**Lab 3: Stacks** last modified by Aaron Bloomfield on September 19, 2013 9:18:21 PM EDT

# Laboratory 3: Stacks

## Week of Monday, September 16, 2013

### Objective:

To understand the workings of a stack as well as postfix notation, and to be introduced to the STL library

### Background:

A stack is a basic data structure similar in use to a physical stack of papers. You can add to the top (push) and take from the top (pop), but you are not allowed to access the middle or bottom. A stack adheres to the LIFO property.

### Reading(s):

1. Readings can be found online on the Readings page

## Procedure

### Pre-lab

1. Read this entire lab document before coming to lab.
2. Go through the first half of the Unix tutorial found in the Collab workspace (in the "03-04-more-unix" directory – the "01-unix-tutorial" directory is from lab 1): Tutorial 3: More UNIX, part 1. This tutorial is originally from the department of Electrical Engineering at the University of Surrey at http://www.ee.surrey.ac.uk/Teaching/Unix/. You should complete the introductory part and sections 1-4. You will be somewhat familiar with some of the materials in the first few of these tutorials, as it was in the Unix tutorial from the first lab. The rest of the tutorial (sections 5-8) are for next week's lab, but feel free to go through it this week, if you are interested.
3. Write up at least one question that you still have on Unix (or things you are still confused about) into unix.questions.txt.
4. Your code for the pre-lab will use the pre-existing STL stack class. The STL is the Standard Template Library, and is a collection of useful routines analogous to the routines in Java's SDK (it contains a vector class, for example).
    - To use the stack STL class, just put #include <stack> at the top of your C++ file. A standard clang++ installation should automatically find the STL stack class (this works in Linux and Cygwin, for example).
    - Documentation on the STL routines can be found at http://www.sgi.com/tech/stl/.
5. Implement a simple postfix stack calculator for integers using your stack.
    - **You should use the STL stack class**, rather than implement your own.
    - An online description of postfix calculators can be found on Wikipedia: http://en.wikipedia.org/wiki/Reverse_Polish_notation - you will need to implement this into postfixCalculator.h and postfixCalculator.cpp
    - Create a simple test driver, testPostfixCalc.cpp, which will be used to demonstrate your calculator. This file should have hard-coded values for input; keyboard input is the in-lab.
    - The last page of this document has some sample test cases you can use.
6. Your code must compile!
7. Be sure to include: your name, the date, and the name of the file in a banner comment at the beginning of each file you submit.
8. Files to download: none
9. Files to submit: postfixCalculator.h, postfixCalculator.cpp, testPostfixCalc.cpp, unix.questions.txt

### In-lab

1. Come to class with a *working prelab*.
2. Run your postfix calculator on the test sequences posted on the board by the TAs. Since your code only can handle hard-coded values, this will require a code modification and a recompilation to test each case. If your program does not calculate the correct result, use the debugger to find the errors and correct them. These modifications will be submitted to the in-lab.
    - Be sure you are able to explain how all parts your code work. You will be responsible for this material for the midterms and final exam.

3. You need to expand your pre-lab code to handle keyboard input. See the specifications in the in-lab section for how to handle the input.
4. The files you submit should be a FULLY WORKING postfix calculator.
5. Start working on the post-lab (implementing your own stack class) if you get your calculator fully working before lab ends.
6. Files to download: none (just your pre-lab source code)
7. Files to submit: postfixCalculator.h, postfixCalculator.cpp, testPostfixCalc.cpp

