CISC 105 Spring 2006 Project 2 beta

Due Thursday, May 11th, 11:59 pm

on MyCourses, paper in class Friday.

This project is about using structs to do word counts and bi-gram frequencies.

Check this directory periodically for updates. When I get questions from students that I think are of interest to the whole class I will put them in an FAQ file in this project's directory.

Overview

The first part of the project consists of reading in an 850 word vocabulary of Basic English, an artificial "constructed" language created by C.K. Ogden in 1934 because he thought a simplified version of English would be simple to teach. Similar languages were used by various military and political groups (especially in World War II) for radio broadcasts to non-fluent English speakers. We use it here because there are texts available in this language, and an array of size 850^2 will fit easily on Strauss (this only matters for the last parts). Your program should be written so that it can easily be used with a vocabulary of different size (there are Klingon vocabularies and texts online, too!).

After reading in the vocabulary, your program will look at the text of *Gulliver's Travels* by Jonathan Swift and possibly other texts. As it walks through the text, it will find each word it encounters in the vocabulary array and update the counter for that word by one. When the end of the story is reached, you will have frequency counts for every word in Basic English from this text. You will demonstrate sorting the vocabulary by alphabetical order and by frequency.

When Dr. McCoy visited our class, she spoke of the importance of word prediction to augmentative communication. A very simple, but reasonably effective, way to predict words is to compute bi-gram frequencies. Bi-gram is a fancy name for a word pair (there are also tri-grams and *n*-grams.) To find bi-gram frequencies, count the number of times each word *pair* occurs. If you find that the count of the pair "apple butter" is the highest count pair beginning with "apple", then when someone types in "apple" it might be reasonable to predict that the next word will be "butter".

As you code, remember to write small functions and test them thoroughly. You must use all the functions I describe, but you may also want more - feel free. Style matters! Also, there is no need to start using the real data files right away; make arrays or files of ten words for testing your program to simplify your task.

Check the FAQ in this directory frequently! You are responsible for its contents.

Task

1. (10 pts) Declare a struct to hold one word from the vocabulary file and the associated count. Make an array of structs, size 850.

2. (20 pts) Write a function to read the words from the vocabulary file into the struct array, and initialize all word counts to zero. The function will take an open file pointer, the struct array, and the array size as parameters (why?).

3. (5 pts) Write a function to print the vocabulary array with the word counts. Put five words or more on a line, but your lines should not wrap at all.

4. (20 pts) Write a function to read the data file. You will give the data file name as command line argument to main (as in lab 9). **Do not** store words from the data file in an array. Instead, read one word, find its struct by traversing the struct array, then increment the counter inside that struct. If a word is not in your vocabulary array (there will be lots of these, since finding plural forms is hard), print a message that, for example, the word "sailors" is not found (What about "sailor"? Why?). You could do all of this in one function, but why not write two or three smaller, simpler ones?

5. (15 pts) Sort your array of structs using a selection sort function. Your sort function should take an extra parameter that tells it whether you want to sort alphabetically by word, or by word frequency.

6. (15 pts) Traversing the array for every word in a data file could be very time consuming. Convert your word search to a binary search.

7. (5 pts) Setting aside a large number of chars for a word in every struct in the vocabulary array is wasteful (and would be *really* wasteful if we were using a full size English vocabulary!). After carefully saving a backup copy of your working program, change your struct to hold a char pointer instead of a char array. Then as you read in each vocabulary word, check its length and allocate just enough space to hold that word. (Thought: where in memory will your array of structs be? Where will the vocabulary words themselves be?)

8. (10 pts) Bi-gram counts are maintained in a 2-d array. Because our counts will be fairly small, you will not use integers. Instead, you will use the type **short** which takes half as much space (hmm... how big a number can it hold?).

   Each cell in the array represents a word pair. Consider the sentence "The quick brown fox jumped the spam." The word "the" does not follow anything, so the first pair is "The quick". Add one to the bi-gram frequency, that is, to the cell in the array for row "the" and column "quick". Since array indices are integers, you'll need to look up the index of each word in your alpha sorted vocabulary array using binary search. After processing the whole example you'll have the array shown in Figure 1.

	the	quick	brown	fox	jumped	spam
the	0	1	0	0	0	1
quick	0	0	1	0	0	0
brown	0	0	0	1	0	0
fox	0	0	0	0	1	0
jumped	1	0	0	0	0	0
spam	0	0	0	0	0	0

Figure 1: A two dimensional array of type short for storing bi-gram frequency.

When you process the data file, every word is part of a bi-gram except the very first word and the very last.

Submission

Design and test your program carefully using data that you make up. Check the calculations! The day before the project is due you will be given new test data and a project submission sheet. You must show that your program performs correctly for all test data, and you must complete the **submission sheet** and turn it in with your project's **paper copy** (when?). Of course, code for your program must be submitted to **MyCourses** along with your script file.