CIS 291 – Data Structures in C++ - Spring 2005
Homework #2

HW #2 due: Thursday, February 3rd, BEGINNING of class

Purpose:

Make use of abstract data types --- here, stack --- given UML depictions of those ADT's and implementations of those ADT's.

How this will be turned in:

When you wish to submit one or more files, call:

~st10/291submit

...from the cs-server prompt while in the directory where the files to submit reside. It will prompt you for the number of the homework and for the names of the files you wish to submit. Only files submitted using this tool by the deadline will be accepted for credit.

[IF YOU ENCOUNTER ANY PROBLEMS ACCESSING cs-server, LET ME KNOW A.S.A.P. This is even worthy of a brief phone message to 825-7727 (my home phone), so I can contact the lab administrator as promptly as possible. It is another example of the importance of starting assignments early. One more comment in this regard: what if cs-server has a burp early in the week? I would strongly advise beginning your development on paper, or on redwood or sorrel (whose g++ versions may be slightly older, so be careful!) or Dev-C++, and then using **sftp** to transfer your work-in-progress to cs-server when you are able.]

[Remember the scheduled power shutdown for NHW and NHE this weekend, also; you should have received an e-mail message about it.]

Homework Problems:

NOTE: for all course programming, you are expected to follow the course coding standards as described in lecture and lab, and as demonstrated in posted course examples. You are also expected to follow the style demonstrated in the posted course templates.

Also, some friendly C++ reminder/hints:

* remember, when you use the type **string**, you need to #include **<string>**...

* while you do not usually need #include's in .h files, you do if one of the parameter types in the prototype declaration needs it. So, if you have a **string** parameter, you'll need #include **<string>** (and using namespace std!) in the .h file, too.

Next page: a pseudo-UML depiction of the **stack** ADT, close to what was described in lecture:

Initial "UML" for **stack** class (revised 1-27-05)

adapted from Ch. 7, Savitch and Main, "Data Structures and Other Objects Using C++"

Template Class: `stack`
/* a collection of items such that entries can be inserted and removed at only one end (called the **top**). */

Member data and related details:
/* contains elements of type **`value_type`**; this is set to be the value of template parameter **`Item`**

Constructors:
/* postcondition: creates an empty **stack** instance */

```
stack( );
```

Accessors and other constant member functions:
/* postcondition: returns **true** if stack is empty, and returns **false** otherwise */

```
bool     is_empty( ) const;
```

/* precondition: is_empty() == false */
/* postcondition: returns the value of the top item of the stack, BUT the stack is unchanged. */

```
Item     get_top( ) const;
```

Modifiers and other modifying member functions:
/* postcondition: a new copy of **entry** has been pushed onto the (top of the) stack */

```
void     push(const Item& entry);
```

/* precondition: is_empty() == false */
/* postcondition: the top item of the stack has been removed, and a reference to it is returned */

```
Item&    pop( );
```

1. You should copy the following files from the course web page (also from "Homeworks and Handouts", where you grabbed this handout from): **stack.h**, **stack.template**.

Notice how that second file ends with **.template** instead of with **.cpp**? That's because it is an implementation of a **template class** instead of a "normal" C++ class. We'll have to talk about implementing template classes, because there are some syntax issues to discuss to be able to write them. However, once written, template classes are a breeze to use!

The point of a template class is that, for some situations, you have a whole "family" of classes that are really the same, except for which type they involve. (For example, consider a list of strings, a list of ints, a list of lists...) A template class lets you write that class but defer the specification of which type is involved in that class until compile-time. You can specify a particular type when you use a class, and use different types for different instances of the same class. Or, if I have a stack template class, I can write a single stack template class, and use it to create an int stack here, and a string stack there --- and these two instances could even be within the same program, using the same stack template class. This is convenient!

SO --- given a template class **george**, that expects a single **template parameter**. If I'd like that template parameter to be **int** for george instance **numeric_g1** and to be char for george instance **letter_g2**, then I just put the desired type (for the template parameter) in angle brackets after the template class name in the declaration:

george<int> numeric_g1;
george<char> letter_g2;

Then, I just call the methods of **numeric_g1** and **letter_g2** in the normal way.

