

STANDARD TEMPLATE LIBRARY STACKS

Problem Solving with Computers-II

C++

```
#include <iostream>
using namespace std;

int main(){
    cout<<"Hola Facebook\n";
    return 0;
}
```

C++ STL

- The C++ Standard Template Library is a very handy set of three built-in components:
 - Containers: Data structures
 - Iterators: Standard way to search containers
 - Algorithms: These are what we ultimately use to solve problems

C++ STL container classes

```
array  
vector  
forward_list  
list  
set  
stack  
queue  
priority_queue  
multiset (non unique keys)  
deque  
unordered_set  
map  
unordered_map  
multimap  
bitset
```

Stacks – container class available in the C++ STL

- Container class that uses the Last In First Out (LIFO) principle
- Methods
 - i. `push()`
 - ii. `pop()`
 - iii. `top()`
 - iv. `empty()`

Lab05 – part 1: Evaluate a fully parenthesized infix expression

$(4 * ((5 + 3.2) / 1.5))$ // okay

$(4 * ((5 + 3.2) / 1.5)$ // unbalanced parens - missing last ‘)’

$(4 * (5 + 3.2) / 1.5)$ // unbalanced parens - missing one ‘(’

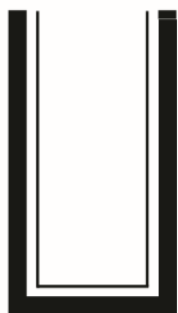
$4 * ((5 + 3.2) / 1.5)$ // not fully-parenthesized at ‘*’ operation

$(4 * (5 + 3.2) / 1.5)$ // not fully-parenthesized at ‘/’ operation

Example: $((2 * 2) + (8 + 4))$

How do we figure out if the parentheses are balanced using stacks?

Initial
empty
stack



Read
and push
first (



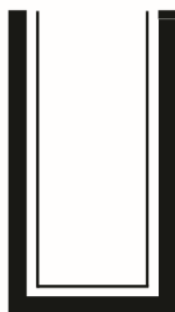
Read
and push
second (



(We can safely ignore all characters that aren't (or) right now)

$((2 * 2) + (8 + 4))$

Initial
empty
stack



Read
and push
first (



Read
and push
second (

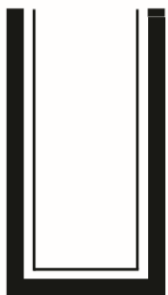


What should be the next step after the first right parenthesis is encountered?

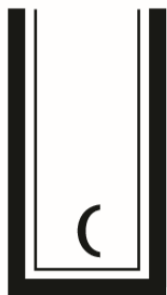
- A. Push the right parenthesis onto the stack
- B. If the stack is not empty pop the next item on the top of the stack
- C. Ignore the right parenthesis and continue checking the next character
- D. None of the above

$((2 * 2) + (8 + 4))$

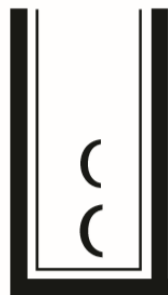
Initial
empty
stack



Read
and push
first (



Read
and push
second (



Read first
) and pop
matching (



Read
and push
third (



Read
second)
and pop
matching (



Read third
) and pop
the last (



Evaluating a fully parenthesized infix expression

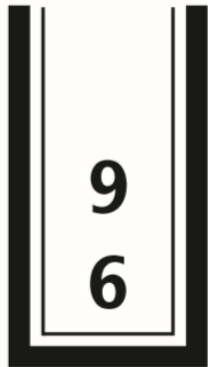
$((6 + 9) / 3) * (6 - 4)$

Evaluating a fully parenthesized infix expression

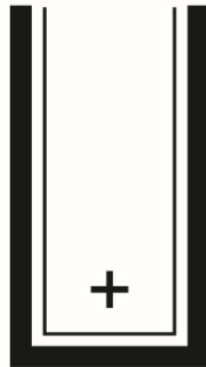
Characters read so far (shaded):

`((6 + 9) / 3) * (6 - 4)`

Numbers



Operations

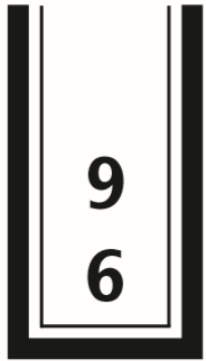


Evaluating a fully parenthesized infix expression

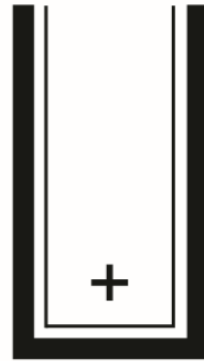
Characters read so far (shaded):

((6 + 9) / 3) * (6 - 4))

Numbers



Operations



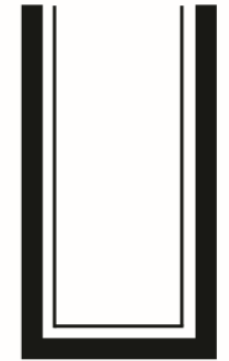
Before computing 6 + 9

6 + 9 is 15

Numbers



Operations



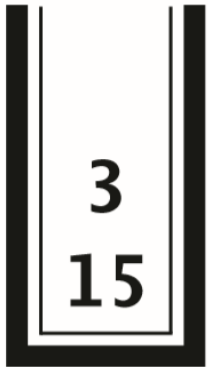
After computing 6 + 9

Evaluating a fully parenthesized infix expression

Characters read so far (shaded):

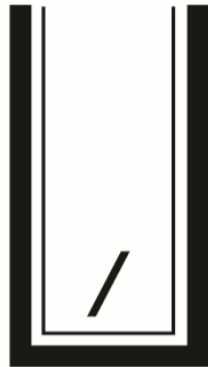
`((6 + 9) / 3) * (6 - 4)`

Numbers



Before computing 15/3

Operations



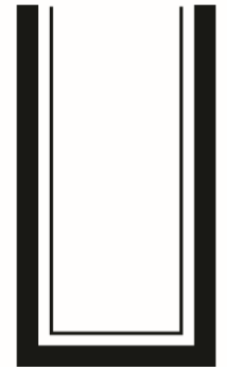
15 / 3 is 5

Numbers



After computing 15/3

Operations



Notations for evaluating expression

- Infix number operator number
 - (Polish) Prefix operators precede the operands
 - (Reverse Polish) Postfix operators come after the operands
-
- $3 * 5$
 - $4 / 2$
 - $7 + (3 * 5)$
 - $(7 + (3 * 5)) - (4 / 2)$

Lab 05, part2 :

Evaluating post fix expressions using a single stack

Postfix: 7 3 5 * + 4 2 / -

Infix: (7 + (3 * 5)) - (4 / 2)

Small group exercise

Write a ADT called in minStack that provides the following methods

- `push()` // inserts an element to the “top” of the minStack
- `pop()` // removes the last element that was pushed on the stack
- `top ()` // returns the last element that was pushed on the stack
- `min()` // returns the minimum value of the elements stored so far

