CPP -Exercise

Abdullah Al Naffakh

Contents

[Usage details 2](#_Toc450037937)

[Implementation 3](#_Toc450037938)

[Testing 7](#_Toc450037939)

[Requirements review (verification) 9](#_Toc450037940)

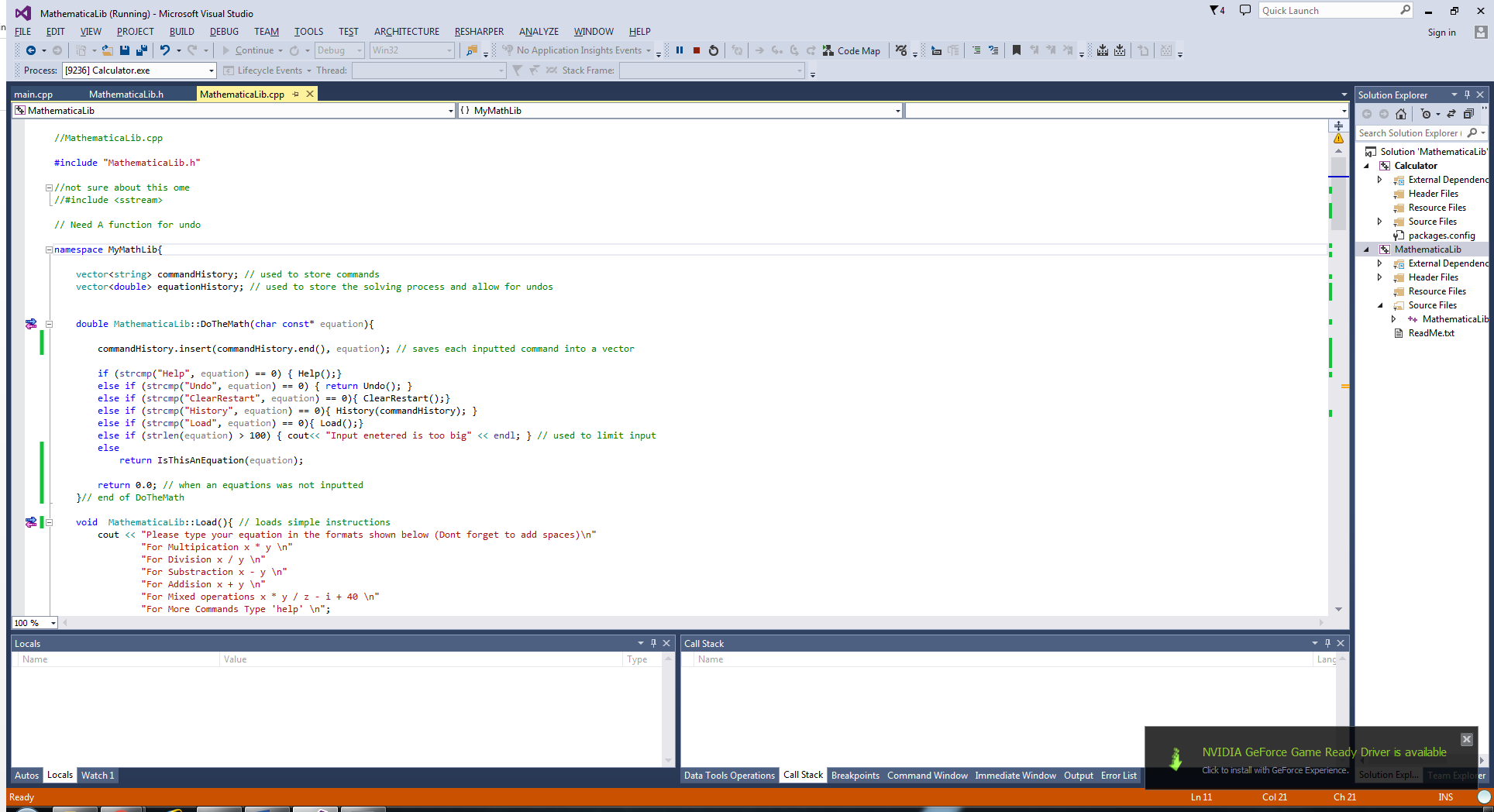
[Compile/build instructions 9](#_Toc450037941)

[Future work 10](#_Toc450037942)

[Learning outcomes 10](#_Toc450037943)

# Usage details

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



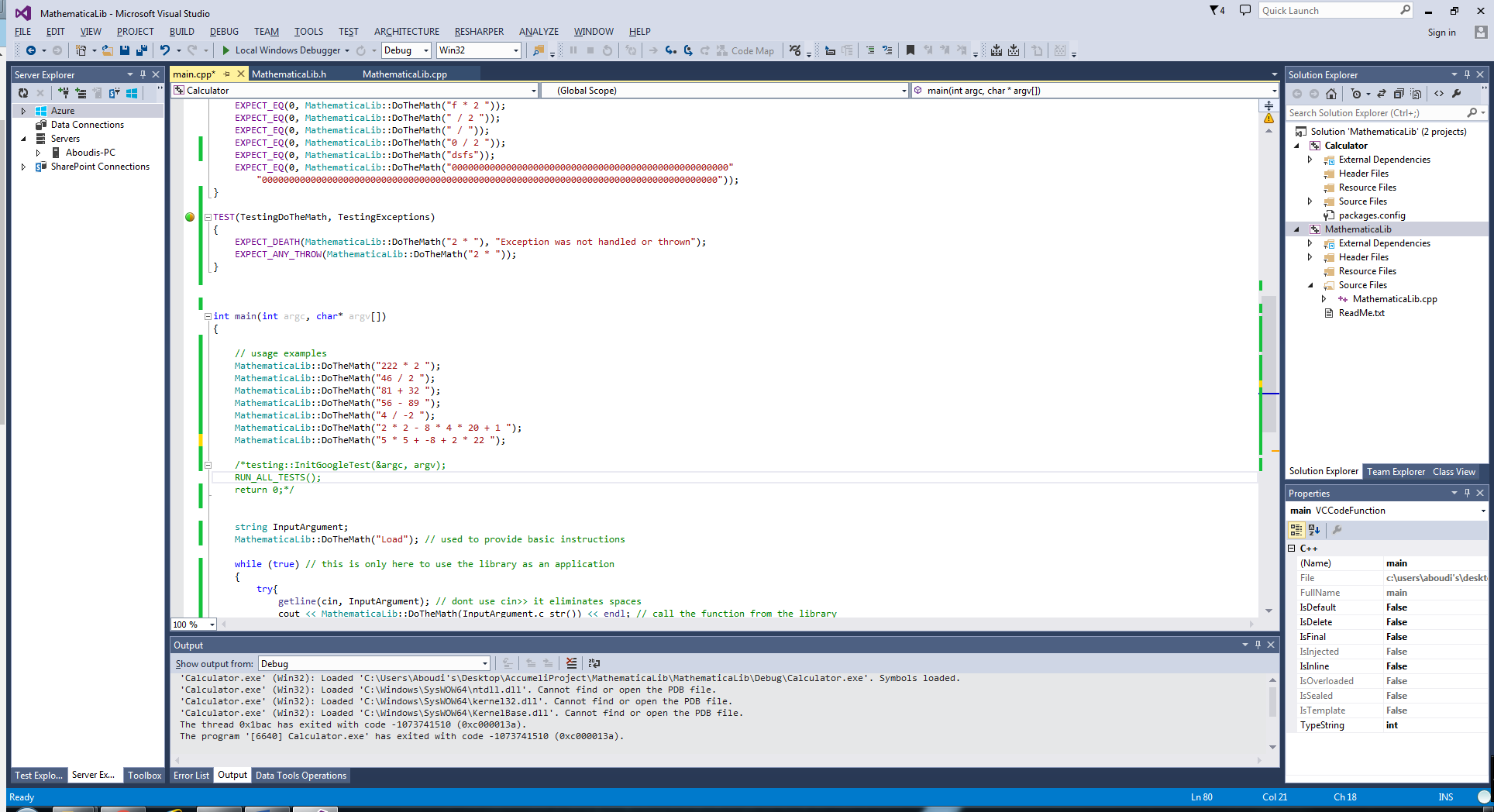
The library fulfils two purposes, both fulfilled by a single function call:

* 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:



