COP4530 – Project 3

Stack and Its Applications

Submission:

1. Create a tar file with the required files.
2. Make an electronic submission of your tarball file on Canvas.

Due Date: Oct 27, 2019, 11:50pm for all students
Late Penalty: maximum one day at the penalty of 15%.

1. Overview

In this project, you shall implement a generic stack container as an adaptor class template, and then implement two programs that are representative applications of Stack. One program converts infix expression to postfix expression. The other program evaluates postfix expression using the stack container.

The objectives of this project include the following:

- Understand the stack ADT and its applications.
- Understand infix to postfix conversion and postfix expression evaluation.

2. Description of Provided Source Files

A tar file `proj3src.tgz` is available on Canvas. To extract these files from the tar file `proj3src.tgz`, you may use the following command **on the `linprog` machines, where your code will be tested and graded**.

- `tar zxf proj3src.tgz`

This project includes two sets of documents. One is essential for the project completion. The other set is provided to encourage the use of the `flex` tool for lexical analysis in this project.

Essential files

You can three essential documents as described below. These provided documents provide a good set of sample tests. You are welcome to come up with more test cases, for testing Stack features and for testing your Infix-To-Postfix program.

- `test_stack1.cpp:` A sample test driver for the Stack class.
- `testcases:` A directory with 10 different test cases (`test[0-9].txt`) for your in2post program. The expected output running your in2post program with these test files are also provided in the file `outputs.txt`.
- `in2post:` It is an example test program. You can run it with any of the test files. For example, you can run with the command below.

```
in2post test2.txt
```

You are not supposed to include these files in your final submission. Thus any change you make to these files will not be graded.

Sample token generation using flex

Throughout your undergraduate program, you will have, or may already have had, multiple needs to parse an input file and generate many different operators, keywords, numerical values and identifiers. Three files are provided here to demonstrate the generation of operators, numerical values for this project.

- `opnum.h:` A header file with the definition of some constants and the declaration of utility functions for flex-based token generation.
- `opnum.fl:` The main file defines how to match the tokens from an input file.
- `test_flex.cpp:` A sample driver program that takes an input file and print out them separately line by line.
- `Makefile:` A sample Makefile that allows you to compile test_flex.cpp for the driver program test_flex. You can run the test program using commands below.
    - `make test_flex`
    - `test_flex test2.txt`

You will need to make some minor modifications so that these utility files can support your in2post program correctly. If your in2post program is dependent on your modifications on these files, please include your modified files as part of your project 3 submission.

3. Project Description

A Stack is a type of data container/ data structure that implements the LAST-IN-FIRST-OUT (LIFO) strategy for inserting and recovering data. This is a very useful strategy, related to many types of natural programming tasks as we have discussed in class.

Remember that in the generic programming paradigm, every data structure is supposed to provide encapsulation of the data collection, enabling the programmer to interact with the entire data structure in a meaningful way as a container of data. By freeing the programmer from having to know its implementation details and only exporting the interface of its efficient operations, a generic Stack provides separation of data access/manipulation from internal data representation. Programs that access the generic Stack only through the interface can be re-used with any other Stack implementation. This results in modular programs with clear functionality and that are more manageable.

A. Goals:

- Implement a generic Stack as an adaptor class template
- Write a program that parses infix arithmetic expressions to postfix arithmetic expressions using a Stack
- Write a program that evaluates postfix arithmetic expressions using a Stack

More detailed descriptions for each of the above tasks are provided in the rest of this document.

B. Task1: Implement Stack adaptor class template:

- Stack MUST store elements internally using a proper C++ STL container (of course, you cannot use the C++ STL stack container).
- Your Stack implementation MUST:
    - be able to store elements of an arbitrary type.
    - Every Stack instance MUST accept insertions as long as the system has enough free memory.
- Stack MUST implement the full interface specified below
- You MUST provide the template and the implementation in two different files stack.h and stack.hpp, respectively. You must include stack.hpp into stack.h
- Your stack MUST be in the namespace cop4530 (see Listh.h in Project 2 for an example).

Stack Interface:

The interface of the Stack class template is specified below.

