Andrew Tran

Lab Experience Nine

Do the following problems:

Page 384: 2 – 10 evens, 16 – 22 evens

Problems 2 – 10 even

**a = 7.0, b = 4.0, c = 3.0, d = -2.0**

**2.) a b c + / d \***

7 4 3 -2 + / \*

**28**

**4.) a b + c + d +**

7 4 + 3 + -2 +

**12**

**6.) a b c + + d +**

7 4 3 + + -2 +

**12**

**8.) a b – c – d –**

7 4 – 3 - -2 –

**2**

**10.) a b c - - d –**

7 4 3 - - -2 –

**8**

16 – 22 evens

**16.) a + b / c + d**

**a b c / + d +**

**18.) a + b / (c + d)**

**a b c d + / +**

**20.) (a – b) \* (c – (d+ e) )**

**a b – c d e + - \***

**22.) a – (b – (c – (d – e) ) )**

**a b c d e - - - -**

Programming Problem:

Write a program to implement the algorithm for evaluating postfix expressions that involve only single-digit integers and the integer operations: +, - , \*, and /. To trace the action of postfix evaluation, display each token as it is encountered, and display the action of each stack operation. Your program should output an error message if the postfix expression is not well formed. I.e. not correct.

For example given the expression 9 2 1 + / 4 \* the output of your program should resemble the following:

Token = 9 Push 9

Token = 2 Push 2

Token = 1 Push 1

Token = + Pop 1 Pop 2 Push 3

Token = / Pop 3 Pop 9 Push 3

Token = 4 Push 4

Token = \* Pop 4 Pop 3 Push 12

Token = Pop 12

The algorithm for evaluating postfix expressions is given below.

**ALGORITHM TO EVALUATE POSTFIX EXPRESSIONS**

Each time an operand is encountered, it is pushed onto the stack. When an operator is encountered, the top two values are popped from the stack, the operator applied to them, and the result pushed back onto the stack. The following algorithm summarizes this procedure:

1. Initialize an empty stack

1. Repeat the following until the end of the expression is encountered:
2. Get the next token (character, variable, arithmetic operator) in the postfix expression.

b) If token is an operand, push it onto the stack. If it is an operator, do the following:

i) Retrieve and pop the top two values from the operand stack. If the operand stack does not contain two items, an error due to a malformed postfix expression has occurred and evaluation is terminated.

ii) Apply the operator token to these two values.

1. Push the resulting value back onto the operand stack.
2. When the end of the expression is encountered, its value is on top of the stack (and in fact must be the only value in the stack).

**What to hand in:**

1. Print-out of your program containing the solution of the above algorithm.
2. A word document containing screen shots of your program executing showing that it works for multiple postfix expressions and also detects a malformed expression.
3. A print out of a word document containing the answers to exercises assigned above.
4. Compress the word document and all files for your program solution into a single file called yournamelab9.zip.

**LStack.h – HEADER**

/\*-- LStack.h --------------------------------------------------------------

This header file defines a Stack data type.

Basic operations:

constructor: Constructs an empty stack

empty: Checks if a stack is empty

push: Modifies a stack by adding a value at the top

top: Accesses the top stack value; leaves stack unchanged

pop: Modifies stack by removing the value at the top

display: Displays all the stack elements

Note: Execution terminates if memory isn't available for a stack element.

--------------------------------------------------------------------------\*/

#include <iostream>

#include <string>

using namespace std;

#ifndef LSTACK

#define LSTACK

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Uncomment the following section to use the class for your lab

// assignment.

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/\*class IdentOrTemp

{

public:

int temp; // used as as counter for temp1, temp2 etc.

string ident; // used to specify a , b, c, etc.

};

//-- Empty output operator to use for Stack display()

inline ostream & operator<<(ostream & out, IdentOrTemp it)

{ out << ""; return out; }

typedef IdentOrTemp StackElement;

\*/

typedef int StackElement; // comment out or remove after testing.

class Stack

{

public:

/\*\*\*\*\* Methods \*\*\*\*\*/

/\*\*\*\*\* Constructors \*\*\*\*\*/

Stack();

/\*-----------------------------------------------------------------------

Construct a Stack object.

Precondition: None.

Postcondition: An empty Stack object has been constructed

(myTop is initialized to a null pointer).

