CPP -Exercise

ABDULLAH AL NAFFAKH

Contents

Usage details

The library currently provides only one public function that behaves autonomously depending on the parameter passed into the function

```
double MathematicaLib::DoTheMath(char const* equation)
```

The library fulfils two purposes, both fulfilled by a single function call:

- Can be used as a calculator application.
- Provides a method of solving mathematical equations

The implementation of a single function call is done deliberately to simplify the use of the library, the parameters which can be passed to this function are shown in the table below:

Valid inputs (All inputs must be a *char*)		
Input/Format	Return value	purpose
"Help"	0.0	Displays command and descriptions
"Load"	0.0	Loads instructions
"ClearRestart"	0.0	Resets the system and deletes history and results
"History"	0.0	Loads a history of commands and equations entered, chronologically.
"Undo"	0.0	Undo the solving process
"x * y "	Result	Performs a multiplication operation
"x / y "	Result	Performs a division operation
"x + y "	Result	Performs a addison operation
"x – y "	Result	Performs a subtraction operation
"x * y + k – y * j / p + o "	Result	Solves an entire equation consisting of a mixed operations
other	0.0	Any other value is invalid

Usage examples: Note that the use of spaces is mandatory between values and characters and a space at the end of each equation passed, as shown below. Spaces are not required when pacing commands however they are case sensitive:

```
// usage examples
MathematicaLib::DoTheMath("222 * 2 ");
MathematicaLib::DoTheMath("46 / 2 ");
MathematicaLib::DoTheMath("81 + 32 ");
MathematicaLib::DoTheMath("56 - 89 ");
MathematicaLib::DoTheMath("4 / -2 ");
MathematicaLib::DoTheMath("2 * 2 - 8 * 4 * 20 + 1 ");
MathematicaLib::DoTheMath("5 * 5 + -8 + 2 * 22 ");
```

Implementation

The functionalities implemented within the system will be explained in a coherent order, in order to clarify how each functionality relates to another and how it was implemented.

The *DoTheMath* function initially checks the input parameter for any commands, each time a valid command is detected then a related method is called, for example when the command *Help* is passed into the function then the *Help* function executes, however if the passed parameter is not a valid command then function invokes another function that checks whether this passed parameter is an equation, if so then it returns the result of solving the equation.

Each time the function is called then the passed parameter is stored within a vector, this is done to provide a mechanism to save the commands history, the vector is global to the library therefore it will not be reset each time the *DoTheMath* function is invoked, rather it gets cleared when the *ClearRestart* Function is called as part of the reset functionality.

The function is limits the size of the parameter that can be passed to it, reducing its vulnerability to stack overflow attacks. Any undefined or illegal behaviour will be handled by the try catch block, which will throw an exception rather cause the programme to crash, currently exception are not handled.

```
double MathematicaLib::DoTheMath(char const* equation){

    commandHistory.insert(commandHistory.end(), equation); // saves each inputted command into a vector
    try{
        if (strcmp("Help", equation) == 0) { Help(); return 0.0; }
        else if (strcmp("Undo", equation) == 0) { return Undo(); }
        else if (strcmp("ClearRestart", equation) == 0){ ClearRestart(); return 0.0; }
        else if (strcmp("History", equation) == 0){ History(commandHistory); return 0.0; }
        else if (strcmp("Load", equation) == 0){ Load(); return 0.0; }
        else if (strlen(equation) > 100) { cout << "Input enetered is too big" << endl; return 0.0; } // used to limit input
        else
        {
            return IsThisAnEquation(equation);
        }
    }
    catch (exception e){ return 0; }
    return 0.0; // when an equations was not inputted
}// end of DoTheMath
```

Each one of the commands has a related function as shown below:

The *Load* function prints out a series of statements on the screen providing instructions on how to write an equation that is accepted by the function:

```
void  MathematicaLib::Load(){ // loads simple instructions
    cout << "Please type your equation in the formats shown below (Dont forget to add spaces)\n"
            "For Multipication x * y \n"
            "For Division x / y \n"
            "For Substraction x - y \n"
            "For Addision x + y \n"
            "For Mixed operations x * y / z - i + 40 \n"
            "For More Commands Type 'help' \n";
}
```

The *ClearRestart* function simply behaves as a reset mechanism by resetting all global variables within the system, such as command history:

```
void  MathematicaLib::ClearRestart(){ // clears progress and restarts the Application

    commandHistory.clear(); // used to display command history
    equationHistory.clear(); // use to allow for undo methods
}
```