### Post-lab

1. Implement a stack class (into files stack.h and stack.cpp). **You can NOT use an STL container class for this** (list, vector, stack, etc.) for this, but you can use the STL string class. You should use either your List class from the last lab (if it works), or write up new stack class based on either the lecture notes or the textbook pages on stacks. Note that your stack class can contain a LinkedList object, and a stack class method can just pass the value onto the appropriate method in the LinkedList class. You don't need to implement all possible stack methods (in particular, you can ignore the copy constructor, operator=(), etc.) – just the four mentioned in the pre-lab (push, top, pop, and empty). After this lab, it is expected that you will be able to implement a stack class in C++.
2. Modify your postfix calculator to use the stack class that you have implemented.
3. Be sure to include: your name, the date, and the name of the file in a banner comment at the beginning of each file you submit. Your submission must contain the following code:
    1. Your stack code. This will likely be stack.h/cpp, and may (or may not; your choice) include all of the List.h/cpp, ListItr.h/cpp, ListNode.h/cpp files from lab 2
        - For your stack code, you are welcome to submit it in many files, as long as it will compile with 'clang++ *.cpp', and as long as the total number of files submitted does not exceed 11 files (you can submit 12 files total, but you need to submit a text file, described below, as well)
    2. A listing of your in-lab calculator code and your calculator test code: postfixCalculator.h/cpp, testPostfixCalc.cpp
4. Submit, in addition to your code, a paragraph (in a file called difficulties.txt) describing what difficulties you encountered getting your code working and what you did to solve them.
5. The files you submit should be a FULLY WORKING postfix calculator. Your code must compile! Even if it doesn't work perfectly, make sure it compiles. In particular, make sure that the capitalization case of the #includes (i.e. #include "Stack.h" versus #include "stack.h") is correct.
6. Files to download: none (just your in-lab source code)
7. Files to submit: stack.h, stack.cpp, postfixCalculator.h, postfixCalculator.cpp, testPostfixCalc.cpp, difficulties.txt
    - You may submit additional stack/list files as well, if you want

---

## Pre-lab

### Code Description

In this lab, you will:

- Implement a stack class that handles a stack of integer numbers. This stack implementation is done for the post-lab; for the pre-lab and the in-lab, you will be using a pre-existing stack class from C++'s standard template library (STL).
- Write a program that uses this class to implement a postfix calculator. This will include the following files:
    - postfixCalculator.h, which is the class declaration of the calculator
    - postfixCalculator.cpp, which is the implementation of the postfix calculator
    - testPostfixCalc.cpp that has a hard-coded expression (see below) and evaluates that expression.

The various parts of the lab develop the entire program:

- The pre-lab develops the calculator itself, without dealing with user input or the stack class
- The in-lab develops the user input routines
- The post-lab develops the stack class that your calculator uses

Note that the program is fully able to be run after each lab portion.

### Stacks

A stack is a container that implements the LIFO (last in, first out) property. Often it internally uses a linked list, array, vector, or a doubly-linked list to contain the elements. In general, a stack needs to implement the following interface and functionality:

- void push(int e): This adds the passed element to the top of the stack.
- int top(): This returns the element on the top of the stack. It does not remove that element from the top. If the stack is empty, then somehow an error must be indicated– printing an error message and exiting is fine for reporting errors for this lab.
- void pop(): This removes the element on the top of the stack, but does not return it. If the stack is empty, then somehow an

error must be indicated – for this lab, you can just print out an error message and then exit.
- bool empty(): This tells whether there are any elements left in the stack (false) or not (true).

Often, the top() and pop() functionality are joined as an int pop() function, but in this lab, it is beneficial to separate them.

For this lab, you must implement the stack so there is no maximum capacity! For now if pop() or top() are called on an empty stack, terminate the program with the function call exit(-1), which is from the <cstdlib> library.

For this lab, you will use a stack of int values.

## Input

For this part of the lab, you will not deal with keyboard input (that's in the in-lab) – thus, your submitted program will always compute the exact same value each time it is run. You will need to hard-code, into the main() method, the values to be operated on by your calculator.

A sample main() method that might work is as follows – this should be modified for your particular situation (i.e. how you declare your class, your method names, etc.). This main() method uses the first sample input given at the very end of this document.

```
int main() {
        PostfixCalculator p;
        p.pushNum (1);
        p.pushNum (2);
        p.pushNum (3);
        p.pushNum (4);
        p.pushNum (5);
        p.add();
        p.add();
        p.add();
        p.add();
        cout << "Result is: " << p.getTopValue() << endl;
}
```

Keep in mind that you can type up a few of the blocks, and comment them out with the /* … */ comment syntax that you are familiar with from Java – this will allow you to easily switch between the different hard-coded input test cases.

## Stack Calculator Implementation:

Your calculator must implement the following arithmetic operations: ~, +, -, *, and /. The tilde (~) is the unary negation operator – it negates the top element of the stack, and (unlike the other four operators) does not use a second number from the stack. Note that negative numbers still use a regular minus sign (i.e. '-3') – this just involves putting the negative number on the stack. But if you want to do negation (which involves popping the top value, negating it, and pushing that new value back on the stack), then you would use the tilde. For the non-commutative operators (operators where the order of the numbers matters, such as minus and divide), the first value you pop we'll call x, the second value you pop we'll call y; the result should be y-x, NOT x-y (i.e., the "lower" one in the stack minus/divided by the "higher" one in the stack).

