, pop operators off our stack until it is empty, an open parenthesis or an operator with less than or equal precedence.

if (isOperand(\*cPtr)) { postFixS += \*cPtr; }

else if (isOperator(\*cPtr)) {

while (!stackPtr->isEmpty() && stackPtr->peek() != '(' && compareOperators(stackPtr->peek(), \*cPtr) <= 0) {

postFixS += stackPtr->peek();

stackPtr->pop();

}

stackPtr->push(\*cPtr);

}

// Simply push all open parenthesis onto our stack

// When we reach a closing one, start popping off operators until we run into the opening parenthesis.

else if (\*cPtr == '(') { stackPtr->push(\*cPtr); }

else if (\*cPtr == ')') {

while (!stackPtr->isEmpty()) {

if (stackPtr->peek() == '(') { stackPtr->pop(); break; }

postFixS += stackPtr->peek();

stackPtr->pop();

}

}

// Advance our pointer to next character in string.

cPtr++;

}

// After the input expression has been ran through, if there is any remaining operators left on the stack

// pop them off and put them onto the postfix string.

while (!stackPtr->isEmpty()) {

postFixS += stackPtr->peek();

stackPtr->pop();

}

// Show the postfix string at the end.

cout << "Postfix Expression is: " << postFixS << endl;

for (int i = 0; i < postFixS.length(); i++)

{

if (isOperand(postFixS[i]))

stackPtrE->push(postFixS[i]);

else{

op1 = stackPtrE->peek() - '0';

stackPtrE->pop();

op2 = stackPtrE->peek() - '0';

stackPtrE->pop();

switch (postFixS[i])

{

case '+':

resl = op2 + op1;

break;

case '-':

resl = op2 - op1;

break;

case '\*':

resl = op2 \* op1;

break;

case '/':

resl = op2 / op1;

break;

}

stackPtrE->push(resl + 48);

}

}

cout << "The result is: " << stackPtrE->peek() - '0' << endl;

cout << "\nWould you like to do another expression? \nif yes, enter y or Y; or hit any other key quit ";

cin >> repeat;

if (repeat == 'Y' || repeat == 'y')

redo = true;

else

redo = false;

} while (redo);

return 0;

}

**LinkedStack.h**

// Created by Frank M. Carrano and Tim Henry.

// Copyright (c) 2013 \_\_Pearson Education\_\_. All rights reserved.

/\*\* ADT stack: Link-based implementation.

Listing 7-x.

@file LinkedStack.h \*/

#ifndef \_LINKED\_STACK

#define \_LINKED\_STACK

#include "StackInterface.h"

#include "Node.cpp"

#include "PrecondViolatedExcep.cpp"

/\*\* ADT stack - Linked implementation. \*/

template<class ItemType>

class LinkedStack : public StackInterface<ItemType>

{

private:

Node<ItemType>\* topPtr; // Pointer to first node in the chain;

// this node contains the stack's top

public:

// Constructors and destructor:

LinkedStack(); // Default constructor

LinkedStack(const LinkedStack<ItemType>& aStack); // Copy constructor

virtual ~LinkedStack(); // Destructor

// Stack operations:

bool isEmpty() const;

bool push(const ItemType& newItem);

bool pop();

/\*\* @throw PrecondViolatedExcep if the stack is empty \*/

ItemType peek() const throw(PrecondViolatedExcep);

}; // end LinkedStack

#endif

**LinkedStack.cpp**

// Created by Frank M. Carrano and Tim Henry.

// Copyright (c) 2013 \_\_Pearson Education\_\_. All rights reserved.

