Computer Science II — CSci 1200
Lab 10
Stacks and Queues

Introduction to 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.

Stacks and queues are interesting because their simple functionality provides an appropriate model for many common operations. Here are some examples:

- The management of memory associated with function calls uses a stack. Each function call causes the creation of a "frame" for the new function, including space for parameters and local variables. This frame is "pushed" onto the memory stack at the start of operations for the function. When a function ends, this frame is "popped" off the memory stack, and the previous function, whose memory is now on the top of the stack, resumes its execution.

- The allocation of computing resources, including the CPU, printers and communication channels, is often based on the FIFO nature of a queue. Certainly, these allocation methods are more sophisticated than purely FIFO, but FIFO is the starting idea.

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, a mismatch error has also occurred — there aren't enough closing parentheses. To fully understand the foregoing, try a few examples yourself.

Stacks and Queues and the Standard Library

Stacks and queues are implemented in the standard library as templated containers. The include files are just called `stack` and `queue`, as in

```
#include <stack>
#include <queue>
```

The definition of stack and queue objects is pretty much what you might guess, e.g.

```
  std::stack<int> s;
  std::queue<char> q;
```

Figures 1 and 2 summarize the public interfaces to stack and queue objects. These summaries are taken from

    http://www.sgi.com/tech/stl/stack.html

and

    http://www.sgi.com/tech/stl/queue.html

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.

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.

Figure 1: Operations on a `std::stack`.

Files to Download

Before proceeding to the lab checkpoints, download two files from the course web site:

```
http://www.cs.rpi.edu/academics/courses/spring07/cs2/lab10/cs2stack.h
http://www.cs.rpi.edu/academics/courses/spring07/cs2/lab10/cs2queue.h
```

After doing so, turn off your network connections.

Checkpoints

1. Write a program that uses a stack and a queue to determine if the alphabetic characters in a line of input form a palindrome. The program must read in the characters from `cin` one at a time until encountering the '\n' char (use `cin.get(c)` rather than `cin >> c`), convert all alphabetic chars to lower case, and store the letters in a stack or a queue or both. After the line of input has been read, the program

3

Member	Description
`value_type`	The type of object stored in the `queue`. 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 `queue` contains no elements, and `false` otherwise. `Q.empty()` is equivalent to `Q.size() == 0`.
`size_type size() const`	Returns the number of elements contained in the `queue`.
`value_type& front()`	Returns a mutable reference to the element at the front of the queue, that is, the element least recently inserted. Precondition: `empty()` is `false`.
`const value_type& front() const`	Returns a const reference to the element at the front of the queue, that is, the element least recently inserted. Precondition: `empty()` is `false`.
`value_type& back()`	Returns a mutable reference to the element at the back of the queue, that is, the element most recently inserted. Precondition: `empty()` is `false`.
`const value_type& back() const`	Returns a const reference to the element at the back of the queue, that is, the element most recently inserted. Precondition: `empty()` is `false`.
`void push(const value_type& x)`	Inserts `x` at the back of the queue. Postconditions: `size()` will be incremented by `1`, and `back()` will be equal to `x`.
`void pop()`	Removes the element at the front of the queue. [3] Precondition: `empty()` is `false`. Postcondition: `size()` will be decremented by `1`.
`bool operator==(const queue&, const queue&)`	Compares two queues for equality. Two queues 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 queue&, const queue&)`	Lexicographical ordering of two queues. This is a global function, not a member function.

Figure 2: Operations on a `std::queue`.

must empty the container(s) used, determine if the line is a palindrome while doing so, and output an appropriate message.

2. The downloaded file `cs2stack.h` contains a partial implementation of the stack in terms of a 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.

4