Project 12: Expression Evaluation

In this project, you will write a basic interpreter, which can evaluate mathematical expressions. Knowing how to do this will come in handy if you ever have to add scripting functionality to a project that you’re working on.

This assignment will teach you how to perform evaluation of simple mathematical expressions, containing:

1. Numbers (positive integers)
2. Operators: for multiplication, division, addition, and subtraction
3. Parentheses.

The project should be completed in stages:

1. Scanning a number from a string and converting it to an integer.
2. Tokenizing a string.
3. Testing for balanced parentheses.
4. Validating syntax for an infix expression, and converting it to postfix.
5. Evaluating a postfix expression to obtain a final result.

Here is a description of the files in this project:

* ExpressionManager.h: contains important definitions for you to use (described later)
* ExpressionMaager.cpp: contains the factory class, which you will need to modify.
* TestCase.h and TestCase.cpp: these should be old-hat, by now.
* Main.cpp: contains all the test cases you’ll need to pass off.

# Getting everything set up

To get set up:, start by unpacking everything into a directory, and open it up in Visual Studio.

The first thing you’ll need is to define a class that implements IExpressionManager. This is kind of like what you did with the sort demo, but in the case of this project you only need to make one new class. This class will have the code for your entire assignment. So create a new class like you did in the sort demo project (right-click on your solution, and from the pop-up menu select New🡪Class…). Follow the instructions. If you’re using Xcode, you’ll need to create a header file and a .cpp file.

## Setting up your header file:

Open up your header file, and do the following. First, you’ll need to #include two STL header files:

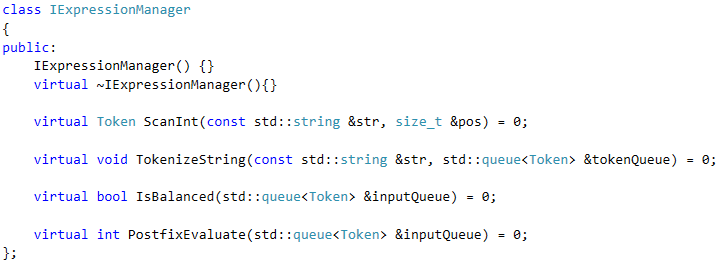


Next, you’ll need to include the header file for this project.



After that, you’ll need to define your class. You can call it anything you like, it doesn’t matter. Make sure you inherit IExpressionManager.

Next, you’ll need to copy all the method declarations out of the IExpressionManager interface (look in ExpressionManager.h), and paste the methods into your new class. This is the same thing you did with the sort-demo project. Make sure you copy all the methods out of IExpressionManager, there are five:



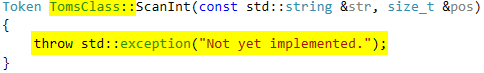
Finally, in your new class, you’ll have to delete the =0 at the end of each method. You did this in the sort-demo project, too.

## Setting up your .cpp file

Start out by making stub methods for everything that you defined in your .h file

1. Delete the virtual keyword
2. Remember to put your class’s name with a :: in front of each method name.
3. Put a body that throws a not-implemented exception, until you’re ready to implement each method:

Here is an example (I’ve called my class TomsClass):



Please do not cram all your code into your .h file! I’ll take off 10 points if I see this. You’re not doing a template class for this project, so I want to see a .h file where you define your class, and a .cpp file where you define implementations for each of the methods in IExpressionManager. I will allow an exception to this rule for constructors, or for any method that is O(1). If you declare an inline method, it should have three lines or less, it may have no more than one if-statement, and it must contain no loops or recursion.

# The ScanInt() method

The ScanInt method extracts a number from a string, converts it to an integer (int), and returns the value in a new Token object.



I’ve described the algorithm for ScanInt in the slide deck. ScanInt takes two parameters:

1. an STL string that contains an expression
2. an integer which contains a starting position within the string where ScanInt should begin scanning.

Requirements:

1. If the starting position is greater than or equal to the string’s length, throw an exception.
2. If the starting position does not point to a digit, throw an exception.
3. The parameter, pos, is a reference to an integer. You are expected to update this value as you scan in the integer.
4. You can use any algorithm you like to convert a string to an integer, as long as you can adapt it to the requirements of this program.
5. ScanInt must return an instance of the Token object. The Token class is defined in ExpressionManager.h

The parameter, pos, is used to indicate the index of the character where you want to start scanning, like so:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |  |
| … | \* |  | ( | 6 | 2 | 3 | 4 |  | - |  | 3 | 4 | … |

When ScanInt returns, the value of pos should point to the first character after the number that was just scaned in:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |  |
| … | \* |  | ( | 6 | 2 | 3 | 4 |  | - |  | 3 | 4 | … |

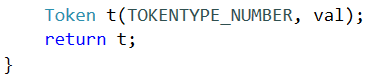
You will find these tips helpful:

1. The ScanInt method uses a STL string. You can look up examples using the search engine of your choice by querying things like “stl string get length”, or “stl string convert to char\*”
2. You can get the length of a string by calling its length() method.
3. You can use array indexing to get the individual characters in the string. For example, str[5] will get the sixth character (the character at position 5) in the array.
4. You can use the isdigit macro to determine whether or not the current character is a digit. This macro is defined in ctype.h. Here is an example: isdigit(str[5])

When you’re finished scanning in the integer, ScanInt must return a Token object. The token class has two public fields:

* Type: this is one of the TOKENTYPE\_xxx enumerated type values.
* Value: this is used only when Type == TOKENTYPE\_NUMBER. It is ignored, otherwise.