/\*\* @file LinkedStack.cpp \*/

#include <cassert> // For assert

#include "LinkedStack.h" // Stack class specification file

#include <iostream>

using namespace std;

template<class ItemType>

LinkedStack<ItemType>::LinkedStack() : topPtr(NULL)

{

} // end default constructor

template<class ItemType>

LinkedStack<ItemType>::LinkedStack(const LinkedStack<ItemType>& aStack)

{

Node<ItemType>\* origChainPtr = aStack->topPtr; // Points to nodes in original chain

if (origChainPtr == NULL)

topPtr = NULL; // Original stack is empty

else

{

// Copy first node

topPtr = new Node<ItemType>();

topPtr->setItem(origChainPtr->getItem());

// Copy remaining nodes

Node<ItemType>\* newChainPtr = topPtr; // Points to last node in new chain

while (origChainPtr != NULL)

{

// Advance original-chain pointer

origChainPtr = origChainPtr->getNext();

// Get next item from original chain

ItemType nextItem = origChainPtr->getItem();

// Create a new node containing the next item

Node<ItemType>\* newNodePtr = new Node<ItemType>(nextItem);

// Link new node to end of new chain

newChainPtr->setNext(newNodePtr);

// Advance pointer to new last node

newChainPtr = newChainPtr->getNext();

} // end while

newChainPtr->setNext(NULL); // Flag end of chain

} // end if

} // end copy constructor

template<class ItemType>

LinkedStack<ItemType>::~LinkedStack()

{

// Pop until stack is empty

while (!isEmpty())

pop();

} // end destructor

template<class ItemType>

bool LinkedStack<ItemType>::isEmpty() const

{

return topPtr == NULL;

} // end isEmpty

template<class ItemType>

bool LinkedStack<ItemType>::push(const ItemType& newEntry)

{

Node<ItemType>\* newNodePtr = new Node<ItemType>(newEntry, topPtr);

topPtr = newNodePtr;

newNodePtr = NULL;

return true;

} // end push

template<class ItemType>

bool LinkedStack<ItemType>::pop()

{

bool result = false;

if (!isEmpty())

{

// Stack is not empty; delete top

Node<ItemType>\* nodeToDeletePtr = topPtr;

topPtr = topPtr->getNext();

// Return deleted node to system

nodeToDeletePtr->setNext(NULL);

delete nodeToDeletePtr;

nodeToDeletePtr = NULL;

result = true;

} // end if

return result;

} // end pop

template<class ItemType>

ItemType LinkedStack<ItemType>::peek() const throw(PrecondViolatedExcep)

{

// Enforce precondition

if (isEmpty())

throw(PrecondViolatedExcep("peek() called with empty stack."));

// Stack is not empty; return top

return topPtr->getItem();

} // end peek

// End of implementation file.

**precondViolaedExcep.h**

// Created by Frank M. Carrano and Tim Henry.

// Copyright (c) 2013 \_\_Pearson Education\_\_. All rights reserved.

/\*\* Listing 7-5.

@file PrecondViolatedExcep.h \*/

#ifndef \_PRECOND\_VIOLATED\_EXCEP

#define \_PRECOND\_VIOLATED\_EXCEP

#include <stdexcept>

#include <string>

using namespace std;

class PrecondViolatedExcep : public logic\_error

{

public:

PrecondViolatedExcep(const string& message = "");

}; // end PrecondViolatedExcep

#endif

**PrecondViolaedExcep.cpp**

// Created by Frank M. Carrano and Tim Henry.

// Copyright (c) 2013 \_\_Pearson Education\_\_. All rights reserved.

/\*\* Listing 7-6.

@file PrecondViolatedExcep.cpp \*/

#include "PrecondViolatedExcep.h"

PrecondViolatedExcep::PrecondViolatedExcep(const string& message): logic\_error("Precondition Violated Exception: " + message)

{

} // end constructor

// End of implementation file.

**StackInterface.h**

// Created by Frank M. Carrano and Tim Henry.

// Copyright (c) 2013 \_\_Pearson Education\_\_. All rights reserved.

/\*\* @file StackInterface.h \*/

#ifndef \_STACK\_INTERFACE

#define \_STACK\_INTERFACE

template<class ItemType>

class StackInterface

{

public:

/\*\* Sees whether this stack is empty.

@return True if the stack is empty, or false if not. \*/

virtual bool isEmpty() const = 0;

/\*\* Adds a new entry to the top of this stack.

@post If the operation was successful, newEntry is at the top of the stack.

@param newEntry The object to be added as a new entry.

@return True if the addition is successful or false if not. \*/

virtual bool push(const ItemType& newEntry) = 0;

/\*\* Removes the top of this stack.

