Nicholas Mills

CMSC 330

10/14/18

Project #2: Expression Evaluator

**Process:**

The three major changes requested for this project were adding operators, reading input from a file with new line separation, and changing the result type from double to integer.

Adding the arithmetic operators was simple. I simply copied add.h, modified it slightly for each of the other arithmetic operators, included them and added cases for them in the switch statement within subexpression.cpp. Additionally, I added checking for divide by 0 in divide and printed an error message to the file in this event. ‘|’, ‘&’, and ‘=’ were also relatively simple to add. Since any positive value can be evaluated as true, I simply checked to see if left and right evaluate were greater than 0 and performed the proper logical operators on that result.

‘!’ and conditionals posed a greater challenge, since they didn’t follow the standard <operand> <op> <operand> format of the other operators. To deal with this, I created two other constructor types for SubExpression, one of which has a left, middle, and right expression and the other of which only has a single left expression. I then created Conditional and Not classes which used those constructors respectively. I added in a second switch statement before the existing one, after parsing only the left Operand, since all operations require at least the left. In this switch statement, I check for case ‘:’ and case ‘!’, retrieve the proper number of Operands for those operators, and create the proper object to return. If neither of those cases occurs, I can safely retrieve one more operand and check for the other operators, since the remaining operators all need two.

In order to read in from a file, I created an ifstream from a file specified by the user. I then passed a reference to that file stream all the way down to any functions which used cin and replaced it with my file stream. In order to process multiple lines, I put the bulk of main within a while(!in.eof()) loop and advanced to the next paren at the end each loop. This forces the eof trigger before the next loop. Now, this means that my code would work whether there is a newline separating each expression or not, but I did not account for this since it was said in the prompt that we could assume that syntax was always going to be correct. I also decided to have the output placed in a file determined by the user as well for easier side by side comparison of the input and output.

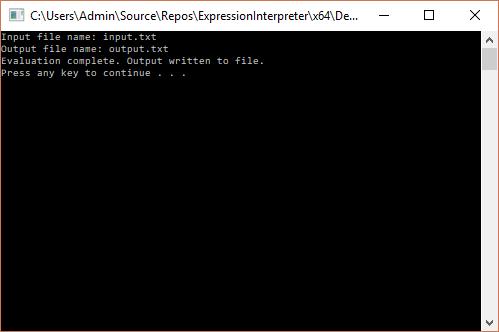
Another issue encountered in this step was the fact that SymbolTable was storing values for variables in between expression parses. That resulted in previous values being used for calculations even if new values were declared. My solution for this was to reset the SymbolTable after each loop. I accomplished this by writing a reset() function that cleared the vector storing the Symbol objects.

For the last step, I simply changed all references from doubles to integers. This doesn’t result in any major loss of functionality since doubles never processed correctly anyway, i.e. they already get cut off at the decimal. The only functionality loss is the ability to use double variables, which I’m unsure was an intended feature to begin with.

**Test Cases:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Purpose | Input | Expected Output | Actual Output | P/F |
| Example Case, tests addition and multiplication. | (x + (y \* 3)), x = 2, y = 6; | Value = 20 | Value = 20 | P |
| Test subtraction, division and negative results. | (x – (y / 2)), x = 6, y = 20; | Value = -4 | Value = -4 | P |
| Test logical operators true | (((x !) & y) | x), x = 0, y = 1; | Value = 1 | Value = 1 | P |
| Test logical operators false | (((x !) & y) | x), x = 0, y = 0; | Value = 0 | Value = 0 | P |
| Test conditional true | (10 : 5 ? z), z = 1; | Value = 10 | Value = 10 | P |
| Test conditional false | (10 : 5 ? z), z = 0; | Value = 5 | Value = 5 | P |
| Test relational operators true | (((x = y) | (y > z)) | (x < z)), x = 1, y = 1, z = 0; | Value = 1 | Value = 1 | P |
| Test relational operators false | (((x = y) | (y > z)) | (x < z)), x = 5, y = 1, z = 2; | Value = 0 | Value = 0 | P |
| Prove int not double | (x / 3), x = 10; | Value = 3 | Value = 3 | P |
| Divide by 0 | (x / 0), x = 5; | Problem with line: Divide by zero. | Problem with line: Divide by zero. | P |

**Screenshots:**



**Lessons Learned:**

Seeing as I haven’t coded in C++ in almost 10 years, and even then, we weren’t really using OOP as it was a very basic high school course, this project allowed me an opportunity to see how C++ functions in a variety of ways, especially in how it differs from Java as that is what I’ve grown accustomed to using in UMUC’s Computer Science program. Some major differences that I had to deal with in this project were the separation of specification from implementation through header files, the need to explicitly pass variables by reference in regard to my file input stream, and the inability to reset SymbolTable by simply creating a new one and setting the existing variable to that. Also when writing the error handling for divide by zero, I discovered that a try/catch block to catch the exception thrown by default did not work. It took me a bit to discover that this is technically not an exception for C++. I also became more familiar with switch/case statements, seeing as I typically tend to use if/else more frequently.