VG101 — Intoduction to Computers & Programming

Lab 8

Instructor: Manuel Charlemagne

TA: Yihao Liu – UM-JI (Summer 2018)

Goals of the lab

- Simple I/O in C++
- Parse a string with a loop
- Use Standard Template Library

1 Introduction

Haruka, Kana, and Chiaki are catching up the deadline of a project, but they find out that all of them have left their calculator at home.tor, If you can help them build a simple calculator they will survive from failing in this course!

2 Working Flow

2.1 Stack-based Calculator

Calculators in the computers are usually based on stacks. In most situations, you need two stacks: one for storing the operators and another one for storing the numbers.

In computer science, a stack is an abstract data type that serves as a collection of elements, with two principal operations:

- push, which adds an element to the collection, and
- pop, which removes the most recently added element that was not yet removed.

The order in which elements come off a stack gives rise to its alternative name, LIFO (last in, first out).

The reason why stacks are needed is that operators have their own priorities. For example, the expression `1+2*3-4` has three operators `+`, `*` and `-`, but they don't have the same priority in calculation, so you can't calculate them from left to right. Actually, `*` has the higher priority then other two operators, so `2*3` should be calculated first. Here we can use two stacks to simulate this procedure.

i) Push `1` into the number stack.

ii) The operator stack is empty, push `+` into the operator stack.

iii) Push `2` into the number stack.

iv) The top of the stack `+` has a lower priority than the next operator `*`, so push `*` into the operator stack.

v) Push `3` into the number stack.

vi) The top of the stack `*` has a higher priority than the next operator `-`, so a calculation is made. Pop `3` and `2` from the number stack and push `6` into the number stack. Then pop the operator `*` from the operator stack.

vii) The top of the stack `+` has a same priority as the next operator `-`, so again a calculation is made. Pop `1` and `6` from the number stack and push `7` into the number stack. Then pop `+` from the operator stack.

viii) The operator stack is empty, push `-` into the operator stack.

ix) Push `4` to the number stack.

x) The expression reaches the end, so the final calculations can be made.

xi) Pop `7` and `4` from the number stack and push `3` into the number stack. Then pop the operator `-` from the operator stack.

xii) The operator stack is empty again, and the result of the expression is the top of the number stack: `3`.

So the algorithm can be summarized as follows:

- Push any number you meet into the number stack.
- If the operator stack is empty, or the top of the operator stack has a lower priority than the next operator, push the next operator into the operator stack.
- While the top of the operator stack is an operator with priority no lower than the next operator, pop the top operator and two numbers from the stacks, perform the calculation, and push the calculation result back into the number stack. After that, push the next operator into the operator stack.
- If the expression reaches the end, it can be assumed that calculations can be performed one by one according to the stack order.

2.2 (Mandatory) A simple calculator

Your first task is to write a simple calculator according to the algorithm in the previous part.

You can take advantage of `std::string` and `std::stack` in your implementation.

The input is given by a string in one line, only with integers ranged in $[0, 9]$ and operators `+,-,*,/`. There may be spaces in any part of the input string.
Sample input: `1 + 2*3 - 4`

The output is the result of the input expression.
Sample output: `3`

2.3 (Optional) Using priority map to simplify implementation

Your second task is to add some more functions to the calculator:

- Add an operator ˆ (power), which has higher priority then `+,-,*,/`.

  Note: For the power operator, we have a different calculating convention. For example, when calculating 2^{3^4}, we first calculate 3^4, then calculate 2^{81}. This means that when directly pushing the operator into the stack, we require the priority of the next operator to be strictly less than the priority of the operator on the top of the stack.

- Support arbitrary level of brackets in your calculator.

  Note: You can see that we've provided you with a map of priority relation below. Note that after introducing the bracket into this problem, the priority is a little bit different from before, in that the priority of two brackets are for reference only. When processing multiple brackets, the best way to deal with them is to treat them separately from other operators.

- Support floating type numbers of arbitrary length (eg. 101.101). There won't be spaces inside one number.

  Note: To finish this task, what you need is simply some extra processing when taking in the input.

You won't meet many difficulties if you complete the simple calculator with well-organized data structure and algorithm.

Hint: Since the priority is much more complex than that of the simple calculator, you can use `std::map` or `std::unordered_map` in this part. Here is an example:

```cpp
#include <map>

const map<char, int> priority_map = {
        {')', 0},
        {'+', 1},
        {'-', 1},
        {'*', 2},
        {'/', 2},
        {'^', 3},
        {'(', 4},
}; // Why setting the priority of brackets like this?
```

And then you can simply get the priority of `*` with `priority_map.at('*')`.

2.4 Ending

You are very lucky to survive all of these labs this semester. Wish that you can do well in all your finals.