CS 104 (Fall 2013) — Assignment 5
Due: 10/15/2013, 11:59am

BitBucket directory name for this homework (case sensitive): HW5. Reminder: Only BitBucket submissions will be accepted, and make sure that your Makefile follows the conventions from HW4.

(1) Review C++ Interludes 3, 5 and Chapters 6, 7, 13.1, 13.2, 14.1 from the textbook.

(2) In this question, you will build a simple calculator for vectors, using a stack. The calculator will use "normal" (infix) notation of the form `5+3` instead of the `+ 5 3` version discussed in the textbook for a simple calculator.

(a) Implement a class for (mathematical) vectors of real numbers. Your class needs to provide at least the following functionality:

Constructors: You should have at least the following three constructors:

- a copy constructor
- a constructor that takes a `double` and builds a one-dimensional vector from it.
- a constructor from a string that takes a string of the form `[0.57,-1.414,3.14159]` and constructs from it the corresponding (3-dimensional) vector. Your constructor should figure out the correct dimension of the vector on its own. If the string is not correctly formatted, your constructor should throw a suitable exception.

Functions: `toString` returning a string with some human-digestible version of the vector, such as `[-1.3,5.2]`. If the vector is one-dimensional, it is up to you whether you want to print it as `[-1.3]` or just `-1.3`.

Overloaded Operators: You should have (at least) the following arithmetic operators available.

- `+` (addition) and `-` (subtraction). If the two vectors have different dimensions, your code should throw an exception.
- `*` (multiplication). If the two vectors have the same dimensions, then you should return the inner product (which is a one-dimensional vector). If one (or both) of the vectors has dimension 1, then you should return the result of multiplying the scalar with the vector. In all other cases (mismatching dimensions), your code should throw an exception.
- `=` (assignment)
- `==` (test for equality) and `!=` (test for inequality). It is up to you whether you want to return `false` or throw an exception when the dimensions don't match (though you should document your choice).
- `[]` (accessing individual elements). If the index is out of bounds, you should throw an exeption.

You may find the notes from Shane's lab (posted on Piazza) helpful, as they show how to implement operator overloading for complex numbers.

In your implementation, you should *not* use the C++ STL `Vector`. The recommended way is to just use a dynamically allocated array. But if you want, you can also use your own implementation of the `List` class from HW4.

(b) Implement a class for a member of your stack. As you will see in the next two parts of this problem, you will probably want to put different types of things on your stack, namely: (1) vectors, (2) opening parentheses, (3) mathematical operations. Hence, your class should be able to remember which of the three it is, and return the corresponding data.

(c) Implement a stack (using linked lists or dynamically expanding arrays, your choice; but do not use any STL classes). You can choose if you want to implement it with templates (more work now, but may be useful later), or just build a stack of the type that you defined for the previous subproblem.

Your stack should only have the following functions: `top`, `pop`, and `push`, as well as a default constructor building an empty stack. In particular, you cannot have a function `isEmpty` or another way to test whether the stack is empty. Instead, your functions `pop` and `top` should throw exceptions when called on an empty stack, and your other code should handle those. (Yes, this is not the perfect way to implement this, but we want to make you practice exceptions.)

(d) Use your stack implementation to build a text-based calculator for vectors. The user will type in a string that is a (possibly incorrect) formula, and your calculator should output the correct value/vector. The input formulas you need to be able to process are defined recursively as follows:

- Any vector written in the format `[a1,a2,...]`, where `a1,a2,...` are correctly formatted real numbers.
- Any single real number. (We recommend converting that into a one-dimensional vector.)
- The formulas `(A+B)`, `(A-B)`, `(A*B)`, where `A,B` are themselves formulas.

If the formula is not correctly formatted, you should return some error to the user.

Notice that the restrictions we placed on correctly formatted formulas should simplify your task a lot. You do not need to parse the formulas `[1,1]+[2,0]` (no parentheses) or `([1,1]+[2,0]+[-1,-1])` (three terms).

For your solution, you must use a stack. For once, a solution by recursion is not acceptable for full credit. We recommend the following algorithmic idea: scan your string from left to right. Whenever you encounter an opening parenthesis or arithmetic operation (`+`, `-`, or `*`), push it on the stack. Whenever you encounter an opening bracket, parse the vector and push it on the stack. When you encounter a number, parse is and put a one-dimensional vector of it on the stack. When you encounter a closing parenthesis, pop things from the stack and remember them until you hit an opening parenthesis. Now — assuming everything was correctly formatted — compute the value of the expression in parentheses, and push it onto the stack as a vector. When you reach the end of the string, assuming that everything was correctly formatted (otherwise, report an error), your stack should contain exactly one vector, which you can output.

(3) Chocolate Problem: This problem does not count towards your grade, but if you solve it, you can earn up to 2 chocolate bars. Feel free to specify your preferred type of chocolate — your requests may be accommodated if they are reasonable. Submitting this problem a few days late is ok, but in that case, please submit it directly by e-mail to David.

Extend your solution to the vector calculator problem in the following ways:

(a) Extend your calculator to parse formulas that do not have all those extra parentheses, e.g., `5+3*6`. Notice that this will also involve treating precedence correctly.

(b) Extend your calculator to parse formulas with more than two terms connected, e.g., `[5,3]+[1,2]+[-1,-2]`.

(c) Allow users to insert arbitrary spaces into their formulas (except of course in the middle of numbers).

(d) Add matrices (and matrix multiplication and matrix-vector multiplication) to your calculator. We recommend a format such as `[[5,3],[1,2]]`, but feel free to choose another one.