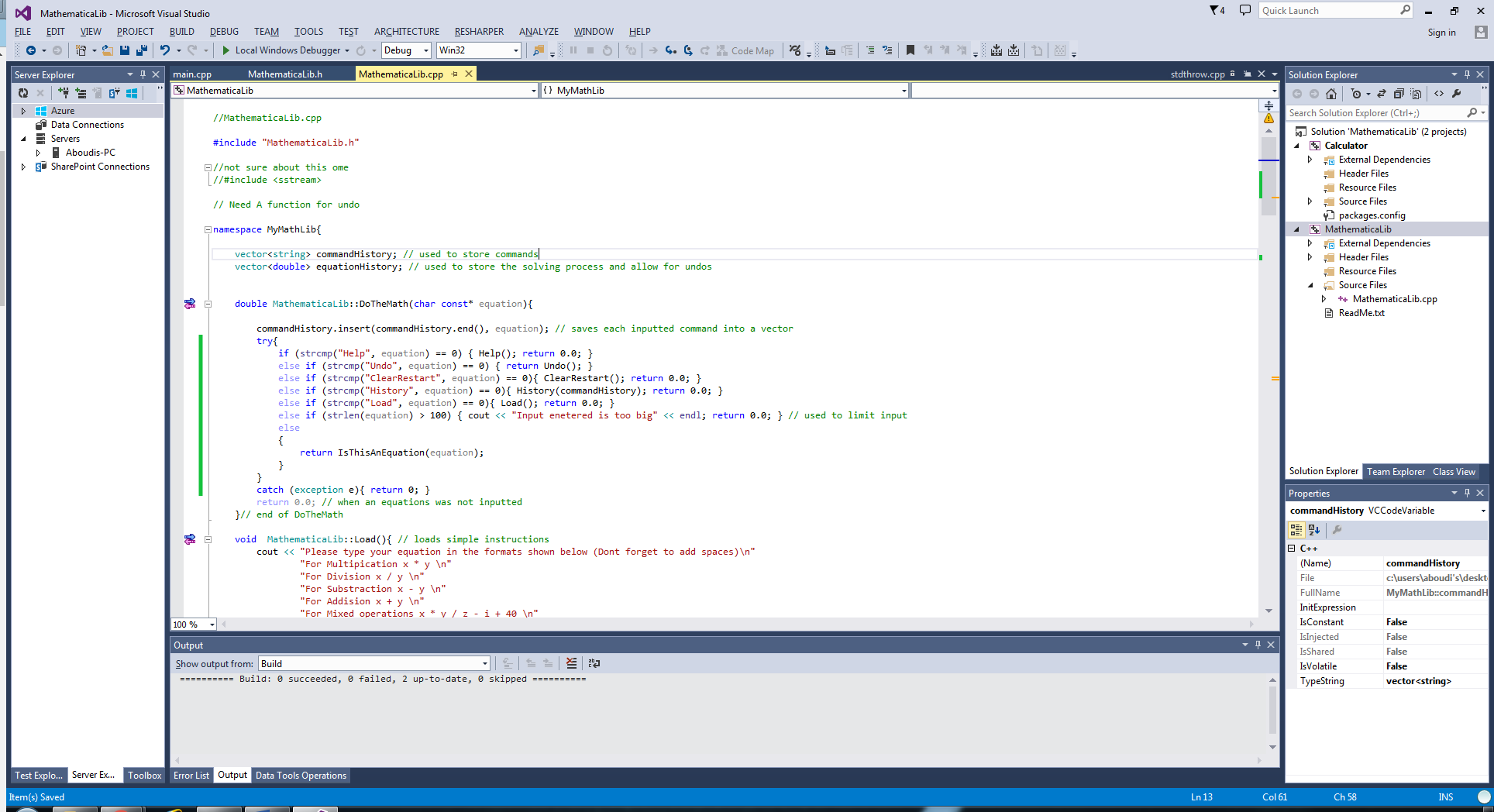
# Implementation

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.

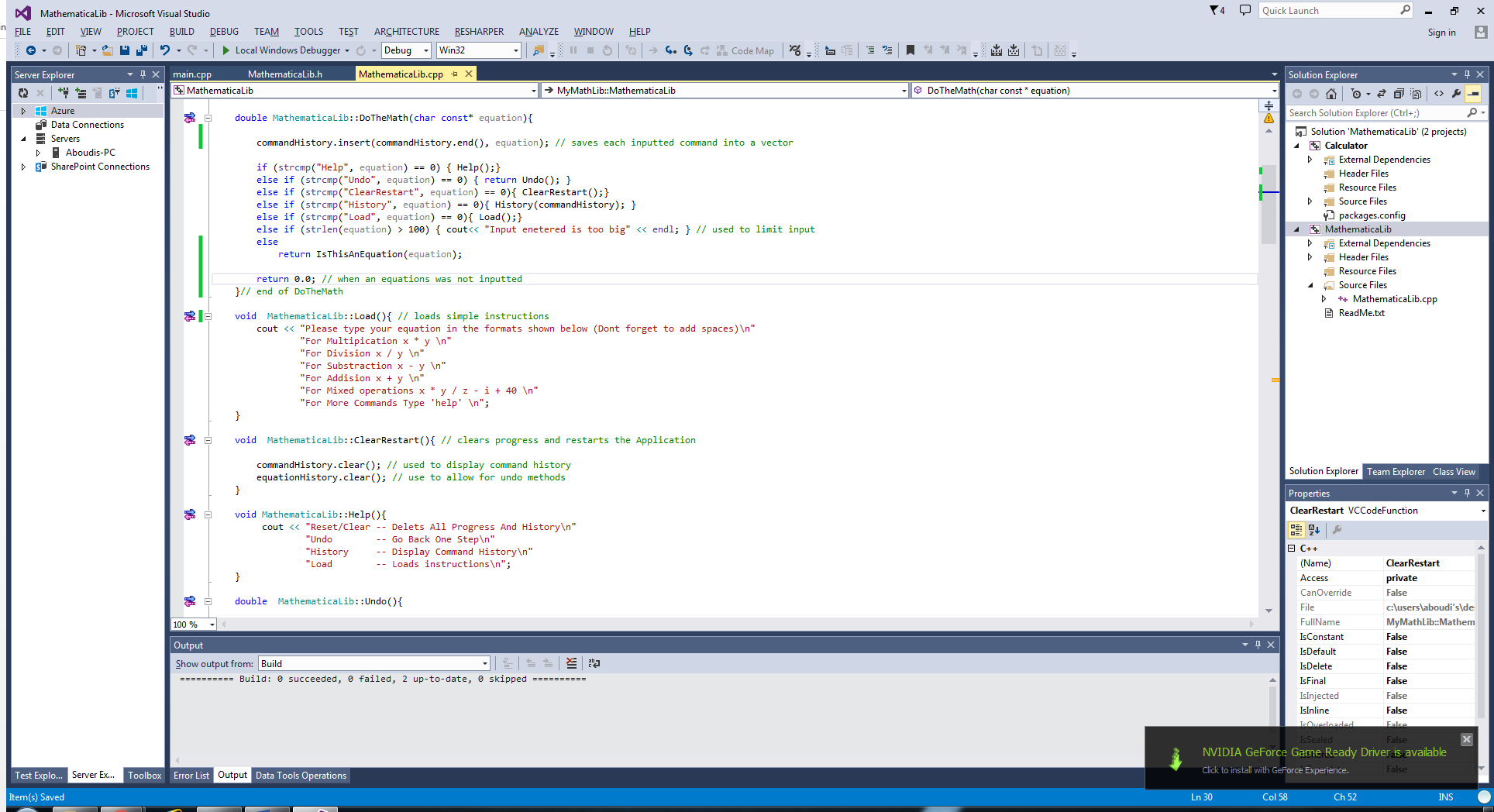
Each time the function is called then the passed parameter is stored within a vector, this is done to provide a mechanism to save the commands history, the vector is global to the library therefore it will not be reset each time the *DoTheMath* function is invoked, rather it gets cleared when the *ClearRestart*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