@post If the operation was successful, the top of the stack

has been removed.

@return True if the removal is successful or false if not. \*/

virtual bool pop() = 0;

/\*\* Returns the top of this stack.

@pre The stack is not empty.

@post The top of the stack has been returned, and

the stack is unchanged.

@return The top of the stack. \*/

virtual ItemType peek() const = 0;

}; // end StackInterface

#endif

**Node.h: header file**

#ifndef \_NODE

#define \_NODE

template<class ItemType>

class Node

{

private:

ItemType item; // A data item

Node<ItemType>\* next; // Pointer to next node

public:

Node();

Node(const ItemType& anItem);

Node(const ItemType& anItem, Node<ItemType>\* nextNodePtr);

void setItem(const ItemType& anItem);

void setNext(Node<ItemType>\* nextNodePtr);

ItemType getItem() const;

Node<ItemType>\* getNext() const;

}; // end Node

#endif

**Node.cpp: cpp file**

//#include "stdafx.h"

#include "Node.h"

#include <cstddef>

template<class ItemType>

Node<ItemType>::Node() : next(NULL)

{

} // end default constructor

template<class ItemType>

Node<ItemType>::Node(const ItemType& anItem) : item(anItem), next(NULL)

{

} // end constructor

template<class ItemType>

Node<ItemType>::Node(const ItemType& anItem, Node<ItemType>\* nextNodePtr) :

item(anItem), next(nextNodePtr)

{

} // end constructor

template<class ItemType>

void Node<ItemType>::setItem(const ItemType& anItem)

{

item = anItem;

} // end setItem

template<class ItemType>

void Node<ItemType>::setNext(Node<ItemType>\* nextNodePtr)

{

next = nextNodePtr;

} // end setNext

template<class ItemType>

ItemType Node<ItemType>::getItem() const

{

return item;

} // end getItem

template<class ItemType>

Node<ItemType>\* Node<ItemType>::getNext() const

{

return next;

} // end getNext

1. **User Document**

This program has an easy to use procedure, but it has certain constraints that must be followed for successful usage. This program does postfix calculation and the user is required to type an infix expression; reading the following instructions could be very much of helpful.

Follow this instructions to run the programs:

* To run the test program calculator.cpp,
* To compile the postfix calculator program, simply enter: g++ calculator.cpp
* To run the program, type in: a.out in the command window
* Once the program starts running input an infix expression and hit enter.
* After that the program goes through the expression and displays the postfix expression of your input its result. Remember to input a single digit number.
* If the program is finding your expression to be other than an infix expression it will ask you type it in again.
* The program re runs every time up on your choice

1. **Test Data Plan**

|  |  |  |
| --- | --- | --- |
| Valid input Values |  |  |
|  | Input: infix expression | Performs its check on the expression |
|  | Input: select y or Y to continue | Repeats the program |
| Boundary values |  |  |
|  | Your infix expression should not exceed 100 characters | Bound error exception |
|  |  |  |
| Invalid input values |  |  |
|  | Input: if invalid operators are used | Error Message:’ retype your expression’ |
|  | Input: if it’s not an infix expression. | Error Message: ‘retype your expression’ |

1. **Summary**

This program uses three classes and an interface class for computing and completing an infix expression. It uses a linked-based stack with exceptions.

Linked lists are a way to store data with structures so that the programmer can automatically create a new place to store data whenever necessary. Specifically, the programmer writes a struct or class definition that contains variables holding information about something, and then has a pointer to a struct of its type.

The main drawback of using linked list stack compared to other sequence containers is that it lacks direct access to the elements by their position; For example, to access the sixth element in a list one must iterate from a known position (like the beginning or the end) to that position, which takes linear time in the distance between these. They also consume some extra memory to keep the linking information associated to each element (which may be an important factor for large lists of small-sized elements).

The program could be extended in several ways that would make it more useful for large projects by utilizing permanent storing systems. This can be achieved using a text file or similar data extensions by utilizing libraries like ‘ifstream’ and ‘ofstream.’ Besides utilizing a permanent storage system, we can also use GUI to make it easier and simpler for users.