CIS 291 – Data Structures in C++ - Spring 2005
Homework #3
HW #3 due: Thursday, February 10th, BEGINNING of class

Purpose:
Implement fixed-capacity versions of the stack and queue ADT's, and use them in a function.

How this will be turned in:
Use **~st10/291submit**, called from the directory on cs-server where the files you wish to submit are stored.

Homework Problems:
IMPORTANT: PUT THE FILES FOR THIS ASSIGNMENT IN A DIFFERENT DIRECTORY THAN USED FOR HW #2 or for the WEEK 3 LAB EXERCISE. We are deliberately creating different versions of files with the SAME NAMES as in those previous activities (for easier switch-ability later on), and I want you to be able to RE-USE EITHER of the versions later on in the semester.

On the course web page, you will find pseudo-UML descriptions of a **fixed-capacity** stack and queue (these have a **2-3-05** revision date).

1. *Carefully* study the pseudo-UML for the 2-3-05 fixed-capacity stack. You'll see some new operations, as well as modifications of the pre-existing operations.

IN A NEW DIRECTORY, create **new** versions of **stack.h** and **stack.template** that use a **static array** implementation to **precisely** implement this fixed-capacity UML.

Part of what I am looking for here is attention to detail, in two senses:

* first, if you are asked (as part of a team) to implement a particular UML, everyone else on that team is depending on you to implement it **precisely** as described. Their code may fail if you vary from that agreed-upon **interface** in any way. Therefore, you will lose credit for **any** deviation from the provided pseudo-UML in your implementation.

* second, I fully expect that you will likely start with the dynamic-array implementation provided as part of HW #2's stack, especially because it is your first example of a template class implementation, However, when you modify such existing code, it is important to remove everything that it not pertinent to the new version, change now-inappropriate or now-misleading identifier names and comments appropriately, etc. Therefore, you will lose credit for any such "artifacts" remaining in your revised version.

Note that, since you are to be creating a **static array** implementation, you do not need to explicitly implement a copy constructor or destructor, and in fact it is a homework specification for this particular assignment that these **not** be explicitly implemented for this particular homework assignment.

Be careful --- note how the template class methods are implemented in the dynamic array stack.template file (for example, how each template class method needs to begin with a **template <typename Item>** line, and instead of **stack::**, you need **stack<Item>::** .) Another point to note: when you refer to a named constant from a template class in the implementation, it, too, needs to be preceded by **stack<Item>::** --- that's why, in the dynamic array implementation of stack's **push** method, you saw an instance of **stack<Item>::INCR_AMT** .

When you are ready, create **test_stack.cpp** to test your new implementation. Make sure that the constructor and all of the methods are attempted, and their results appropriately verified. When you are satisfied with it, redirect its output into a file for turning in:

```
test_stack > hw3_stack_output
```

For this problem, you will turn in your final versions of **stack.h**, **stack.template**, **test_stack.cpp**, and **hw3_stack_output**.

2. *Carefully* study the pseudo-UML for the 2-3-05 fixed-capacity queue. You'll see some new operations, as well as modifications of the pre-existing operations.

IN A NEW DIRECTORY, create **new** versions of **queue.h** and **queue.template** that use a **static array** implementation to **precisely** implement this fixed-capacity UML.

The same attention to detail is expected here as for problem #1, and, likewise, your solution is expected to **not** explicitly implement a copy constructor nor a destructor.

Remember about needing **template <typename Item>** and **queue<Item>::** in **queue.template**, too.

When you are ready, create **test_queue.cpp** to test your new implementation. Make sure that the constructor and all of the methods are attempted, and their results appropriately verified. When you are satisfied with it, redirect its output into a file for turning in:

```
test_queue > hw3_queue_output
```

For this problem, you will turn in your final versions of **queue.h**, **queue.template**, **test_queue.cpp**, and **hw3_queue_output**.

3. Now, some stack-and-queue-in-tandem practice for your new fixed-capacity implementations.

We discussed (during lecture in Week #2) how one can recognize a palindrome string using a stack and a queue: grab each character in the string, and if it is a letter between a-z or A-Z convert it to lowercase and push it on the stack and enqueue it in the queue. Then, while the stack and queue are not empty, pop and dequeue (respectively) comparing the letters. If the string is a palindrome, then the popped/dequeued characters will always match --- if they ever do NOT match, then you know the original string was not a palindrome.

Write a function **is_palindrome** that uses the above algorithm, using your stack and queue implementations from problems #1 and #2. It should expect a C++ **string** as its parameter, and return **bool** true if the passed string is a palindrome that does not exceed the stack/queue maximum capacity, and return **bool** false otherwise. (Because we are talking about fixed-capacity stack and queue implementations here, your function should also return false if the passed string is too long. How can you check this?)

(As implied in the above algorithm description, your function will only consider letters a-z (regardless of case) in its consideration; other characters should be ignored. That is:

```
string phrase = "Madam, I'm Adam";
cout << (is_palindrome(phrase) == true) << endl;
```

...would print a 1 (is_palindrome() would return true for this string phrase).)

(Hint: when you #include **string** and/or **iostream**, you also include such nifty functions as **isalpha** which, when given a **char** as an argument, returns **bool** true if it is an alphabetic character and false otherwise, and **tolower** which, when given an alphabetic character, returns the lowercase version of that character. Also, note how strings are used in the **reverseString.cpp** example.)

(Hint #2: in writing is_palindrome.h, note that #include's are only needed in a .h file when the function prototype has a **parameter** type that requires it...)

Yes, you are required to appropriately use both a stack and a queue in your solution. Be sure, amongst your examples, to include at least one even-length palindrome and at least one odd-length palindrome (traditionally this can be a sticking point in some algorithms).

You will turn in **is_palindrome.cpp**, **is_palindrome.h**, **test_is_palindrome.cpp**, and the results of running **test_is_palindrome** redirected to a file named **hw3_pal_output**.

Note that, when you are finished with this homework, you will be submitting **10** files:

stack.h, **stack.template**, **test_stack.cpp**, **hw3_stack_output**,
queue.h, **queue.template**, **test_queue.cpp**, **hw3_queue_output**,
is_palindrome.cpp, **is_palindrome.h**, **test_is_palindrome.cpp**, and **hw3_pal_output**.