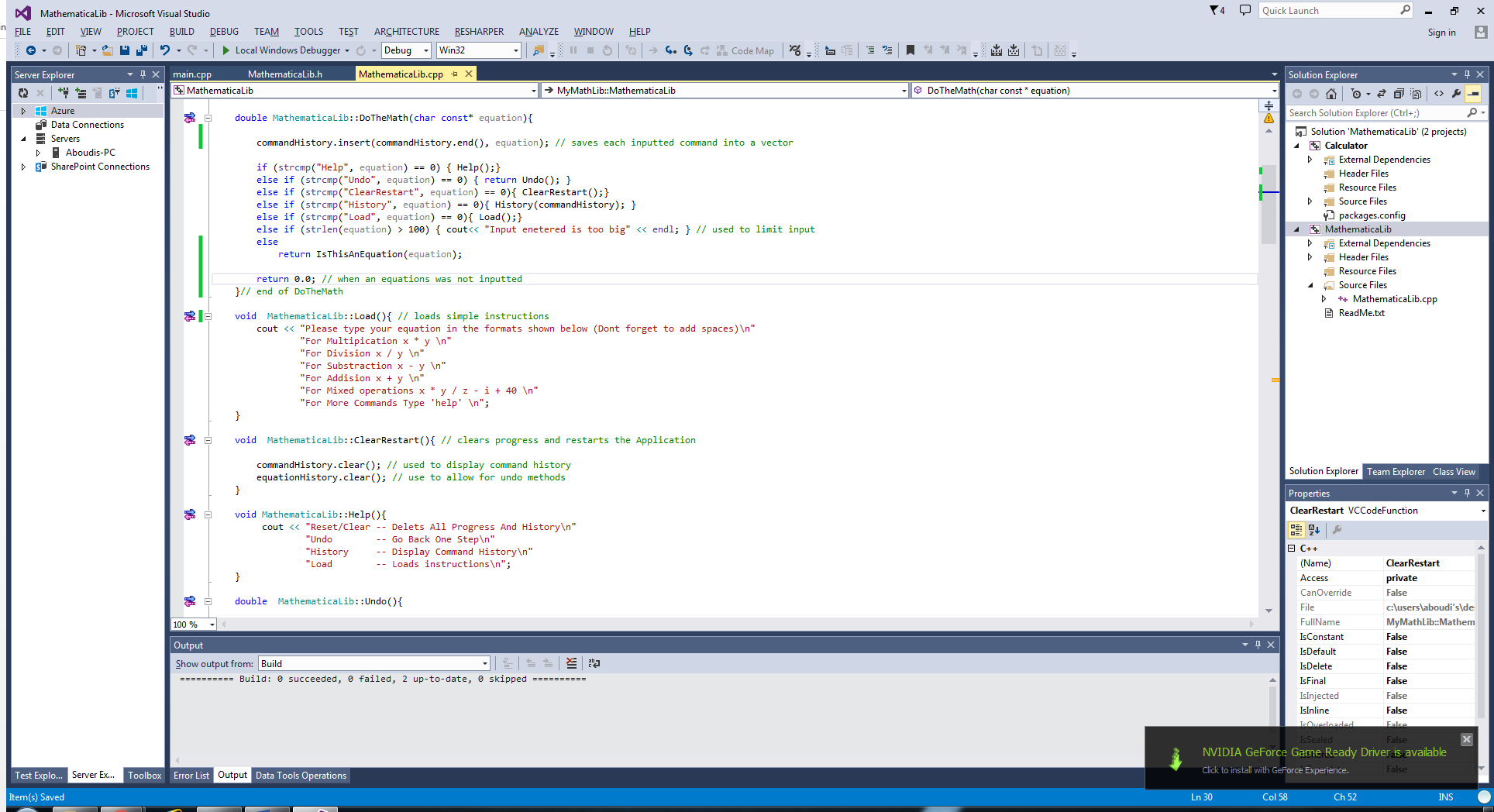


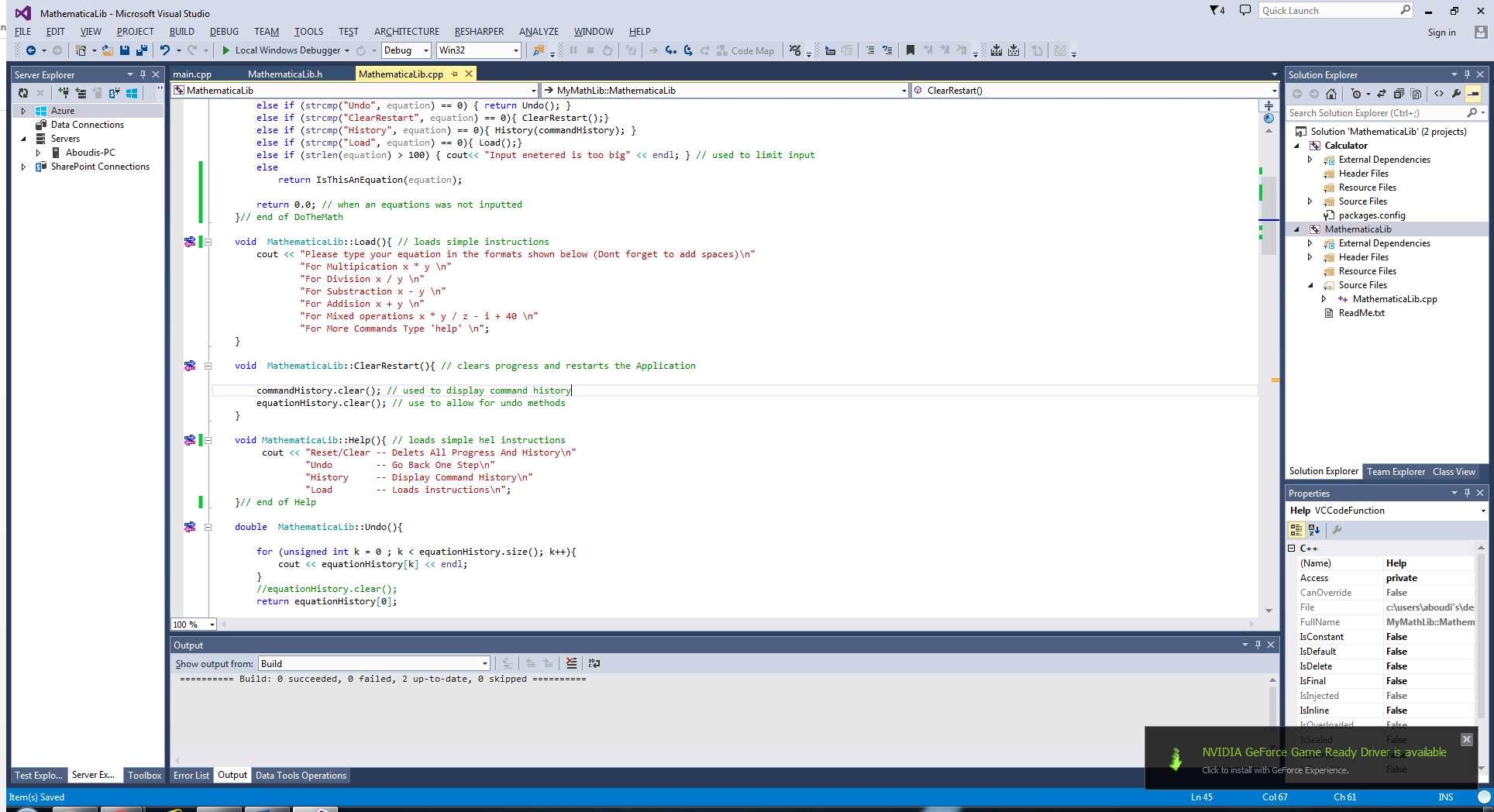
Each one of the commands has a related function as shown below:

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:

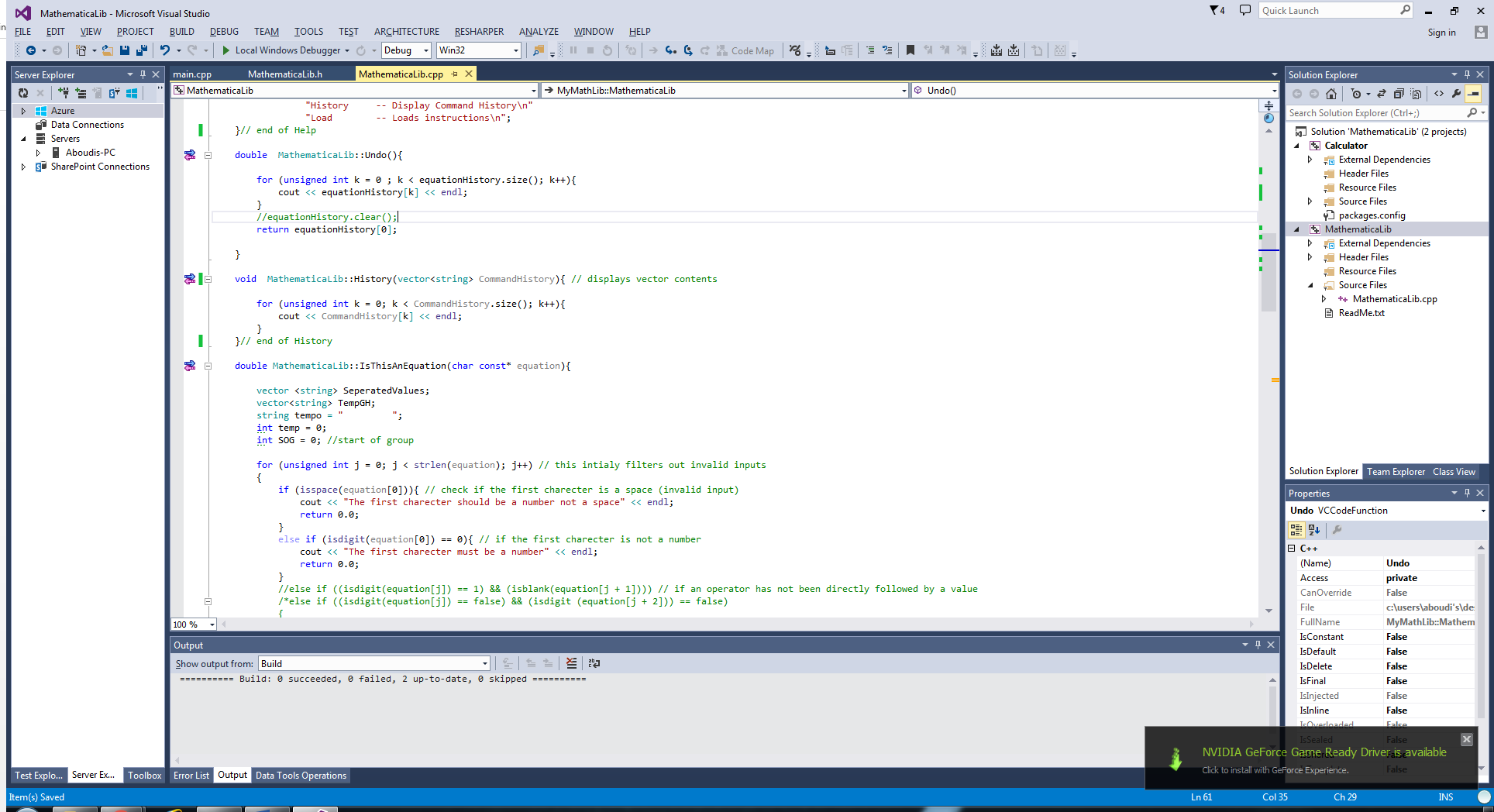


The *ClearRestart* function simply behaves as a reset mechanism by resetting all global variables within the system, such as command history:

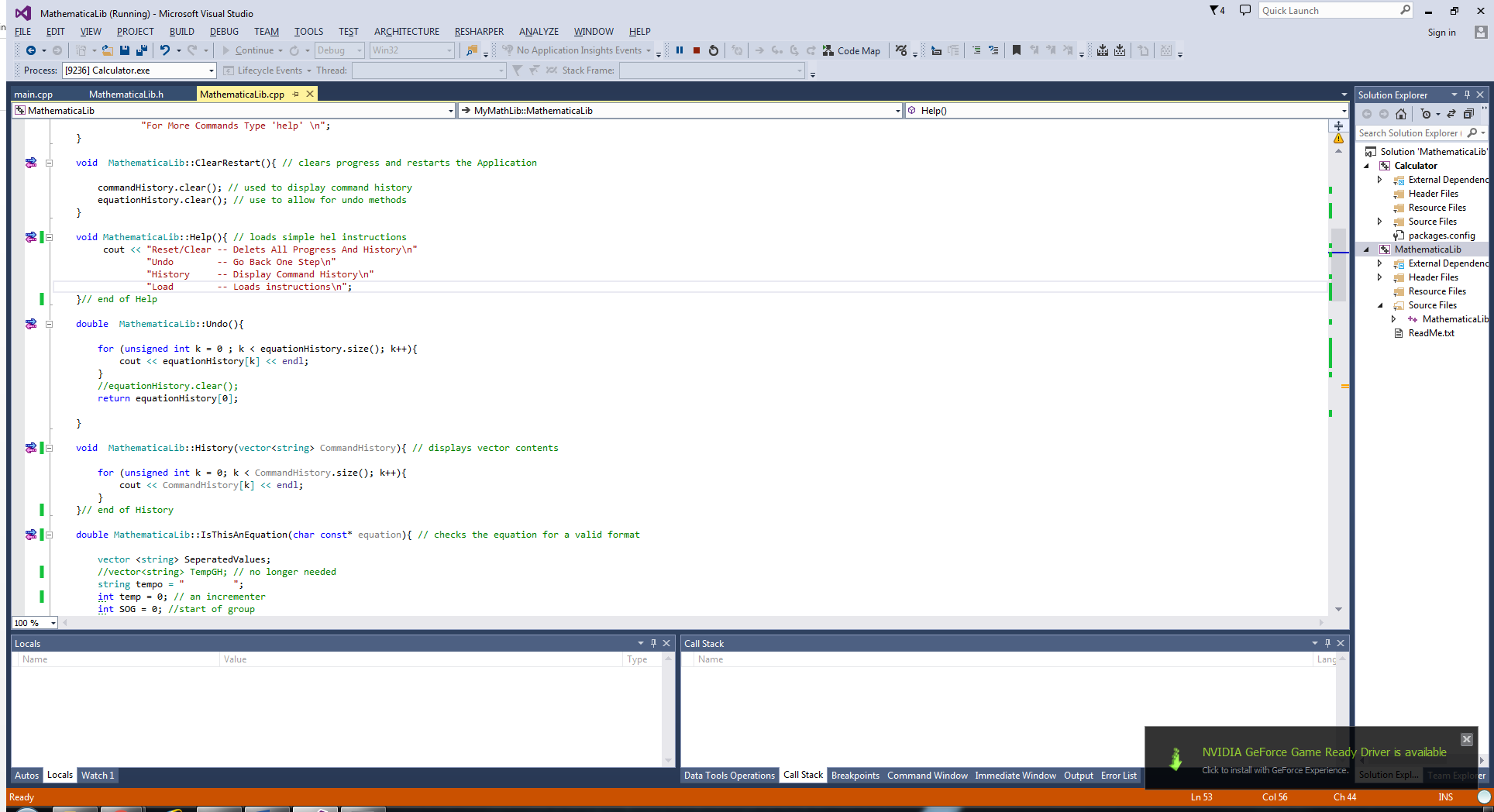


The *Help* function provides information on available commands and the purpose of each:

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.

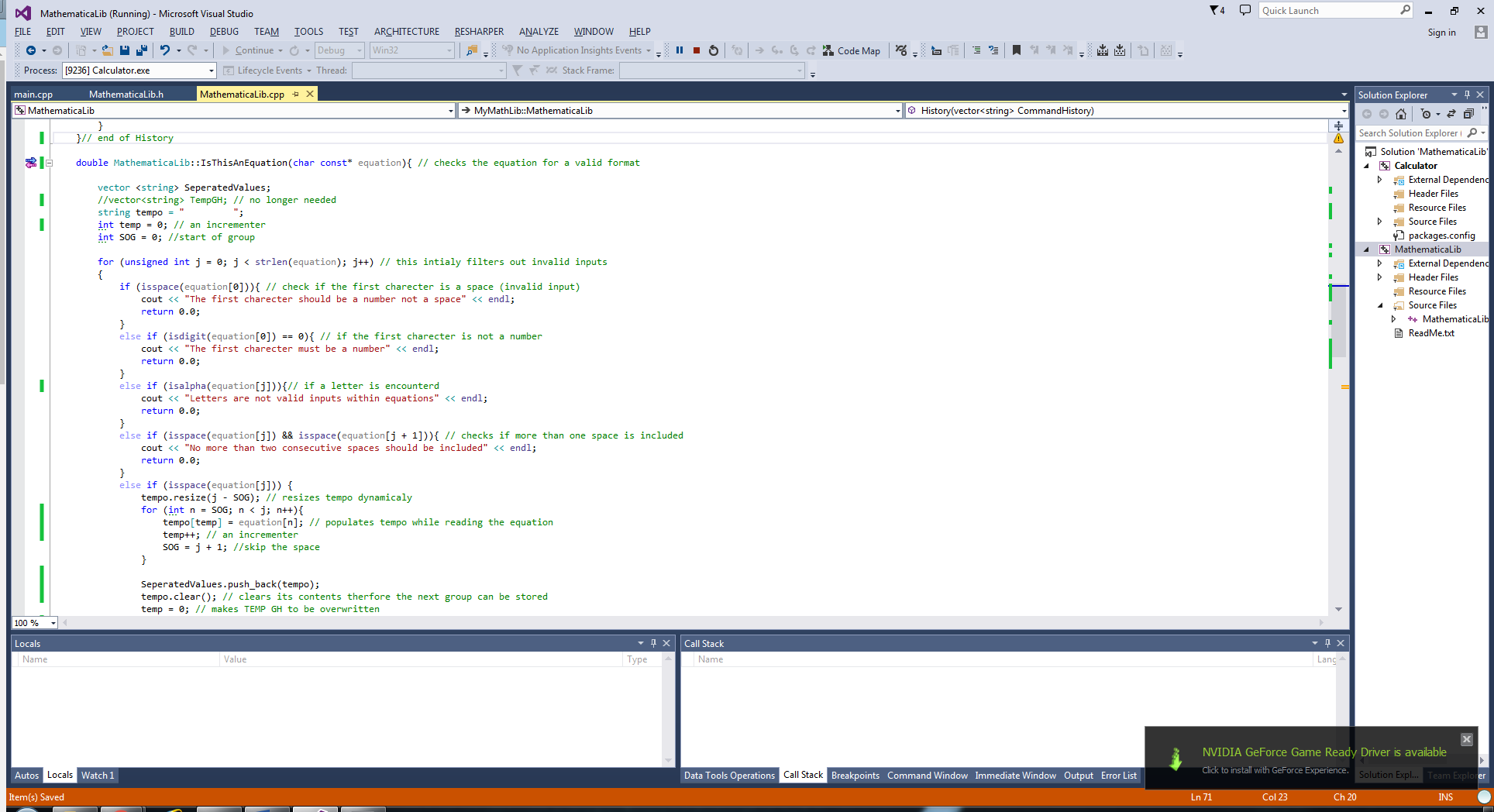


The *Undo* method behaves in a similar way, however it saves the order in which the equation was solved and displays it step by step:



