/\*\*

\* Class: CMSC 330

\* Instructor: Amitava Karmaker

\* Project Number: 2

\* Name: Nabeel Hussain

\* Date: 02/28/2017

\* Platform/Compiler: Microsoft Visual Studio 2015

\*

\* The main function reads in an input file, allowing multiple expressions arranged one

\* per line. For each expression in the file, it will call upon the static function parse

\* of the SubExpression class to parse it, and build an arithmetic expression tree.

\* It then calls the subordinate function parseAssignments to parse the assignments and

\* enter them into the symbol table, and then evaluates the expression and displays the

\* result.

\*/

Project 2 involves completing and extending the C++ program that evaluates statements of an expression language contained in the module 3 case study. The statements of that expression language consist of an arithmetic expression followed by a list of assignments. Assignments are separated from the expression and each other by commas. A semicolon terminates the expression. The arithmetic expressions are fully parenthesized infix expressions containing integer literals and variables.

**Your first task is to complete the program provided by providing the three missing classes, Minus, Times and Divide.**

The implementation of these classes are in the files **minus.h**, **times.h**, and **divide.h** respectively.

**Next, you should extend the program so that it supports relational, logical and conditional expression operators as defined by the following extension to the grammar:**

<exp> -> '(' <operand> <op> <operand> ')' **|**

'(' <operand> ':' <operand> '?' <operand> ')' **|**

'(' <operand> '!' ')'

<op> -> '+' | '-' | '\*' | '/' | '>' | '<' | '=' | '&' | '|'

The implementation of these new operators are in the files **greaterthan.h**, **lessthan.h**, **isequalto.h**, **and.h**, and **or.h**

When the expression is being evaluated, there are three different options to choose from depending on what the third character in the expression is. If it’s an **‘!’** character, then you follow the '(' <operand> '!' ')' model for a negation statement. If it’s a **‘:’** character, then you follow the

'(' <operand> ':' <operand> '?' <operand> ')' model for evaluating a conditional statement. Otherwise, it will be an **<op>** character, so you will need to follow the '(' <operand> <op> <operand> ')' model for evaluating an arithmetic, relational, or logical statement.

For evaluating a conditional statement, I created a new class called ConditionalExp, which is implemented in the file **conditonalexp.h**