------------------------------------------------------------------------\*/

Stack(const Stack & original);

/\*-----------------------------------------------------------------------

Copy constructor to create another Stack object.

Precondition: Another stack

Postcondition: A copy of the stack has been created.

------------------------------------------------------------------------\*/

/\*\*\*\*\* Destructor \*\*\*\*\*/

~Stack();

/\*------------------------------------------------------------------------

Class destructor

Precondition: None

Postcondition: The linked list in the stack has been deallocated.

------------------------------------------------------------------------\*/

/\*\*\*\*\* Assignment \*\*\*\*\*/

const Stack & operator= (const Stack & rightHandSide);

/\*------------------------------------------------------------------------

Assignment Operator

Precondition: rightHandSide is the stack to be assigned and is

received as a const reference parameter.

Postcondition: The current stack becomes a copy of rightHandSide

and a const reference to it is returned.

------------------------------------------------------------------------\*/

bool empty() const;

/\*------------------------------------------------------------------------

Check if stack is empty.

Precondition: None

Postcondition: Returns true if stack is empty and false otherwise.

-----------------------------------------------------------------------\*/

void push(const StackElement & value);

/\*------------------------------------------------------------------------

Add a value to a stack.

Precondition: value is to be added to this stack

Postcondition: value is added at top of stack provided there is space;

otherwise, a stack-full message is displayed and execution is

terminated.

-----------------------------------------------------------------------\*/

void display(ostream & out) const;

/\*------------------------------------------------------------------------

Display values stored in the stack.

Precondition: ostream out is open.

Postcondition: Stack's contents, from top down, have been output to out.

-----------------------------------------------------------------------\*/

StackElement top() const;

/\*------------------------------------------------------------------------

Retrieve value at top of stack (if any).

Precondition: Stack is nonempty

Postcondition: Value at top of stack is returned, unless the stack is

empty; in that case, an error message is displayed and a "garbage

value" is returned.

-----------------------------------------------------------------------\*/

void pop();

/\*------------------------------------------------------------------------

Remove value at top of stack (if any).

Precondition: Stack is nonempty.

Postcondition: Value at top of stack has been removed, unless the stack

is empty; in that case, an error message is displayed and

execution allowed to proceed.

-----------------------------------------------------------------------\*/

private:

/\*\*\* Node class \*\*\*/

class Node

{

public:

StackElement data;

Node \* next;

//--- Node constructor

Node(StackElement value){data = value; next = NULL;}

/\*-------------------------------------------------------------------

Precondition: None.

Postcondition: A Node has been constructed with value in its data

part and its next part set to link (default 0).

-------------------------------------------------------------------\*/

}; // end of class Node

typedef Node \* NodePointer;

/\*\*\*\*\* Data Members \*\*\*\*\*/

NodePointer myTop; // pointer to top of stack

}; // end of class declaration

#endif

**LStack.cpp – IMPLEMENTATION**

//--- LStack.cpp -------------------------------------------------

#include <new>

using namespace std;

#include "LStack.h"

//--- Definition of Stack constructor

Stack::Stack(){myTop = NULL;}

//--- Definition of Stack copy constructor

Stack::Stack(const Stack & original)

{

myTop = NULL;

if (!original.empty())

{

// Copy first node

myTop = new Stack::Node(original.top());

// Set pointers to run through the stacksÕ linked lists

Stack::NodePointer lastPtr = myTop,

origPtr = original.myTop->next;

while (origPtr != NULL)

{

lastPtr->next = new Stack::Node(origPtr->data);

lastPtr = lastPtr->next;

origPtr = origPtr->next;

}

}

}// end copy constructor

//--- Definition of Stack destructor

Stack::~Stack()

{

// Set pointers to run through the stack

Stack::NodePointer currPtr = myTop; // node to be deallocated

while (currPtr != NULL)

{

myTop = myTop->next;

delete currPtr;

currPtr = myTop;

}

}//end destructor

//--- Definition of assignment operator

const Stack & Stack::operator=(const Stack & rightHandSide)

{

if (this != &rightHandSide) // check that not st = st

{

this->~Stack(); // destroy current linked list

// if (rightHandSide.empty()) // empty stack

// myTop = 0;

//else

//{ // copy rightHandSide's list

// Copy first node

myTop = new Stack::Node(rightHandSide.top());

// Set pointers to run through the stacks' linked lists

Stack::NodePointer lastPtr = myTop,

rhsPtr = rightHandSide.myTop->next;

while (rhsPtr != 0)

