Project 3: Stack and Its Applications

Due: 3/1/2018

Educational Objectives: Understand the stack ADT and its applications. Understand infix to postfix conversion and postfix expression evaluation.

Statement of Work: Implement a generic stack container as an adaptor class template. Implement a program that converts infix expression to postfix expression and implement a program that evaluates postfix expression using the stack container you develop.

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.

Goals:

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

More detailed descriptions for each of the above tasks are now provided.

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 implementation MUST be in the namespace cop4530.

Interface:

The interface of the **Stack** class template is specified below.

Stack(): 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.

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(): mutator that returns a reference to the most recently added element of the **Stack**.

const T& top() const: accessor that returns the most recently added element of the **Stack**.

int size() const: returns the number of elements stored in the **Stack**.

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 global 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 **Stack**s have the same elements, in the same order.

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 **Statck 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 <= b_i, until **a** is empty.

Analyzing the worst-case run-time complexity of the member function clear() of Stack. Give the complexity in the form of Big-O. Your analysis can be informal; however, must be clearly understandable by others. Note that the time complexity of the function clear() depends on the underlying adaptee class you use for the implementation of Stack. Name the file containing the complexity analysis as "analysis.txt", in which, you should state the time complexity of clear() in the Big-O notation, your discussions of why it is so, in particular, you need to include the information on the underlying adaptee class you used in the implementation of the Stack.

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.

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 ")"

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 (i.e., "-", "*") 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 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 –

3 5 * c – 10 /

Example outputs:

89

34

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.

For more information on the evaluation of postfix notation arithmetic expressions, look up the lecture notes and Section 3.6 of the textbook.

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.

Restrictions

The infix to postfix conversion MUST be able to convert expressions containing both numbers and variable names.

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 for mismatched parentheses (this should be taken care of if you infix to postifx expression conversion is performed properly.

Your program MUST check invalid infix expressions and report errors. We consider the following types of infix expressions as invalid expressions: 1) an operator does not have the corresponding operands, 2) an operand does not have the corresponding operator; or) mismatched parentheses. Note that some checks can be performed in the expression conversion or postfix evaluation.

Your program MUST re-prompt the user for the next infix expression. Your program must be able to process several inputs before terminating (check the provided executable in2post.x to see the behavior of the program).

Deliverable Requirements

Your implementation should be contained in three files, which MUST be named stack.h, stack.hpp and in2post.cpp. Submit your implementation in a tar file including the three files (stack.h, stack.hpp, and in2post.cpp), the makefile you use, and the analysis.txt file for time complexity of the clear() function of Stack.

Sample Executable Code

You can download the tar file containing a sample executable code of the project (compiled on linprog) and a test driver for Stack. There files are included in the tar file:

1. in2post.x: executable program of the project
2. test_stack1.cpp: source code of the test driver
3. test_stack1.x: executable code of the test driver
4. test0.txt: example test cases for in2post.x

Note

This is our first capstone assignment. Make sure you do well for this project in order to pass this class. As discussed at the beginning of this course, failing a capstone assignment automatically means a fail of the class.