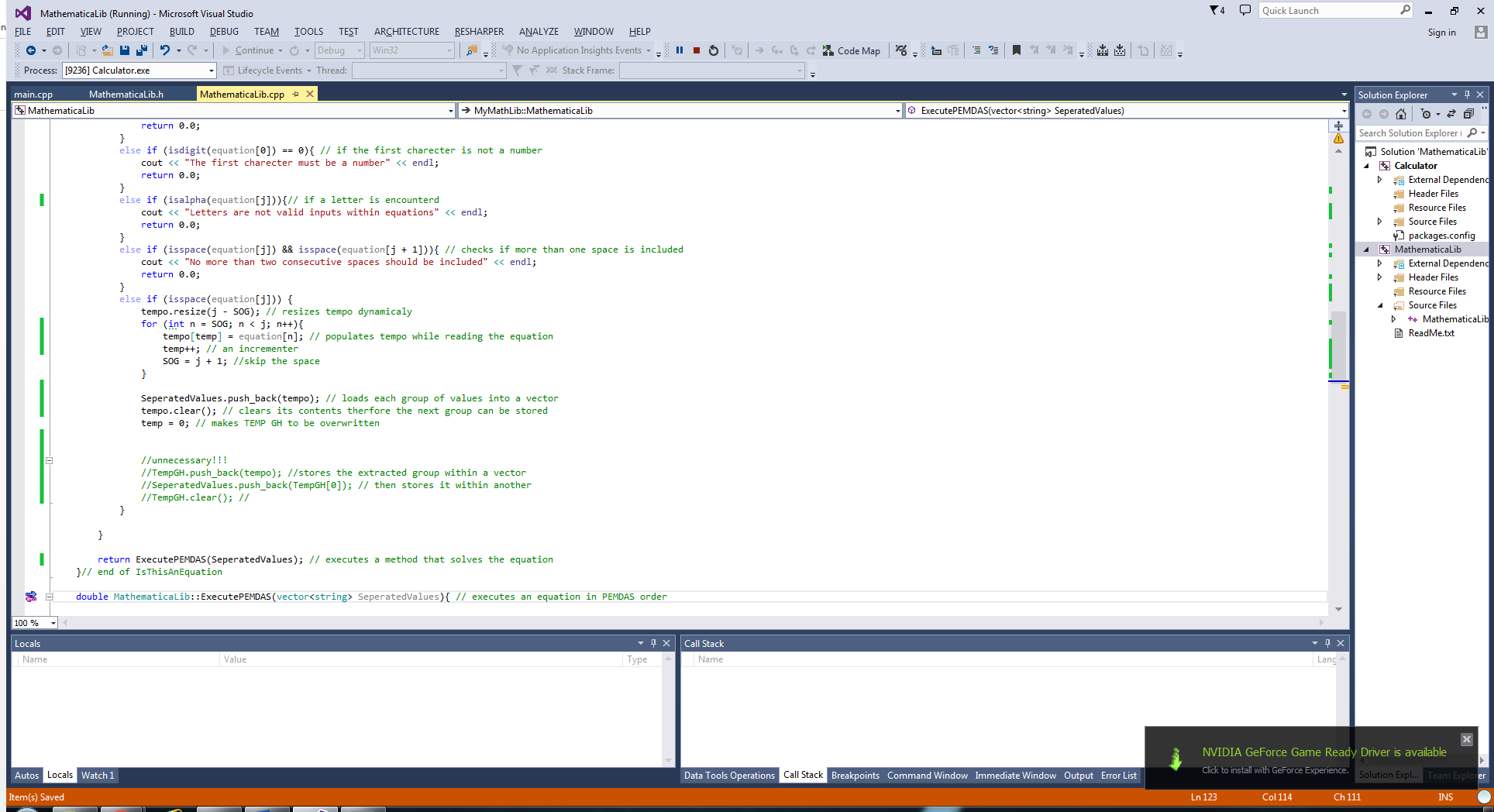
The function below is used to check whether the equation passed to the function is in a valid format or not, it executes a number of checks prior to solving the equation, if the parameter passed to the function fails these checks then a message is displayed and 0 is returned.

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.

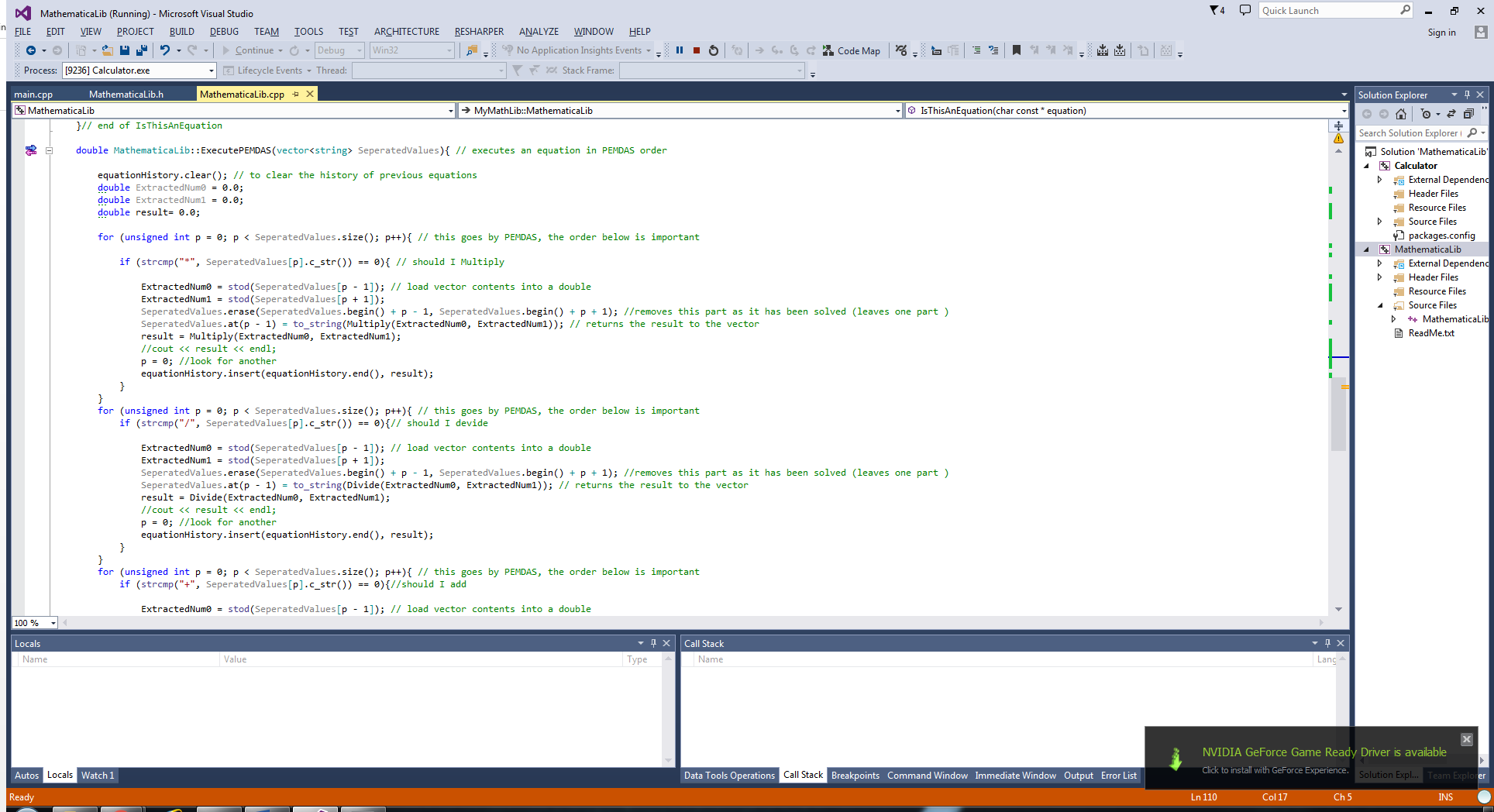


By the time the last conditional *else-if* statement is reached, it begins to separate, extract and group values together from the input parameter, it extracts numerical values and operator characters, storing each within a group, each extracted group is stored within a vector.

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

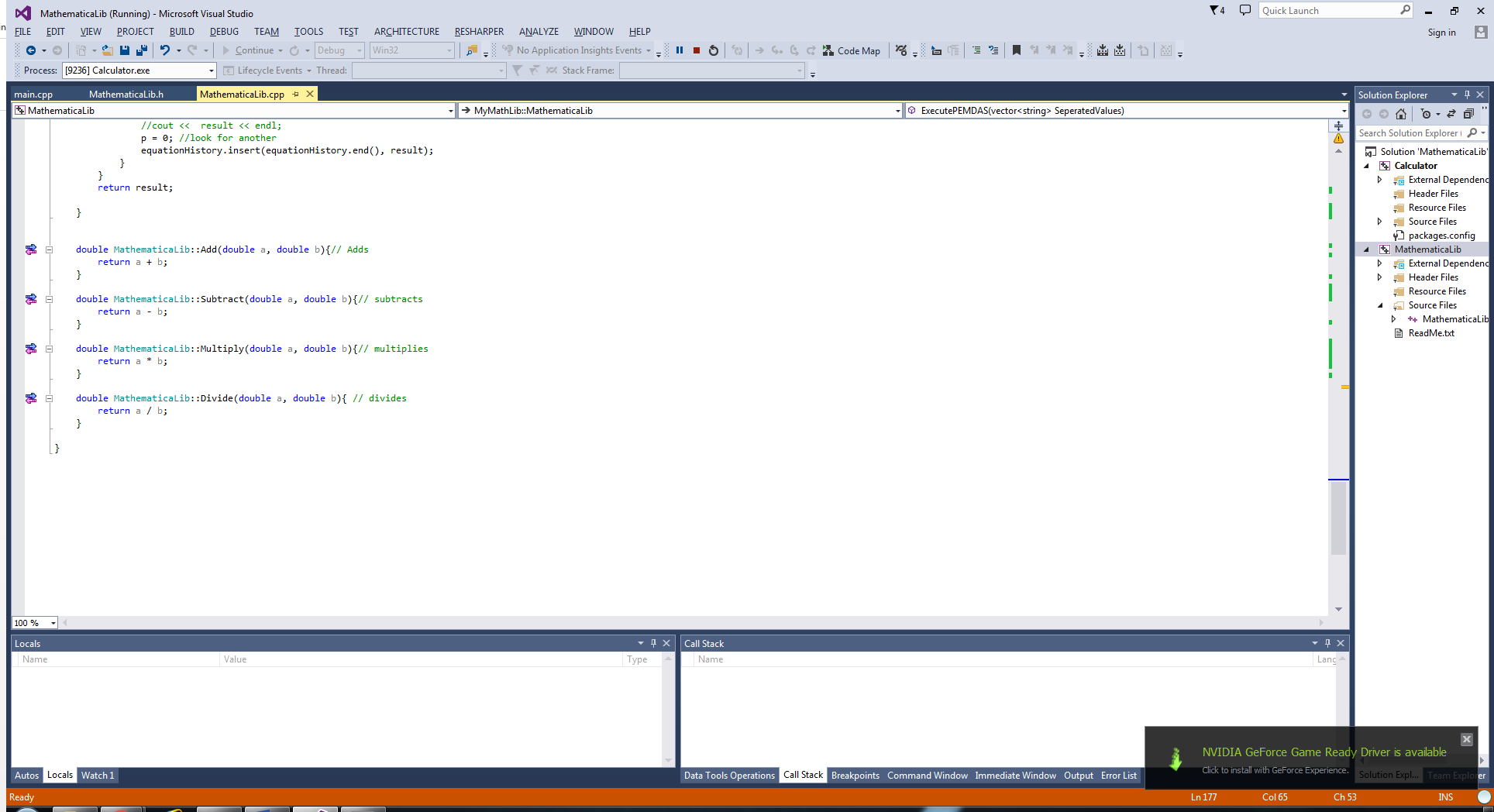


The *ExecutePEMDAS* function reads from the vector that held the separated values and solves the equation in the PEMDAS (Parenthesis, Exponents, Multiplication, Division, Addison, and Subtraction), it does so by looping through the vector and looking for arithmetic operators, each time an operator is 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.

