Book Indexing

Due Date:

Milestone 1: Due Monday Feb 2, 2015 - the input file parsed and output to the terminal, each word on a separate line. Pushed to github (get tag name from TA)

Complete Project: Monday Feb 9, 2015 - pushed to github (get tag name from TA)

Introduction

Professor Johnson just got assigned to be the editor of a riveting textbook titled Advanced Data Structure Implementation and Analysis. She is super excited about the possibility of delving into the material and checking it for technical correctness. Another of her tasks is to create an index for the book.

Everyone has used the index at the back of a book before. An index organizes important words or phrases in alphabetical order along with a “pointer” (not in the C++ sense, but in the index sense) to where that phrase can be found in the book. In most books, this “pointer” consists of a list of page numbers where that particular word or phrase can be found in the text. But, who or what creates these indexes? Are they created by humans? Are they created by computers?

Professor Johnson knows that an automated indexer is the way to go (not only because its faster, but because it can be reused later when she finishes writing her own book), but she just doesn't have time to edit the book and create the auto-indexer software in the short time frame. She has asked for your help! In this project, you will implement a program that can help Professor Johnson index a text.

Your Task

You will implement a piece of software which can read in a book file (raw ASCII text with page indications) and a category file, process the book data using the category file, and output the complete index to separate file.

Implementation Details

The **input text file** will contain the text from the book to be indexed organized by pages from the book. The end of the book will be signaled by [-1] at the end of the file. All punctuation will have been removed from the file, but capitalization will still be the way it was in the original book. Here is a (very very) simple input text file:

```
[1]
This is the text that appears on page one of the book Note that there is no
[2]
punctuation anywhere but that sentence-based capitalization is still present
[3]
You only need to worry about indexing individual words not phrases
[4]
Lorem ipsum dolor sit amet consectetur adipiscing elit Phasellus condimentum
[-1]
```

Important points about the input file:

- No word in the input file will exceed 30 characters in length.
- Page numbers may not necessarily always be in order.
- No page number will be repeated.
- Indexing needs to be case-insensitive. In other words, cat and Cat should be considered the same word.

The **category file** will contain information on categories to handle when indexing. You have seen this in indexes before where it might say something like:

For spaniel, see dog.

Each entry in the list will be organized in the following way:

```
category = word1 word2 word3 word4
```

This means that where **word1** would appear in your index, you would output **"For word1, see category"**. Additionally, the pages where word1 appears should be associated with the list for category in your implementation. Here is a sample category file:

```
apple = iphone ipad macbook
automobile = car truck toyota ford
```

The output file will be organized alphabetically. Each letter will appear in square brackets followed index entries that start with that letter in ascending alphabetic order. An index entry will consist of the indexed word, a colon, then a list of page numbers where that word was found in ascending order OR a reference to that word's category. No output line should be longer than 50 characters. The line should wrap before 50 characters and subsequent lines for that index entry should be indented 4 spaces.

Here is an example output file:

```
[A]
apple: 4, 12, 66, 131
asymptotic: 44, 45, 46
[B]
binary: 25, 26, 29, 88
[C]
comeonthisisacrazylongword: 13, 14, 16, 33, 101,
    102, 103, 105
[I]
For ipad, see apple.
```

Professor Johnson is a purist and doesn't trust many of the container classes and algorithms from the STL. **Therefore, she has instructed you to not use any of them. This includes her aversion to string objects.** However, through much pressuring from her students and colleagues, she has come to accept and trust the streaming libraries that are part of the STL (iostream, fstream, etc.). Johnson trusts your skills though, so she encourages you to implement your own container class(es).

Assumptions

You may make the following simplifying assumptions in your project:

- The input files (text and category) will be properly formatted according to the rules above
- Punctuation will already be removed from the text file
- No need to worry about indexing phrases (this is much harder to do)
- No line of text in either file will contain more than 80 characters
- No word will be longer than 30 characters
- Hyphenated words may exist in the files, but are to be considered simply one word (**sentence-based** is an individual word that is separate from the word **sentence** and the word **based** and would have its own index entry unless it was part of a category).
- No terms that are in one category will be in another category (e.g. ipad won't appear in the apple category and also the tablet category)
- Categories are only one level deep (e.g. a category term will not also be in the list of specific terms for another category)
- Categorical terms may also appear in the text
- You don't have to worry about various forms of the same word (e.g. run, runs, and running would each be considered individual words)

Execution

The executable for this project will be run from the command line with three arguments:

- the name of the input text file,
- the name of the category file, and
- the name of the output file to write the index to.

