Kendall Belles

CS-2413 Data Structures

Project 2

March 22, 2013

Table of Contents

[Design Documents: 2](#_Toc351718696)

[Class Layout: 2](#_Toc351718697)

[Flow Charts: 3](#_Toc351718698)

[Additional Pages: 7](#_Toc351718699)

[ReadMe File: 11](#_Toc351718700)

[User Manual 12](#_Toc351718701)

[Introduction: 12](#_Toc351718702)

[Functionality: 12](#_Toc351718703)

[Errors: 13](#_Toc351718704)

[Source Code: 14](#_Toc351718705)

[project2.cpp 14](#_Toc351718706)

[calculator.h 15](#_Toc351718707)

[calculator.cpp 16](#_Toc351718708)

[Compile: 27](#_Toc351718709)

[Example Output: 27](#_Toc351718710)

# Design Documents:

These are the initial design documents for the calculator module and driver. Some changes may have been made during implementation that is not reflected in these documents. Please refer to full code for current version.

## Class Layout:

|  |  |  |
| --- | --- | --- |
| union Data | |  |
| double | num | Post validation operand |
| char | op | Post validation operator |

|  |  |  |
| --- | --- | --- |
| struct Expression | |  |
| Data | data | Pertinent data of expression at the moment |
| bool | isNumber | If true, “data” stores an operand |
| int | positionID | Position in original expression, for error tracking |

|  |  |  |  |
| --- | --- | --- | --- |
| class Calculator | | |  |
| attributes | | |  |
| list <Expression> | infix |  | Parsed infix expression |
| list <Expression> | postfix |  | Parsed postfix expression |
| string | exp |  | Raw input |
| double | ans |  | Evaluation of expression |
| bool | valid |  | True if expression evaluated correctly |
| bool | errAllow |  | True if error messages can be displayed |
| operations | | |  |
|  | Calculator | string, bool = true | Conducts all operations |
|  | ~Calculator | () |  |
| bool | parseInfix | () | Populate and validate “infix”, false if error |
| bool | infixToPostfix | () | Populate “postfix”, false if error |
| bool | evalPostfix | () | Populate “ans”, false if error |
| int | precedence | char | Returns the precedence of passed operator |
| bool | isDouble | string, double& | Double validator, false if error |
| bool | isOperator | string, char& | Operator validator, false if error |
| bool | isValid | () | “valid” accessor |
| double | answer | () | “ans” accessor |

## Flow Charts:











# ReadMe File:

=== calculator.h ===

The calculator module consists of two files, calculator.h and calculator.cpp. The calculator module evaluates an infix expression and stores the result.

=== Description ===

The calculator module takes an infix expression as a string object and an optional Boolean value. The Boolean value will allow/disallow error reporting for the calculator module and is defaulted to true. On initialization of the calculator object, the expression will be evaluated. If the evaluation was successful, then the function isValid() will return true and the function answer() will return the result. If an error was encountered during and error reporting is allowed, then an error will be outputted to the screen.

=== Use ===

Extract the files to the desired directory and add #include “calculator.h” to the project file. Compile normally.

To implement the calculator object, it is recommended that a pointer to class Calculator is created. Prompt the user for the infix expression and store it as a string object. Use the pointer to initialize a new Calculator object passing the expression as an argument. Check the return value of isValid() and, if true, retrieve answer(). Optionally, delete the calculator class to prepare for the next execution.

=== Example ===

Calculator \*calc1;

cout << "Enter an infix expression:" << endl;

getline (cin, infix);

calc1 = new Calculator(infix, false);

if ((\*calc1).isValid())

cout << "The result is: " << (\*calc1).answer() << endl;

delete calc1;

=== Author and Contact ===

Primary author: Ken Belles, kendall.belles@cameron.edu

Contributions by: Dr. Chao Zhao, chaoz@cameron.edu

# User Manual

## Introduction:

The calculator module allows a program to read an expression and compute its result. If the expression contains errors, the module can print error messages. These error messages can also be turned off.

## Functionality:

The calculator module accepts real, positive numbers. Commas and scientific notation will result in an error and the expression not being evaluated by the calculator module. Examples:

|  |  |
| --- | --- |
| Acceptable | Unacceptable |
| 1 | -4 |
| 12.56 | 1,951 |
| 0.00547 | 1e15 |

The calculator module can process addition, subtraction, multiplication, division, exponentials, and parenthesis.

Exponentials:

Exponential expressions use the carat (^) character.