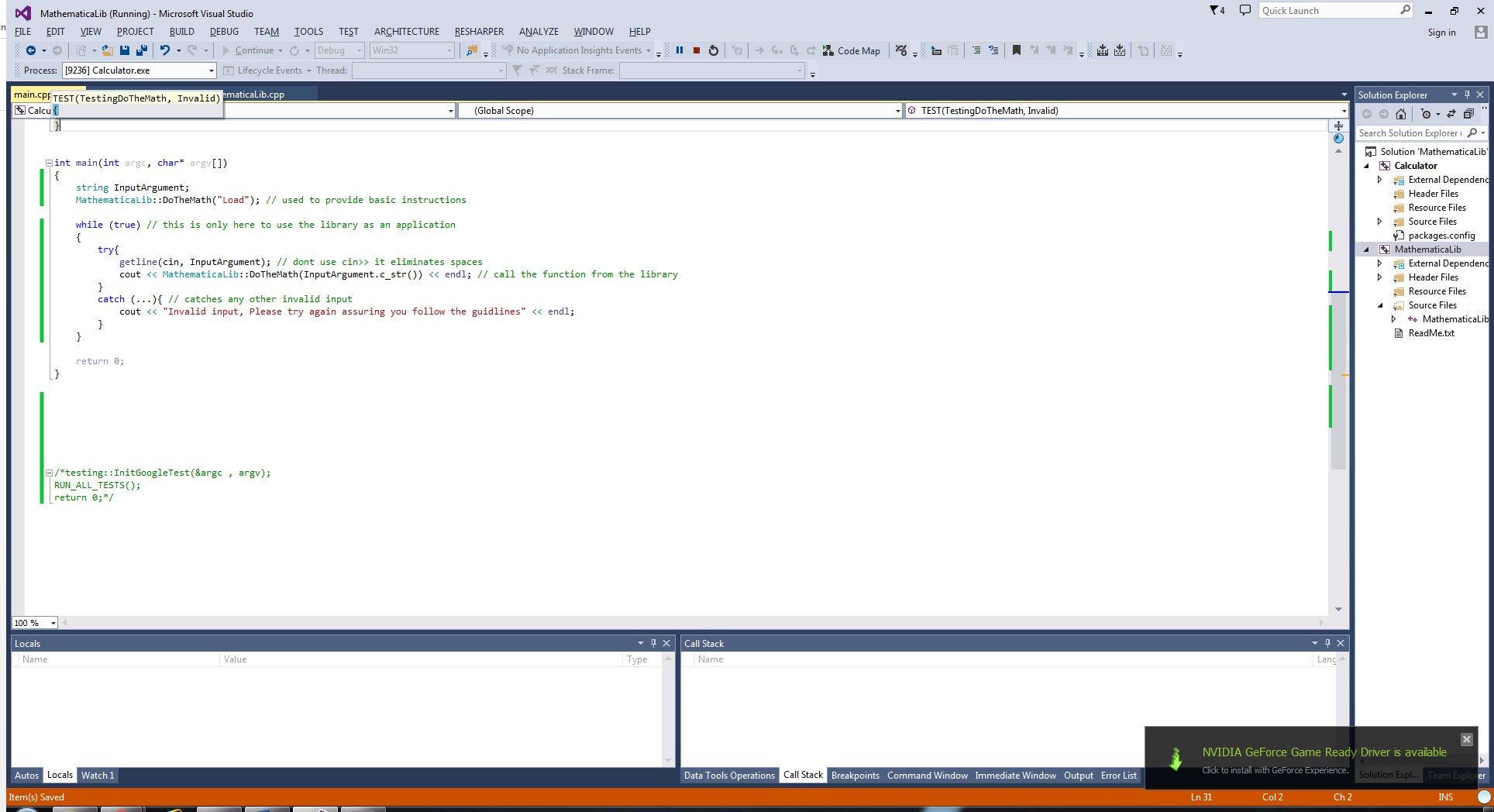


Please refer to the project files in order to view the full source code.

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



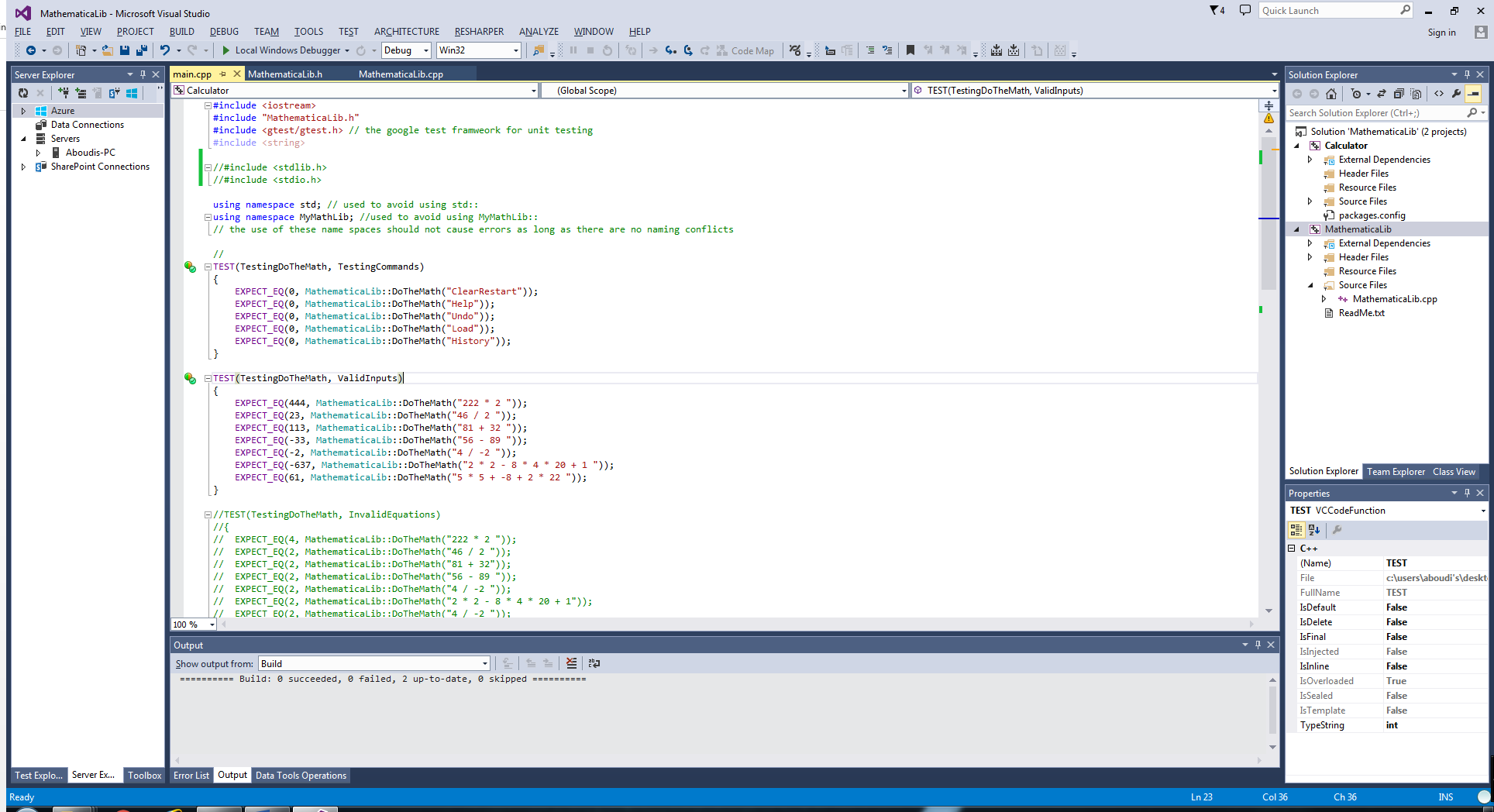
The main file is shown below, it demonstrates how the function provided by the library, fulfils both purposes. The infinite loop is only included to allow the user to re-enter inputs rather than only invoke the function once and cause the programme to exit:

