YALE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

CPSC 427: Object-Oriented Programming *Handout #8*
Professor M. J. Fischer *April 18, 2016*

Problem Set 5

Due before midnight on Friday, April 29, 2016.

1 Assignment Goals

- Get practice reading other people's code.
- Learn how to extract a module from existing code and turn it into a generally-useful template class.
- Learn what a wrapper class is and why it is sometimes needed.
- Use the stopwatch tool introduced in lecture 14 to explore the runtime efficiency of various sorting methods.

2 Problem

For this assignment, you will be working with two existing projects, `Sensor` and `SortTest`, which can be found on the Zoo in `/c/cs427/assignments/ps5`.

`Sensor` reads a large file of sensor readings from a networked digital thermometer, sorts them in order of increasing temperatures, and writes out the sorted readings in a modified format. The sorting method it uses is radix sort. (See https://en.wikipedia.org/wiki/Radix_sort.) The particular version implemented here is an iterative lest significant digit sort, similar to what is described in section 2.3 of the Wikipedia page, except that the sort works on 8-bit bytes instead of on decimal digits.[1]

`SortTest` is derived from class demo `14-StopWatch`. It performs timing tests on various sort implementations. The two sorts that are tested here are `std::sort()` and a highly-optimized implementation of `quicksort` by Alice Fischer. The list of items to be sorted can be represented in various ways. The built-in `std::sort()` is tested on both an `int[]` array and on a `vector<int>`. Alice's `quicksort` is only tested on an `int[]` array but could easily be generalized to work on `vector<int>`.

The goal of this assignment is to extract the radix sort implemented in `Sensor`, convert it to a template class so that it can work on element type `T`, write new code to take the input `int[]` array and put it into a form that can be handled by the templatized `RadixSort` class, and add it to the sorts tested by `SortTest`.

In greater detail, you should do the following:

1. The sort algorithm `Sensor` is currently written to work on a `Queue` of `Reading`, where each object of type `Reading` represents one sensor data point. You should convert both `Queue` and `RadixSort` to template classes that will work for objects of type `T`.

[1] `Sensor` is derived from code written by Kevin Steele and used with permission.

2. Make a copy of project `Sensor` called `Sensor2`. Backport the new template classes of part 1 into `Sensor2`, replacing the old `Queue` and `RadixSort` classes. Make minor modifications to the remaining code to use the new template classes instead of the original ones. `Sensor2` should produce identical output to `Sensor`.

3. Create a new class `RadixInt` that uses the template classes of part 1 to sort an array of non-negative integers. `RadixInt` is an adaptor class that contains code to convert an `int[]` array to the form required as input by `RadixSort<>` and to copy the sorted output from `RadixSort<>` back to the original `int[]` array.

4. Make a copy of project `SortTest` called `SortTest2`. Add the new `RadixInt` and related classes to `SortTest2`, and put an additional timing test into `main()` to measure the run time for radix sort. You should measure separately the setup time, the actual sort time, and the cleanup time, just as is already done for the other three timing tests. The setup time is the time used to prepare the linked list queue needed by `RadixSort<>`. The cleanup time is the time used to copy the sorted results from the queue back to the original `int[]` array and to delete the queue and associated data structures.

3 Programming Notes

Radix sort (as implemented here) assumes a 4-byte sort key, where the keys are ordered as if they represented a 32-bit unsigned integer. If applied to 32-bit signed integers, it will sort the records correctly, except that the negative numbers will end up in revers order *after* the non-negative ones. For example, given the input list $[-3, -2, -1, 0, 1, 2, 3]$, the "sorted" order would be $[0, 1, 2, 3, -3, -2, -1]$. This makes sense when written in binary:[2]

0	00000000
1	00000001
2	00000010
3	00000011
-3	11111101
-2	11111110
-1	11111111

Nevertheless, it works fine for 32-bit *non-negative* `int`'s.

Radix sort makes four passes over the data. On pass k ($k = 0, 1, 2, 3$), it looks at byte k of the key (counting from right to left), interprets that byte as a number b between 0 and 255, and places the record on queue number b. At the end of the pass, the 256 queues are concatenated together into a single list, ready for the next pass.

To get the 4-byte sort key, `sort()` calls `Cell::getKey()`, which in turn calls `Reading::getKey()`. In the Sensor applicaiton, the temperature is represented as a `float`, so `Reading::getKey()` uses a reinterpret cast in order to view the 32 bits of the `float` as if it were an `unsigned int`. It's a property of the IEEE floating point standard that non-negative `float`'s sort the same way whether interpreted as floating point numbers or as unsigned integers. Thus, if we view the temperatures as unsigned integers, they will sort the desired way.

Unfortunately, temperatures can be negative, so we can't assume they're non-negative. To circumvent this problem, Sensor converts Celsius temperatures to Kelvin, sorts the data, and then converts the temperatures back to Celsius when writing the output.

[2]The table shows 8-bit binary numbers, but the pattern is the same for 32-bit numbers.

When we templatize class `RadixSort`, we want to be able to sort lists of arbitrary type `T` by instantiating the template `RadixSort<T>`. For this to work, type `T` must have a member function `getKey()` defined that returns an `unsigned int` on which the list will be sorted. Thus, if `x` is an object of type `T`, we must be able to write `x.getKey()` and have an unsigned sort key returned. Since class `Reading` has such a method, we can use the templatized sort code in the Sensor application instead of the original non-templatized code by simply replacing `RadixSort` with `RadixSort<Reading>` wherever it occurs.

However, there is a problem when trying to use the templatized radix sort to sort lists of integers. If we write `RadixSort<int>`, then there is no place to put the needed `getKey()` function. If `x` is an `int`, we can't meaningfully write `x.getKey()`.

The solution to this problem is create a wrapper class `Integer` for the primitive type `int`. Then, to sort a list of `int`, one constructs a corresponding `Queue<Integer>`, uses `RadixSort<Integer>` to sort it, and then unwraps the resulting list. Class `Integer` has a member function `getKey()` that returns the sort key needed by `RadixSort<Integer>`, and all is well.

4 Expected Output

Your program `Sensor2` should produce output identical to that produced by `Sensor`.

Your program `SortTest2` should produce output similar to that produced by `SortTest`, except that it should include the output from the radix sort test. I have provided a sample output file. Obviously, your numbers will not match mine since these are real execution times on particular machines. Do not be surprised if you see big differences between different systems, not only in absolute speed of the machine but also in the relative efficiency of the different sorting methods.

5 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 for each project `Sensor2` and `SortTest2` that specifies compiler options `-O3 -g -Wall -std=c++11`. (Notice the higher optimization level than we usually use.)
2.	5	Running `make` in each of the projects `Sensor2` and `SortTest2` compiles and links the project and results in executable files `analyze` and `sorttest`, respectively.
3.	10	The templatized files `RadixSort.hpp` and `Queue.hpp` are identical in both projects `Sensor2` and `SortTest2`, and the differ from the corresponding files in `Sensor` only in ways essential to the conversion from non-templatized to templatized code.
4.	10	Running `analyze` from `Sensor2` produces identical output to that produced by `analyze` from `Sensor`.
5.	20	Running `sorttest` from `SortTest2` produces reasonable output for all four tests.
6.	10	Each class and function definition is preceded by a comment that describes clearly what it does.
7.	30	All of the instructions in sections 2 and 3 are followed. Good OO design principles are followed. No global variables (other than the stop watch). No setters (without convincing justification). No public data members.
8.	10	All relevant requirements from previous problem sets are followed.
	100	Total points.

Figure 1: Grading rubric.