Multiplication:

Multiplication uses the asterisk (\*) character. Multiplication must always be explicitly stated in the expression. Implied multiplication, such as will result in an error.

Division:

Division uses the forward slash (/) character. Division by zero will result in an error.

Addition:

Addition uses the addition (+) character.

Subtraction:

Subtraction uses the dash (-) character. This character always means subtraction. It never means a negative number. Negative numbers can be created like so:

Parenthesis:

Parenthesis use the open parenthesis and close parenthesis characters. Every open parenthesis must be matched to a closed parenthesis, and vice versa, or the calculator module will generate an error.

## Errors:

The following table shows the errors that the calculator module can generate and a brief explanation of each one:

|  |  |
| --- | --- |
| Evalution of the expression was terminated. | The calculator module encountered an error. This message will appear with every error. |
| The character, “x” in the expression is invalid. | The calculator module found an illegal character in the expression. This halted processing of the expression. |
| The expression cannot be all spaces. | Self-explanatory. |
| The expression is too short to be valid. | If the expression is empty of characters, the calculator module will return this error. |
| The expression result is out of range. | During processing of the expression, a number became too large or too small for accurate processing. |
| The expression results in division by zero. | During processing of the expression, a number that is the denominator in division became zero. |
| The number, “xxx” in the expression is invalid. | The calculator module encountered a number that was not properly formatted. |
| The operand “(”, at the shown position, was not matched with a “) “. | The calculator module encountered an improperly pared set of parenthesis. Only the open parenthesis is present. |
| The operand “)”, at the shown position, was not matched with a “(“. | The calculator module encountered an improperly pared set of parenthesis. Only the close parenthesis is present. |
| There are too many operands in the expression. | There were too many numbers without enough mathematical operators to modify them in the expression. |
| There are too many operators in the expression. | There were too many mathematical operators in the expression without enough numbers for them to modify. |

# Source Code:

## project2.cpp

#include <iostream>

#include "calculator.h"

using namespace std;

bool isValDone (string); //validates 'y' or 'n', case-insensitive

int main() {

cout << endl << "Welcome to the basic expression calculator!" << endl << endl;

cout << "If you have any quetions about how to use this calculator" << endl;

cout << "or what a particular error means, then please check the" << endl;

cout << "user's manual." << endl;

//Calculator pointer, as recommended by the readMe file

Calculator \*calc1;

bool done;

string infix;

string reply;

//while the user is not done

do{

cout << endl << "Enter an infix expression:" << endl;

//while infix == ""

do{

getline (cin, infix);

}while(infix.length() < 1);

//create new Calculator instance to run computation, error reporting on

calc1 = new Calculator(infix);

//if successfully completed, print result

if ((\*calc1).isValid())

cout << "The result is: " << (\*calc1).answer() << endl;

delete calc1;

//while not y/n

do{

cout << "Do you have another expression to evaluate? y/n" << endl;

cin >> reply;

//validate reply

if (!isValDone(reply))

cout << "Please answer only \"y\", \"Y\", \"n\", or \"N\" without the quotes." << endl;

}while(!isValDone(reply));

if (reply[0] == 'n' || reply[0] == 'N')

done = true;

else

done = false;

}while(!done);

}

bool isValDone (string ans) {

if (ans.length() > 1)

return false;

if (ans[0] == 'y' || ans[0] == 'Y' || ans[0] == 'n' || ans[0] == 'N')

return true;

else

return false;

}

## calculator.h

#ifndef CALCULATOR\_H

#define CALCULATOR\_H

#include <string>

#include <list>

using namespace std;

union Data {

double num; //postvalidation operand

char op; //postvalidation operator

};

struct Expression {

Data data; //pertinent data of expression at the moment

bool isNumber; //if true, data stores an operand

int positionID; //error tracking

};

class Calculator {

private:

list <Expression> infix; //parsed infix expression

list <Expression> postfix; //parsed postfix expression

string exp; //unparsed input

double ans; //evaluated answer to expression

bool valid; //all parts of parsing and evaluation completed correctly

bool errAllow; //stores weather the class can show error messages

public:

Calculator(string, bool); //master of operations

~Calculator(); //delete instance of Calculator class

bool parseInfix(); //populate and validate infix list, return false if not completed

bool infixToPostfix(); //populate postfix, return false if not completed

bool evalPostfix(); //populate ans, return false if not completed

int precedence(char); //returns the precedence of the operator

bool isDouble(string, double&); //validate double from string or return false

bool isOperator(string, char&); //validate char from string or return false

bool isValid(); //"valid" accessor

double answer(); //"ans" accessor

};