Example:

```
prompt$ ./indexer book.dat categ.txt index.output
```

Implementation Details:

You don't have any idea how many individual words, index entries, etc. will be present in the input data file. And since Johnson doesn't like the container classes from the STL, you cannot make use of the amazing vector class that automatically grows as you insert elements into it. You'll need to implement some “data structure” that is capable of “growing” as needed. Using dynamic memory allocation smartly, you can simulate the idea of an array growing in capacity. Therefore, this would allow you to handle really small books as well as really large books without being insensitive to memory usage issues. There is no need (and you definitely shouldn't) allocate arrays with 50,000 elements and cross your fingers in hopes that you'll not encounter 50,001 items that need to go into your array.

In particular, you'll implement functionality that will resize your arrays as needed for the various different dimensions of your data structure (such as words, pages, page list, etc.). Figure 1 shows an overview of a potential memory layout for this project:

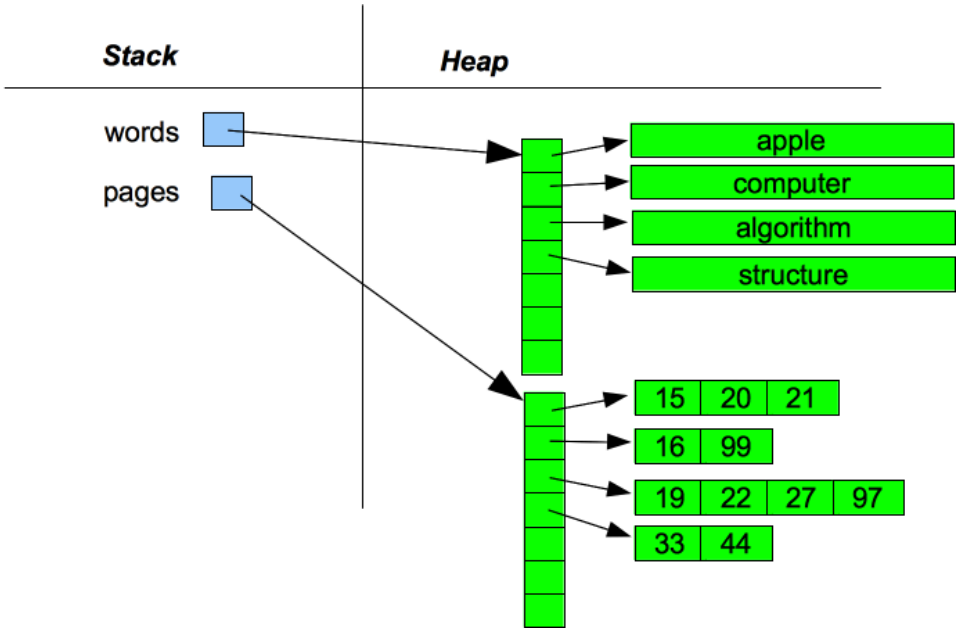

Figure 1 - Memory diagram for project

At a minimum, your implementation should contain a pointer to a char pointer (char**) and a pointer to an int pointer (int**) at a minimum. You may have other data members as you see fit. Your implementation will likely always have some "extra" space to store more words and their page lists. **However, you may never have more than 10 unused spaces in your data structure.**

What to Submit

You should submit:

- well formatted and documented source code
- any design documents you created up front in order to help you get started on the project
    - Keep anything you jot down while thinking about how to structure the project. Scan it, take a picture of it, or otherwise reproduce it as part of your submission.
- any sample data files you used to test your program.

Strategies for Success`

Just some friendly words of wisdom from your professor:

- The first 10% of a project is always the hardest. Don't sit down in front of an empty .cpp file hoping/waiting for inspiration. This is likely to turn into exasperation, desperation, exhaustion, etc. very quickly
- THINK BEFORE YOU CODE.
    - Design before you start. Draw class diagrams, connect the classes with lines. Brainstorm about what classes/functionality you'd need to make this

happen. Think about the major steps of processing that you'll have to go :q
- through. ***Do this step with a friend/buddy/pal/BFF that's in the class. That is completely acceptable. Challenge each other's deisgn. Critique, question, explore.***
- Consider the analogue of writing a paper by starting with an outline. After reading this handout in detail, what are the big "roman numeral" things that have to get done. Try to keep the list to 5 or less big tasks. Write them down (or type them in to a Word doc). Break each of them into smaller tasks.
- When coding, THINK BEFORE YOU TYPE. You don't want a carpenter to start randomly putting nails in walls or drilling holes in your ceiling before they measure, remeasure, think about it, etc. Don't just mindlessly write code. Be intentional about every line you write.

Your TA's will also give you their guidance in each of the respective labs. Please don't dismiss our suggestions; they come from experience of making many mistakes. This is a completely do-able project in the time frame you've been given as long as you use your time wisely.
vim

	Points Possible	**Points Awarded**
Correct Underlying Data Structure Implementation	30	
Complete functionality implemented	30	
Dynamic memory managed correctly	20	
Proper class infrastructure (constructors, destructors, accessors, mutators, etc.) and design	10	
Documentation, formatting, comments	10	