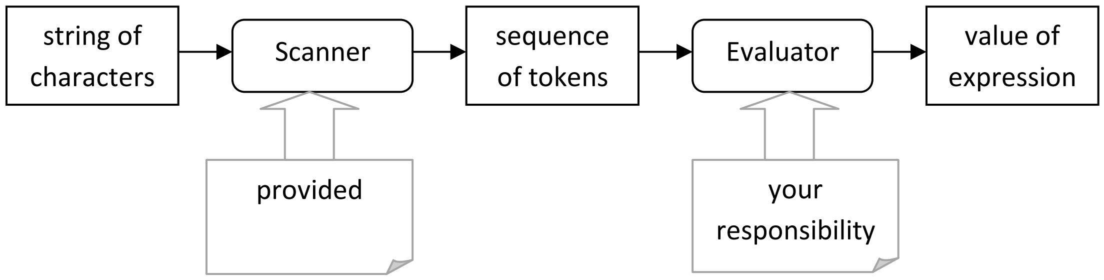
Programming Assignment 3
Due: Friday, October 12, 11:55PM

Objective: Build a text based calculator. Your calculator will accept *postfix expressions*, also known as Reverse Polish Notation (RPN) expressions, evaluate the expressions and print the result. You will be given a `Scanner` class that converts character strings containing expressions into a sequence of tokens. You will process the sequence of tokens using a *stack-based evaluation algorithm*.

RPN Expressions: Wikipedia has a reasonable entry on Reverse Polish Notation, refer to it if you are not familiar with this type of expression. If you've used a calculator that expects operators to be pressed after the operands are entered, then you've probably used RPN. Examples:

RPN	equivalent infix	value
`3 4 5 + *`	`3 * (4 + 5)`	`27`
`6 2 + 9 1 - /`	`(6 + 2) / (9 – 1)`	`1.5`

System Overview: We'll approach this problem in a manner that is similar to how compilers work. The user will type an expression, which will be stored in a character string. A scanner will break the string into words (numbers and operators). The individual words are called *tokens*. The sequence of tokens is then passed into an expression evaluator, which interprets them in some manner to determine their meaning. For this exercise, the meaning of an expression will be a single value that is the result of the implied arithmetic.



Evaluator Algorithm: The algorithm uses a stack of real numbers to hold values until operators can be applied to them. This algorithm assumes all input expressions are well-formed and does not indicate how to handle errors.

```
Algorithm PostfixEvaluator:
  input: a sequence of tokens representing a postfix expression
  output: a single real number, which is the value of the expression

while there are tokens remaining:
        if next token is a number:
                push it on the stack
        if next token is an operator:
                pop the top two values from the stack
                apply the operator to those values
                push result onto the stack
        advance to next token
pop single value from stack and return it
```

Provided Code: You'll be given a `Scanner` class and a `Token` class.

Instances of the `Scanner` class will accept expressions stored in STL `string` objects, and return a sequence of `Token` objects.

`Scanner` Public Interface:

- `A` default constructor, which creates a `Scanner` with an empty `Token` sequence.
- `void Scanner::processExpression(const string& s)`: Scans the `string` contained in the parameter `s` and stores the resulting tokens for later retrieval through the `nextToken` method. If the scanner detects any error in the expression, it will abort and clear the sequence of `Token` objects.
- `bool Scanner::nextToken(Token& token)`: Returns the next `Token` in the sequence, if one is available. The return value indicates whether the returned token is valid (`true`) or if there are no remaining tokens (`false`).

`Token` objects represent the "words" in the input expression. Tokens can be numeric values or character operators. The possible operator values are `'+'`, `'-'`, `'*'` and `'/'`with the usual arithmetic meaning. Numeric values can be any real number. It is the programmer's responsibility to determine the token type and call the appropriate method to extract a numeric value or a character operator.

`Token` objects should be considered "read-only" or "constant" objects, since they are created by a `Scanner` object and only access in your program.

`Token` Public Interface:

- Default constructor, copy constructors and assignment operators are provided.
- `bool Token ::isOperator()` : Returns true if this token is an operator, otherwise false.
- `bool Token ::isValue()` : Returns true if this token is a number, otherwise false.
- `float Token::getValue()` : Returns the numeric value of this token, if it is a value. If this token is not a value, the return value is undefined.
- `char Token::getOperator()` : Returns the character ('+', '-', '*' or '/') representation of the operator, if this token is an operator. If this token is not an operator, the return value is undefined.

In addition to the two classes, you'll be given a program that reads expressions from the keyboard, stores them in an STL `string` object, passes the string to a `Scanner` object and prints the resulting sequence of `Token` objects. See the code itself for details.

Expressions Accepted by the Scanner: The scanner expects all tokens to be separated by at least one space. Numbers can start with a minus sign, can contain zero or one decimal points and can have zero or more digits before and after the decimal point. Illegal expressions will be rejected, with a minimal message printed to `cerr`. Here are some examples of illegal expressions:

Illegal expression	**Reason**
`3 6 9 +* 2 /`	No space between + and * operators.
`4.5 12.34e5 +`	`12.34e5` is not a valid numeric token.
`1 2 3 + 4 *`	Insufficient number of operators.

Expression Length and Capacities: Since expressions will be typed at the keyboard, we can make the following assumptions and avoid capacity problems in our arrays and data structures:

- Each input expressions will be no more than 100 characters long.
- Each expression will have at most 51 tokens.
- Each expression will have at most 26 values and 25 operators.
- Your stack will never exceed a size of 26.

General Programming Rules for this Assignment:

- You may not alter files containing the `Token` class or `Scanner` class in any way. The only exception would be to modify the C library include files in Scanner.cpp, if the libraries in your programming environment are different.
- You can use the `stack` class we developed in lecture, the STL `stack` class, or you can develop your own stack implementation.
- Basic input error checking is managed by the `Scanner` class. You do not have to implement input validation.
- The PostfixEvaluator algorithm did not account for errors in the sequence of tokens. You do need to determine these potential errors, detect them and alert the user when they are detected. For example, if the value stack is empty when you need to pop values to apply an operator, your program must print an error message.

Development Hints:

- Typing and retyping expressions while debugging a program like this can be very tedious. To avoid this, put test expressions directly in your program and bypass the keyboard input until your evaluator is working properly for most expressions.
- You'll need to erase the elements in your stack between expressions, in case a malformed expression left some values in the stack.

What to Submit:

- All source code that you wrote or modified. Your name should appear near the top of all submitted code files.
- A screen capture (or multiple screen captures) showing how your program responded to a set of test expressions.