YALE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

CPSC 427a: Object-Oriented Programming *Handout #2*
Professor M. J. Fischer *September 10, 2010*

Problem Set 1

Due before midnight on Friday, September 17, 2010.

1 Assignment Goals

1. To familiarize yourself with the tools needed for this course: the Zoo computer facility, your Zoo course account, a good text editor or IDE, the `C++` compiler and linker (`g++`), a Linux command shell, and basic Linux commands.
2. To learn how to compile, link, and run a multimodule `C++` program.
3. To learn how to use the `submit` script to submit your assignment.
4. To learn to distinguish programming constructs for describing computation from those used to control, modularize, and constrain the code.
5. To have the experience of repurposing existing code.

2 Problems

The class demo `02-InsertionSortCpp` illustrates how a `C++ class` and multiple source files can be used to put structure on a program. In this problem set, you will be asked to read the code carefully in order to distinguish which parts of the program support modularity, code reuse, and robustness, and which parts comprise the actual executable code. You will then be asked to repurpose the code for a different but related application.

2.1 Written Part

Recall from class that the insertion sort demo does the following:

1. It prints a banner.
2. It prompts the user to enter a file name.
3. It opens the specified file name and reads up to `LENGTH` floats from it or until end-of-file is reached (or a read error occurs), storing the list of numbers read.
4. It prints the list of numbers.
5. It uses insertion sort to sort the list. (N.B. Insertion sort is an $O(n^2)$ algorithm and is only suitable for use on relatively short lists.)
6. It prints the sorted list.
7. It frees storage and exits.

This isn't a particularly interesting application in its own right except to serve as a unit test for verifying the correctness of the sort function.

Many people, when asked to write a program to do what is described above, would come up with a monolithic program such as you will find in `02-InsertionSortMonolith`, where everything is contained in `main()` (which you will find in the file `sort.cpp`). Indeed, this program does exactly the same thing as `02-InsertionSortCpp` (with one minor difference), but the code looks very different.

For this problem, I want you to print out files `sort.cpp` from `02-InsertionSort-Monolith`, and files `main.cpp`, `datapack.hpp`, and `datapack.cpp` from `02-Insertion-SortCpp`. Then for each line in `sort.cpp`, find whether or not that line appears in one of the other files, and if so, highlight it there. If it appears but in a slightly altered form, highlight it in a different color (or otherwise indicate that it corresponds but has been changed slightly).

Now, the lines that are not highlighted in the demo files are the ones that do not appear in the monolithic version. These lines are there for the purpose of putting structure on the code. You will see that they define subunits such as classes, function declarations, and function definitions. Now go back to `sort.cpp` and mark the lines that belong to the same subunit in the demo program. For example, you might use the notation "sd" to mark all of the lines in `sort.cpp` that came from the definition of `DataPack::sortData()`. I don't care what notation you use as long as it is clear and unambiguous.

After you have identified the code-structuring parts of the demo program, write a brief paragraph on each, describing how its use contributes (or not) to the goals of modular, reusable, robust programming. Also, for any lines that appear in both versions of the code but with modifications, describe why the monolithic version of the program would not work if the lines were the same as in the demo program.

2.2 Programming Part

You are to modify the demo program (*not* the monolithic code—I do want you to succeed in getting your program to work) to change its behavior in two ways:

1. You should remove the limitation on the size of the file that the program can handle. To do this, your program should check n before inserting a new number into the `DataPack`. If the `DataPack` is full, it should allocate a new array of size to hold `2*max` elements of type `BT`, copy the contents of `store` into the new array, delete the old array, make `store` point to the new array, and adjust `max` accordingly. Now the `DataPack` is no longer full and data reading can continue. For this assignment, `LENGTH` should be set to 4 so that the code for expanding the `DataPack` can be easily tested. (In practice, one might well wish to start with a considerably larger initial size.)

   I have described the algorithm for expanding the `DataPack`. I will leave it to you to decide how this code should be integrated into the `DataPack` class in as clean and modular a way as possible. In thinking about where to put the code, you should ask yourself the following questions:

   - Where in the existing code is data inserted into the `DataPack`?
   - Is the code for expanding the `DataPack` related to the code that is inserting the data, or are they logically distinct?

- Can you imagine future extensions that would also need to expand the `DataPack`? If so, how should you write the expansion code so as to avoid code duplication, both now and in the future?

2. You should repurpose the code to solve the following. Instead of reading one file into one instance of a `DataPack`, your new program will read two files, each into its own instance of a `DataPack`. It will then sort both, find those elements that are common to both files, put them into a third `DataPack`, and print it out. The program should print both files, before and after sorting as the original demo does, followed by a printout of the `DataPack` containing the common elements. Then it should clean up after itself, deleting all storage that has been explicitly allocated using `new`.

   Common elements of two sorted lists can be found by a simple merge. Namely, compare the first element of each list. If they are the same, then a common element has been identified which should be put into the third `DataPack`. Discard the common element (and any duplicates) from both lists and continue. If they are different, discard the smaller element from its list and continue. Stop when either list is exhausted.

   The code for carrying out the merge should go into a member function in either the `DataPack` class or a new class. It should *not* be put into `main()` or into a global (`C`-style) function. You should give some thought to this design decision and explain it in the notes accompanying your submission.

   As part of your testing, you should run your code under `valgrind` to make sure that there are no storage leaks.

3 Deliverables

You will probably find it most convenient to submit the written part of your assignment on paper rather than electronically unless you have annotation software available. Alternatively, you can submit *legible* PDF scans of your paper solutions. Papers can be submitted in class, given directly to the TA, or put in my AKW mailbox, #408 (in which case you should write the date and time of submission on the paper). *Remember to write your name on your papers and to staple or clip them together.*

You should subit the programming part of this assignment using the `submit` script that you will find in `/c/cs427/bin/` on the Zoo. Remember to submit the following items:

1. Source code and header files.

2. A `makefile` and any other files necessary to build your project (If you're using Eclipse with the default settings, the makefiles will be found in the directory `Debug`.).

3. One or more test files and corresponding output files showing that your program correctly implements both of the required extensions.

4. A brief report named `report.txt` (or any other common format such as `.pdf`) describing the design choices you made in implementing the required code extensions. You can also put any other information here that the TA should know about your program.