The *Help* function provides information on available commands and the purpose of each:

```
void MathematicaLib::Help(){ // loads simple hel instructions
     cout << "Reset/Clear -- Delets All Progress And History\n"
             "Undo        -- Go Back One Step\n"
             "History     -- Display Command History\n"
             "Load        -- Loads instructions\n";
}// end of Help
```

As mentioned previously during the explanation of the *DoTheMath* function, that in order to implement the functionality of displaying command history then there must be method to store and keep track of each command entered, which explains why each time a parameter is passed into the function, then that parameter gets stored within a vector, thus fulfilling the functionality of storing values. The *History* function simply iterates through the vector and displays its contents.

```
void  MathematicaLib::History(vector<string> CommandHistory){ // displays vector contents

    for (unsigned int k = 0; k < CommandHistory.size(); k++){
        cout << CommandHistory[k] << endl;
    }
}// end of History
```

The *Undo* method behaves in a similar way, however it saves the order in which the equation was solved and displays it step by step:

```
double  MathematicaLib::Undo(){

    for (unsigned int k = 0 ; k < equationHistory.size(); k++){
        cout << equationHistory[k] << endl;
    }
    //equationHistory.clear();
    return equationHistory[0];

}
```

The function below is used to check whether the equation passed to the function is in a valid format or not, it executes a number of checks prior to solving the equation, if the parameter passed to the function fails these checks then a message is displayed and 0 is returned.

The invalid values could have been simply ignored or handled by an exception however it would be more informative this way, as it informs the use of what exactly went wrong and avoids the system from crashing.

```
double MathematicaLib::IsThisAnEquation(char const* equation){ // checks the equation for a valid format

    vector <string> SeperatedValues;
    //vector<string> TempGH; // no longer needed
    string tempo = "         ";
    int temp = 0; // an incrementer
    int SOG = 0; //start of group

    for (unsigned int j = 0; j < strlen(equation); j++) // this intialy filters out invalid inputs
    {
        if (isspace(equation[0])){ // check if the first charecter is a space (invalid input)
            cout << "The first charecter should be a number not a space" << endl;
            return 0.0;
        }
        else if (isdigit(equation[0]) == 0){ // if the first charecter is not a number
            cout << "The first charecter must be a number" << endl;
            return 0.0;
        }
        else if (isalpha(equation[j])){// if a letter is encounterd
            cout << "Letters are not valid inputs within equations" << endl;
            return 0.0;
        }
        else if (isspace(equation[j]) && isspace(equation[j + 1])){ // checks if more than one space is included
            cout << "No more than two consecutive spaces should be included" << endl;
            return 0.0;
        }
```

By the time the last conditional *else-if* statement is reached, it begins to separate, extract and group values together from the input parameter, it extracts numerical values and operator characters, storing each within a group, each extracted group is stored within a vector.

The vector holding separated values then gets passed into a function that solves the equation. The result of solving the equation is then returned.

```
        else if (isspace(equation[j])) {
            tempo.resize(j - SOG); // resizes tempo dynamicaly
            for (int n = SOG; n < j; n++){
                tempo[temp] = equation[n]; // populates tempo while reading the equation
                temp++; // an incrementer
                SOG = j + 1; //skip the space
            }

            SeperatedValues.push_back(tempo); // loads each group of values into a vector
            tempo.clear(); // clears its contents therfore the next group can be stored
            temp = 0; // makes TEMP GH to be overwritten


            //unnecessary!!!
            //TempGH.push_back(tempo); //stores the extracted group within a vector
            //SeperatedValues.push_back(TempGH[0]); // then stores it within another
            //TempGH.clear(); //
        }

    }

    return ExecutePEMDAS(SeperatedValues); // executes a method that solves the equation
}// end of IsThisAnEquation
```

