**Project-6**

**Name: Pawan Dhungana**

**Project no: 6**

**Due Date: October 22, 2018**

**Design Document**

**Introduction**

The conventional notation in which we are accustomed to write arithmetic expressions is called infix notation, since in it, operators are written between their operands: X + Y. In postfix notation, the operator follows the operands: X Y +. Postfix expressions can be easily evaluated with a stack-based algorithm.

Here, a program is designed, implemented and tested to read postfix expressions from a file, one expression per line. The program echoes the expressions, evaluates them using the stack-based algorithm described in class, and reports their values.

**Data Structures**

The program uses a class Stack. An array data[CAPACITY] is used in the program. It holds the Items in a stack in positions 0 through used-1. The value of CAPACITY is set to be 100. Item is used as the data type of the integers in the file. The default constructor Stack() is used that initializes the value of used to be zero. The integer ‘used’ stores the number of items in the stack.

**Functions**

The program uses the default constructor as an inline function. The function apply() is declared outside the class, it is the function that returns the value that results when the operator optr is applied to the two operands that are the function's second and third parameters. There are five other functions inside a class declared under public member function. They are: push(), pop(), empty(), size(), and postFix(). The push() function inserts a new element at the top of the stack, above the current top element. The pop() function removes the element on top of the stack, effectively reducing its size by one. The empty() function returns whether the stack is empty(it checks of its size is zero). The size() function returns the number of elements in the stack. The postFix() returns the value of the postfix expression after calculation.

**Main Program**

In the main program, the user is asked to input the file that contains the postfix expressions. The program reads the name of a file, then from the file reads syntactically correct postfix expressions involving single-digit integers and the binary arithmetic operators +, -, \*, and /. If the file is not found, the program terminates with a message. The program prompts for the input file name and prints to the terminal each expression in the file and its value.

**User Document**

The conventional notation in which we are accustomed to write arithmetic expressions is called infix notation, since in it, operators are written between their operands: X + Y. In postfix notation, the operator follows the operands: X Y +. Postfix expressions can be easily evaluated with a stack-based algorithm.

Here, we design, implement, document, and test a program that reads postfix expressions from a file, one expression per line. The program echoes the expressions, evaluates them using the stack-based algorithm described in class, and reports their values.

The program's name is Project6.cpp, to compile and run it, simply enter:

g++ Project6.cpp

a.out

A run of the program might look like this:

Enter input file name: postfix.dat

Expression: 3 4 + 3 \*

Value = 21.

Expression: 5 4 3 2 1 - + / \*

Value = 5.

EXPRESSION: 9 5 2 4 + - 2 \* \*

Value = -18.

**Code Listing:**

#include<iostream>

#include<cstdlib>

#include<fstream>

#include<cstring>

#include<cctype>

#include<stack>

using namespace std;

int apply(char optr, int opnd1, int opnd2)

//precondition: optr is binary arithmetic operator.

//postcondition: opnd1 and opnd2 are the operands evaluated by the optr and the result is returned

{

if(optr == '+')

return (opnd1+opnd2);

else if (optr == '-')

return (opnd1-opnd2);

else if (optr == '\*')

return (opnd1\*opnd2);

else if(optr == '/')

return(opnd1/opnd2);

}

class Stack

{

public:

typedef int Item;

static const Item CAPACITY = 100;

Stack(){used = 0;}//constructor

//modification member functions

void push(Item entry);

Item pop();

//constant member functions

bool empty()const{return used == 0;}

int size()const{return used;}

int postFix(string postfix, Stack s);

private:

Item data[CAPACITY];

int used;

};

int main()

{

Stack s;

ifstream infile;

string temp;

int pop1, pop2;

int final;

char ch;

string filename;

cout<<"Enter input file name : ";

cin>>filename;

infile.open(filename.c\_str());

if(!infile)

{

cout<<"Invalid! File not found."<<endl;

}

while (getline(infile, temp))

{

cout << "Expression: " << temp << endl;

cout << "Value = " << s.postFix(temp,s) << "." << endl << endl;

}

infile.close();

return 0;

}

int Stack::postFix(string postfix,Stack s)

{

int i = 0;

char ch;

int val;

while (i < postfix.length()) {

ch = postfix[i];

if(ch != ' ')

if (isdigit(ch)) {

s.push(ch-'0');

}

else {

int op1 = s.pop();

int op2 = s.pop();

val = apply(ch, op2, op1);

s.push(val);

