**Chris Ceron**

**CSCI 220 – MW 2:40P.M**

**Project Number 2**

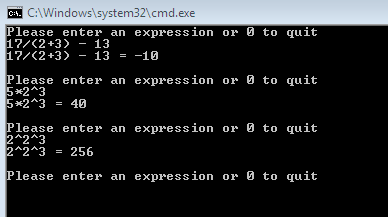
**Microsoft Visual Studio**

**Notes**

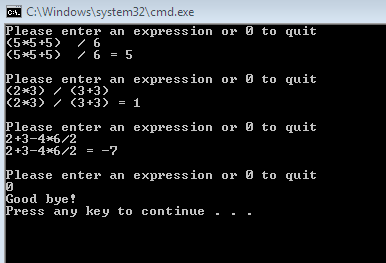
This program is 100% functional and **includes extra credit 1.** I implemented a stack class using an array for simplicity and to work on the Calculator class without worrying about a functional linked list. If I had more time, I would go back to my stack class and implement a linked list instead as it is more efficient than a fixed array. I did not encounter many issues with this project, but I created pseudocode to make the logic implementation of the program easier. To manage the order of operations, I found that giving each operand a weight helped in the implementation as well. The program also manages negative numbers.

For the extra credit, I implemented return statements for an invalid amount of parenthesis, for characters that are not valid operands or integers (such as adding a variable to the expression), and having too many operands (such as 17 ++5).

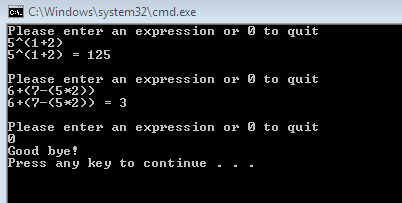
**Basic examples of expressions**



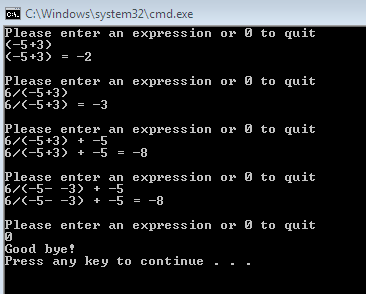
**Order of operation examples**



**More complicated expressions**

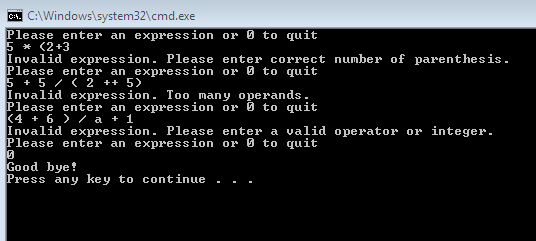


**Implementation of negative numbers**

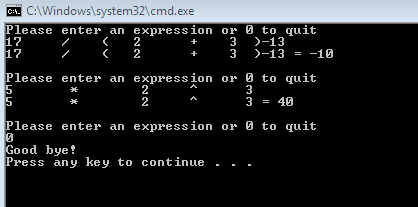


**Extra Credit**

**Prints out errors with illegal expressions**



**Manages spacing**



**Source Code**

#pragma once

#include<string>

template <class E> class Stack

{

private:

E expression[20];

int numElements;

int insertion;

public:

Stack();

int size() const;

bool empty() const;

const E top();

void push(E input);

E pop();

};

template<class E> inline Stack<E>::Stack()

{

numElements = 0;

insertion = -1;

}

template<class E> inline int Stack<E>::size() const

{

return numElements;

}

template<class E> inline bool Stack<E>::empty() const

{

if (numElements == 0)

return true;

else return false;

}

template<class E> inline const E Stack<E>::top()

{

if (Stack<E>::empty())

{

cout << "Stack is empty" << endl;

return NULL;

}

else return expression[insertion];

}

template<class E> inline void Stack<E>::push(E input)

{

if (numElements == 20)

{

cout << "Stack is full. Did not add: " << input << endl;

}

else

{

numElements++;

insertion++;

expression[insertion] = input;

}

}

template<class E> inline E Stack<E>::pop()

{

if (Stack<E>::empty())

{

cout << "Stack is empty" << endl;

return NULL;

}

else

{

numElements--;

return expression[insertion--];

}

}

#pragma once

#include "Stack.h"

#include <string>

using namespace std;

class Calculator

{

private:

string infixExpression;

string postfixExpression;

public:

string infixPostfix(string expression);

int evaluation(string postfix);

int operandWeight(char operand);

int balance(string input);

};

#include "Calculator.h"

#include <string>

#include<iostream>

#include<math.h>

using namespace std;

string Calculator::infixPostfix(string expression)

{

Stack<char> toPostfix;

string postFix = "";

string previous = "";

int i = 0;

while( i < expression.length())

{

int weightOfChar = operandWeight(expression[i]);

if (weightOfChar == 0)

{

while (expression[i] >= '0' && expression[i] <= '9')