The *ExecutePEMDAS* function reads from the vector that held the separated values and solves the equation in the PEMDAS (Parenthesis, Exponents, Multiplication, Division, Addison, and Subtraction), it does so by looping through the vector and looking for arithmetic operators, each time an operator is encountered then the predecessor and successor of the operator get converted into doubles and a calculation is perform, the type of calculation is dependent on the operator, the order which the search takes place for these operators is in correspondence to PEMDAS. Each time a calculation is performed then the values converted get erased from the vector and replaced with the result, until only one value is remaining.

```
double MathematicaLib::ExecutePEMDAS(vector<string> SeperatedValues){ // executes an equation in PEMDAS order

    equationHistory.clear(); // to clear the history of previous equations
    double ExtractedNum0 = 0.0;
    double ExtractedNum1 = 0.0;
    double result= 0.0;

    for (unsigned int p = 0; p < SeperatedValues.size(); p++){ // this goes by PEMDAS, the order below is important

        if (strcmp("*", SeperatedValues[p].c_str()) == 0){ // should I Multiply

            ExtractedNum0 = stod(SeperatedValues[p - 1]); // load vector contents into a double
            ExtractedNum1 = stod(SeperatedValues[p + 1]);
            SeperatedValues.erase(SeperatedValues.begin() + p - 1, SeperatedValues.begin() + p + 1); //removes this part as it has been solved (leaves one part )
            SeperatedValues.at(p - 1) = to_string(Multiply(ExtractedNum0, ExtractedNum1)); // returns the result to the vector
            result = Multiply(ExtractedNum0, ExtractedNum1);
            //cout << result << endl;
            p = 0; //look for another
            equationHistory.insert(equationHistory.end(), result);
        }
    }
    for (unsigned int p = 0; p < SeperatedValues.size(); p++){ // this goes by PEMDAS, the order below is important
        if (strcmp("/", SeperatedValues[p].c_str()) == 0){// should I devide

            ExtractedNum0 = stod(SeperatedValues[p - 1]); // load vector contents into a double
            ExtractedNum1 = stod(SeperatedValues[p + 1]);
            SeperatedValues.erase(SeperatedValues.begin() + p - 1, SeperatedValues.begin() + p + 1); //removes this part as it has been solved (leaves one part )
            SeperatedValues.at(p - 1) = to_string(Divide(ExtractedNum0, ExtractedNum1)); // returns the result to the vector
            result = Divide(ExtractedNum0, ExtractedNum1);
            //cout << result << endl;
            p = 0; //look for another
            equationHistory.insert(equationHistory.end(), result);
        }
    }
```

Please refer to the project files in order to view the full source code.

The set of functions below is used to carry out arithmetic operations, these functions are used when solving equations:

```
double MathematicaLib::Add(double a, double b){// Adds
    return a + b;
}

double MathematicaLib::Subtract(double a, double b){// subtracts
    return a - b;
}

double MathematicaLib::Multiply(double a, double b){// multiplies
    return a * b;
}

double MathematicaLib::Divide(double a, double b){ // divides
    return a / b;
}
```

The main file is shown below, it demonstrates how the function provided by the library, fulfils both purposes. The infinite loop is only included to allow the user to re-enter inputs rather than only invoke the function once and cause the programme to exit:

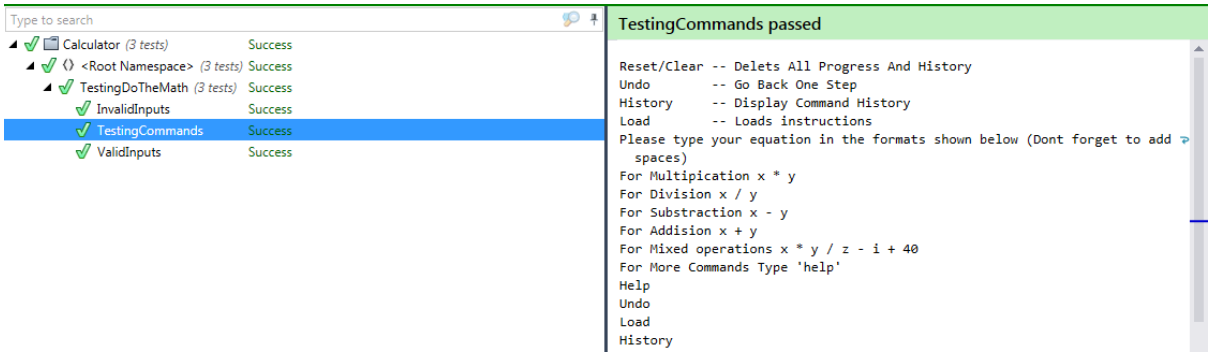
```
int main(int argc, char* argv[])
{
    string InputArgument;
    MathematicaLib::DoTheMath("Load"); // used to provide basic instructions

    while (true) // this is only here to use the library as an application
    {
        try{
            getline(cin, InputArgument); // dont use cin>> it eliminates spaces
            cout << MathematicaLib::DoTheMath(InputArgument.c_str()) << endl; // call the function from the library
        }
        catch (...){ // catches any other invalid input
            cout << "Invalid input, Please try again assuring you follow the guidlines" << endl;
        }
    }

    return 0;
}
```