- explicit Stack(int initSize = 0): zero-argument constructor.
- ~Stack (): destructor.
- Stack (const Stack<T>&): copy constructor.
- **Stack(Stack<T> &&)**: move constructor.
- Stack<T>& operator= (const Stack <T>&): copy assignment operator=.
- **Stack<T> & operator=(Stack<T> &&)**: move assignment operator=
- bool empty() const: returns true if the Stack contains no elements, and false otherwise.
- void clear(): delete all elements from the stack.
- int size() const: returns the number of elements stored in the Stack.
- void push(const T& x): adds x to the Stack. copy version.
- **void push(T && x)**: adds x to the Stack. move version.
- void pop(): removes and discards the most recently added element of the Stack.
- T& top(): returns a reference to the most recently added element of the Stack (as a modifiable L-value).
- const T& top() const: accessor that returns the most recently added element of the Stack (as a const reference).
- void print(std::ostream& os, char ofc = ' ') const: print elements of Stack to ostream os. ofc is the separator between elements in the stack when they are printed out. Note that print() prints elements in the opposite order of the Stack (that is, the oldest element should be printed first).

The following non-member functions should also be supported.

- std::ostream& operator<< (std::ostream& os, const Stack<T>& a): invokes the print() method to print the Stack<T> **a** in the specified ostream
- bool operator== (const Stack<T>&, const Stack <T>&): returns true if the two compared Stacks have the same elements, in the same order, and false otherwise
- bool operator!= (const Stack<T>&, const Stack <T>&): opposite of operator==().
- bool operator<= (const Stack<T>& a, const Stack <T>& b): returns true if every element in Stack a is smaller than or equal to the corresponding element of Stack b, i.e., if repeatedly invoking top() and pop() on both **a** and **b**, we will generate a sequence of elements a_i from a and b_i from b, and for every i, $a_i \leq b_i$, until **a** is empty.

C. Task2: Convert infix arithmetic expressions into postfix arithmetic expressions and evaluate them (whenever possible)

For the sake of this exercise, an arithmetic expression is a sequence of space-separated strings. Each string can represent an operand, an operator, or parentheses.

Operands: can be either a numerical value or a variable. A variable name only consists of alphanumerical letters and the underscore letter "_". A variable name starts with a English letter. Numerical operands can be either integer or floating point values. Two simplifications on operand calculations:

1) No signs ('+' or '–') are used for operands.

2) No scientific notations (1.23e4 or 1.23E4) are used for numerical operators.

Examples of operands: `"34", "5", "5.3", "a", "ab", "b1", and "a_1"`

Operators: one of the characters `'+', '-', '*', or '/'`. As usual, '*' and '/' are regarded as having higher precedence than '+' or '-'. Note that all supported operators are binary, that is, they require two operands.

Parentheses: `'(' or ')'`. No empty pairs of parentheses `()` are allows.

An Infix arithmetic expression is the most common form of arithmetic expression used.

Examples:

- (5 + 3) * 12 - 7 is an infix arithmetic expression that evaluates to 89
- 5 + 3 * 12 - 7 is an infix arithmetic expression that evaluates to 34

For the sake of comparison, postfix arithmetic expressions (also known as `reverse Polish notation`) equivalent to the above examples are:

- 5 3 + 12 * 7 -
- 5 3 12 * + 7 -

Two characteristics of the Postfix notation are (1) any operator, such as '+' or '/' is applied to the two prior operand values, and (2) it does not require the use of parenthesis.

More examples:

- `a + b1 * c + ( dd * e + f ) * G` in Infix notation becomes
- `a b1 c * + dd e * f + G * +` in Postfix notation

To implement infix to postfix conversion with a stack, one parses the expression as sequence of space-separated strings. When an operand is read in the input, it is immediately output. Operators (e.g., '-', '*') may have to be saved by placement in an operator stack. We also stack left parentheses. Start with an initially empty operator stack.

Follow these 4 rules for processing operators/parentheses:

1) If input symbol is '(', push it into stack.
2) If input operator is '+', '-', '*', or '/', repeatedly print the top of the stack to the output and pop the stack until the stack is either (i) empty ; (ii) a '(' is at the top of the stack; or (iii) a lower-precedence operator is at the top of the stack. Then push the input operator into the stack.
3) If input operator is ')' and the last input processed was an operator, report an error. Otherwise, repeatedly print the top of the stack to the output and pop the stack until a '(' is at the top of the stack. Then pop the stack discarding the parenthesis. If the stack is emptied without a '(' being found, report error.
4) If the end of input is reached and the last input processed was an operator or '(', report an error. Otherwise print the top of the stack to the output and pop the stack until the stack is empty. If an '(' is found in the stack during this process, report error.

For more details on how the conversion works, look up the lecture notes and Section 3.6 of the textbook.