#include "calculator.cpp"

#endif

## calculator.cpp

#include <cfloat>

#include <cmath>

#include <stack>

#include <iostream>

#include <iomanip>

using namespace std;

Calculator::Calculator(string input, bool err = true) { //master of operations

exp = input;

errAllow = err;

valid = true;

parseInfix();

if (valid)

infixToPostfix();

if (valid)

evalPostfix();

if (!valid) {

if (errAllow)

cout << "Evalution of the expression was terminated." << endl;

}

}

Calculator::~Calculator() { //delete instance of Calculator class

infix.clear();

postfix.clear();

}

bool Calculator::parseInfix() { //populate infix list, return false if not completed

if (exp.length() < 1) {

if (errAllow)

cout << "The expression is too short to be valid." << endl;

valid = false;

return false;

}

string temp;

double tempDbl;

char tempChar;

Expression tempExp;

//for each character of exp

for (unsigned i=0; i<exp.length(); ) {

temp = "";

//if valid character (digit, operator or space)

if (int(exp[i]) > 39 && int(exp[i]) < 58 && int(exp[i]) != 44 || exp[i] == '^' || exp[i] == ' ') {

//if digit or decimal

if (int(exp[i]) > 47 && int(exp[i]) < 58 || exp[i] == '.') {

//while digit or decimal

while (int(exp[i]) > 47 && int(exp[i]) < 58 || exp[i] == '.') {

temp += exp[i];

i++;

}

//if temp is valid double type

if (isDouble(temp, tempDbl)) {

tempExp.data.num = tempDbl;

tempExp.isNumber = true;

tempExp.positionID = i-1;

infix.push\_back(tempExp);

//temp not valid double type

} else {

valid = false;

if (errAllow)

cout << "The number, \"" << temp << "\" in the expression is invalid." << endl;

return false;

}

//if operator

} else if (int(exp[i]) != 32) {

temp = exp[i];

if (isOperator(temp, tempChar)) {

tempExp.data.op = tempChar;

tempExp.isNumber = false;

tempExp.positionID = i;

infix.push\_back(tempExp);

i++;

}

//if space

} else

i++;

//invalid character

} else {

temp = exp[i];

valid = false;

if (errAllow)

cout << "The character, \"" << temp << "\" in the expression is invalid." << endl;

return false;

}

}

//if infix was not populated

if (infix.empty()) {

if (errAllow)

cout << "The expression cannot be all spaces." << endl;

valid = false;

return false;

}

return true;

}

bool Calculator::infixToPostfix() { //populate postfix, return false if not completed

stack <Expression> opStk;

list <Expression>::iterator itr;

itr=infix.begin();

//while itr still points within infix

while (itr!=infix.end()) {

//if itr is pointing to a number

if ((\*itr).isNumber) {

postfix.push\_back(\*itr);

itr++;

//if itr is pointing to an operator

} else {

switch ((\*itr).data.op) {

//if operator is '('

case '(':

opStk.push(\*itr);

itr++;

break;

//if operator is ')'

case ')':

while (!opStk.empty() && (opStk.top()).data.op != '(') {

postfix.push\_back(opStk.top());

opStk.pop();

}

if (opStk.empty()) {

valid = false;

if (errAllow) {

cout << setw((\*itr).positionID+1) << "^" << endl;

cout << "The operand \")\", at the shown position, was not matched with a \"(\"." << endl;

}

return false;

} else {

opStk.pop();

itr++;

}

break;

//all other operators

default:

if (opStk.empty()) {

opStk.push(\*itr);

} else if (precedence((\*itr).data.op) > precedence((opStk.top()).data.op)) {

opStk.push(\*itr);

} else {

while (!opStk.empty() && (opStk.top()).data.op != '(' && precedence((\*itr).data.op) <= precedence((opStk.top()).data.op)) {

postfix.push\_back(opStk.top());

opStk.pop();

}

opStk.push(\*itr);

}

itr++;

break;

} //end switch

}

} //end while

while (!opStk.empty()) {

if ((opStk.top()).data.op == '(') {

valid = false;

if (errAllow) {

cout << setw((opStk.top()).positionID+1) << "^" << endl;

cout << "The operand \"(\", at the shown position, was not matched with a \")\"." << endl;

}

return false;

