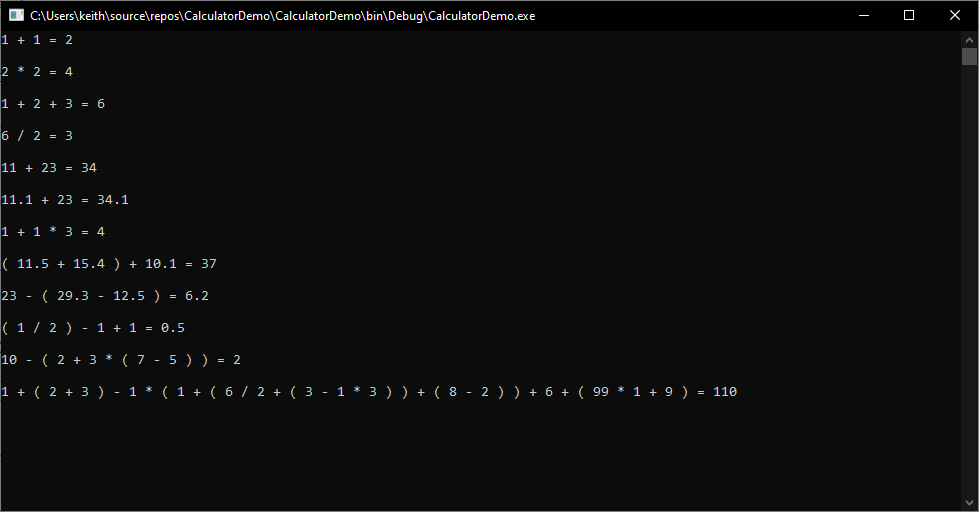
**Total Time Used:** 10 Hours

**Calculator Demo**



**Type:** Console Application

**.NET Framework:** 4.7.2 (Visual Studio 2019)

The proposed solution is to take the string of sum and split the string into List via .Split(‘ ‘).ToList() with space as separator.

Then we’ll check if the List actually contains any bracket(s), if yes then we’ll process the values within brackets first.

**Function** ProcessBrackets

For handling the brackets, first we’ll get all the indexes of Left and Right brackets, putting them into separate Lists. Then we’ll loop through the Left Brackets List from the last index till the first, and get the correct Right Bracket belong to each of the Left Bracket. If we detect that the Bracket itself is Nested, then we’ll store in a List containing the indexes of Left and Right Bracket (of the Nested), and continue the iteration till we locate the Main Bracket that contains these Nested Bracket(s).

**Function** HandleBracketValues

Once we’ve located the Main Bracket, we’ll start to process each Bracket, starting from the most inner Nested bracket, and all the way till it reaches the Main bracket. Each Nested bracket will store its total Sum (value of the calculation) before end of their iteration, and when it reaches the Main bracket (or Nested bracket that contains another Nested bracket), we’ll replace the Nested bracket e.g. “( 1 + 1 )“ with the actual sum of the Nested bracket instead e.g. “2”.

The end result of processing the Brackets will be as follows: -

|  |  |
| --- | --- |
| Before | After |
| ( 1 + 1 + ( 2 + 3 + ( 4 + 6 ) ) + 3 ) | 1 + 1 + 15 + 3 |

Using the “After” result, we’ll process the calculation and return the total Sum of every Main bracket (**just the Main Bracket Sum** that already includes Sums from Nested(s)).

Once we’ve received the Sum of every Main Bracket, we’ll replace the entire Main Bracket with the Sum value instead.

|  |  |
| --- | --- |
| Before | After |
| 5 + ( 1 + 1 + ( 2 + 3 + ( 4 + 6 ) ) + 3 ) \* 3 | 5 + 20 \* 3 |

**Function** ProcessMultiplyDivide

For handling Multiply and Divide scenarios, first we’ll locate the indexes of all Multiple (\*) and Divide (/) syntax from the List. Then we’ll loop through the indexes, locating the numbers before and after the syntax (we assume the numbers before/after the index will be actual numbers).

I’ve also added additional checking to see if the next iteration is next to the current index or not (separated by 2). If yes, we’ll nullify current sum and update the number after current syntax with the total sum instead. This is to cater for scenarios e.g. “2 \* 2 \* 2”. Each iteration by the end will store the Start/End Index of the syntax and its numbers, along with the total sum into a List.

Once the loop is complete, we’ll loop through the mentioned List above from the last index till the first (to prevent index error), and replace the value e.g. “2 \* 2” with the actual sum e.g. 4. If the sum found is null, then we’ll remove the value “2 \* 2” only.

The end result if “2 \* 2 \* 2” will be “8” based on the logic above.

**Function** GetTotalSum

For handling Plus and Minus scenarios e.g. “1 + 2 + 3”, we’ll loop through the passed in List, and check if the current iteration value is a double or not. If it’s not, we’ll assume it’s either Plus or Minus syntax, and store the syntax in a local variable. When the Loop iterates to a Number row, we’ll do the calculation based on the stored Syntax earlier with earlier totaled up value.

Once the Loop is complete, the end result of “1 + 2 + 3” will return value of “6”.

**Struct** ValuePair

I’ve also added a Struct to include properties like Start Index, End Index, and the total Sum for the Start/End Index calculation. This is being used for Brackets and Multiply/Divide scenarios.

**Limitation**

The solution doesn’t have any **Unit Test** implemented as I’m not that familiar with the flow.