

HOMEWORK #6:

No Parenthesis Necessary

Due Date: [Wednesday, March the 16th, 11:59.59pm](#)

For this assignment, submit any necessary files, but your `main()` function should be in a file called `robotcalc.cpp`. Your program will be **compiled** using the command `fg++ *.cpp`, so please don't submit extra files that may interfere with compilation. Remember to put your **name** and **section** at the top of your program files. Your program should expect all input to come from `cin`, and all your output should be to `cout`.

Problem:

It is well known that robots are superior to humans. One way in which robots are better than humans is in the way they evaluate arithmetic expressions. Instead of using *in-fix* notation like humans do, robots use *post-fix* notation, in which the operator follows the operands. For example, while a human would write:

$$5 + 4$$

A robot would write:

$$5\ 4\ +$$

This notation allows robots to avoid wasting precious memory with silly parentheses and baroque operator precedence. While humans need parentheses to indicate the order of evaluation like in:

$$7 * (5 - 3)$$

Robots simply follow the order of operations from left to right, as in :

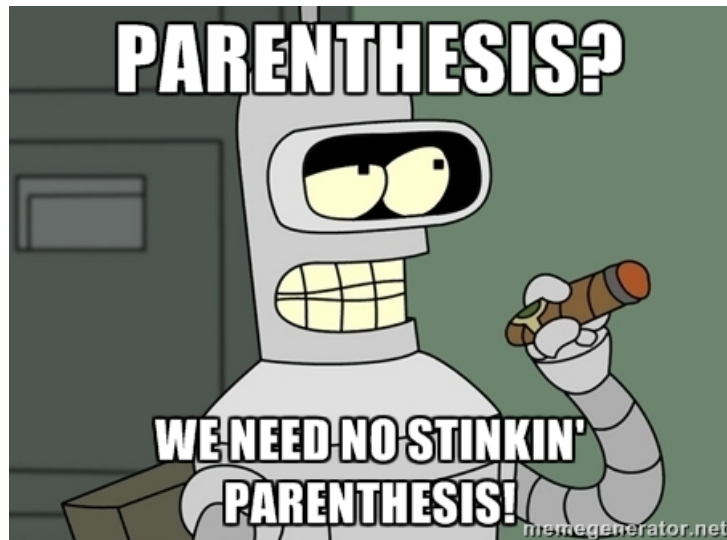
$$5\ 3\ -\ 7\ *$$

or alternatively:

$$7\ 5\ 3\ -\ *$$

This super-human ability is possible because robots organize their short term memory as a **stack**. When robots read an *operand* they push it into their short-term memory stack. When they read an *operator*, they pop the necessary operands and push the result into the memory **stack**.

Your job is to write a program that simulates the way robots process integer expressions and create a “*Robot Calculator*”. Your program will use a stack to simulate a robot's short term memory.



(Typical Robot)

Input:

The input will consist of a series of **integer** expressions in *post-fix* notation. Elements of the expression are separated by spaces.

You will implement the following robot integer expression operators:

- Binary operators '+', '-', '*', '/', '%' with their usual meanings.
- Unary operator '~' negation... (Example... 5 ~ produces -5)
- Aggregate operator 's' : the sum all the elements in the stack.
- Aggregate operator 'p' : the product of all the elements in the stack.
- Stack operator '!' : prints the content of the stack.
- Stack operator '\$' : clears the stack.

Character '.' will denote the end of the input.

Output:

Simulate a robots expression evaluation and output the contents of the stack (formatted as in the sample) whenever the '!' command is found.

REMEMBER: When your program reads an operator, it should pop the corresponding number of operands off the stack, apply the operator, and push the result back into the stack.

Implementation Requirements / Details:

- Use a **stack** data structure to simulate a robot's computational processes.
- Your stack implementation **should be a subclass** of the provided "AbstractStack" class.
- Use a '*.h' and '*.cpp' file pair **for your own stack class**.
- Your **'main()' function** should be inside a file called '**robotcalc.cpp**'

- All operands are integers.
- An expression may take more than one line.
- All expression elements are separated by at least a single space.
- The stack is printed only when the '!' command is encountered.
- [Create .zip or .tar to submit files together.](#)

Sample

Input	Output
1 2 3 ! \$	[3 2 1]
4 3 * ~ !	[-12]
20 3 / ! \$!	[6 -12]
62 ! \$	[]
2 3 8 * + !	[62]
4 48 4 2 + / !	[26]
S ! \$	[8 4 26]
1 2 3 4 5 !	[38]
P ~ !	[5 4 3 2 1]
.	[-120]

Reference:

http://en.wikipedia.org/wiki/Reverse_Polish_notation

Hints:

Study the following code sample:

```
#include <cstdlib>
#include <iostream>
using namespace std;
int main ()
{
    string s1, s2;
    int x, y;

    cin >> s1 >> s2;
    x = atoi( s1.c_str() );
    y = atoi( s2.c_str() );
    cout << s1 << " * " << s2 << " = " << x*y << endl;
    return 0;
}
```