Note: all source code included in this project was written by myself, Ryan San Miguel, during the spring 2018 semester with the exception of ‘ArgumentManager.h’ which was provided by Dr. Giulia Toti to process command-line arguments and the few changes described in section (IV) which were made during the spring 2019 semester.

I. Purpose

The purpose of this project is to convert a given infix expression to its equivalent form in postfix notation and to compute the result of this expression. This project was done as part of the course work for COSC 2430 - Programming and Data Structures at the University of Houston. The solution was designed to meet the requirements of the assignment as set out by the instructor, Dr. Giulia Toti. Errors were detected after the assignment was submitted and subsequently corrected. Changes made after submission are detailed in section (IV).

II. Background

Prefix, infix, and postfix notations are three different, yet equivalent ways of writing arithmetic expressions. This project is only concerned with the latter two, specifically converting from infix to postfix.

Infix expressions are written as two operands separated by an operator. Equation (1), shown below, is an example of a simple infix expression.

|  |  |  |
| --- | --- | --- |
|  |  | (1) |

They require special properties of operators such as operator precedence and associativity as well as the use of parentheses to indicate the order of operations. For example, consider equation (2) shown below.

|  |  |  |
| --- | --- | --- |
|  |  | (2) |

The procedure for evaluating this expression is as follows.

|  |  |  |
| --- | --- | --- |
| Step 1: |  | (3) |
| Step 2: |  | (4) |
| Step 3: |  | (5) |

Postfix expressions, one the other hand, are written as two operands *followed by* an operator. Equation (6), shown below, is an example of a simple postfix expression and is equivalent to equation (1).

|  |  |  |
| --- | --- | --- |
|  |  | (6) |

Unlike infix expressions, they do not require operator precedence nor associativity and the use of parentheses is unnecessary. The order of operations is determined by the order in which the operators appear. That is the leftmost operator indicates the first operation to be performed; the rightmost operator, the last. It is important to note that for subtraction and division, which are not commutative, the order of the operands matters. For example, in equation (6), the variable is the dividend and is the divisor. Consider equation (7) shown below.

|  |  |  |
| --- | --- | --- |
|  |  | (7) |

The procedure for evaluating this expression is as follows.

|  |  |  |
| --- | --- | --- |
| Step 1: |  | (8) |
| Step 2: |  | (9) |
| Step 3: |  | (10) |

Note that the order of operations for evaluating both equations (2) and (7) is the same - addition, multiplication, division. Also, note that equations (3), (4), and (5) are equivalent to equations (8), (9), and (10), respectively. This is because equation (7) is the postfix equivalent of the infix expression in equation (2). The process for converting from equation (2) to (7), and indeed any infix to postfix conversion, is described in the section below.

III. Method

a. Infix to Postfix Conversion

The instructions for the assignment define two special symbols which the program must be able to handle - CE, for clear everything, which clears all preceding symbols in the infix expression and C, for clear, which clears just the symbol immediately preceding it. The first step in the conversion process is to check for these symbols in the infix expression and to modify it accordingly.

The next step is to check that the parentheses are balanced. That is for every open parenthesis, there exists a matching closed parenthesis. This is done using the following procedure. Iterate through each symbol in the infix expression. If a symbol is an open parenthesis, push it onto the top of the stack. If a symbol is a closed parenthesis and the stack is empty, then the expression contains at least one more closed parenthesis and is unbalanced. If after iterating through the entire expression, the stack is not empty, then the expression contains at least one more open parenthesis and is, again, unbalanced. It is possible for one of the special symbols to cause the expression to become unbalanced. Therefore, it is important to handle them first.

The last check is to ensure that no two operands and no two operators are adjacent as this constitutes an invalid infix expression. Once again, iterate through the expression. If a symbol is an operand and the previous symbol was not, then append it to the string containing the resulting postfix expression. If a symbol is an operator and the previous symbol was not, then refer to the stack. If the stack contains operators of greater precedence, append them to the postfix expression. Then, push the current symbol on top of the stack. If a symbol is an open parenthesis, then push it on top of the stack. If a symbol is a closed parenthesis, then pop the top of the stack and append it to the postfix expression. Continue doing so until a matching closed parenthesis is present atop the stack. Pop the closed parenthesis, but do not append it to the postfix expression. After iterating through the entire infix expression, pop the remaining elements and append them to the postfix expression. For a simple example, consider equation (2) where . Table (1) shown below details this procedure where current represents the current symbol in the input infix expression and top represents the element atop the stack.

