Project-7

Name: Pawan Dhungana

Project no: 7

Due Date: October 30, 2018

Design Document

Introduction

The conventional notation in which we usually write arithmetic expressions is called infix notation; in it, operators are written between their operands: X + Y. Such expressions can be ambiguous; do we add or multiply first in the expression 5 + 3 * 2? Parentheses and rules of precedence and association clarify such ambiguities: multiplication and division take precedence over addition and subtraction, and operators associate from left to right.

Here, a program is designed, implemented and tested to exercise a stack-based algorithm that evaluates infix expressions according to the conventional rules of precedence and association. The program echoes the expressions, evaluates them using the stack-based algorithm described in class, and reports their values.

Data Structures

The data structure used here is stack. The program uses a class named Stack. Here, a stack abstract data type is implemented in a class using a linked (pointer-based) stack implementation. Item is used as the data type of the integers in the file. The default constructor stack() is used that initializes the value of top to be NULL and count to be zero.

Functions

The program uses the default constructor as an inline function that initializes the stack to be empty. A destructor is used to return all the stack's dynamic memory to the heap. 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. Precedence() is also declared outside the class that returns the operator precedence level number. There are four other functions inside a class declared under public member function. They are: push(), pop(), empty(), size() and peek(). 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 peek() function returns the top value of the stack without changing the stack.

Main Program

In the main program, the user is asked to input the file that contains the infix expressions. The program reads the name of a file, then from the file reads syntactically correct infix expressions involving single-digit integers and the binary arithmetic operators +, -, *, and /. If the file is not found, the program terminates. 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 usually write arithmetic expressions is called infix notation; in it, operators are written between their operands: X + Y. Such expressions can be ambiguous; do we add or multiply first in the expression 5 + 3 * 2? Parentheses and rules of precedence and association clarify such ambiguities: multiplication and division take precedence over addition and subtraction, and operators associate from left to right.

Here, a program is designed, implemented and tested to exercise a stack-based algorithm that evaluates infix expressions according to the conventional rules of precedence and association. The program echoes the expressions, evaluates them using the stack-based algorithm described in class, and reports their values.

The program's name is Project7.cpp, to compile and run it, simply enter:

g++ Project7.cpp

a.out

A run of the program might look like this:

Enter input file name: infix.dat

Expression: 5 + 7 * (9 - 6) + 3
Value = 29.

Expression: 8 + 4 / 2
Value = 10.

Expression: (8 + 4) / 2
Value = 6.

Expression: (6 + (7 - 3)) * ((9 / 3) + 2) * 4
Value = 200.

