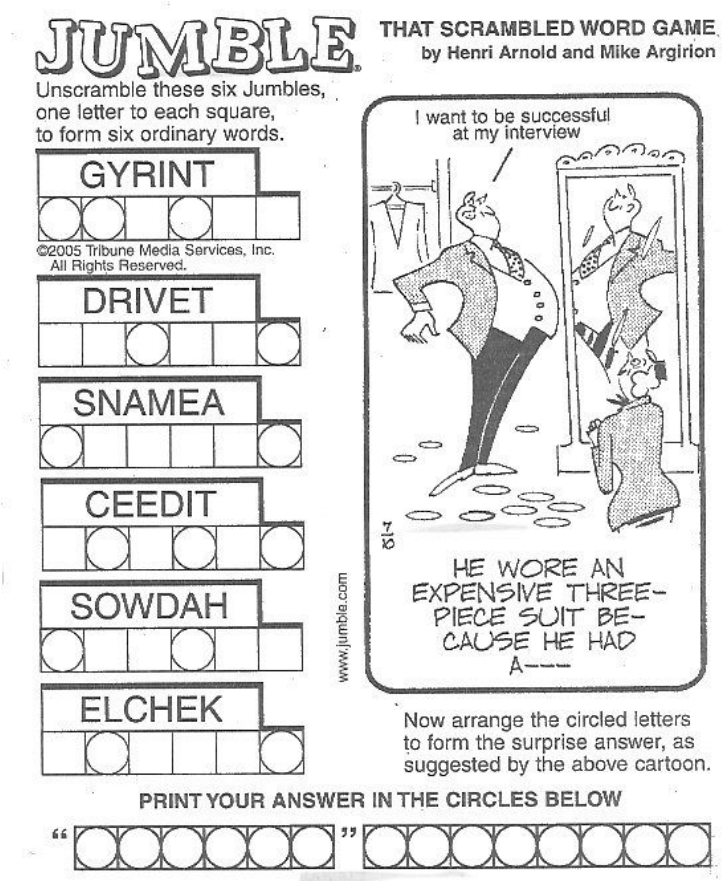
CS251 -
A Jumble Solving Program
Due: Mon, April 22 at 11:59pm

NO LATE SUBMISSIONS FOR THIS ASSIGNMENT

BIG PICTURE:

- This is a "mini-program" assignment. It will be assigned a weight fo ½ that of a "full" programming assignment.
- The assignment uses hash-tables / hash-maps. However, you will not be implementing a hash table data structure yourself; instead, you will use the `unordered_map` class from the Standard Template Library (STL).
- Unlike most other CS251 programming assignments, your deliverable will be a complete working program which you will write from scratch.
- You will submit a single file `jumble.cpp`

In this project you will use a clever application of hash tables to solve "word jumble" puzzles. You have probably seen these puzzles in the newspaper. An example is below.



Your program will implement a strategy described below in the section labeled Algorithm. But first, let's see how the program should behave

from the user's viewpoint.

Overview of Program Behavior

The program is called jumble and requires a single command-line argument specifying a dictionary file. For example:

```
./jumble dictionary-small.txt
```

(You have been given two dictionary files dictionary-small.txt and dictionary-big.txt; each of them contains just a bunch of English words -- over 250k words in the case of dictionary-big.txt).

After the program sets up its internal data structures, it enters an simple interactive loop which behaves as follows:

I. The user does one of the following:
   - **A.** enter a string of characters (presumably a jumbled version of one or more English words).
   - **B.** or types ctrl-d to terminate the interactive loop.

II. If the user enters a string (1a above), a list of *all* English words in the given dictionary that are rearrangements (anagrams) of the user input. The list can appear in any order. If there are no such English words, the program reports "no matches." The user is then prompted for another input.

III. When the user terminates the interactive loop (1b above), the program produces a report with the following information and then the program terminates:
   - The **number of words read** from the input file.
   - The **number of "equivalence-classes"**: every word in the input file belongs to exactly one subset of the dictionary where all words in the subset are rearrangements of each other. For instance, "stake", "skate", "steak", "takes" all belong to the same equivalence class. This part of the report gives the number equivalence classes formed by the words in the dictionary.
   - The **size of the largest equivalence class**.
   - The **"key" associated with the largest equivalence class** (see the Algorithm section).
   - Finally, the **members of the largest equivalence class**. The ordering of the members can be arbitrary as long as all members are listed.

Example run (using the given dictionary file dictionary-small.txt):

```
$ ./jumble dictionary-small.txt
reading input file...
start entering jumbled words (ctrl-d to terminate)

> islme
English Anagrams Found:
    miles
    slime
    smile
> retumpoc
English Anagrams Found:
    computer
> crleov
English Anagrams Found:
    clover
> able
English Anagrams Found:
    able
    bale
> abinst
no anagrams found...try again
> ilpsl
English Anagrams Found:
    spill
> atrce
English Anagrams Found:
    cater
    crate
    react
    trace
> REPORT:

  num_words             :  21876
  num_classes           :  21086
  size-of-largest-class :  4
  largest-class key     :  'aprt'
  members of largest class:

   'part'
   'rapt'
   'tarp'
   'trap'
Johns-MacBook-Air:jumble-c++ lillis$
```

Algorithm

You are required to implement a specific Hash-Table/Hash-Map based algorithm to solve this problem. You will write a program which utilizes a standard C++ HashMap implementation in the Standard Template Library: the unordered_map class.

