CSCI-1200 Computer Science II — Fall 2006
Lab 11 — Stacks and Queues

Stacks and queues are very simple sequence containers in which items are only added and removed from the end. In a stack, all work is done on just one end, called the `top`. Hence, when an item is removed, it will be the item most recently added. As a result, a stack is called a LIFO structure, for "Last In First Out". In a queue, items are added to the end, usually called the `rear` or `back`, and removed from the other end, usually called the `front`. Hence, when an item is removed it will be the item that has been in the queue longer than any other item currently in the queue. As a result, a queue is called a FIFO structure, for "First In First Out". A fundamental property distinguishing stacks and queues from other containers is that **items in the middle of the sequence may not be accessed or removed.** One effect of this is than neither stacks nor queues have iterators.

A simple example will help illustrate further the use of a stack. The goal is to determine whether or not an expression has a balanced set of parentheses. Here the term parentheses is meant to include the characters '(', '[', '{', ')', ']', '}'. We use a stack of chars. Each time an "open" parenthesis char — '(', '[', '{' — is read in the input, the char is pushed onto the stack. Every time a "close" parenthesis — ')', ']', '}' — is read in the input, the top char of the stacked is checked:

- If the stack is empty, the parentheses are unbalanced, so an error has occurred.
- If the top char is the matching "open" parenthesis char — e.g. '(' on top of the stack when ')' is read, etc. — there is a correct match of parentheses. In this case, the top char is popped off the stack, and both it and the input char are discarded (no longer considered).
- If the top char is not the matching "open" parenthesis — e.g. a '(' is on top of the stack when a '}' is read — then a error has been detected.

This process of reading chars and doing the outlined push / comparison / pop operations continues until an error is found or until there is no more input associated with the expression. If the stack is not empty at the end of the input, there aren't enough closing parentheses.

Stacks and queues are implemented in the standard library (`#include <stack>` and `#include <queue>`) as templated containers (e.g., `std::stack<int> s;` and `std::queue<char> q;`). Summaries of the `stack` and `queue` operations from `http://www.sgi.com/` are included for your reference. Interestingly, these classes are implemented in terms of other standard library containers rather than being implemented "from scratch". We'll explore this issue in the checkpoints below.

Download these two files from the course web site and then turn off your network connections.

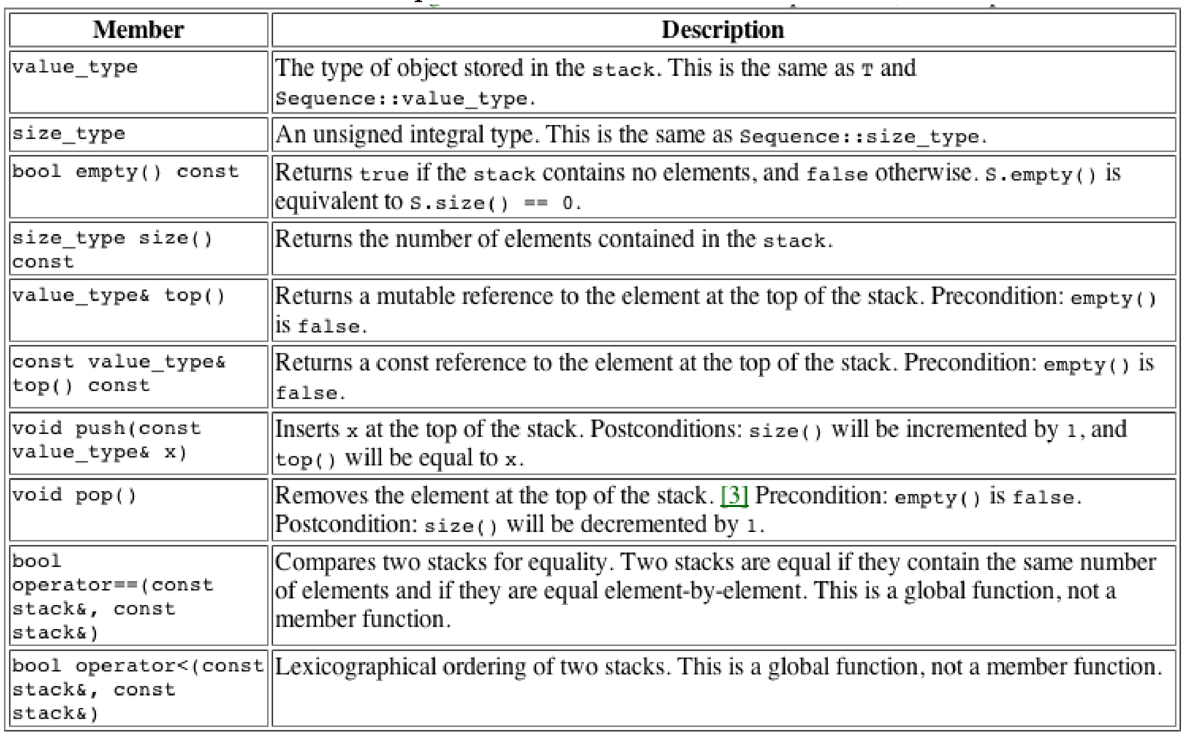
```
http://www.cs.rpi.edu/academics/courses/fall06/cs2/labs/11_stacks_queues/cs2stack.h
http://www.cs.rpi.edu/academics/courses/fall06/cs2/labs/11_stacks_queues/cs2queue.h
```

Checkpoints

1. Write a program that uses the standard library `stack` class as described above to see if the parentheses, curly braces, square brackets, *and angle brackets* are balanced. Make up some simple test cases to be sure your program works correctly. Also test your program on various C++ files you have created this semester. Do programs that compile without error always have balanced pairs of these characters? Why or why not?

2. The file `cs2stack.h` contains a partial implementation of `stack` in terms of `vector`. Complete this implementation and write a short main program to test it. Be sure to test all member functions. You should keep the implementation entirely inside `cs2stack.h` and you are welcome to inline functions. Carefully study the preconditions and output an error message is they are not met.

Operations on a `std::stack`



Member	Description
`value_type`	The type of object stored in the `stack`. This is the same as `T` and `Sequence::value_type`.
`size_type`	An unsigned integral type. This is the same as `Sequence::size_type`.
`bool empty() const`	Returns `true` if the `stack` contains no elements, and `false` otherwise. `S.empty()` is equivalent to `S.size() == 0`.
`size_type size() const`	Returns the number of elements contained in the `stack`.
`value_type& top()`	Returns a mutable reference to the element at the top of the stack. Precondition: `empty()` is `false`.
`const value_type& top() const`	Returns a const reference to the element at the top of the stack. Precondition: `empty()` is `false`.
`void push(const value_type& x)`	Inserts `x` at the top of the stack. Postconditions: `size()` will be incremented by `1`, and `top()` will be equal to `x`.
`void pop()`	Removes the element at the top of the stack. [3] Precondition: `empty()` is `false`. Postcondition: `size()` will be decremented by `1`.
`bool operator==(const stack&, const stack&)`	Compares two stacks for equality. Two stacks are equal if they contain the same number of elements and if they are equal element-by-element. This is a global function, not a member function.
`bool operator<(const stack&, const stack&)`	Lexicographical ordering of two stacks. This is a global function, not a member function.