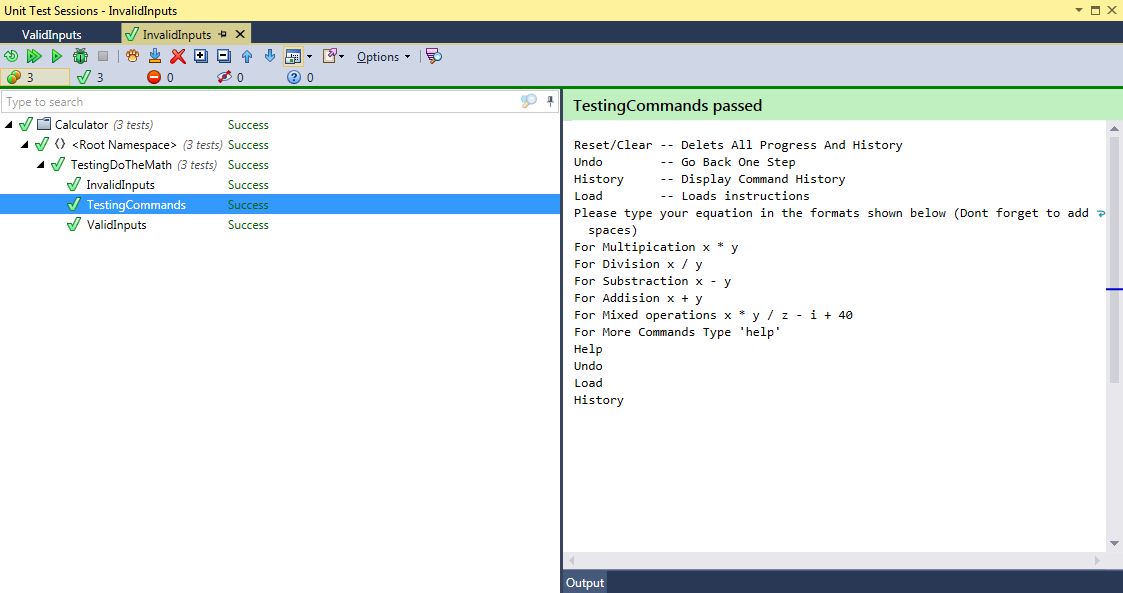


# Testing

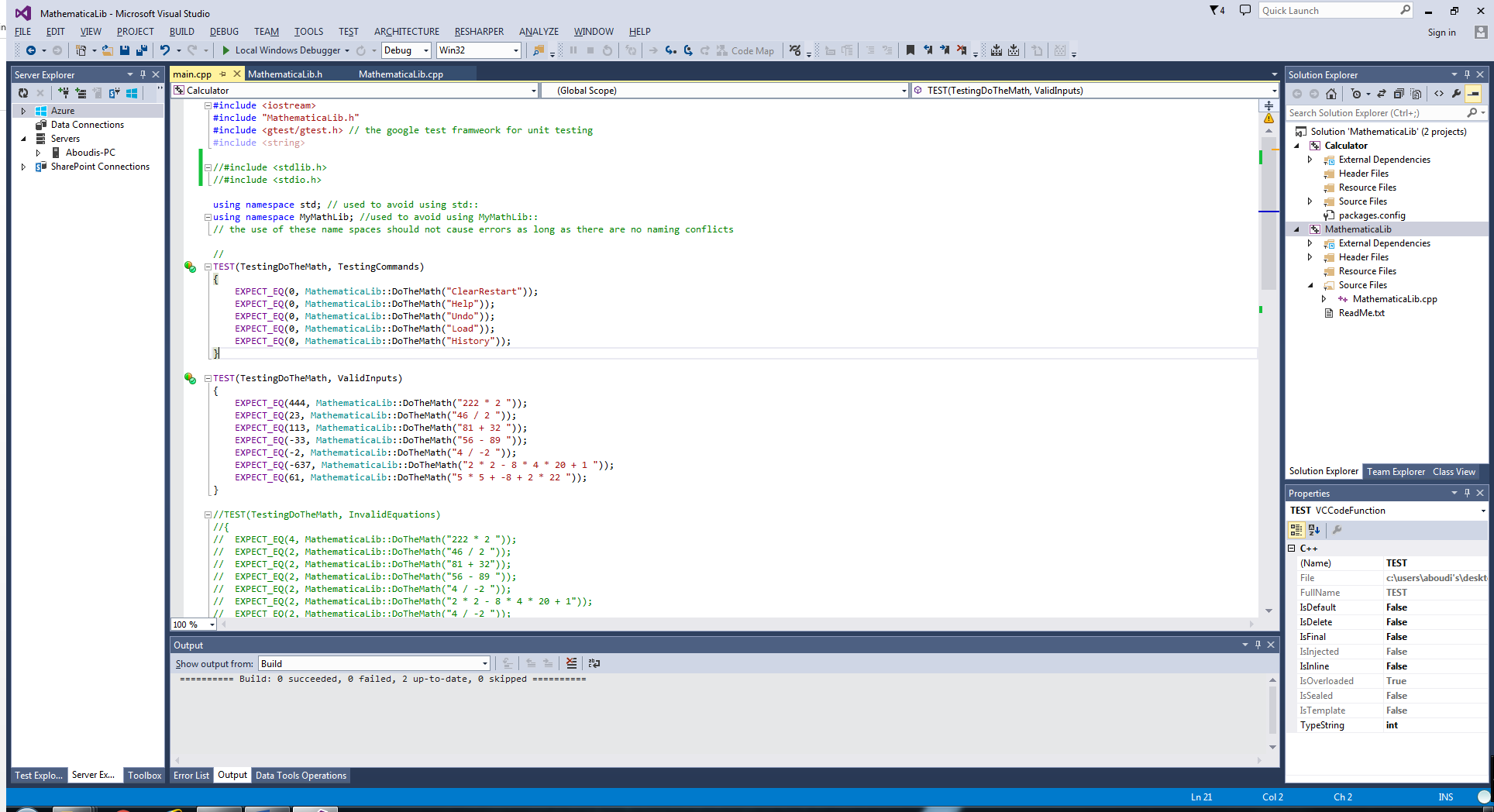
The google test framework has been used to carry out a number of unit test, to confirm successful implementation of the system:

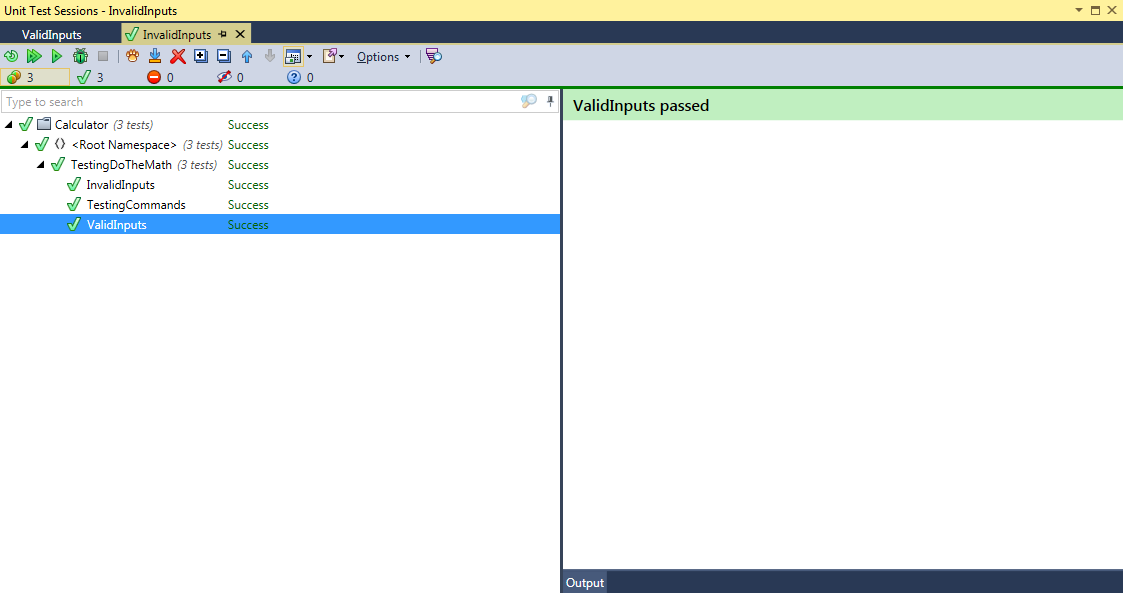
The first test is to test passing commands to the library, such as *History* and *Help*, the test was successful and the outputs are as expected, 0 should be returned each time a text command is executed and a few statements should be printed on the screen relevant to each command:



