Project 3: Stack and Its Applications

Due: 10/22/2007

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. 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. For instance:

- Keeping track of nested invocation calls in a procedural programming language, such as C/C++.
  - Each function call results in a new entry being placed into the program run-time stack. This new entry contains memory space for local variables (which can grow dynamically) and for a return pointer to the instruction in the function that invoked the current function (caller/callee). As functions terminate, their stack entry is "popped out," with the return values written to the proper location in the caller.
  - Since nested procedural/ function invocation levels are entered and exited in LIFO order, a stack is the most appropriate data structure to handle this functionality.
- Evaluating arithmetic expressions.
  - Stacks can be used to parse arithmetic expressions and evaluate them efficiently, as we shall see as part of this assignment.
- To eliminate the need for direct implementation of recursion.
  - As recursive function calls require a lot of overhead, it is often the case that recursive algorithms are "unrolled" into non-recursive ones. Since recursive calls are entered/exited in LIFO order the use of stacks to mimic recursion is a natural choice.

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 only 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
2. Write a program that parses Infix artithmetic 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 a GenericStack:

- **GenericStack** MUST store elements internally in a **BasicContainer** developed in project 2.
- It should use only the public interface to **BasicContainer**, as we will compile/link your **GenericStack** implementation with our **BasicContainer** class.
- Therefore, in particular **GenericStack** MUST:

- be able to store elements of an arbitrary type.
- have a no-argument constructor that initializes it to some size. This size MUST NOT be so large as to prevent a calling program to initialize several **GenericStack** instances in a single execution. Suggested length values are 10, 16, or 256.
- Every **GenericStack** instance holding *n* elements MUST accept insertions as long as the system has enough free memory to dynamically allocate an array of *2n+1* elements at insertion request time, regardless of the initial capacity of the **GenericStack**. If the system does not have such free memory, **GenericStack** MUST report an error message as the result of the insertion request.

- **GenericStack** MUST implement the full interface specified below
- You MUST provide both the template (in a class named GenericStack.h) and the implementation (in a class named GenericStack.cpp).

Interface:

The interface of **GenericStack** is specified below. It provides the following public functionality, all of which can be efficiently achieved using an internal array representation for the data.

GenericStack(): no-argument constructor. Initializes the Stack to some pre-specified capacity.

~GenericStack (): destructor.

GenericStack (const GenericStack <T>&): copy constructor.

GenericStack<T>& operator= (const GenericStack <T>&): assignment operator

bool empty() const: returns true if the **GenericStack** contains no elements, and false otherwise.

void push(const T& x): adds x to this **GenericStack**.

void pop(): removes and discards the most recently added element of the **GenericStack**.

T& top(): mutator that returns a reference to the most recently added element of the **GenericStack**.

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

int size(): returns the number of elements stored in this **GenericStack**.

void print(std::ostream& os, char ofc = '\0') const: print this instance of GenericStack to ostream os. **Note that gs.print() prints elements in opposite order as if bc.print() was invoked in the underlying BasicContainer.**

The following non-member functions should also be supported.

std::ostream& operator<< (std::ostream& os,const GenericStack<T>& a): invokes the **print()** method to print the **GenericStack<T>** a in the specified ostream

bool operator== (const GenericStack<T>&,const GenericStack <T>&): returns true if the two compared "**GenericStack**"s have the same elements, in the same order.

bool operator< (const GenericStack<T>& a, const GenericStack <T>& b): returns true if every element in **GenericStack a** is smaller than corresponding elements of **GenericStatck b**, i.e., if repeatedly invoking top() and pop() on both a and b 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.

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.

Examples of operands: "34", "5", "a", and "b1" (alphanumeric)

Operators: one of the characters "+", "-", "*", or "/". As usual, "/" and "*" are regarded as having higher precedence than "+" or "-"

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 ever 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 (i.e., an alphanumeric string) is read in the input, it is immediately output. Operators (i.e., "-", "*") may have to be saved by placement in the 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 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 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 equal to the input.

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 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".

Your program should convert all Infix expressions to Postfix expressions, including expressions that contain variable names. 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

1. You MAY NOT use any C++ STL classes in addition to the GenericStack, BasicContainer, and ANSI string class for all your data structures.

2. The Infix to Postfix conversion MUST be able to convert expressions containing both numbers and variable names (alphanumeric strings). Variable names may be arbitrary strings, such as "sci_fi_freak88"

3. You MAY use the "ctype.h" (alpha/digit check) and "stdlib.h" (alpha to digit/float conversion) library.

4. Their program MUST be able to produce floating number evaluation (i.e., deal with floats correctly).

5. 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.

6. Your MUST check for mismatched parentheses.

7. Their program MUST re-prompt the user for the next infix expression. Your program must be able to process several inputs at a time.

Deliverable Requirements

Your implementation should be contained in three files, which MUST be named GenericStack.h, GenericStack.cpp and in2post.cpp. Note that we will use our own implementation of BasicContainer.cpp to test your code. So make sure that all calls made by your GenericStack class to your BasicContainer class use the public interface of BasicContainer. Submit your implementation in a tar file including the three files (GenericStack.h, GenericStack.cpp, and in2post.cpp) and the makefile you use.

If your program does not run in the same way as the sample executable code provided (below), you also need to write a README file to inform our TA how to run your program. This README file should also be included in the tar file.

Sample Executable Code

You can download the executable code of an implementation of the project (compiled on Linux).