Code Listing:

```cpp
#include<iostream>

#include<cstdlib>

#include<fstream>

#include<cstring>

#include<cctype>

#include<stack>


using namespace std;


int apply(char optr, int opnd1, int opnd2)
//precondition: char optr is binary arithmetic operator.
//postcondition: opnd1 and opnd2 are evaluated by the optr and
//                          the result is returned.
{
        if(optr == '+')                         //if operator is +, it adds them.
                return (opnd1+opnd2);
        else if(optr == '-')            //if operator is -, it substracts them.
                return (opnd1-opnd2);
        else if(optr == '*')            //if operator is *, it multiplies them.
                return (opnd1*opnd2);
        else if(optr == '/')            //if operator is /, it divides them.
                return (opnd1/opnd2);
}


int Precedence(char optr)
//postcondition: The level of the operator precedence level number is returned.
```

```cpp
{
    int level;
    if (optr == '*' || optr == '/')
        level = 6; // Operator precedence level number in C++
    else if (optr == '+' || optr == '-')
        level = 5;

    return level;
}

class Stack
{
public:
    typedef int Item;
    // Constructor
    Stack() { first = NULL; count = 0; }            // Inline

    // Destructor
    ~Stack();

                //modification member functions
    void push ( const Item& entry );
    Item pop ();

                //constant member functions
    size_t size () const { return count; }      // Inline
    bool empty () const { return first == NULL; }  // Inline
```

```cpp
    Item peek () const;


  private:
    // Data members
    struct Node
    {
      Item data;
       Node *next;
    };
    Node *first;
    int count;
};


int main()
{
    Stack Operand;
    Stack Operator;
    ifstream infile;

    int Operand1, Operand2, value;
        char ch, character;


        string temp;
        string filename;


    cout << "Enter the input file name : ";
    cin >> filename;
```

```cpp
infile.open(filename.c_str());   // Opens the file


if (!infile)          // If file fails to open, program terminates
{
   cout << "Could not find input file"<<endl;
}


while(getline(infile,temp))     // While file is not empty
{
     cout << "Expression: "<<temp<<endl; //print the expression


     infile.get(ch);  // Read each character one by one
   while (!infile.eof() && ch > ' ' )   // While not a new line
   {
       if(ch<='9' && ch >= '0')        // Single digit integers
     {
            // Conversion of characters into integers
         Operand.push(ch-'0');        // push the integer into the stack
     }


     else if (ch == '(')
     {
        Operator.push(ch);
     }


     else if (ch == '+' || ch == '-' || ch == '*' || ch == '/') // For all operators
     {
```

```
        while( (!Operator.empty()) && (Operator.peek()!='(') &&
(Precedence(Operator.peek()) >= Precedence(ch)))
        {
            Operand2 = Operand.pop();        // pop the top Operand
            Operand1 = Operand.pop();        // pop the top Operand again


            character = Operator.pop();


            value = apply(character,Operand1,Operand2);    // Call the function to do