## Useful Information

Postfix notation (also known as reverse Polish notation) involves writing the operators after the operands. Note how parentheses are unnecessary in postfix notation.

- Infix: ( 3 + 6 ) - ( 8 / 4 )
- Postfix: 3 6 + 8 4 / -

An online description of postfix calculators can be found on Wikipedia: http://en.wikipedia.org/wiki/Reverse_Polish_notation - note that you do NOT need to print out the infix form of the postfix expression; you only need to print the final answer. See the last page of this lab for example input streams and expected output.

When you start handling input (in the in-lab), you will want to store your read-in values into strings. You can use the string compare() method to compare them, but realize that it returns 0 if the are *equal*, and non-zero if they are not equal.

If you want to see some quick code for converting a string to an int, see the StringToInt() function at the bottom of this page. Warning: just copying that function without understanding it will only make your life more difficult.

### Hints

In the past, students have run into a few problems with this lab. We list them here in an effort to prevent these particular problems from being encountered again.

- When compiling your code, remember to compile ALL of your cpp files in the compile command:

```
clang++ postfixCalculator.cpp, testPostfixCalc.cpp
```

- Remember to put using namespace std; at the top of EACH file you write. Even if you don't use anything from the standard namespace, putting that at the top of the file will not hurt.

---

## In-lab

The purpose of the in-lab is first to ensure that your pre-lab code (the postfix calculator) is working properly. Then, you will need to add keyboard input to your lab. For the post-lab, you will be implementing your own stack. It will be much easier to debug if you know that your calculator code works – then, you'll know that your bugs (if any) are in your input routines or your stack code.

If you finish your in-lab before the end of the lab session, start working on your post-lab.

### Input

For your keyboard input, your program should read in just a single line. You should read this in using STL strings (if you are looking at building a tokenizer, then you are making it much more difficult than it need be). Once you encounter the end of a line, you can assume that there is no more input to read in. Your program should read in a single line, compute the result, and exit (i.e. don't prompt the user for more input). When processing input, you can't know before you read something if it will be an operand (a numeric value) or an operator (a character), so you must read in each space-separated part into a string variable, and analyze that.

All input is read from standard input (i.e. cin)! You should not be dealing with files for this lab. You can use the getline() method in the cin object for this. Once you read in a line, your program should exit. When we test your program, we will only be providing it with one line of input, so if your program is waiting for more, that will be a problem.

You need to accept both negative numbers (-5 for example), and numbers with multiple digits (34 is the number thirty-four, not the separate numbers three and four).

We provide you with a number of input files that match the input shown at the end of this lab document. Recall that you can supply the contents of a file as input to your program (as described in the Unix tutorial):

```
./a.out < addition.txt
```

### Reading in Tokens

A token is a single 'thing' passed to the postfix calculator. It can be an operator or a number, but is always separated by spaces. Thus, it is an entire number that is passed to the calculator, and not part of a number. The following code will read in the tokens for this program.

```
while(true) {
        string s;
        cin >> s;
        if(s == "")
                break;
        cout << s << endl;
}
```

Each string s that is read in must then be processed to determine if it's a number or an operator. The difficult part is if a minus sign is the first character of the token – it could be a subtraction sign or the beginning of a negative number (recall that the unary negation operator is the tilde).

You may find it useful to use the isdigit() or atoi() functions provided in <cstdlib> in this lab. Try searching on the web for info on these routines. The atoi() function operates on a C-style string, which is an array of characters. You can convert a C++ string to one of these by calling the c_str() member function on the C++ string object. More string functions can be found at [http://www.sgi.com/tech/stl/](http://www.sgi.com/tech/stl/).

Also, to check if there is any more input, you can use the good() method in cin.

### Assumptions:

1. Assume that the input, i.e. the postfix expression, is entered in on one line and that all numbers and operators are separated by a single space. We will only provide you with valid input.
2. You can assume that users will enter the proper number of operands/operators. In other words, if an invalid postfix expression is entered, your program can do anything (including crashing) and we won't take off points.

### Terminating Input

How should the program know when you are finished providing input? There are a couple of ways to do this.

- Only read in one line, and not accept any more input
- Read in input until cin.good() method returns false; **this will require entering a Control-D at the end of the provided input.**

Either way is fine. Our test scripts will send in all the input on a single line, and will provide a Control-D if necessary. So whichever means you use to determine the end of your input is fine.