}

postfix.push\_back(opStk.top());

opStk.pop();

}

return true;

}

bool Calculator::evalPostfix() { //populate ans, return false if not completed

stack <double> operands;

list <Expression>::iterator itr;

double num1, num2;

for (itr=postfix.begin(); itr!=postfix.end(); itr++) {

if ((\*itr).isNumber)

operands.push((\*itr).data.num);

else {

if (operands.size() < 2) {

valid = false;

cout << "There are too many operators in the expression." << endl;

return false;

} else {

num1 = operands.top();

operands.pop();

num2 = operands.top();

operands.pop();

switch ((\*itr).data.op) {

case '^':

if ((num1 > 0 && log(DBL\_MAX) / log(num2) >= num1) || (num1 < 0 && log(DBL\_MIN) / log(num2) <= num1)) {

operands.push(pow(num2,num1));

break;

} else {

valid = false;

if (errAllow)

cout << "The expression result is out of range." << endl;

return false;

}

case '\*':

if (num1 == 0 || num2 == 0) {

operands.push(num2 \* num1);

break;

} else if (num1 > -1 && num1 < 1 && num2 > -1 && num2 < 1) {

if (num1 > 0 != num2 > 0) {

if ((num1 < 0 && -DBL\_MIN/num2 <= num1) || (num2 < 0 && -DBL\_MIN/num1 <= num2)) {

operands.push(num2 \* num1);

break;

} else {

valid = false;

if (errAllow)

cout << "The expression result is out of range." << endl;

return false;

}

} else if ((num1 > 0 && DBL\_MIN/num1 <= num2) || (num1 < 0 && DBL\_MIN/num1 >= num2)) {

operands.push(num2 \* num1);

break;

} else {

valid = false;

if (errAllow)

cout << "The expression result is out of range." << endl;

return false;

}

} else if ((num2 > 0 && num1 > 0) || (num2 < 0 && num1 < 0)){

if ((num2 > 0 && DBL\_MAX / num2 >= num1) || (num2 < 0 && DBL\_MAX / num2 <= num1)) {

operands.push(num2 \* num1);

break;

} else {

valid = false;

if (errAllow)

cout << "The expression result is out of range." << endl;

return false;

}

} else {

if ((num1 < 0 && -DBL\_MAX / num2 <= num1) || (num2 < 0 && -DBL\_MAX / num1 <= num2)) {

operands.push(num2 \* num1);

break;

} else if ((num1 > -1 && num1 < 1) || (num2 > -1 && num2 < 1)) {

operands.push(num2 \* num1);

break;

} else {

valid = false;

if (errAllow)

cout << "The expression result is out of range." << endl;

return false;

}

}

case '/':

if (num1 == 0.0) {

valid = false;

if (errAllow)

cout << "The expression results in division by zero." << endl;

return false;

} else if (num2 == 0) {

operands.push(num2 / num1);

break;

} else if (num2 > 0 == num1 > 0) {

if ((num1 > -1 && num1 < 1) != (num2 > -1 && num2 < 1)) {

if (num2 >= 1 && DBL\_MAX \* num1 >= num2) {

operands.push(num2 / num1);

break;

} else if (num1 < -1 && DBL\_MIN \* num2 <= 1 / num1) {

operands.push(num2 / num1);

break;

} else {

valid = false;

if (errAllow)

cout << "The expression result is out of range." << endl;

return false;

}

} else {

operands.push(num2 / num1);

break;

}

} else if (num1 > -1 && num1 < 1) {

operands.push(num2 / num1);

break;

} else {

if (num2 >= 1 && -DBL\_MAX \* num1 >= num2) {

operands.push(num2 / num1);

break;

} else if (num2 > 0 && -DBL\_MIN \* num1 <= num2) {

operands.push(num2 / num1);

break;

} else {

valid = false;

if (errAllow)

cout << "The expression result is out of range." << endl;

return false;

}

}

case '+':

if (num1 >= 0 != num2 >= 0) {

operands.push(num2 + num1);

break;

} else if (num1 >= 0 && DBL\_MAX - num1 >= num2) {

operands.push(num2 + num1);

break;

} else if (num1 < 0 && -DBL\_MAX - num1 <= num2) {

operands.push(num2 + num1);

break;

} else {

valid = false;

if (errAllow)

cout << "The expression result is out of range." << endl;

return false;

}

default:

if (num2 < 0 && num1 >= 0) {

if (-DBL\_MAX + num1 <= num2) {

operands.push(num2 - num1);

break;

