CS 132 - Intro to Computer Science II - Spring 2005
Week #8 Lab Exercise and Homework #6

Week #8 Lab Exercise due: Wednesday, March 9th, END of lab
HW #6 due: Wednesday, March 23rd, beginning of lab

Purpose: get some experience with "modern" C++ type casting, template functions, and template classes.

Week #8 Lab Exercise

1. It has come to my attention that our course text does not use "modern" C++ type casting, which is unfortunate, in my opinion. To make sure, then, that you are reminded of this syntax:

To cast an int to a double: (here, casting the int literal 3 into a double instance)

```
static_cast<double>(3)
```

to cast a c-string literal to a **string**: (here, casting a c-string literal into a string instance)

```
static_cast<string>("hello")
```

(and you can cast variables and other expressions, too...)

Let's give you a quick chance to practice this (and to practice compiling a program that uses a template function). Consider **findMax.template** and **test_findMax.cpp**, which were discussed in lecture and now are available from the course web page.

Modify test_findMax.cpp within a new file **test_findMax2.cpp** so that, instead of using **string** variables **s1** and **s2**, it instead performs type casting of the literal values "pork" and "beans" within that call to **findMax** instead.

(Note: I'm not saying that it is bad to use s1 and s2 here --- this is simply a low-impact way to let you quickly practice modern C++ type-casting.)

When you are satisfied with your modifications, run:

```
test_findMax2 > lab6_1_out
```

2. Write a template function **orderPair** (putting it in a file **orderPair.template**, as done with **findMax.template**). **orderPair** expects two arguments of the same type, which can be any type that has defined operators > and =. It doesn't return anything, but its effect is that, if the first argument is greater than the second, it swaps their values so that the arguments are now "in order" --- the first argument's value is now less than or equal to the second argument's value. This function needs the usual full-function opening comment block (see the opening comment block templates on the course web page for a reminder of what needs to be included). (for the contract, look at template function findMax's contract to remind you how we are handling the type-to-be-specified-later in template function contracts).

Then, write a testing function **test_orderPair.cpp** that runs and tests your function's examples. It can use the opening comment block for testing main's (see that same opening comment block template link), but make sure that your orderPair tests (and examples!) involve at least **three** different data types of your choice.

When you are done with all of this, create an example output file with:

```
    test_orderPair > lab6_2_out
```

When you are done with both of the above, put your name on the **Next:** list on the board to have your work checked over.

HOMEWORK #6:

1. Write a template function **getIndex** (putting it in a file **getIndex.template**, as done with **findMax.cpp**). **getIndex** expects 3 arguments: an array of a type to be specified later, a size, and a target value able to be in the array. It tries to find the target value within the array --- if it CAN, it returns the index of its LATEST instance. (That is, if several copies of the index appear at indices 3, 5, and 18, it would return 18.) If it CANNOT, it returns -1. This function needs the usual full-function opening comment block (for the contract, look at template function findMax's contract to remind you how we are handling the type-to-be-specified-later in template function contracts).

Then, write a testing function **test_getIndex.cpp** that runs and tests your function's examples. It can use the opening comment block for testing main's (see that same opening comment block template link), but note that it **also** needs to include at least one call where the size is passed using a **named constant**, at least one call where the size is passed using an **int literal** (e.g., 3, 27, etc.) , and at least two calls involving arrays of **different** types of elements (your choice of type, as long as they are different from one another).

When you are done with all of this, create an example output file with:

```
test_getIndex > hw6_1_out
```

2. On the course web page, you will see a minimal implementation of a **set**. You'll find **set.h**, **set.cpp**, and a playing-around-function (too informal/incomplete to be a tester!) **trySet.cpp**.

Create a new directory **132hw06**, and within it create a **template class** set, creating appropriate files **set.h** and **set.template**.

Then, play around with your template class **set** by modifying **trySet.cpp** to play with this new template class version. In addition to other changes needed, add a **mySet3** containing **string** instances, which is played around with analogously to how **mySet1** is played around with.

When you are done with all of these, create an example output file with:

```
trySet > hw6_2_out
```

NOTES to keep in mind:
* remember: we are using **typename** instead of **class** in our template prefixes.

Make sure, when you are done, that you have submitted to me the following files (using **~st10/132submit** on cs-server):

getIndex.template
test_getIndex.cpp
hw6_1_out
set.h
set.template
trySet.cpp
hw6_2_out