YALE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

CPSC 427: Object-Oriented Programming *Handout #4*
Professor M. J. Fischer *February 22, 2016*

Problem Set 3

Due before midnight on Monday, March 7, 2016.

1 Assignment Goals

- Explore design patterns for modularity.
- Get practical experience in refactoring existing code to enable it to support new requirements.
- Learn how to use a vector of pointers to manage dynamically allocated data.
- Learn how to write a comparison function to sort on multiple fields.
- Learn how to write a static class function.
- Learn how to implement a real-world program specification.

2 Problem

This assignment is to complete the skater ranking program begun in assignment 2 by computing the final placement of the skaters according to the 6.0 judging system.

Here's an overview of the steps the program should take once the data has been read in from the file:

1. For each judge, compute the *ordinal* number corresponding to each mark.
2. For each skater, compute the several ranking factors based on the skater's ordinals. These factors are called *absolute majority* (`M`), *greater majority* (`GM`), *total ordinals of majority* (`TOM`), and *total ordinals* (`TO`).
3. Sort the skaters based on the ranking factors.
4. Assign final places, taking into account possible ties.
5. Print out the results

The program design problem in this assignment is to organize the data in ways that facilitate each of these steps. A simpler organization of the data into just two classes `SixOh` and `Score` was adequate for assignment 2.

For this assignment, you must change the data organization according to the following.[1]

- In addition to `skaterID`, `judgeID`, and `mark`, class `Score` should have `ordinal` as an additional data member.

[1]There are many ways to write this code, but I am asking you to do it in the way I describe here so that you can understand the advantages (and possible disadvantages) of doing it that way.

- Because the information needed to compute `ordinal` is not available when each `Score` object is instantiated, `Score` objects must now be *mutable*, meaning that they can change after they have been constructed.

- Once the `ordinal` data member has been set in the score object, it needs to be visible in all copies of that object—both in the list of scores for a given judge and in the list of scores for a given skater. To accomplish that, you should have only a single `Score` object for each such skater-judge, and what were formerly lists of type `vector<Score>` will become lists of type `vector<Score*>`, that is, lists of pointers to scores.

- You should create the `Score` objects in dynamic storage using `new`. (Make sure you understand why!)

- You should create a new class `Judge` to represent a single judge. It's data members are `judgeID` (an `unsigned`), `name` (a `string`), and `myScores` (a list of type `vector<Score*>`) containing pointers to all scores for this judge. Public member functions include `addScore`, which puts a `Score*` onto the list `myScores`, and `computeOrdinals()`, which performs step 1 above.

- You should create a new class `Skater` to represent a single skater. Its data members are `skaterID` (an `unsigned`), `name` (a `string`), and `myScores` (a list of type `vector<Score*>`) containing pointers to all scores for this skater. In addition, there are `unsigned` data members for the four ranking factors and the final place, and there is a boolean value to indicate if this skater is tied for the place assigned to it. Public member functions include `computeRankFactors`, which performs step 2 as well as setter functions to set the final place and the tie fields.

- Steps 3 and 4 will be carried out by a new member function `rankAll()` in class `SixOh`.

- Because class `Score` is tightly coupled with classes `Judge` and `Skater`, it is permissible for `Score` to give friendship to both class `Judge` and class `Skater`.

3 Programming Notes

Here are some more details for how to compute each of the steps above.

3.1 Ordinal step

The ordinals are computed separately for each judge, using the `myScores` list of score pointers described above. The method is to sort `myScores` in decreasing order of the `mark` field in each `Score` object. This will require that you use the 3-argument version of `sort`, where the 3rd argument is a comparison function that must return `true` if the first mark is strictly better than the second and `false` otherwise. You should make this comparison function a private static function in class `Judge`.

Once the marks have been sorted, the ordinals are assigned in decreasing order of marks. In case a judge gives the same mark to two different skaters, both skaters should be assigned the same ordinal, and the next ordinal should be skipped. More generally, if the next k skaters all have the same mark, then all should be assigned the next ordinal, and the following $k - 1$ ordinals are skipped.[2]

Here's an example that might make this clearer. The first row represents the marks given by a single judge to 10 skaters. The second row are the assigned ordinals.

$Marks:$	3.4	3.3	3.3	3.0	2.9	2.9	2.9	2.8	2.5	2.1
$Ordinals:$	1	2	2	4	5	5	5	8	9	10

3.2 Ranking factors step

The ranking factors are at the heart of the 6.0 judging system. They are computed for each skater based on the ordinals for that skater from each of the judges.

The following table gives an example of how to compute the ranking factors for a skater, given the ordinals from 7 judges.

Judges:	**J6**	**J2**	**J1**	**J5**	**J7**	**J4**	**J3**	**Summary Value**
$Ordinals:$	1	1	1	2	2	3	3	
$M:$				$\uparrow$				2
$GM:$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$			5
$TOM:$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$			$7 = 1 + 1 + 1 + 2 + 2$
$TO:$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$13 = 1 + 1 + 1 + 2 + 2 + 3 + 3$

- **Absolute majority** `M` is ordinal in the middle position when the ordinals are arranged in increasing order. With 7 judges, the middle is the 4th position.[3] In this example, J5 is in the middle position, and `M` is 2.
- **Greater majority** `GM` is the *number* of judges whose ordinals are less than or equal to `M`. In this example, the set of judges with ordinals less than or equal to `M` is $\{J6, J2, J1, J5, J7\}$. It has cardinality 5.
- **Total ordinals of majority** `TOM` is the *total* of the ordinals that were counted in the greater majority. In this example, it is the sum of the ordinals given by judges J6, J2, J1, J5, and J7, so TOM is 7.
- **Total ordinals** `TO` is the sum of all ordinals for that skater. In the above example, `TO` = 13.

3.3 Sorting step

The ranking factors allow two skaters to be compared to see which is better. For `M`, `TOM`, and `TO`, smaller is better, but for `GM`, larger is better (more judges in favor of the skater).

Skaters are compared on the factors `M`, `GM`, `TOM`, `TO`, in order. The first factor in which the two skaters differ determines which is better. Only if they are the same is the next factor examined. If two skaters are the same in all four factors, then they are *tied*.

In this step, all skaters are sorted according according to the "better than" order described above.[4]

[2]In real competitions, judges are discouraged from giving two skaters the same marks since their job is to make fine distinctions between the competitors.

[3]When the number n of judges is even, the middle position is defined to be $n/2 + 1$. The element in the middle position is also called the *median* of the set of ordinals.

[4]Note that the comparison function used with `std::sort()` *must* define a strict weak order. This means that if two

3.4 Final placement step

The final placement is given by the position of the skater in the sorted list, except that ties must be handled as in the first step. Namely, two skaters who are tied both receive the higher place and the next place is skipped.

4 Grading Rubric

Your assignment will be graded according to the scale given in Figure 1.

#	Pts.	Item
1.	5	A well-formed `Makefile` or `makefile` is submitted that specifies compiler options `-O1 -g -Wall -std=c++11`.
2.	5	Running `make` successfully compiles and links the project and results in an executable file `scorer`.
3.	40	Running `scorer` on the graders' test files produces correct output or a reasonable error message in case of bad input.
4.	10	Each class and function definition is preceded by a comment that describes clearly what it does.
5.	30	All of the instructions in sections 2 and 3 are followed.
6.	10	All relevant requirements from PS1 and PS2 are followed.
	100	Total points.

Figure 1: Grading rubric.

skaters are tied, neither is "better than" the other. Otherwise, `std::sort()` may go into an infinite loop, exhaust the stack, and get a segmentation fault.