Spring 2009 CS 32

**Programming Assignment 4  
Anagrams**

**Time due: 9:00 PM Thursday, June 4**

An anagram of a collection of letters is a word that is a rearrangement of all the letters in that collection. For example, anagrams of "idte" are "diet", "edit", and "tide". An anagram of "importunate" is "permutation". An anagram of "excitation" is "intoxicate".

Here is the interface for a class that stores words and lets you find anagrams:

class Dictionary

{

public:

Dictionary();

~Dictionary();

void insert(const std::string& word);

void lookup(const std::string& letters,

void callback(const std::string&)) const;

};

As you would expect, the constructor creates an empty dictionary, and the insert function adds a word to the dictionary. The lookup function takes a string and a *callback* function, and for every word in the dictionary that is an anagram of the letters in the string, it calls the callback function with that word as an argument. (This should be clearer when you examine the code we provide.)

We have written [a correct but horridly inefficient Dictionary implementation](http://www.cs.ucla.edu/classes/spring09/cs32/Projects/4/anagrams.zip). Your assignment is to write a more efficient correct implementation. If you wish, you can do this by starting with our implementation, and changing the data structures and algorithms that implement the class and its member functions. You may add, change, or delete classes or functions if you want to. Correctness will count for 40% of your score, although if you turn in a correct implementation that is no faster than the horribly inefficient one we gave you, you will get zero correctness points (since you could just as well have turned in that same horribly inefficient program).

Of course, we have to give you some assumptions about the way clients will use Dictionary so that you know what needs to be faster. The client may call insert tens of thousands of times. The collection of letters for which we want to find all the anagrams is typically about four to eight letters, but may well be more than that. Your program should be able to process thousands of requests for anagrams quickly.

Performance will count for 50% of your score (and your report for the remaining 10%). To earn any performance points, your program must be correct. (Who cares how fast a program is if it produces incorrect results?) This is a critical requirement — you *must* be certain your program produces the correct results and does not do anything undefined. Given that you are starting from a correct program, this should be easier than if you had to start from scratch. The faster your program performs on the tests we make, the better your performance score.

When we do the performance tests on your program, we will start the clock, create a dictionary, insert a bunch of words, do a lot of lookups, destroy the dictionary, and stop the clock. This means that although you have to correctly handle the insert function being called after the lookup function, such an insertion doesn't have to be particularly efficient.

To give you a taste of how fast the program can be, we did this project and produced [this executable program](http://www.cs.ucla.edu/classes/spring09/cs32/Projects/4/fastanagrams.exe). Put it in the same directory as the words.txt file we supply in anagrams.zip, and try the program on words.txt with an collection of letters like "intoxicate" or "Veronica Snot".

By default, when you build an executable file, it builds it in the Debug configuration. This is nice when you're not yet sure your program is correct, because the compiler inserts extra code to support giving you extra information if your program crashes. This extra code takes time to execute, however.

When you are sure your program is correct, and you're ready to test its speed, you'll want to build your program in the Release configuration. To do this in Visual C++,

1. Add the following two lines to the very top of the file Dictionary.cpp, *before* any #include lines:
2. #define \_HAS\_ITERATOR\_DEBUGGING 0
3. #define \_SECURE\_SCL 0
4. In the **Build** menu, select **Configuration Manager**.
5. In the drop-down list under **Active Solution Configuration**, select **Release** instead of **Debug**, and then close that dialog.
6. To run your program, you'll want to Start without Debugging.

Here are some requirements your program must meet:

* You must not change Dictionary.h in any way. (In fact, you will not turn that file in; when we test your program, we'll use ours.) You can change Dictionary.cpp however you want, subject to this spec. (Notice that by giving Dictionary just one private data member, a pointer to a DictionaryImpl object (which you can define however you want in Dictionary.cpp), we have given you free rein for the implementation while still preventing you from changing the interface in any way. This is an example of what is known as the [pimpl idiom](http://www.gotw.ca/gotw/024.htm) (from "**p**ointer-to-**impl**ementation").
* You may design interesting data structures of your own, but the only containers from the C++ standard library you may use are vector, list, stack, and queuestring). If you want anything fancier, implement it yourself. (It'll be a good exercise for you.) Although we're limiting your use of *containers* from the library, you are free to use library *algorithms* (e.g., sort).
* The Dictionary::lookup function must call the provided callback function once for every word that is an anagram of the given collection of letters. If more than one word is an anagram of that collection, the order in which those words are passed to the successive calls to the callback function is up to you. For example,
* Dictionary dict;
* dict.insert("cat");
* dict.insert("act");
* dict.lookup("cat", f); // void f(const string& s);

will call either f("cat") followed by f("act"), or f("act") followed by f("cat"), your choice.

* Duplicate words are allowed in the input, and each instance of a duplicate counts as an anagram. For example,
* Dictionary dict;
* dict.insert("cow");
* dict.insert("cow");
* dict.lookup("woc", f); // void f(const string& s);

will call f("cow") twice, since "cow" appears twice in the dictionary. This requirement makes things easier for you, since there's nothing special for you to check.

* A project consisting of your Dictionary.cpp, and our Dictionary.h and testDictionary.cpp from our inefficient implementation must build correctly, and when executed, must run to completion without error.
* During execution, your program must not perform any undefined actions, such as dereferencing a NULL or uninitialized pointer.

**Turn it in**

By Wednesday, June 3, there will be a link on the class webpage that will enable you to turn in your source files and report. You will turn in a zip file containing two files:

* Dictionary.cpp. You will not turn in Dictionary.h or a main routine, since we'll supply our own.
* report.doc, report.docx, or report.txt, a report containing
  + a description of your algorithms and data structures (good diagrams may help reduce the amount you have to write), and why you made the choices you did. You can assume we know all the data structures and algorithms discussed in class and their names.
  + [pseudocode](http://www.cs.ucla.edu/classes/spring09/cs32/pseudocode.html) for non-trivial algorithms.
  + a note about any known bugs, serious inefficiencies, or notable problems you had.

Your report will count for 10% of your score. We'll be reading it with this in mind: If you were standing around a whiteboard talking about your design and gave this description, would a CS grad student know enough about what you had in mind to be able to code it up without asking you any further questions?