**Table 1**: Example of How to Convert from Infix to Postfix

|  |  |  |  |
| --- | --- | --- | --- |
| Input: “2 \* ( 1 + 3 ) / 4” |  |  |  |
| Current | Top | Postfix | Comment |
| 2 | NULL | “” | append 2 to postfix |
| \* | NULL | “2” | push \* onto stack |
| ( | \* | “2” | push ( onto stack |
| 1 | ( | “2” | append 1 to postfix |
| + | ( | “2 1” | push + onto stack |
| 3 | + | “2 1” | append 3 to postfix |
| ) | + | “2 1 3” | pop stack and append to postfix |
| ) | ( | “2 1 3 +” | pop stack |
| / | \* | “2 1 3 +” | pop \* from stack and append to postfix |
| / | NULL | “2 1 3 + \*” | push / onto stack |
| 4 | / | “2 1 3 + \*” | append 4 to postfix |
| /0 | / | “2 1 3 + \* 4” | pop remaining elements and append to postfix |
| /0 | NULL | “2 1 3 + \* 4 /” | final result |

b. Evaluating Postfix Expressions

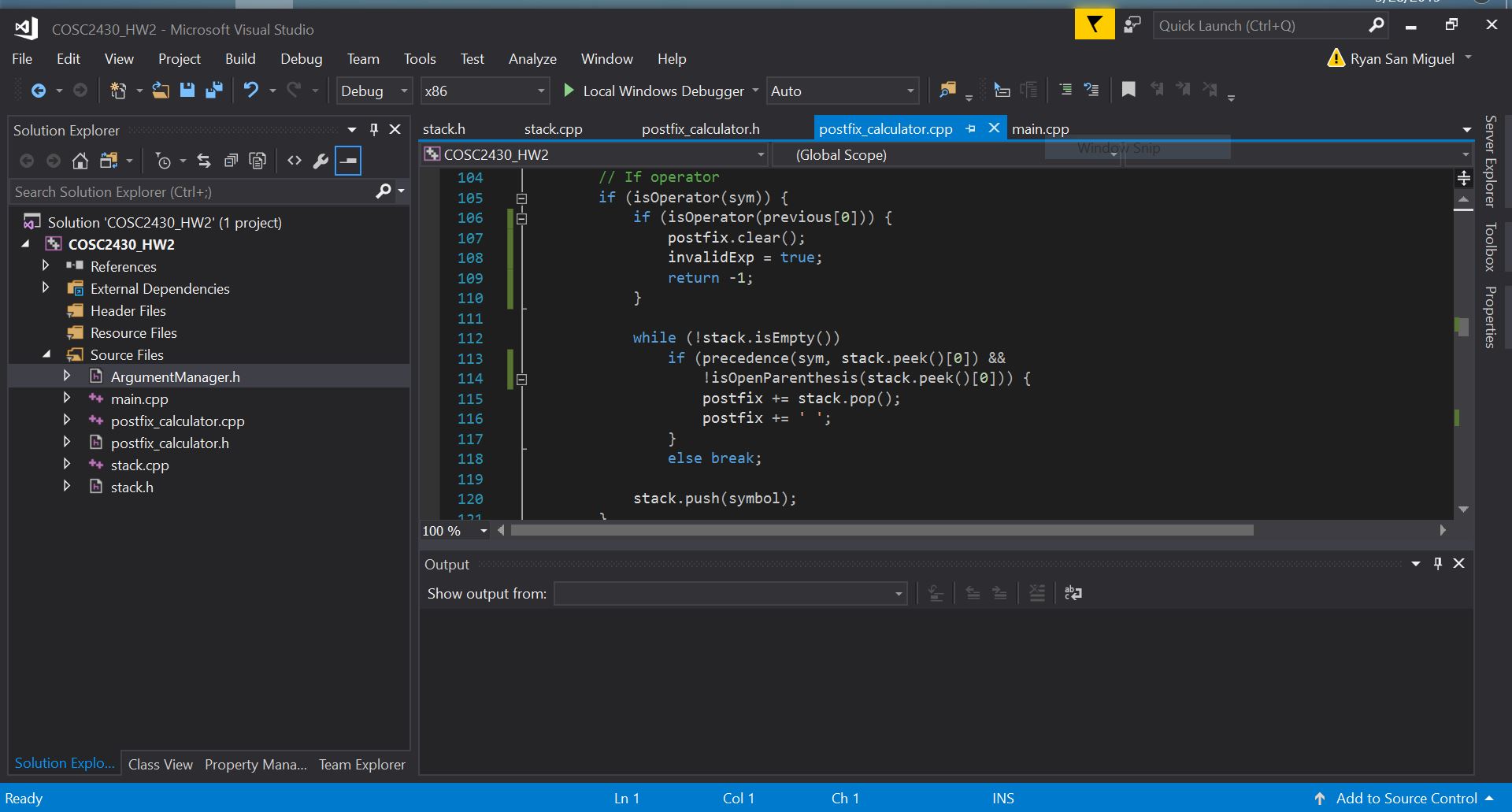
Once the given infix expression has been converted, the resulting postfix expression can be evaluated. This can be done with the use of a stack. This particular solution uses a linked list-based implementation of a stack which, for simplicity, was designed to store strings only. The procedure is described below.

