CSE 232 Spring 2016

Programming Project #6

Assignment Overview:

Create a book cipher. Experience with files, vectors, strings and maps.,
The assignment is worth 40 points (4% of the course grades), and must be completed before 11:59 pm on Tuesday, March 15th (because of spring break).

Background:

A Book Cipher (https://en.wikipedia.org/wiki/Book_cipher) is another kind of cipher that uses some particular text as a key. It is also a homophonic cipher (https://en.wikipedia.org/wiki/Substitution_cipher#Homophonic_substitution) as it can use multiple cipher symbols to represent each letter/word.

There are two kinds of book ciphers: **word based** ciphers and **letter based** ciphers. In a word based cipher, a particular word is encoded as a wod location in a book text. Thus you might encode a word as a combination page:line:word , that is the word is on a page, on a line on that page, a particular word on that line. To make that work you need to have a pretty good sized text to make sure you have the words you need. In a letter based cipher, you can work with a much smaller text, and each cipher encoding uses the first letter of a particular word to help spell the encoded word. This is also homophonic, but can make the cipher text very long.

Choosing books

We are going to do a word based book cipher based on a provided text. We can use nearly any text, but I have chosen David Thoreau's "Walden" as copied from Project Gutenberg (http://www.gutenberg.org/files/205/205-0.txt) and the Gettysburg Address. Both are included in the project directory.

For "Walden" I took the text file "as is", with whatever was included. This is pretty important for a book cipher as you have to agree with your contacts the exact text you are going to use. This particular version includes stuff from Project Gutenberg (its license for example) as well as the essay "On the Duty of Civil Disobedience". I changed nothing from that download. Take a look at it. It has a lot of blank lines, lots of punctuation such as stars, commas, periods, semicolons etc., words in all caps, words that are Capitalized, all kinds of things. There are misspellings, we accept that.

For the Gettysburg address, I took just the address itself as text. This is because Gutenberg adds lots of text as part of its legal process and I just wanted a very short file, for your sake mostly.

A Split Function

We have written a `split` function before, and for the most part it would work on the Gettysburg address. However, parsing a line into words can be a little challenging with Walden and will require a more robust `split` function. See the specification in the Program Overview section.

Walden has lines such as the following (I added the line numbers, assume starting from line 0 of the raw text):

```
1411 few are able to tell exactly what their houses cost, and fewer still, if
1412 any, the separate cost of the various materials which compose them:--
1413
1414   Boards.......................... $ 8.03-1/2, mostly shanty boards.
1415   Refuse shingles for roof sides...  4.00
1416   Laths............................  1.25
```

We will **only collect alphabetic** words. We will not collect numbers, and when we collect words that occur with punctuation, we will strip the punctuation from either end. For example, `"them:--"` at the end of 1412 should be turned into the word `"them"`, Line 1414 should yield only the words: `"boards", "mostly", "shanty", "boards"`. Notice the second `"boards"` has the period stripped off. The only non-alphabetic character exception is an apostrophe for a contraction, Thoreau does use `"don't",` and possessives (`"men's", "another's", "mountain's"`) . These we will keep as is.

Program Overview

As there are no pages in a text file, we will work with only line counts and word counts. To stay in tune with C++, line count starts at 0 and the word count in each line starts at 0.

1. **Initialization**: Process the file gberg-walden.txt (Gutenberg version of Walden) as follows:
    a. read in each line and remember the line number, starting at 0
    b. split the line into individual words
        i. for each word remember which word it is in the line. You already know the line number from a.
    c. strip all punctuation and non-alnum characters from the end/beginning of each word
        i. lower case each word
    d. add each word to an `encode_map`. The key to the `encode_map` is the processed word, the value of the `encode_map` is a `vector<pair<long,long>>`. Each word has a `pair<long,long>` value where the pair's first long is the line number and the pair's second long is the word number in that line. As there will be repeats of words, making this a homophonic cipher (which was the point), we need a vector to store all occurrences of each word.
    e. you will also add the word to the `decode_map`. The key to the `decode_map` is the `pair<long, long>` from the word as above, and the value is the word.
2. **Encoding**: Encode the `to_encode` string using the `encode_map`. For each word in the `to_encode` string
    a. see if the `to_encode` word is in the `encode_map`. If so, encode that word in the following form `"line_num:word_num"` .
    b. If there are multiple entries, ***pick the first entry*** (the first one encountered in the text, in first in the `vector<pair<long,long>>` for the word). That is counter to the idea of a good cipher but makes grading and testing easier.
    c. if the `to_encode` word is not in the `encode_map`, it is ***skipped*** and not encoded

3. **Decoding**: Decode the `to_decode` string by looking for the `"line_num:word_num"` as a `pair<long,long>` in the `decode_map`. It should be there, and when found yields the appropriate word.

Program Specifications:

- `void split(string line, vector<string>&words, string good_chars)`
  - Find words in the line that consist of `good_chars`. Any other character is considered a separator.
  - Once you have a word, convert all the characters to lower case.
  - You then push each word onto the reference vector words.

  Important: `split` goes in its own file. This is both for your own benefit, you can reuse `split`, and for grading purposes. We will provide a `split.h` for you.

- ```
  void process_file(string filename,
                    map<string, vector<pair<long,long>>>& encode_map,
                    map<pair<long,long>, string>& decode_map)
  ```
  Process the file as indicated in step 1 **Initialization** above.

- ```
  string encode(string to_encode,
                map<string, vector<pair<long,long>>>& encode_map)
  ```
  Encode `to_encode` using the `encode_map` as described above in **Encoding**. Returns the encoded string as a sequence of space separated elements of the form `"line_num:word_num"`

- ```
  string decode (string to_decode,
                 map<pair<long,long>, string>& decode_map)
  ```
  Decode `to_decode` using the `decode_map` yielding a string of space separated words as described above in **Decoding**.

Deliverables

You must use handin to turn in a ***two files*** called `functions-06.cpp` and `split.cpp`.
The files `main-06.cpp`, `functions-06.h` and `split.h` are provided for you. Please be sure to use the specified file names, and save a copy of both `functions-06.cpp` and `split.cpp` to your H drive as a backup.
You will electronically submit a copy of the file using the "handin" program.

Assignment notes:

1. Some tests are provided in the `main` function
2. There are more than 10,000 lines and 100,000 words in the walden.txt. If you start out testing with the full text, well you will have a very hard time. Instead start with the easier Gettysburg address (gburg.txt) as an example: 25 lines and 270 or so words. Much smaller, easier to test against. Note however, will test against both in the end
3. You clearly need to split a line of text into words, but it will have to be a little more robust than the one you did in lab. Up to you how to write that.
  ```

    a. some of those unusual "find" algorithms might be useful. Take a look at string methods `find_first_of` and `find_first_not_of` and their kin
4. In fact, you may need lots of other functions. You can place them in `functions-06.cpp` and **not in** `functions-06.h` (you can't change `functions-06.h`) meaning that they are private and cannot be used in the `main` program. They might include (but up to you):
    a. split function
    b. strip function
    c. function to print your map (or some part of your map) to check yourself