Problem Set 1

Due before midnight on Friday, September 16, 2011.

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-InsertionSortMonolith`, and files `main.cpp`, `datapack.hpp`, and `datapack.cpp` from `02-InsertionSortCpp`. 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

The code in `02-InsertionSortCpp` reads in a file of floats and prints it out in both file order and in increasing order. You are to repurpose this code to get a program that reads in a file of baseball statistics and prints it out in both file order and in *decreasing* order of batting average.

A sample statistics data file `data.csv` is provided in the Zoo directory `/c/cs427/assignments/ps1`. This file consists of two kinds of lines:

1. Comment lines, which begin with a `'#'` character.

2. Stat lines, which consist of three tab-delimited fields: name, batting average, and year. Each describes a player's batting performance in the given year.[1]

In the original code, each data record consisted of a single `float`. In the stats file, each record contains three fields. You will need to define a `class Player` in which to store a record. Just as a `float` can be read and printed using the C++ I/O operators `>>` and `<<`, a `Player` can be read and printed using those same operators *after* you have defined them appropriately. Similarly, you will need to define the comparison operator `<=` to compare two `Player`s for use in sorting them. Since the `sort` function in `DataPack` puts the smallest element first, then the "smallest" player should be the one with the *largest* batting average. Given two players with the same batting average, the "smaller" should be the one whose

[1] Data obtained from URL http://www.baseball-reference.com/leaders/batting_avg_season.shtml.

name occurs first in alphabetical order. Given two players with the same batting average and same name, then they should be ordered by year.

I have given you the header file `player.hpp` in `/c/cs427/assignments/ps1` for `class Player`. Other than defining the implementation file `player.cpp`, you should make *almost no changes* to any of the other files. In particular, the only file that needs to be changed at all is `datapack.hpp`, where three small changes are required:

1. You will need to change the `typedef` for `BT`;
2. You will need to include the new header file `player.hpp`;
3. You will need to make the length of the `DataPack` large enough to accommodate the data file, which contains 100 records.

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

3 Programming Notes

There are two ways to store a string in `C++` . First is as a `char` array as one does in `C` . The other is to use the standard `string` library. To use it, you must put

```
#include <string>
```

at the top of your code. This makes `string` available as a new type, which you can use to declare variables and parameters. A variable of type `string` acts just like one would expect. You can store a string literal of any length into it. You can copy one string to another using the assignment operator. You can print it using the `<<` operator.

For this problem, you want to read all of the characters from the beginning of a line up to the first tab character as a single string and store it in the `name` field of a `Player`. This can be done with one function call using the 3-argument form of `string::getline()`, where the third argument is set to the tab character `'\t'`. (See `getline()` documentation.)

The other problem you will face is to discard comment lines—lines that begin with `'#'`. There is a new function `cleanline()` in `tools` that skips over the rest of the current input line. After calling `cleanline()`, the next character read from the input stream is the first character of the following line.

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

1. All source code and header files needed to build your project, including copies of my files that you have not changed.

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 its requirements. You might need to make up some dummy test files to ensure that the program takes reasonable action on bad files (e.g., reporting an error and quitting gracefully as opposed to going into an infinite loop or crashing).

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