---

## Post-lab

For the post-lab, you will be implementing your own stack. This can be code that you write yourself, or you can re-use your List code from lab 2 (make sure it works before you re-use it, though!).

You will also have to write up the difficulties.txt file, as described above in the lab procedure section.

Note that you only have to implement the four stack methods described in the pre-lab section (and the constructor, of course): push(), pop(), top(), and empty(). The other methods (copy constructor, operator=(), etc.) do not need to be implemented for this lab.

If you are using an array-based implementation, you must be able to handle when the array fills up; you can't use the vector class to handle this.

### Submitting the stack / list files

Depending on how you implement the stack class, you may just need the stack.h/cpp files, in addition to the three postfix calculator files (postfixCalculator.h/cpp and testPostfixCalc.cpp). Or you may need stack.h/cpp and stacknode.h/cpp in addition to the three postfix calculator files. Or you may want to include the six List/ListItr/ListNode files from lab 2 as well as stack.h/cpp and the three postfix calculator files. How you do this is up to you - as long as it works, we don't really care, provided that:

1. It compiles with 'clang++ *.cpp'
2. The total number of C++ files you submit is 11 or fewer (you can submit 12 files total, but you need to submit a the text file called difficulties.txt as well)

---

## Test files

These files are all available in Collab.

[addition.txt](addition.txt)

```
1 2 3 4 5 + + + +
```

Expected Final Output from addition.txt: 15

[subtraction.txt](subtraction.txt)

```
20 10 - -3 10 - - 2 -
```

Expected Final Output from subtraction.txt: 21

multiplication.txt

```
-1 -2 -5 3 * 2 -2 * * * *
```

Expected Final Output from multiplication.txt: 120

division.txt

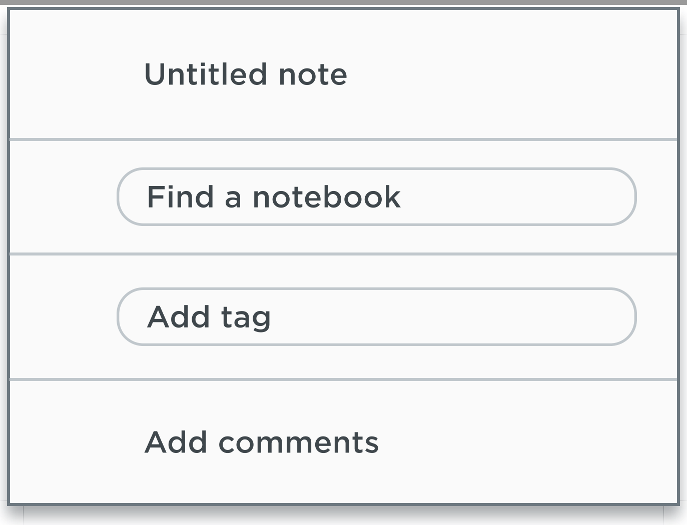
```
-1512 -12 -2 / / -2 / 3 /
```

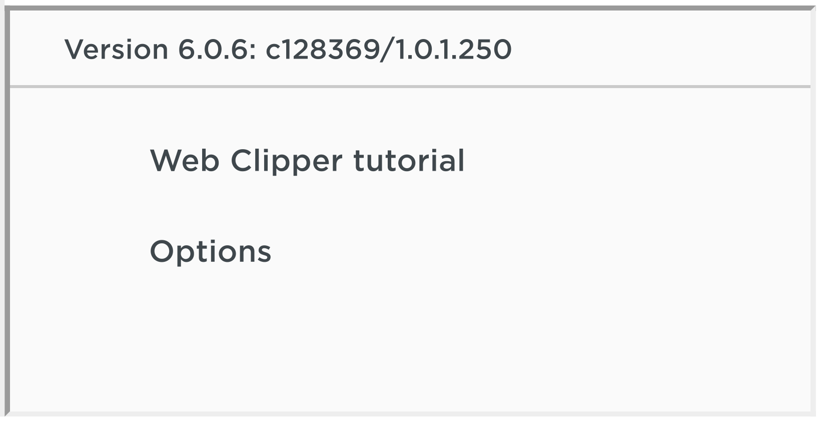
Expected Final Output from division.txt: 42

negation.txt

```
-1 ~ ~ ~
```

Expected Final Output from negation.txt: 1




Version 6.0.6: c128369/1.0.1.250