In this project, the postfix expression to be evaluated is stored as a string of characters. The first step is to iterate through each character in the string. If the character represents an operand, push it onto the stack. In this case, operands are guaranteed to be numerical values. Both integer and decimal values are acceptable. Operands and operators are separated by blank space. If a character represents an operator, pop the top two elements from the stack and perform the appropriate operation using these two values as the operands. Then, push the result of this operation onto the stack and continue iterating through the string. Finally, after reaching the end of the string, pop the top element from the stack; this is the final result. As an example, once again, consider equation (7). For this example, we will assign values such that . Table (1) shown below details this procedure where ‘current’ represents the current character in the string; ‘top’, the element at the top of the stack.

**Table 2**: Example of How to Calculate the Result of a Postfix Expression

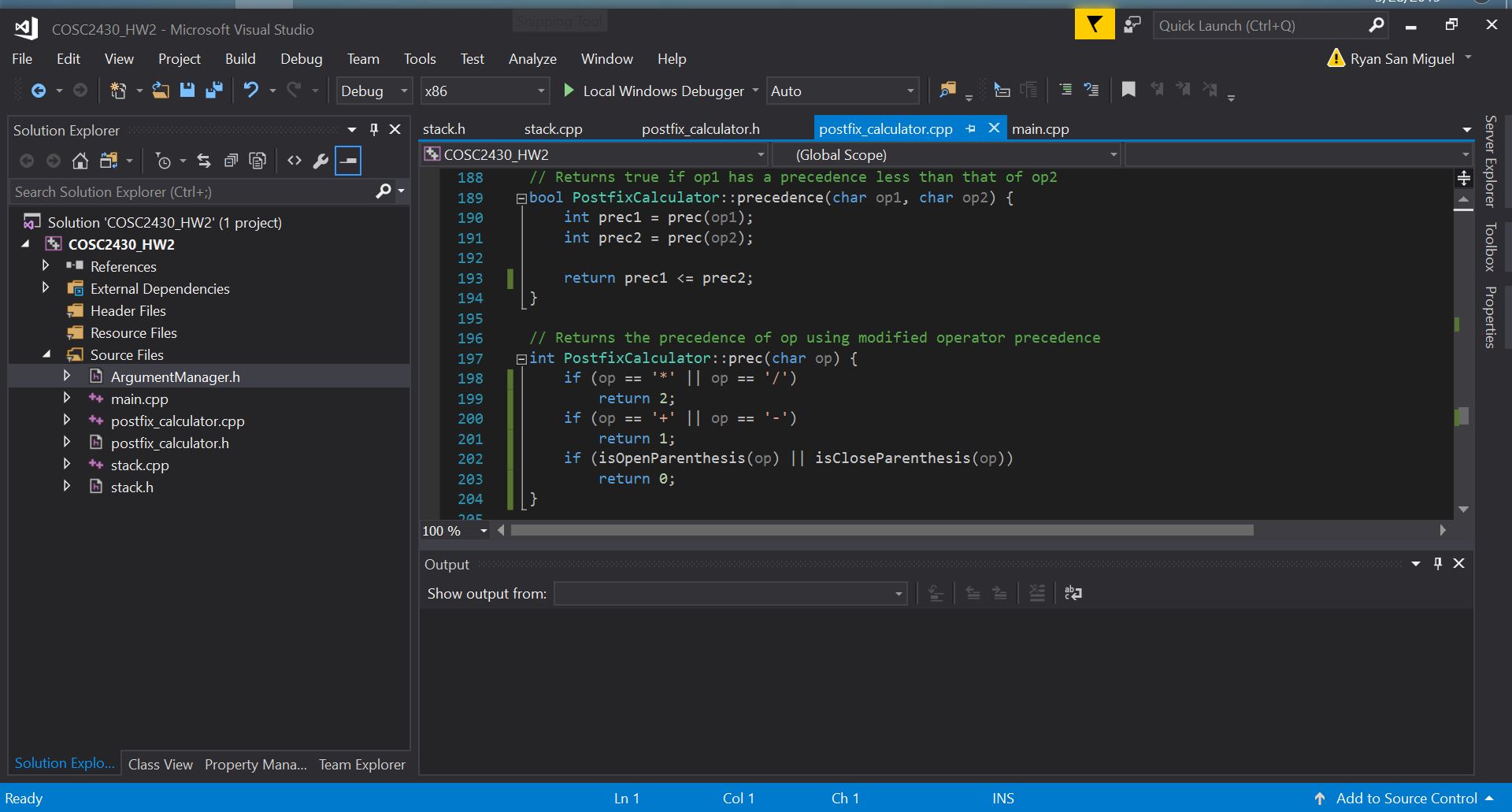
|  |  |  |
| --- | --- | --- |
| Input: “2 1 3 + \* 4 / ” |  |  |
| Current | Top | Comment |
| 2 | NULL | push 2 onto the stack |
| 1 | 2 | push 1 onto the stack |
| 3 | 1 | push 3 onto the stack |