appropriate calculation
            Operand.push(value);    // Result is pushed back again
        }
                        Operator.push(ch);
                }


                else if (ch == ')')
        {
            ch= Operator.pop();
            while (ch != '(')
            {
                Operand2 = Operand.pop();        // pop the top Operand
                Operand1 = Operand.pop();        // pop the top Operand again
                value = apply(ch,Operand1,Operand2);    // Call the function to do appropriate
calculation
                Operand.push(value);    // Result is pushed back again
                ch = Operator.pop();
            }
        }
```

```cpp
        infile.get(ch)

    }

        while (!Operator.empty())

        {

           Operand2 = Operand.pop();       // pop the top Operand

           Operand1 = Operand.pop();       // pop the top Operand again

           ch = Operator.pop();

           value = apply(ch,Operand1,Operand2);    // Call the function to do appropriate
calculation

           Operand.push(value);    // Result is pushed back again

        }

        cout << endl;

                cout << " Value = " << Operand.peek() << endl;   // Display the remaining item
from stack

        }

        infile.close();    // Close the file

        return 0;

}


Stack::~Stack( ) //Destructor

{

        Node* temp;

        while ( first != NULL )

        {

         temp = first;

         first = first -> next;

         delete temp;

        }
```

```
}

void Stack::push (const Item& entry) //pushes the item onto the top of the stack

{

        Node* temp;

        temp = new Node;

        temp->data = entry;

        temp->next = first;

        first = temp;

        ++count;

}


int Stack::pop() //pops the item from the top of the stack

{

        Node* temp= first;

        Item popped = first->data;

        first = first->next;

        delete temp;

        --count;

        return popped;

}


int Stack::peek() const //returns the top value of the stack

{

        if (!empty());

                return first->data;

}
```
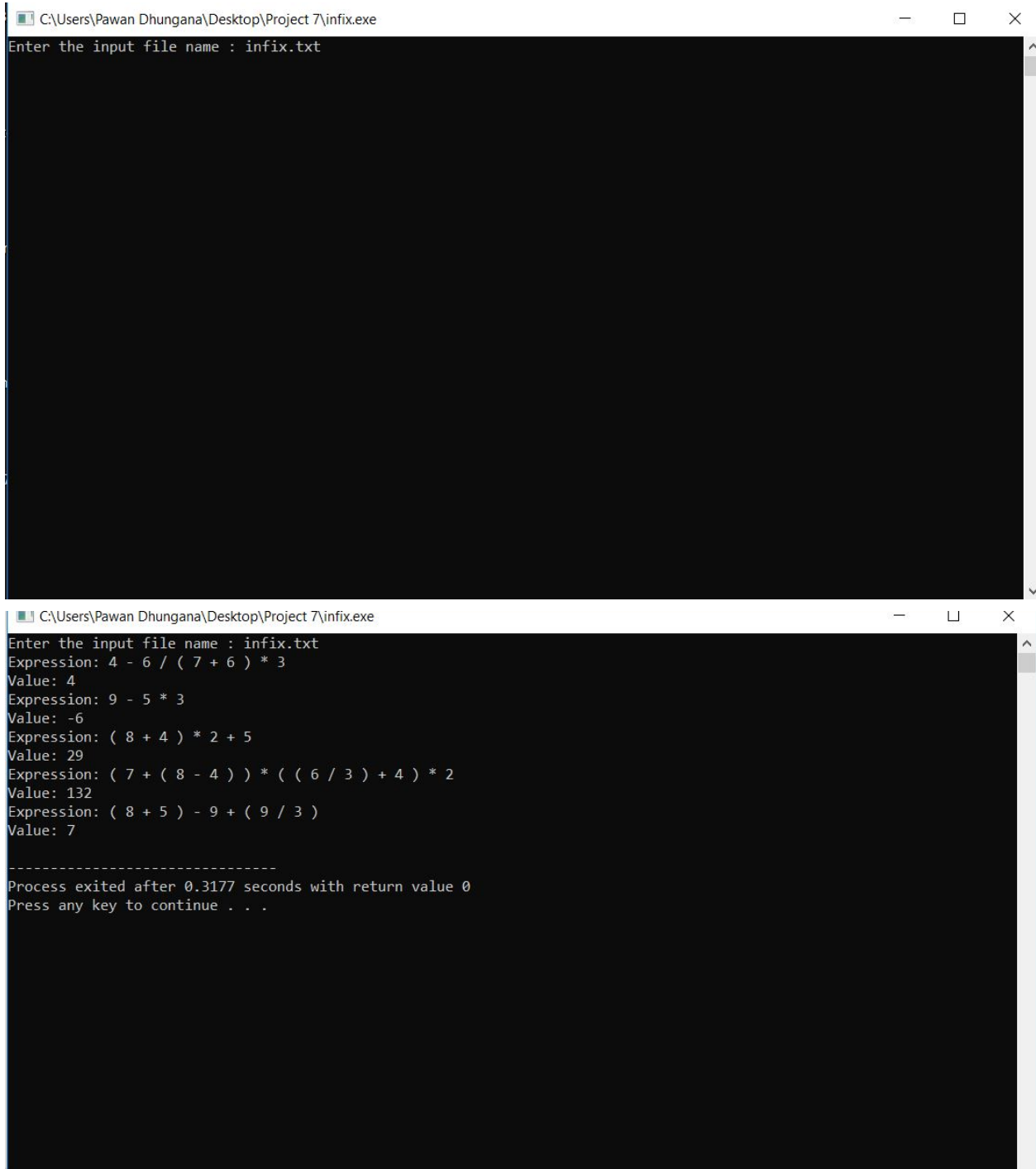
Test Document

```
Enter the input file name : infix.txt
```

```
Enter the input file name : infix.txt
Expression: 4 - 6 / ( 7 + 6 ) * 3
Value: 4
Expression: 9 - 5 * 3
Value: -6
Expression: ( 8 + 4 ) * 2 + 5
Value: 29
Expression: ( 7 + ( 8 - 4 ) ) * ( ( 6 / 3 ) + 4 ) * 2
Value: 132
Expression: ( 8 + 5 ) - 9 + ( 9 / 3 )
Value: 7

--------------------------------
Process exited after 0.3177 seconds with return value 0
Press any key to continue . . .
```

```
infix - Notepad                                                        —   □   ✕
File  Edit  Format  View  Help
4 - 6 / ( 7 + 6 ) * 3
9 - 5 * 3
( 8 + 4 ) * 2 + 5
( 7 + ( 8 - 4 ) ) * ( ( 6 / 3 ) + 4 ) * 2
( 8 + 5 ) - 9 + ( 9 / 3 )
```

Summary

In this project, we implemented a program that reads infix 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.

I learned about how the stacks work in C++ from this project. I encountered problems if I calculate the infix expressions in a different function so I had to just put it in the main function.