National University of Singapore
School of Continuing and Lifelong Education

TIC1002: Introduction to Computing and Programming II
Semester II, 2020/2021

Problem Set 3
Calculating with Stacks

Release date: 15th March 2021, 12:00 mn
Due: 4th April 2021, 11:59 pm

Stack ADT

The C++ Standard Template Library comes with a stack ADT. The member functions of the stack ADT are as follows:

- `stack<T> s` - Creates a stack s for elements of type T.
- `void s.push(T item)` - Adds item into stack s.
- `void s.pop()` - Removes an item from stack s.
- `T s.top()` - Returns the item at the top of stack s.
- `bool s.empty()` - Returns `true` if stack s is empty.

We encourage you to learn to use the STL stack for this problem set. It is very similar to the homemade stack discussed in lecture. The differences to note are i) the functions come behind the variable, ii) `s.pop()` does not return the item that is popped, and iii) `s.top()` is used to examine the top, and not `peek()`. To use the STL stack ADT, you will put `#include <stack>` at the top of your file.

However, if you wish to use the homemade stack ADT discussed in lecture, it is supplied in Coursemology. You will put `#include "stack.cpp"` at the top of your file instead.

Introduction

For this problem set, we will be coding a calculator that can evaluate a simple mathematical expression.

To keep things simple, our calculator will only **deal with non-negative integers**, and support the following operators:

Precedence	Operator	Description
1	`^`	Exponent. `x ^ y` means x to the power of y.
2	`* / %`	Multiplication, integer division, and modulo (remainder).
3	`+ -`	Addition and subtraction.

The lower the number, the higher the order of precedence of the operator. Operators with the same level of precedence will be evaluated left to right.

Task 1: Bracket Matching (5 marks)

Our mathematical expression can also include brackets `()` `{}` `[]` to denote the order of operation. For the expression to be correct, every open bracket must be matched to the same closing pair.

Deliverable: `bool match_bracket(string expression)`
The function takes a mathematical expression as a string, and returns true if the brackets match, and false otherwise.

Hint: You may assume that the expression contains only valid characters.

Task 2: Token ADT (5 marks)

It is quite tedious to deal with strings and characters, when what we are interested in are operators and operands. Thus, it will be easier if we can simply work with a vector of operators/operands.

However, C++ requires that a vector contains elements of the same type, and does not allow mixing of types. No worries, one way to get around this is to define our own Token type.

Our Token type can either contain an operator (which is a `char`) or an operand (which is an `int`). We will define the following functions to support our Token ADT:

Deliverable:

- `struct Token { ... };` a structure to represent a `Token`.
- `Token make_token(char optr)` returns a `Token` that contains the given operator.
- `Token make_token(int opnd)` returns a `Token` that contains the given operand.
- `char get_optr(Token t)` returns the `char` that `Token t` represents.
- `int get_opnd(Token t)` returns the `int` that `Token t` represents.
- `bool is_opnd(Token t)` returns true if `Token t` represents an operand, and false if it represents an operator.

Decide on how to structure the `Token` struct and provide an implementation of Token and all the supporting functions.

Task 3: Infix to Postfix (10 marks)

The traditional way of writing mathematical expressions is infix notation, with the operator in between the operands. The drawback of this notation is that brackets have to be used to specify certain order of operations.

There are better notations known as prefix and postfix, or Polish Notation and Reverse Polish Notation. With prefix the operator is placed before the operands and with postfix the operator is placed after the operands. With these notations, the order of operation is unambiguous and no brackets are needed.

For example:

Infix: `(1 + 2) * 3 ^ 4`

Prefix: `* + 1 2 ^ 3 4`

Postfix: `1 2 + 3 4 ^ *`

In addition to being unambiguous, it is also easier to evaluate an expression in postfix notation. Thus, the first thing we will do is convert an infix expression to postfix notation.

To convert infix to postfix, we make use of a stack and begin reading the infix expression token by token and applying the following rules:

1. If the token is an operand, i.e. an integer, output it directly.
2. If the token is an open bracket, push it onto the stack. For simplicity, we assume that only rounds brackets ( ) are used.
3. If the token is an operator, then push it onto the stack only if the operator on top of the stack has lower precedence, or is a bracket. Otherwise, pop and output operators off the stack until you can push.
4. If the token is a close bracket, pop and output operators off the stack until you reach an open bracket. Then just pop and discard the open bracket.

Once the end of the expression is reached, simply pop and output everything on the stack.

Our function will take in a vector of `Token` as an input, with element being a token. The function `tokenize` is given and it converts a string expression to a vector of `Token`s.

Deliverable: `vector<Token> int_to_post(vector<Token> infix)`
The function takes an infix mathematical expression as a vector, and returns a vector with the tokens in the respective postfix notation.

Hint: You might want to try out the algorithm with some examples by hand first to understand how a stack is being used here.

Task 4: Evaluating the Expression (10 marks)

Now that we have the postfix expression, we can begin evaluating it. As before, we will make use of a stack, but this time we will be putting operands in it.

The rules when reading the tokens from the vector is as follow:

1. If the token is an operand, push it onto the stack.
2. If the token is an operator, pop two operands from the stack and apply the operator, and push the result back onto the stack.

Once the end of the expression is reached, the stack should contain the result.

Deliverable: `int calculate(vector<Token> postfix)`
The function takes a postfix expression as a vector of tokens, and outputs the final result of the expression.