Testing

The google test framework has been used to carry out a number of unit test, to confirm successful implementation of the system:

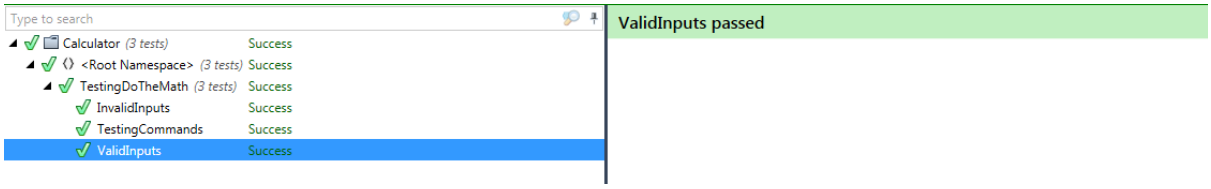
The first test is to test passing commands to the library, such as *History* and *Help*, the test was successful and the outputs are as expected, 0 should be returned each time a text command is executed and a few statements should be printed on the screen relevant to each command:

```
TEST(TestingDoTheMath, TestingCommands)
{
    EXPECT_EQ(0, MathematicaLib::DoTheMath("ClearRestart"));
    EXPECT_EQ(0, MathematicaLib::DoTheMath("Help"));
    EXPECT_EQ(0, MathematicaLib::DoTheMath("Undo"));
    EXPECT_EQ(0, MathematicaLib::DoTheMath("Load"));
    EXPECT_EQ(0, MathematicaLib::DoTheMath("History"));
}
```



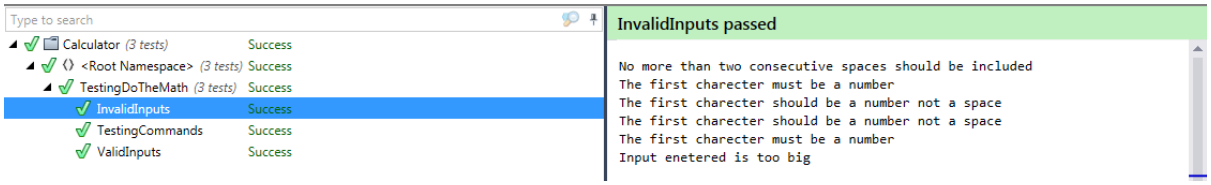
The second test consisted of passing equations to the function and checking the value returned by the function against the value derived from solving the equation, the test was also successful:

```
TEST(TestingDoTheMath, ValidInputs)
{
    EXPECT_EQ(444, MathematicaLib::DoTheMath("222 * 2 "));
    EXPECT_EQ(23, MathematicaLib::DoTheMath("46 / 2 "));
    EXPECT_EQ(113, MathematicaLib::DoTheMath("81 + 32 "));
    EXPECT_EQ(-33, MathematicaLib::DoTheMath("56 - 89 "));
    EXPECT_EQ(-2, MathematicaLib::DoTheMath("4 / -2 "));
    EXPECT_EQ(-637, MathematicaLib::DoTheMath("2 * 2 - 8 * 4 * 20 + 1 "));
    EXPECT_EQ(61, MathematicaLib::DoTheMath("5 * 5 + -8 + 2 * 22 "));
}
```