{

lastPtr->next = new Stack::Node(rhsPtr->data);

lastPtr = lastPtr->next;

rhsPtr = rhsPtr->next;

}

//}

}

return \*this;

}// end overloaded assignment operator

//--- Definition of empty()

bool Stack::empty() const

{

return (myTop == NULL);

}// end empty

//--- Definition of push()

void Stack::push(const StackElement & value)

{

Stack::NodePointer top = new Stack::Node(value);

top->next = myTop;

myTop = top;

//myTop = new Stack::Node(value, myTop);

}// end push

//--- Definition of display()

void Stack::display(ostream & out) const

{

Stack::NodePointer ptr;

for (ptr = myTop; ptr != 0; ptr = ptr->next)

out << ptr->data << endl;

}// end display

//--- Definition of top()

StackElement Stack::top() const

{

if (!empty())

return (myTop->data);

else

{

cerr << "\*\*\* Stack is empty "

" -- returning garbage \*\*\*\n";

StackElement \* temp = new(StackElement);

StackElement garbage = \*temp; // "Garbage" value

delete temp;

return garbage;

}

}// end top

//--- Definition of pop()

void Stack::pop()

{

if (!empty())

{

Stack::NodePointer ptr = myTop;

myTop = myTop->next;

delete ptr;

}

else

cerr << "\*\*\* Stack is empty -- can't remove a value \*\*\*\n";

}// end pop

**Lab9.cpp – DRIVER**

#include<iostream>

#include<cctype>

#include<cstdlib>

#include"LStack.h"

using namespace std;

//prototypes

int operation(int op1, int op2, char operate);

int eval(char postfix[], int size);

void removeSpace(char postfix[]);

int main()

{

char postfix[30] = { 0 };

cout << "Enter a postfix operation: " << endl;

cin.getline(postfix,30);

removeSpace(postfix);

int val = eval(postfix, strlen(postfix));

}

//returns a value of the two popped value with the desired expression

//Precons: operator 1, operator 2, and the desired operation

//Postcons: int answer of the expression

int operation(int op1, int op2, char operate) {

switch (operate)

{

case '\*': //multiplication

cout << "Token = \*" << " " << "Pop " << op1 << " " << "Pop " << op2 << " " << "Push " << op2 \* op1 << endl;

return op2 \* op1;

case '/': //division

cout << "Token = /" << " " << "Pop " << op1 << " " << "Pop " << op2 << " " << "Push " << op2 / op1 << endl;

return op2 / op1;

case '+': //addition

cout << "Token = +" << " " << "Pop " << op1 << " " << "Pop " << op2 << " " << "Push " << op2 + op1 << endl;

return op2 + op1;

case '-': //subtraction

cout << "Token = -" << " " << "Pop " << op1 << " " << "Pop " << op2 << " " << "Push " << op2 - op1 << endl;

return op2 - op1;

default: return 0;

}

}

//determines logic for stack and traverses for single digit numbers and operators

//Precons: postfix char array, int size of the array

//Postcons: int value of the entire expression

int eval(char postfix[], int size)

{

Stack s;

char ch;

int val;

for (int i = 0; i < size; i++)

{

ch = postfix[i];

if (isdigit(ch)) //digit detection

{

s.push(ch - '0');

cout << "Token = " << postfix[i] << " " << "Push " << postfix[i] << endl;

}

else //operator detection

{

if (s.empty()) //malformed expression detection

{

cout << "Malformed Expression" << endl;

exit(1107);

}

int op1 = s.top(); //set op1 to top of stack

s.pop(); //pop stack

if (s.empty()) //malformed expression detection

{

cout << "Malformed Expression" << endl;

exit(1107);

}

int op2 = s.top(); //set op2 to top of stack

s.pop(); //pop stack

val = operation(op1, op2, ch); //all info gathered; call evualuation function

s.push(val);

}

}

s.pop();

cout << "Pop " << val << endl;

return val;

}

//removes white space the user may have entered into the char array

//Precons: postfix char array

//Postcons: none

void removeSpace(char postfix[])

{

int i, j;

for (i = 0; postfix[i] != 0; i++)

{

if (isspace(postfix[i]))

{

for (j = i; postfix[j] != 0; j++)

{

postfix[j] = postfix[j + 1];

}

}

}

}