**Your final task is to make the following two modifications to the program:**

**- The program should accept input from a file, allowing for multiple expressions arranged one per line.**

**- All results should be changed from double to int. In particular, the evaluate function should return an int**.

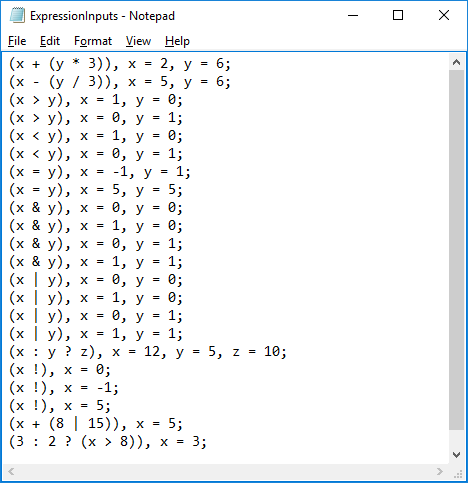
**Test Plan:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case** | **Input:** | **Expected Output** | **Did Test Pass?** |
| 1 | **(*x* + (*y* \* 3)), *x* = 2, *y* = 6;**  \*Will test an expression containing the addition and multiplication arithmetic operators | Value = 20 | Y |
| 2 | **(*x* - (*y* / 3)), *x* = 5, *y* = 6;**  \*Will test an expression containing the subtraction and division arithmetic operators | Value = 3 | Y |
| 3  4 | **(x > y), x = 1, y = 0;**  **(x > y), x = 0, y = 1;**  \*Will test the ‘>’ operator | Value = 1  Value = 0 | Y |
| 5  6 | **(x < y), x = 1, y = 0;**  **(x < y), x = 0, y = 1;**  \*Will test the ‘<’ operator | Value = 0  Value = 1 | Y |
| 7  8 | **(x = y), x = -1, y = 1;**  **(x = y), x = 5, y = 5;**  \*Will test the ‘=’ operator | Value = 0  Value = 1 | Y |
| 9  10  11  12 | (**x & y), x = 0, y = 0;**  **(x & y), x = 1, y = 0;**  **(x & y), x = 0, y = 1;**  **(x & y), x = 1, y = 1;**  \*Will test the ‘&’ operator | Value = 0  Value = 0  Value = 0  Value = 1 | Y |
| 13  14  15  16 | **(x | y), x = 0, y = 0;**  **(x | y), x = 1, y = 0;**  **(x | y), x = 0, y = 1;**  **(x | y), x = 1, y = 1;**  \*Will test the ‘|’ operator | Value = 0  Value = 1  Value = 1  Value = 1 | Y |
| 17 | **(x : y ? z), x = 12, y = 5, z =10;**  \*Will test a conditional expression, where symbols are reversed and the third operand represents the condition. The first operand is the value when true and the second the value when false | Value = 12 | Y |
| 18  19  20 | **(x!), x = 0;**  **(x!), x = -1;**  **(x!), x = 5;**  \*Will test the ‘!’ operator | Value = 1  Value = 0  Value = 0 | Y |
| 21 | **(x + (8 | 15)), x = 5;**  \*Will combine the ‘+’ and ‘|’ operators | Value = 6 | Y |
| 22 | **(3 : 2 ? (x > 8)), x = 3;**  \*Will combine a conditional expression with a ‘>’ operator | Value = 2 | Y |
|  | **Input file name:** ExpressionInputs.txt  **(x + (y \* 3)), x = 2, y = 6;**  **(x - (y / 3)), x = 5, y = 6;**  **(x > y), x = 1, y = 0;**  **(x > y), x = 0, y = 1;**  **(x < y), x = 1, y = 0;**  **(x < y), x = 0, y = 1;**  **(x = y), x = -1, y = 1;**  **(x = y), x = 5, y = 5;**  **(x & y), x = 0, y = 0;**  **(x & y), x = 1, y = 0;**  **(x & y), x = 0, y = 1;**  **(x & y), x = 1, y = 1;**  **(x | y), x = 0, y = 0;**  **(x | y), x = 1, y = 0;**  **(x | y), x = 0, y = 1;**  **(x | y), x = 1, y = 1;**  **(x : y ? z), x = 12, y = 5, z = 10;**  **(x !), x = 0;**  **(x !), x = -1;**  **(x !), x = 5;**  **(x + (8 | 15)), x = 5;**  **(3 : 2 ? (x > 8)), x = 3;**  \*Will test if the program can read the expressions in a file, parse them correctly, and evaluate them. | **(x + (y \* 3)), x = 2, y = 6; Value = 20**  **(x - (y / 3)), x = 5, y = 6; Value = 3**  **(x > y), x = 1, y = 0; Value = 1**  **(x > y), x = 0, y = 1; Value = 0**  **(x < y), x = 1, y = 0; Value = 0**  **(x < y), x = 0, y = 1; Value = 1**  **(x = y), x = -1, y = 1; Value = 0**  **(x = y), x = 5, y = 5; Value = 1**  **(x & y), x = 0, y = 0; Value = 0**  **(x & y), x = 1, y = 0; Value = 0**  **(x & y), x = 0, y = 1; Value = 0**  **(x & y), x = 1, y = 1; Value = 1**  **(x | y), x = 0, y = 0; Value = 0**  **(x | y), x = 1, y = 0; Value = 1**  **(x | y), x = 0, y = 1; Value = 1**  **(x | y), x = 1, y = 1; Value = 1**  **(x : y ? z), x = 12, y = 5, z = 10; Value = 12**  **(x !), x = 0; Value = 1**  **(x !), x = -1; Value = 0**  **(x !), x = 5; Value = 0**  **(x + (8 | 15)), x = 5; Value = 6**  **(3 : 2 ? (x > 8)), x = 3; Value = 2** | Y |

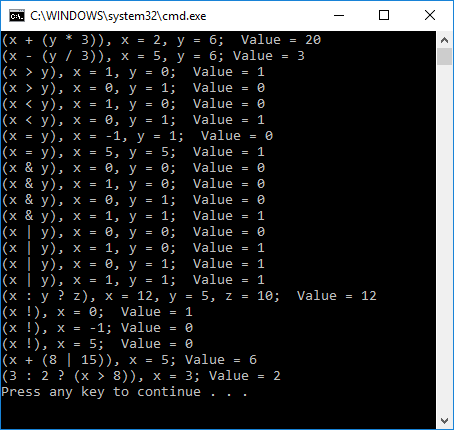
**Screen shot of successful compilation and running for all test cases**

**Test Case:** Evaluating expressions contained in an input file.

**ExpressionInputs.txt**



**Output:**



**Lessons Learned:**

While working on Project 2, there were a lot of new concepts and techniques that I learned about evaluating expression statements and using inheritance to implement the program. It was a bit challenging at first because I didn’t have much experience with C++, since I had mainly worked with Java in my previous courses. I had to go back and refresh my memory on different C++ terminology and basically learn how to program in the language.

One thing I leaned while working on this assignment was how to use inheritance in C++. When a parent class is defined, in which other subclasses will inherit information from, then the use of a virtual function plays a big role. In C++, virtual functions are used to denote those subprograms for which dynamic binding is performed. A subprogram in the derived class with the same name and parameter signature as that of a virtual subprogram in the parent class is also considered virtual.

The most difficult part of this assignment in my mind was to extend the program so that it supports relational, logical and conditional expression operators as defined by the given extension to the grammar. I had to extensively think about how to do this for a very long time, and after a lot of trial and error I got it to work correctly. The logic I followed was that after the first operand has been read in, then the next thing read will either be an <op>, ':', or '!' character. This will determine what type of expression is being evaluated. Using if else statements, if the next operation read in is a ':' character, then it means that it will be a conditional expression. If it's a '!' character, then it means that it will be a negation expression. Otherwise, it will evaluate the expression for an ordinary expression depending on the <op>.

Overall, this assignment was challenging, but it helped me understand and reinforce all the material I learned from the weekly modules during this course. I feel like I am a better programmer now then when I first began this course, which is what I hoped for. Thank You.