COSE213 Data Structures (Fall 2021)
Instructor: Prof. Won-Ki Jeong
Due date: Oct 24, 2021, 11:59 pm.

Assignment 2: Simple Calculator using Stacks (v1.1)

In this assignment, you will implement a simple text-based calculator using a stack. You are required to modify/complete two files: `stack.txx` and `calculator.h`.

1. Stack class implementation (30 pts)

In `stack.h`, the stack abstract data type is defined. You will find function names as well as data. Actual implementation of those functions should be in `stack.txx`. You need to implement the following functions:

- `~Stack()` : destructor
- `type& Top()` : return the top element in the stack. If the stack is empty, throw an exception and print out the message “Error: Queue is empty”.
- `void Push(const type& item)` : push an element to the stack
- `void Pop()` : delete the element at the top. If the stack is empty, throw an exception print out the message “Error: Queue is empty”.
- `bool IsEmpty()` : return true if the stack does not contain any element.

You need to use a linear array to store elements. You also need to dynamically adjust the size of array as needed (initial capacity is 10).

Note that you need to modify `stack.txx`, and this file is included in `stack.h` because stack is a template class (i.e., the implementation must be inlined).

2. Simple text-based calculator (70 pts)

Once you implement the stack data structure, you can implement a simple text-based calculator using stacks. You need to implement `double Eval(char* in)` function in `calculator.h`. This function accepts a Cstring of an **infix** expression and returns the result. For example,

Input : 10 + (20 – (30+40))
Output : -40

The usage example of this function is given in main.cpp as follows:

```
char str1[] ="-10-((-2+(2+4*3))-12) + 122 * (123 +
(120+888) - 300)";
```

```
double res1 = Eval(str1);
```

One way to implement the calculator for infix expression is first converting infix notation to postfix notation, and evaluating the postfix expression using stacks. (as discussed in the class).

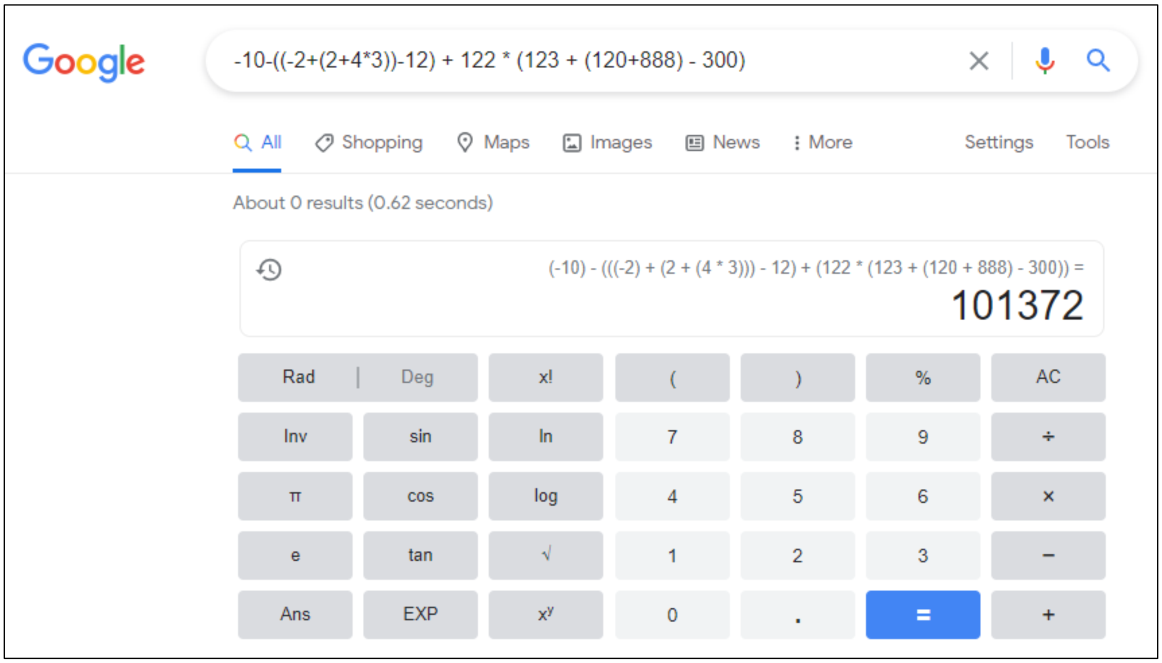
You can assume the following for the input expression:

1. There are **only five operators**, i.e., +, -, *, /, and unary minus.
2. There may be parenthesis in the expression, but the expression may **NOT** be fully parenthesized. For example, (2+3*5)+7 can be a valid input, although the fully-parenthesized expression is ((2+(3*5))+7).
3. Input strings may contain numbers, parenthesis, operators, and space **only**. Example: 10 + { 20 + 30 } is not allowed because { and } are not allowed to use.
4. You can assume that there will be **no** grammatical errors in the given expression. For example, 10 (20+30) is error because * between 10 and (is missing.

Make sure that unary minus operator works correctly. For example, -10+3, - in front to 10 is a unary operator (not a binary operator). In order to distinguish unary and binary operators, you need to check what is in front of -. If there is an operand or right parenthesis in front of -, it is a binary minus. Otherwise, - is a unary operator.

Examples:
10 – 2 : binary minus because 10 is in front of -.
(10 +5) – 8 : binary minus because) is in front of -.
10*5*-2 : unary minus because * is in front of -.
(-5+10)*2 : unary minus because (is in front of -.

Once you finish implementation, check the correctness of your code by comparing the solutions using google (if you cut-and-paste the expression into google search window then google will give you the solution. See below.)



-10-((-2+(2+4*3))-12) + 122 * (123 + (120+888) - 300)

About 0 results (0.62 seconds)

(-10) - (((-2) + (2 + (4 * 3))) - 12) + (122 * (123 + (120 + 888) - 300)) =

101372

I also provide an example parser in the skeleton code (in `calculator.h`). The given parser will parse the input string into numbers and operators, and push those into corresponding stacks. Feel free to modify this parser code for your own implementation.

3. Compile and submit

You must submit the following two files online via blackboard:

- calculator.h
- stack.txx

Note that you are not allowed to modify other files, such as stack.h and main.cpp.

You can compile the code as follows:

```
> make
```

To delete previously compiled code, do as follows:

```
> make  clean
```

The output executable name is assign_2. You can run your code by simply type in this name in the terminal.

```
> assign_2
```

Good luck, and ask TAs and professor if you have any question.