{

postFix += expression[i];

i++;

}

postFix += " ";

}

else if (weightOfChar == -1)

i++;

else if (weightOfChar > 0)

{

if (toPostfix.empty())

{

toPostfix.push(expression[i]);

}

else if (expression[i] == ')')

{

while (weightOfChar > operandWeight(toPostfix.top()))

{

postFix += toPostfix.pop();

postFix += " ";

}

toPostfix.pop();

}

else if (weightOfChar == 3 && weightOfChar == operandWeight(toPostfix.top()))

toPostfix.push(expression[i]);

else if (expression[i] == '-' && operandWeight(previous[0]) > 0 && previous[0] != ')')

{

postFix += expression[i];

}

else if (weightOfChar > operandWeight(toPostfix.top()))

{

toPostfix.push(expression[i]);

}

else if (weightOfChar <= operandWeight(toPostfix.top()))

{

if (toPostfix.top() == '(')

{

toPostfix.push(expression[i]);

}

else

{

while (!toPostfix.empty() && weightOfChar <= operandWeight(toPostfix.top()) && toPostfix.top() != '(')

{

postFix += toPostfix.pop();

postFix += " ";

}

toPostfix.push(expression[i]);

}

}

i++;

}

if(expression[i-1] != ' ')

previous = expression[i-1];

}

while (!toPostfix.empty())

{

postFix += toPostfix.pop();

postFix += " ";

}

return postFix;

}

int Calculator::evaluation(string postfix)

{

Stack<int> evaluate;

int i = 0;

while (i < postfix.length())

{

int weightOfChar = operandWeight(postfix[i]);

int number, first, second;

string temp = "";

if (weightOfChar == 0)

{

while (postfix[i] >= '0' && postfix[i] <= '9')

{

temp += postfix[i];

i++;

}

number = stoi(temp);

evaluate.push(number);

}

else if (weightOfChar == -1)

i++;

else if (weightOfChar > 0)

{

if (postfix[i] == '-' && (postfix[i + 1] >= '0' && postfix[i + 1] <= '9'))

{

temp += postfix[i];

i++;

while (postfix[i] >= '0' && postfix[i] <= '9')

{

temp += postfix[i];

i++;

}

number = stoi(temp);

evaluate.push(number);

}

else

{

second = evaluate.pop();

first = evaluate.pop();

if (weightOfChar == 3)

{

number = pow(first, second);

evaluate.push(number);

}

else if (weightOfChar == 2)

{

if (postfix[i] == '%')

{

number = first % second;

evaluate.push(number);

}

else if (postfix[i] == '/')

{

number = first / second;

evaluate.push(number);

}

else

{

number = first \* second;

evaluate.push(number);

}

}

else if (weightOfChar == 1)

{

if (postfix[i] == '-')

{

number = first - second;

evaluate.push(number);

}

else

{

number = first + second;

evaluate.push(number);

}

}

i++;

}

}

}

int result = evaluate.pop();

return result;

}

int Calculator::operandWeight(char operand)

{

switch(operand)

{

case'+':

case'-': return 1;

case'\*':

case'/':

case'%': return 2;

case'^': return 3;

case'(':

case')': return 4;

case' ': return -1;

default: return 0;

}

}

int Calculator::balance(string input)

{

Stack<char>toBalance;

string previous = "";

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

{

if (input[i] == '(')

toBalance.push(input[i]);

else if (input[i] == ')')

{

if (toBalance.empty())

return 1;

else if (toBalance.top() == '(' && input[i] == ')')

toBalance.pop();

else if (toBalance.top() == '(' && input[i] != ')')

return 1;

}

else if (operandWeight(input[i]) == 0 && input[i] != ' ')

{

if (input[i] > '9' || input[i] < '0')

return 2;

}

else if (operandWeight(input[i]) > 0 && operandWeight(previous[0]) > 0 && input[i] != '-' && previous[0] != ')')

return 3;

if (input[i] != ' ')

previous = input[i];

}

if (toBalance.empty())

return 0;

else return 1;

}

#include<iostream>

#include<string>

#include "Stack.h"

#include "Calculator.h"

using namespace std;

int main()

{

Calculator calc;

string expression, postfix;

char input = NULL;

Stack<int> testing;

bool run = true;

while (run)

{

cout << "Please enter an expression or 0 to quit" << endl;

getline(cin, expression);

if (expression != "0")

{

int validation = calc.balance(expression);

if (validation == 1)

cout << "Invalid expression. Please enter correct number of parenthesis." << endl;

else if (validation == 2)

cout << "Invalid expression. Please enter a valid operator or integer." << endl;

else if (validation == 3)

cout << "Invalid expression. Too many operands." << endl;

else

{

postfix = calc.infixPostfix(expression);

cout << expression << " = " << calc.evaluation(postfix) << endl << endl;

}

}

else run = false;

}

cout << "Good bye!" << endl;

}