Or, if you prefer, look at the implementation of **reverseString.cpp** given on the course web page, under "In-class examples". It uses **stack.h** and **stack.template**, and you can see how I declared **charStack**:

```
stack<char>      charStack;
```

Then **charStack.is_empty()** lets me see if **charStack** is empty, etc.!

Note, too, that when you are using the C++ **string** type, you need to #include **<string>**.

To make sure that you have correctly copied over **stack.h** and **stack.template**, copy over, also, **reverseString.cpp**.

* Write an appropriate header file (.h file) for **reverseString** (using the appropriate posted course template); this file must be named **reverseString.h**

* Write an appropriate testing main function, in its own file **test_reverseString.cpp**, for **reverseString** (using the appropriate posted course template) to run all of the examples given in **reverseString**'s opening comment block. You may add additional tests if you wish, also.

* Another oddness with template classes: for a regular class, when it is used in another, you need its .o file in the linking stage; but a template class is included with the .h file for the template class, and so ends up really being compiled when the file using it is compiled. So, even though:

  reverseString.cpp uses **stack.h**;
  test_reverseString.cpp uses **reverseString.h**;

  ...a similar sequence of compilations as we saw in HW #1 and the Week 1 Lab Exercise works here, too:

```
g++ -c reverseString.cpp
g++ -c test_reverseString.cpp
g++ -o test_reverseString test_reverseString.cpp reverseString.o    // note: templates are weird in that
                                                                     //    they are not compiled until the
                                                                     //    program they are used in is compiled!
```

* When all is well compiling-wise, run and debug your **test_reverseString** until you are satisfied with its output. Then redirect its output into a file for turning in:

```
test_reverseString > test_reverseString_output
```

  For this problem, you will turn in your final versions of **test_reverseString.cpp** and **test_reverseString_output** .

2. Now that you know you have a working stack implementation available, let's use it in another application.

There is a notation known as **reverse Polish notation** (RPN); I believe that is is named after a Polish mathematician. Some calculators use it (or a version of it); its advantage is that parentheses are not needed to disambiguate expressions.

Basically, instead of putting the operator in between the two operands --- "in-fix" --- you write it after the two operands. That is,

instead of...	you'd write:
2+3	**2 3 +**
(2+3)-(8*7)	**2 3 + 8 7 * -**
(2 + (3-8)) * 7	**2 3 8 - + 7 ***
2 + ((3-8) * 7)	**2 3 8 – 7 * +**

It turns out that an expression written in this form is quite easy to calculate with a stack: every time you see a number, you push it on the stack. Every time you see an operator, you pop the top two numbers off of the stack, perform the operation upon them, and push the result back onto the stack. When the expression is completely consumed, only one item should remain on the stack: the computed result.

We're going to practice this in a very restricted setting.

Write a function **rpn_compute**, in file **rpn_compute.cpp**. This function expects a C++ **string** as input, and will only behave reasonably if this string:

* contains **only** the digits 0-9, or the characters +, **-**, *****, or /.
* contains a "legal" (well-formed) RPN expression for consisting of single-digit numbers and operators only. [This restriction is intended to simplify the problem --- you are not being asked to **parse** the input, or to figure out that **12 3** means twelve and three or that **1 23** means one and twenty-three. Instead, you'll see something like **12+** and be able to assume that it means the sum of one and two.]

And, it should return the computed result, or **-1** if, at any time along the way, it runs into problems.

(Hint: C++'s **string** library includes such nifty functions as **isdigit** which, when given a **char** as an argument, returns **true** if that char is a digit character. Also, remember that if **ch** is a char that happens to be a digit (e.g., '6'), then you can get the integer corresponding to that character by subtracting it from the character '0', since C++ char's are weird int's... 8-) That is, '6' - '0' == 6)

You are not required to test for an "illegal" input string; you do need to handle the major categories of possible "legal" inputs, however.

You will turn in **rpn_compute.cpp**, **rpn_compute.h**, **test_rpn_compute.cpp**, and the results of running **test_rpn_compute** redirected to a file named **test_rpn_compute_output**.

Note that, when you are finished with this homework, you will be submitting **6** files:

test_reverseString.cpp, **test_reverseString_output** and
rpn_compute.cpp, **rpn_compute.h**, **test_rpn_compute.cpp**, **test_rpn_compute_output**