}

i++;

}

return val;

}

void Stack::push(Item entry)

{

data[used] = entry;

++used;

}

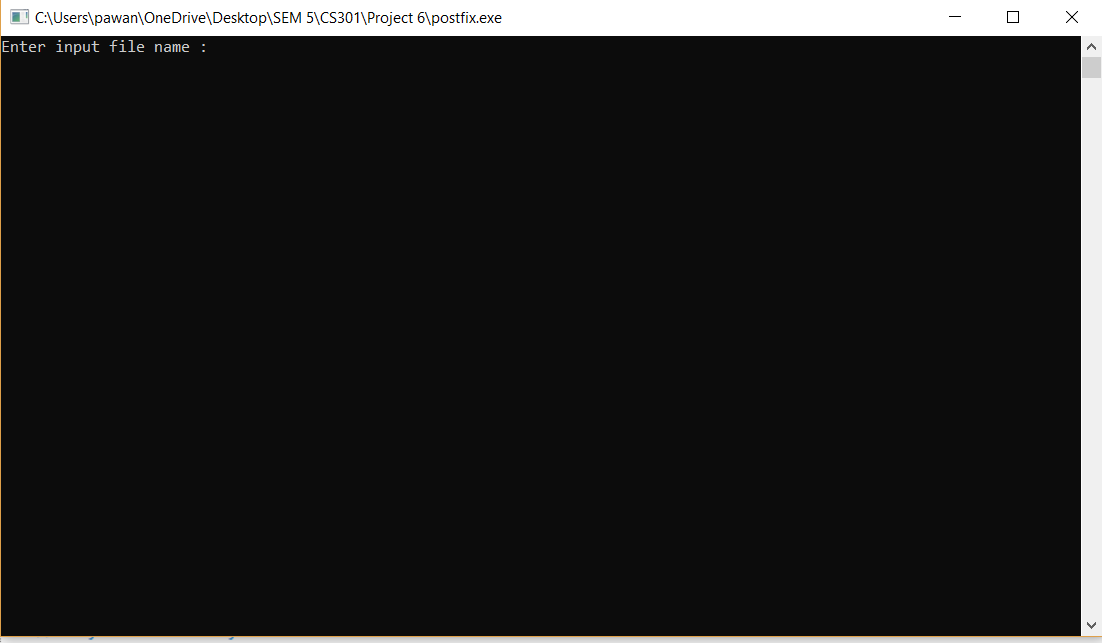
Stack::Item Stack::pop()