A Strawman.
For sake of argument, suppose we built a Hash-Map from the given dictionary where each word in the dictionary becomes an entry in the Hash-Map (as a "KEY"). A Hash-Map like this works great for applications like spell-checking. But does it help us with the Jumble problem? Not much it seems.

For a given jumbled sequence of letters, we *could* enumerate all re-orderings of the sequence and for each of them, check to see if it is in the dictionary.

For example, if we were given "kstae" as in the previous discussion, how many re-orderings would we end up enumerating? The string is of length-5 and no letters are duplicated. Thus there are $5! = 120$ re-orderings we would enumerate (and 120 queries to the Hash Map).

If the given jumbled string is length 6 and all letters are distinct, then we would enumerate all $6! = 720$ reorderings. For example suppose the given jumbled string is "layber" which can be rearranged to form the words "barely", "barley" and "bleary" and we would have 717 queries to the hash-map that would fail.

So...this approach doesn't seem especially elegant or efficient.

Can we avoid explicit enumeration of all reorderings of the given string?

Yes.

A Better Approach.

Short summary of algorithm: From a given dictionary, build a Hash-Map in which

- the keys are sequences of characters/letters *in sorted order;* the

keys are not in general themselves words and
- the value associated with each such key is a list of all words in the dictionary that are rearrangements of the value.

Each "value" in the Hash-Map is a list of dictionary words which are rearrangements of each other (anagrams) and each such list has the letter-level sort of these words as the associated "key."

You can think of each key as a canonical ordering of a collection its associated value as the list of dictionary words which can be formed by rearranging the key.

Thus, to find all words that can be formed from a given string, we first sort the letters of the given string and use it as a key to look up the list of words that can be formed from those letters (if any).

More Discussion

Two strings **s1** and **s2** are rearrangements of each other if and only iff **s1** and **s2** are composed of exactly the same letters and with the exact same frequency. For example **"aekst", "stake"** are rearrangements of each other is TRUE. However **"apple", "aalpe"** are not rearrangements of each -- both strings have the same letters, but not in the same frequency.

Now consider any two words that are rearrangements of each other -- like "stake" and "takes". Now consider the letter-level sorts of each of them. Let **sort(s)** represent the the rearrangement of string **s** such that the individual letters are in sorted order (for example, sort("happy") = "ahppy"**):**

sort("stake"): "aekst"
sort("takes"): "aekst"

It shouldn't come as a surprise that they both result in the same string when letter-level sorted -- after all, they have exactly the same letters in the same frequency.

So instead of storing actual words as "keys" in your hash table you will store sorted re-orderings (as the "key") and a list of all words that have the same sorted ordering (as the "value"). In other words, the table is implementing a function from sorted strings to sets of English words that are formed from exactly those letters.

For example, the anagrams of "takes" might be represented as the following key-value pair in the hashmap:

```
((KEY:  "aekst")
 (VALUE:  ("stake", "takes", "steak", "skate"))
```

Mathematically, the HashMap partitions the dictionary into “equivalence classes” or words that are anagrams of each other (as mentioned above). Each such equivalence class has a unique “id string” -- their letter-level sorted version.

When the user enters a jumbled sequence of letters, you then sort the string and do a lookup on it to retrieve all words that can be formed by rearranging the letters.

For example, if the user enters "ketsa", you would use the sorted version "aekst" as the key for a hash table lookup which would return the English words “steak”, “stake”, “skate” and “takes”

Implementation

Unlike other programming assignments we have done, your deliverable is a complete program. Your program will act as a client of classes already available to you (vector is an example).

Some things that are relevant:

- Command line arguments: as described in Program Behavior, a dictionary file is specified when the program is run through a command line argument.
- Opening and reading from files.
- Reading user input.
- Working with the unordered_map class.
- using the std::sort library routine on strings.

Example Code: You have been given a small program called **freq.cpp** which illustrates all of these concepts through an application that reads in a text file and builds a HashMap (unordered_map) which keeps track of all distinct strings in the input file *and*, for each such string, the number of times it appears in the input file.

This information is naturally represented by a map from strings to integers.

Some Notes and Assumptions

- Your program ***must*** behave as described above. Your may not, for example, have the program prompt the user for the dictionary file -- it must be given as a command line argument.
- Upper-Case vs. Lower-Case -- not an issue at all! To keep things simple, we will assume that upper case and lower case letters are distinguishable. For example, dictionary-small.txt contains "ATM"; we will ***not*** consider it to be an anagram of "mat". Thus, you just treat the letters as they are given.
- You may also assume that your dictionary file does not contain duplicates - no special processing to skip duplicates.

Deliverable:

Your deliverable is just a single program file called **jumble.cpp**. Please use exactly this name.

We should be able to create an executable from your program by running g++ as:

```
g++ -std=c++11 jumble.cpp -o jumble
```

Final Comments:

Your program should not turn out to be very long -- maybe a little over 100 lines of code. Have fun and do some exploring of the C++ libraries.

Command line args. Your program will be called jumble will have one required command line argument -- the dictionary file name. Example:

```
$ jumble dict.txt
```

The dictionary file will simply contain one word per line for easy parsing.