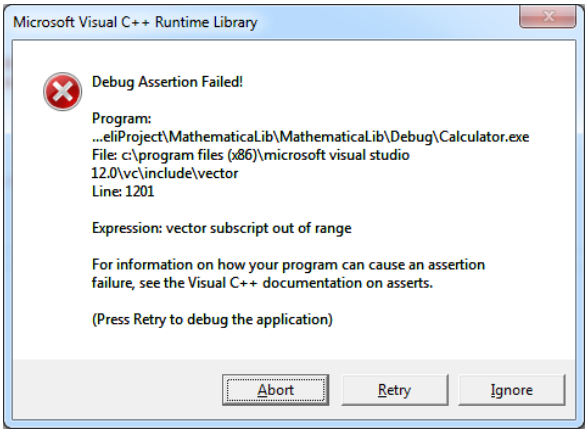
The following test was done to confirm whether the system behaves to invalid inputs as expected or not, the system should behave to invalid inputs by returning a 0 and printing out relevant statements indicating why the input was invalid, the test was also successful:

```
TEST(TestingDoTheMath, InvalidInputs)
{
    EXPECT_EQ(0, MathematicaLib::DoTheMath("2 *  "));
    EXPECT_EQ(0, MathematicaLib::DoTheMath("2 * 0 "));
    EXPECT_EQ(0, MathematicaLib::DoTheMath("f * 2 "));
    EXPECT_EQ(0, MathematicaLib::DoTheMath(" / 2 "));
    EXPECT_EQ(0, MathematicaLib::DoTheMath(" / "));
    EXPECT_EQ(0, MathematicaLib::DoTheMath("0 / 2 "));
    EXPECT_EQ(0, MathematicaLib::DoTheMath("dsfs"));
    EXPECT_EQ(0, MathematicaLib::DoTheMath("0000000000000000000000000000000000000000000000000000"
        "0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000"));
}
```



The final test, was to check whether an exception is thrown when the function has been given an invalid value that has no defined behaviour. The function fails, although try/catch blocks where included throughout the library. The system crashes because one of the vectors became out of range, but such behaviour should have been handled by an exception rather than cause the system to crash:

```
TEST(TestingDoTheMath, TestingExceptions)
{
    EXPECT_DEATH(MathematicaLib::DoTheMath("2 * "), "Exception was not handled or thrown");
    EXPECT_ANY_THROW(MathematicaLib::DoTheMath("2 * "));
}
```



This means that the exception class does not throw exception when vectors go out of range, a solution would be to create a user-defined exception and throw it when vectors go out of range, this can be implemented as part of future work.

Requirements review (verification)

Requirement	Status	Comment
A display history method	Implemented	The current method does not limit the amount of commands stored within the command history
A method of clearing/ resetting the system	Implemented	
An Undo method	Implemented	
A console application that uses the library	Implemented	Does not always throw exceptions.
Carrying out, multiplication, division, addition and subtraction operations and return total	Implemented	
Project documentation	Produced	Further details could have been provided in relation to the design
The use of unit testing	Carried out	My limited knowledge of unit testing may have prevented me from making efficient use of it, I'm sure it offers more than what I have made use of.
Testing documentation	Produced	
Use of the OOP approach	Used	The MathemaicaLib.cpp class contains lots of functionalities that could have been separated into various class
Cross platform compatibility		The current library is not limited to the Windows 7 platform

Compile/build instructions

Option1;

If you are using the visual studio GUI then just open \VisualStudioSln*MathematicaLib**MathematicaLib**MathematicaLib.sln* and build the solution, assuring that the *MTd* run time library is used.

Option2;

If you are using the visual studio command line prompt, then navigate to *\BuildFiles*

```
cl /EHsc MathematicaLib.cpp MathematicaLib.h
LIB.EXE /OUT:MYLIB.LIB MathematicaLib.OBJ
cl /EHsc main.cpp
```

I was unable to test this method, most likely a linking step is missing that should link *main.cpp* with *MathematicaLib.Lib*

Future work

- The library can be expanded to perform further applications such as handling parenthesis and exponents.
- The library currently deals with numerical values as doubles however this is making inefficient use of memory, not all values deal with by the library are doubles, they could be integers or floats, meaning that the constant use of doubles is a waste of memory.
- The storages methods used to store command history and equation history should have limited size to reduce the chance of it being a vulnerability to stack overflow attacks.
- The calculator currently does not support chained operations rather it solves one equation at a time.
- The undo method implemented is not ideal, as all it does is show the order in which the equation is solved rather than undo the most recent step.
- The use of pointers could have been used to utilise memory and performance, as the C++ language copies variable each time they are passed into a function, while the use of pointers eliminates such behaviour as it allows for referencing the variables location within memory rather than copying it.
- Although that bounds checking have been respectively used to limit the size of variables and data structures, also to reduce vulnerability to stack overflow attacks, it has not been used enough, the *CommandHistory* and *EquationHistory* vectors are not protected.

Learning outcomes

As a result of undertaking this exercise, a number of expertise have been gained:

- Library development.
- Use of vectors and its advantageous over arrays.
- The use and familiarisation with unit testing, and various frameworks.
- Shallow knowledge of dynamic and static library linking.
- Use of various functions provided by libraries.
- The constant values of *chars*.