} else {

valid = false;

if (errAllow)

cout << "The expression result is out of range." << endl;

return false;

}

} else {

operands.push(num2 - num1);

break;

}

}

}

}

}

if (operands.size() == 1) {

ans = operands.top();

return true;

} else {

valid = false;

if (errAllow)

cout << "There are too many operands in the expression." << endl;

return false;

}

}

int Calculator::precedence(char temp) { //returns the precedence of the operator

if (temp == '(' || temp == ')')

return 3;

if (temp == '^')

return 2;

if (temp == '\*' || temp == '/')

return 1;

else

return 0;

}

bool Calculator::isDouble(string checkMe, double &returnMe) { //validate double from string or return false

bool correct = false, negativeNumber = false, decSet = false;

int digits = 0;

returnMe = 0;

negativeNumber = false;

for (unsigned i=0; i<checkMe.length(); i++) {

// if first char == '-'

if (int(checkMe.at(i)) == 45 && i == 0)

negativeNumber = true;

// if number char or decimal point

if ((int(checkMe.at(i)) < 58 && int(checkMe.at(i)) > 47) || int(checkMe.at(i)) == 46) {

//if digit

if (int(checkMe.at(i)) < 58 && int(checkMe.at(i)) > 47) {

//if within range

if (DBL\_MAX - returnMe/pow(10.0,digits) >= int(checkMe.at(i)) - 48) {

correct = true;

returnMe \*= 10;

returnMe += int(checkMe.at(i)) - 48;

if (decSet)

digits++;

} else {

correct = decSet = false;

digits = 0;

returnMe = 0.0;

break;

}

//if '.' and '.' has already appeared

} else if (decSet) {

digits = 0;

returnMe = 0.0;

correct = decSet = false;

break;

//char = '.' and '.' has not apeared yet

} else

decSet = true;

// if '-' but not leading or not a valid char

} else if(!negativeNumber || i != 0) {

digits = 0;

returnMe = 0.0;

correct = decSet = false;

break;

}

//end for loop

}

if (checkMe.length() < 1)

correct = false;

// if correct number had a leading '-'

if(negativeNumber)

returnMe \*= -1;

returnMe /= pow(10.0,digits);

return correct;

}

bool Calculator::isOperator(string checkMe, char &good) { //validate char from string or return false

switch (checkMe[0]) {

case '+': good = '+'; return true;

case '-': good = '-'; return true;

case '\*': good = '\*'; return true;

case '/': good = '/'; return true;

case '(': good = '('; return true;

case ')': good = ')'; return true;

case '^': good = '^'; return true;

default: return false;

}

}

bool Calculator::isValid() { //"valid" accessor

return valid;

}

double Calculator::answer() { //"ans" accessor

return ans;

}

# Compile:

C:\Users\Ken\Documents\Data Struct\Project2>cl project2.cpp

Microsoft (R) C/C++ Optimizing Compiler Version 15.00.30729.01 for x64

Copyright (C) Microsoft Corporation. All rights reserved.

project2.cpp

C:\Program Files (x86)\Microsoft Visual Studio 9.0\VC\Include\xlocale(342) : warning C4530: C++ exception handler used, but unwind semantics are not enabled. Specify /EHsc

Microsoft (R) Incremental Linker Version 9.00.30729.01

Copyright (C) Microsoft Corporation. All rights reserved.

/out:project2.exe

project2.obj

# Example Output:

C:\Users\Ken\Documents\Data Struct\Project2>project2.exe

Welcome to the basic expression calculator!

If you have any quetions about how to use this calculator

or what a particular error means, then please check the

user's manual.

Enter an infix expression:

15.24\*(0-123)

The result is: -1874.52

Do you have another expression to evaluate? y/n

j

Please answer only "y", "Y", "n", or "N" without the quotes.

Do you have another expression to evaluate? y/n

Y

Enter an infix expression:

10^310

The expression result is out of range.

Evalution of the expression was terminated.

Do you have another expression to evaluate? y/n

y

Enter an infix expression:

125-(98+48.3/98) /147

The result is: 124.33

Do you have another expression to evaluate? y/n

y

Enter an infix expression:

125-(98+48.3/98 /147

^

The operand "(", at the shown position, was not matched with a ")".

Evalution of the expression was terminated.

Do you have another expression to evaluate? y/n

n

C:\Users\Ken\Documents\Data Struct\Project2>