The Token class has two constructors. You can use whichever one you like. To make ScanInt return a Token object, make an instance and return it, like you would anything else:



# The TokenizeString() method

Tokenizing a string is the first step to being able to work with the data that it contains, and is something basic that all compilers and interpreters do. The TokenizeString() method will scan a string and convert it to a queue of tokens.

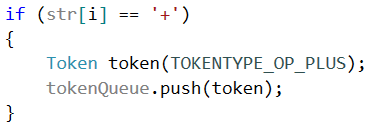


This method has no return type, and takes two parameters:

1. An STL string containing the expression to be parsed.
2. An STL queue object, where you will store the tokens as you scan them from the string.

Use a loop to iterate through all the characters in the input string. As you scan each token, you will insert (enqueue) them into tokenQueue. The tokenQueue parameter is set up for you, and is all ready to use.

You can use array indexing on an STL string to convert each character to a token. Here is how you would create a Token object for a + operator:



If you encounter a digit, you should call ScanInt to scan in the token and move your position to the next character after the number.

You should ignore any whitespace in the string. You can use the isspace macro to find whitespace characters. There can be zero or more whitespace characters between any two tokens. Don’t depend on it being there, and if you do find it don’t assume there’s only one character.

You must throw an exception if you encounter a character that is not one of the following:

1. Digits 0 through 9
2. Operators +, -, \*, or /
3. Parentheses: ( and )
4. Space character: ‘ ‘

# The IsBalanced() method

This method is very simple. All you have to do is dequeue the tokens one at a time and tell whether or not the parentheses in the expression (if there are any) are balanced.



You do not have to preserve the contents of the queue. Just return a true/false value.

1. If the expression has no parentheses, return true
2. If the parentheses are balanced, return true
3. Otherwise, return false.

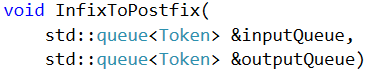
As I mentioned in the slide deck, dequeing a token is simple:



The slide deck contains a description for how to implement this method. If you like, you should try figuring this out on your own.

# The InfixToPostfix() method

You will spend the most time on this method. The algorithm is in the slide deck, and converting it to C++ should be straightforward.



You’ll need to declare an STL stack to keep track of the operators. Other than that, you should be able to use the inputQueue parameter for the input queue (as described in the slide deck), and the outputQueue for your output queue.

To check the token’s type, use its Type property:



If the token is an operator, you may want to define a method to get its precedence. The \* and / operators should have a precedence of 2. The + and – operators should have a precedence of 1. Parentheses have no precedence.

# The PostFixEvaluate() method

This method is described in the slide deck.



Implementation is straightforward. You ought to be able to code this up on your own if you’ve ever owned a TI or HP graphing calculator, which uses reverse-Polish notation.

1. If you get a number, push it onto the stack
2. If you get an operator, pop two numbers from the stack, do the operation, then push the result back onto the stack.

Just remember, the number at the top of the stack is the right operand, and the next number down is the left operand. This is important to remember since – and / are not commutative (a – b ≠ b – a).

# Returning helpful error messages

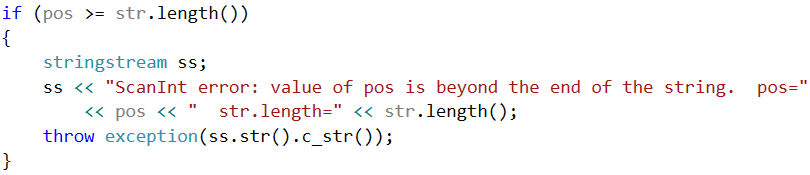
The test cases will print out the message of any exception that you throw, so you might want to take time and put as much detail as you can into them.

I like to use the stringstream class. This class contains a string buffer, which you can add stuff to, just like with cout.

Once you have your error message composed, you’ll need to convert your stringstream object to a char\* (the exception constructor takes only char\*). This conversion happens in two stages:

1. Convert the stringstream to a STL string, using stringstream’s str() method.
2. Convert the STL string to a C-string, using string’s c\_str() method.

Here is a complete example of me throwing an error message:



You’re not required to have robust error messages for this assignment. That said, you might want to get into the habit of doing this, early. If you develop a habit of giving good error info, it will save you time down the road when you’re working on a large project and you have to fix bugs.

# Submitting your project

Please do not submit a single .h file. The easiest way to submit this project is to zip up all your files and submit the whole thing. IMPORTANT: Please un-zip your project and make sure you have everything in it before uploading it to LearningSuite.