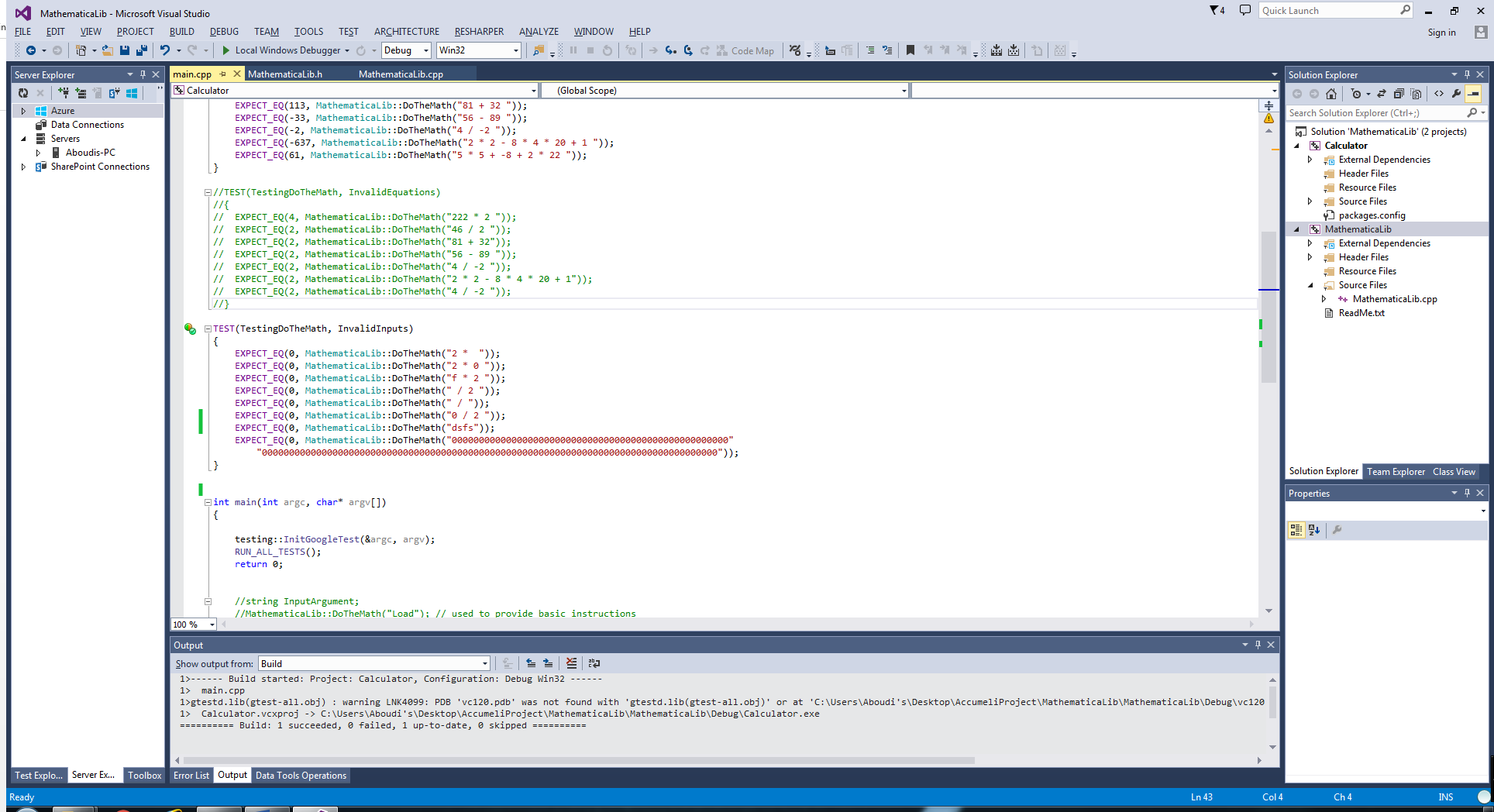


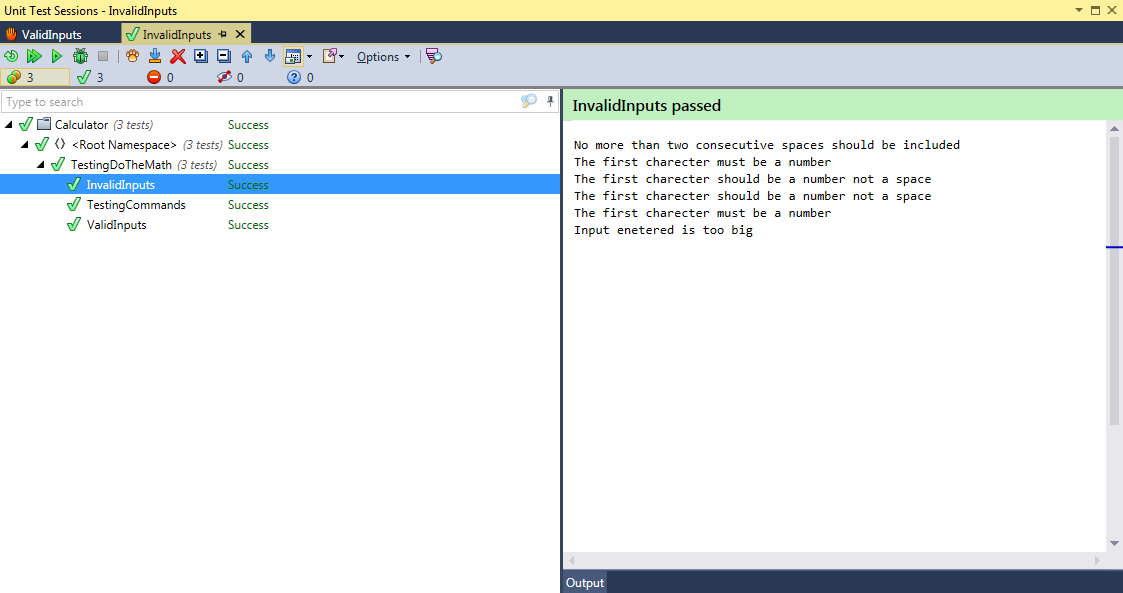
The second test consisted of passing equations to the function and checking the value returned by the function against the value derived from solving the equation, the test was also successful:



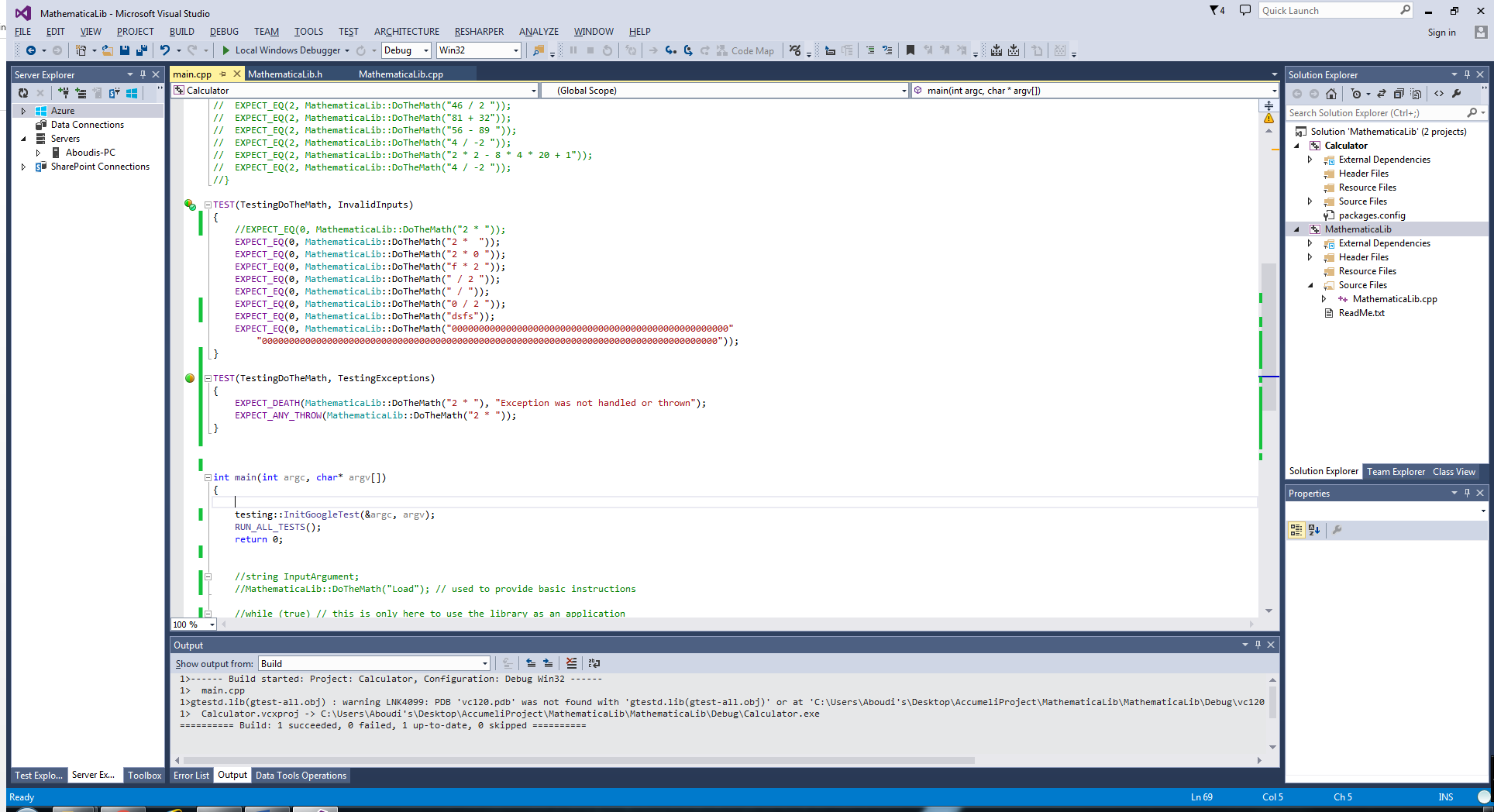


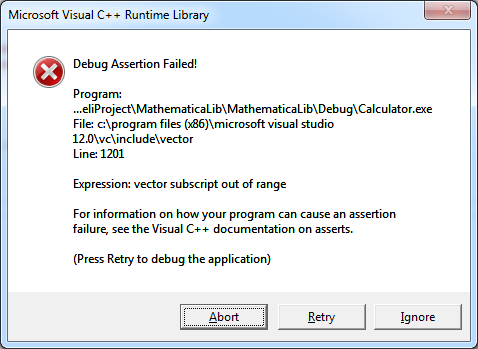
The following test was done to confirm whether the system behaves to invalid inputs as expected or not, the system should behave to invalid inputs by returning a 0 and printing out relevant statements indicating why the input was invalid, the test was also successful:





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:





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*.