CIS 291 – Data Structures in C++ - Spring 2005
Homework #11
HW #11 due: THURSDAY, April 21st, BEGINNING of lecture

Purpose: implementing a given **set** ADT, making design decisions for that implementation

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 #11

On the next page, you will see a familiar pseudo-UML diagram --- it is the same **set** UML that you saw for Exam #2.

Design/write/test/debug this class **set** --- create appropriate files **set.h** and **set.template**. You must implement the given pseudo-UML diagram precisely, you must implement **set** as a template class, and you must follow course style standards, but you get to choose how you will implement it. You may use other classes within your implementation (but submit any additional needed files along with your **set.h** and **set.template**! I should have what I need to run your set implementation if I choose to.)

[Note: you should not be adding operations to those given in the pseudo-UML, EXCEPT for those operations required (but not usually specified in our course UML's) when you have an implementation involving dynamically-allocated memory. If you are not sure what I mean, compare some of the posted implementations to their corresponding UML's, and see how they differ.]

Write a **test_set.cpp** that tests your class (following course style standards), and when you are happy with your code run:

```
test_set > 291hw11_out
```

...and submit your resulting **set.h**, **set.template**, **test_set.cpp**, **291hw11_out**, and any files needed for other classes' you've used in your implementation of **set**.

set pseudo-UML begins on the NEXT PAGE!

"pseudo-UML" diagram for Exam **set** template class

Template Class: `set`

/* an unordered collection of items of a single type, duplicates NOT permitted */

Member data and related details:

* contains elements of type **`value_type`**; this is set to be the value of template parameter **`Item`**
* has a size of **`size_type`**

Constructors:

```
/*   postcondition: creates an empty set instance */
set();
```

Accessors and other constant member functions:

```
/*   postcondition: returns the number of items in the set. */
size_type get_size() const;

/*   postcondition: returns true if the target is in the set, and returns false if it is not. */
bool      contains(const Item& target) const;

/*   postcondition: returns true if there is a valid "current" item that may be returned by the current member function.
     Returns false if there is no valid current item. */
bool      is_item() const;

/*   precondition: is_item() == true
     postcondition: returns the current item in the set's internal iterator. */
Item      current() const;
```

Modifiers and other modifying member functions:

```
/*   postcondition: if target was in the set, remove it and return true; otherwise, return false (and the set is
     unchanged) */
bool      remove(const Item& target);

/*   postconditions: if entry is not already in the set, then add it to the set; otherwise, set is unchanged. */
void      add(const Item& entry);

/*   postcondition: one of the items in the set becomes the current item (but if the set is empty, there is no current
     item) */
void      start();

/*   precondition: is_item() == true
     postcondition: if the current item is the last in the set's internal iterator, then there is no longer any current item.
     Otherwise, the new current item is another set element that has not yet been current during this internal iterator.
     */
void      advance();
```

Other methods

```
/*   postcondition: this returns a new set whose membership is the set-theoretic (classic) union of the calling set and
     <set2> */
set<Item> union(set<Item> set2);

/* postcondition: this returns a new set whose membership is the set-theoretic (classic) intersection of the calling set
     and <set2> */
set<Item> intersect(set<Item> set2);
```