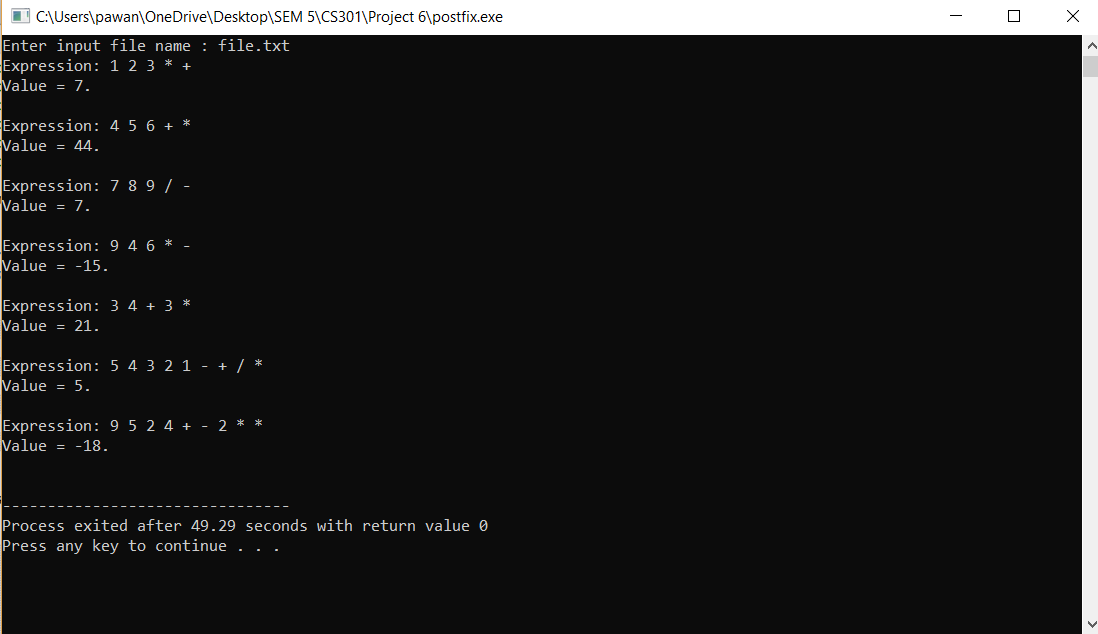
{

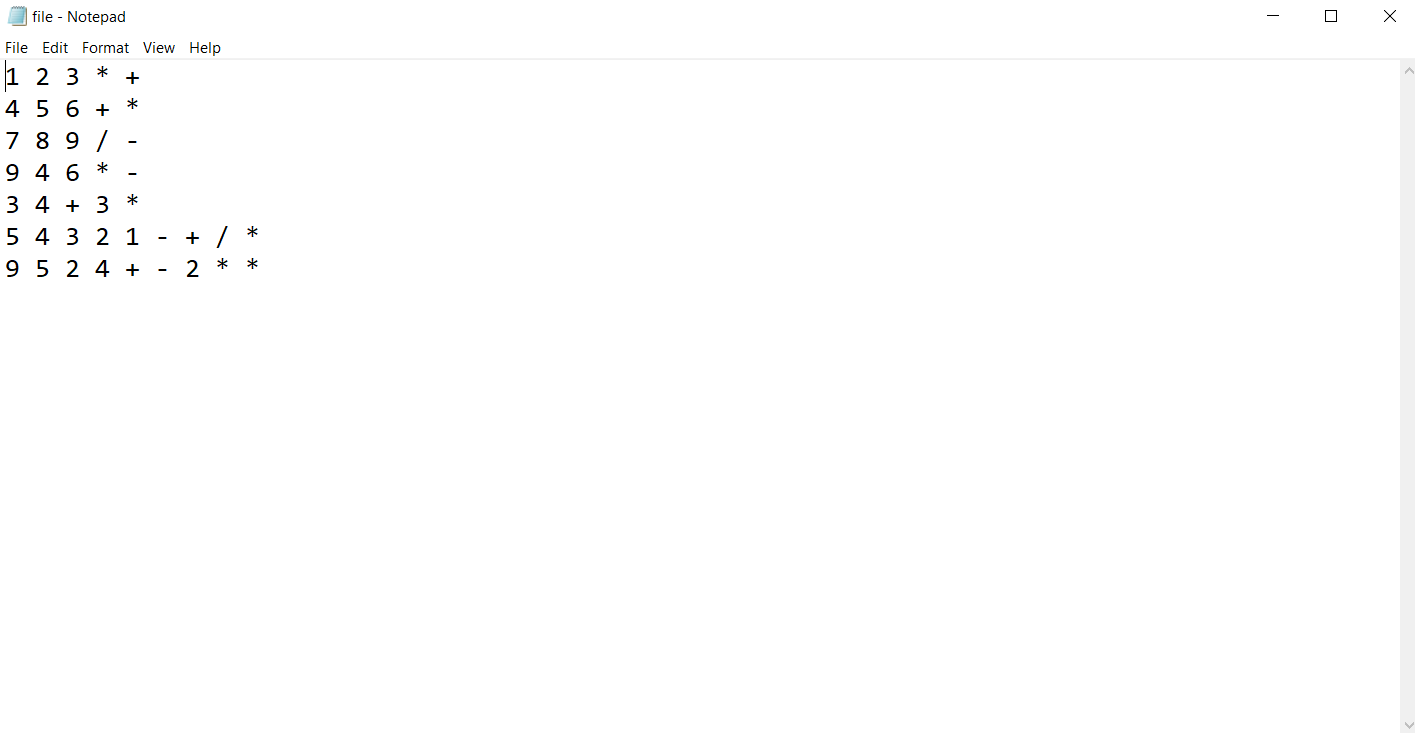
return data[--used];

}

**Test Document**

****

****

****

**Summary**

In this project, we implemented a program that reads postfix expressions from a file, one expression per line, echoes the expressions, evaluates them using the stack-based algorithm described in class, and reports their values.

We implemented stack ADT in C++ in this project. Stack is a last-in first-out structure. Whatever we push an item in the stack goes on top of the stack and the whenever we pop the stack the top item gets returned.