| + | 3 | pop 3 from the stack  rhs = stack.pop(); |
| + | 1 | pop 1 from the stack  lhs = stack.pop(); |
| + | 2 | sum = lhs + rhs;  push sum onto the stack |
| \* | 4 | pop 4 from the stack  rhs = stack.pop(); |
| \* | 2 | pop 2 from the stack  lhs = stack.pop(); |
| \* | NULL | product = lhs \* rhs;  push product onto the stack |
| 4 | 8 | push 4 onto the stack |
| / | 4 | pop 4 from the stack  rhs = stack.pop(); |
| / | 8 | pop 8 from the stack  lhs = stack.pop(); |
| / | NULL | quotient = lhs / rhs;  push quotient onto the stack |
| /0 | 2 | result = stack.pop() |

IV. Post-Submission Changes



Lines 107-109: If there are two adjacent operators in the infix expression, then the expression is invalid. Clear the string containing the postfix expression so that no expression is output to the file. Set the ‘invalidExp’ flag to true and return -1.

Line 114: If the stack contains an open parenthesis, do not append it to the postfix expression. While parentheses are not necessary when using postfix notation, they are not invalid. However, for the purpose of the assignment, the output postfix expression should not include parentheses.



Line 193: The function precedence() was changed to return true if ‘op1’ has a precedence less than *or equal to* that of ‘op2.’

Lines 198-203: The helper function prec() was changed to use standard operator precedence