Evaluating postfix arithmetic expressions

After converting a given expression in infix notation to postfix notation, you will evaluate the resulting arithmetic expression IF all the operands are numeric (int, float, etc.) values. Otherwise, if the resulting postfix expression contains characters, your output should be the same as the input (the postfix expression).

Example inputs:

```
5 3 + 12 * 7 -
5 3 12 * + 7 -
1 3 + 5 * c - 10 /
```

Example outputs:

```
89
34
1 3 + 5 * c - 10 /
```

To achieve this, you will have an operand stack, initially empty. Assume that the expression contains only numeric operands (no variable names). Operands are pushed into the stack as they are ready from the input. When an operator is read from the input, remove the two values on the top of the stack, apply the operator to them, and push the result onto the stack. If an operator is read and the stack has fewer than two elements in it, report an error. If end of input is reached and the stack has more than one operand left in it, report an error. If end of input is reached and the stack has exactly one operand in it, print that as the final result, or 0 if the stack is empty.

Summarizing task 2.

Your program should expect as input from (possibly re-directed) stdin a series of space-separated strings. If you read a1 (no space) this is the name of the variable a1 and not "a" followed by "1". Similarly, if you read "bb 12", this is a variable "bb" followed by the number "12" and not "b" ,"b", "12" or "bb", "1" ,"2". The resulting postfix expression should be printed to stdout.

Your program should evaluate the computed postfix expressions that contain only numeric operands, using the above algorithm, and print the results to stdout.

4. Restrictions

- Your program MUST be able to produce floating number evaluation (i.e., deal with floats correctly).
- Your program MUST NOT attempt to evaluate postfix expressions containing variable names. It should print the postfix-converted result to `stdout` and MAY NOT throw an exception nor reach a runtime error in that case.
- Your program MUST check invalid infix expressions and report errors. For example, the following types of infix expressions are invalid expressions: 1) an operator does not have the corresponding operands, 2) an operand does not have the corresponding operator; or) mismatched parentheses or an empty pair of parentheses. These checks can be performed either in the expression conversion or in the postfix evaluation. Note that the errors are not limited to these three. You shall run the provided program again the test cases for more coverage.
- No need to consider an expression across multiple lines. But your program must be able to process every lins in the input before terminating (check the provided executable in2post to see the behavior of the program).

5. Programs and files you need to submit.

If you do not use `flex` for parsing the input, you only need to submit a total of four files for the completion of this assignment: `stack.h`, `stack.hpp`, `in2post.cpp`, and `Makefile`. Create a tarball file from these four files using the command below, where `fsuid` is your FSU ID.

- `tar zcf <fsuid>_proj3.tgz Makefile in2post.cpp stack.h stack.hpp`

If you do use `flex` for parsing the input, you will need to include two more files, i.e., the `opnum.h and opnum.fl` files with your modifications. Of course, your Makefile should include the command(s) that use these extra files to generate the program `in2post`. Create a tarball file from these six files using the command below (all characters in one line), where `fsuid` is your FSU ID.

- `tar zcf <fsuid>_proj3.tgz Makefile in2post.cpp stack.h stack.hpp opnum.h opnum.fl`

6. General Requirements

- Document your code appropriately so that it is readable and easy to navigate
- You shall NOT use the Stack template library from the STL for this project.
- Make sure your implementation compile and run on linprog.cs.fsu.edu with g++, using the C++11 standard compilation flag. Your Makefile should function in a similar manner to produce the driver programs, as shown in the file `testcases/outputs.txt`.

7. Breakdown of points

The grade breakdown of points for these assessments is provided as follows:

Points	Requirements
15	Have completed the required functions in stack.h and stack.hpp in a reasonably correct manner. Particularly the driver programs can compile and execute.
5	The submitted files in2post.cpp can compile and produce a driver program for the provided test via a modified Makefile.
25	Correct implementation and successful passing of test_stack1.cpp.
25	Correct implementation and successful passing of in2post using test0.txt
20	Correct implementation and successful passing of all other test cases in test files test[1-10].txt. 2 points per test case.
10	Error messages and output are well formatted, similar to the content in outputs.txt.
Bonus: 20	Have used the flex tool for parsing the input and the driver program works correctly on top of the modified flex tool.

8. Miscellaneous

The first person to report any compilation or execution errors in the provided materials will be given 3% extra credit (maximum 15% per person). Automatic plagiarism detection software will be used on all submissions – any cases detected will result in a grade of 0 for those involved.