15-123 Lab1: Dictionary Array

Due Date: Sunday January 23 2011 by 11:59pm

Important: You must log hours and commands that was needed to complete this assignment. This is required as part of the submission. Please write the date and hours spent on the selflog.txt in the given format. Also you SHOULD develop your program on andrew.unix.cmu.edu and BEFORE each session, type: (this will start recording your commands and output to the file session1.txt)

- **script session1.txt**

Before you end the program development session:

- **cntrl-D**

This will end the session and save all commands and output to session1.txt Do this for all sessions.

You can clean up any commands that were not part of your C program development.

All files: session1.txt, session2.txt, …., lab1-selflog.txt are REQUIRED to get 3 points. These log files together with your source code *.c files should be submitted to handin for each assignment.

Objectives:

Lab1 requires you to read and process command line arguments, open files for reading and writing, use a static 2D C array of chars to store the file of data and write a simple algorithm for insertion sort. You will also learn how to use time functions to get runtime approximations and understand strings in C to compare and sort. No dynamic memory allocation is needed in this first lab.

The Assignment:

The purpose of this program is to write a C program that reads in a text file no larger than dictionary.txt (use wc command in unix to get word count. Eg***: wc dictionary.txt)*** and stores each word in a row of a 2D char array. Since the storage array is static, you need to decide how big to make it at compile time, not runtime. No word in the file is longer than 31 characters. We know that this is a gross waste of space for a small file such as small.txt but you will get your chance at dynamic memory and efficiency shortly.

You will be using command line arguments to perform various operations on the file. For example, -s options allows you to read and sort the dictionary file into the array, *keep it sorted as you go*. **Don't load first** and **sort later**! Be sure you sort it AS you load.

Essentially do an insertion sort. **See insertion sort logic in the write up given in this folder.** You will also get a chance to look at insertion sort applied to integers in a skills lab or in a recitation. Do not copy quicksort or other sorting code from the web just to get the fastest load. You will be penalized for making it run faster ☺. You will get a chance to improve a lot in Lab2

Your program will be executed as follows using the executable code: a.out

> ./a.out <option1> <option2> file1 file2

The options are -s for sort , -t for time,
file1 is the input file.
file2 is the output file
The command must have at least 3 arguments, including the command name. Otherwise print an error message that says
a.out: Insufficient Arguments
Usage: a.out <option1> <option2> file1 file2
(follow the standard unix format)

Example runs
> **./a.out** infile outfile % read infile and write to outfile

> **./a.out** -t infile outfile % read infile and write to outfile. Output the time to STDOUT

> **./a.out** -s infile outfile % Sort the infile as you read in (do not sort after reading). Write **sorted** file to outfile.

> **./a.out** -t -s infile outfile % Read the infile, sort as you read, and write the sorted file to outfile and output the processing time to STDOUT

> **./a.out** -s -t infile outfile % same as above

You can assume no other type of a command is allowed.

No matter the order of operations (-s and –t), your program will sort first (if any), and time second (if any). The time is the total time to run the program from beginning of the main to end of the main program. The time is printed in the format (to stdout)
The time to execute this program is _____ seconds

Your book may contain good pieces of code and the **demo1.c** program could serve as a sample code file too.

Getting Started

Step 1: Download the starter files: /afs/andrew/course/15/123/downloads/lab1

(you can copy them to your folder using cp command)
Step 2: Compile lab1.c to make sure the file has no errors
Step 3: Complete the function readFile.
Step 4: Complete the function writeToFile
Step 5: Test the code with data files at /afs/andrew/course/15/123/datafiles
(do not download these input files). For example,

- **./a.out /afs/andrew/course/15/123/datafiles/10-words.txt out.txt**

Step 6: Complete the function insertInOrder. Test the code with data files as in Step 5

Segmentation Faults

Segmentation faults are part of life of a C programmer. Many things can contribute to segmentation faults. Here are some reasons:

1. The allocated array is not enough to hold the data file
2. Attempting to read/write from/to a file that is not opened
3. Many other things that I did not list here. You will learn all of those as part of the course

Expected runtimes

Actual runtime of a program depends on many factors. You generally cannot control system dependent factors such as processor speed or RAM. But you can control your algorithm. If you think about where your code is spending most of the time, you can improve your code. The slowest part of your program in this case is insertInOrder that requires you to search for a position, insert the word in order and adjust the array. Your program should run under 1 second for most data files except 42k-words, 172k-words. You can test with our solution to see if your code runs closer to sample solution runtimes. There are many ways that runtime can be improved. One of the best ways is to use a more efficient sorting algorithm such as quicksort. Although easier to implement, our insertInOrder algorithm is slow for large files. You will be penalized if your runtime is excessively high (in the order of few minutes).
Here are some of the runtimes (in seconds) we get with our solution (-s and –t options):

1. 100-words.txt - < 1 sec
2. 1000-words.txt - < 1 sec
3. 5k-words.txt - 1 sec
4. 10k-words.txt – 2 sec
5. 42k-words.txt - 25 sec
6. 172k-words.txt – 400 secs

Grading:

Your program will be graded as follows.
The following general grading criterion is strictly enforced for **ALL** assignments.

1. A program that does not compile - 0 points
2. A program that is 0-24 hours late – max grade is 50%
3. A program that is 25-48 hours late - – max grade is 20%

4. A program that is more than 48 hours late – 0 points
5. Only one late day can be used per assignment (you have a total of 3 for the semester)

Grading Programming Components

1. Correct command line argument processing – 20 points
2. correct reading and writing files – 20 points
3. correct implementation of the sorting algorithm – 40 points
4. correct use of string functions – 10 points
5. Style points – 7 points (style points are based on indentation, proper use of variable names, structure of your program, handin proper files, removal of unused variables. Your TA can provide more guidance on this. Please ask)
6. Submission of the log files– 3 points (keep track of hours on the selflog.txt file and submit at the end). You also need to submit each session log files.
   If you'd like to see an example, see the folder called **samplelogfiles** that is given with your starter code.

Solution Executable: See solution executable (named **solution**) to see how your program should run. Do not submit this executable with your code.

Commenting your code: Not necessary for *this* lab. But we will require you to comment code as we progress.

Handing in your Solution

Your solution must be in the form of a .zip file. Just hand in all the .c files in your program (This program has only one source file and the file must be named lab1.c). **Don't hand in input, obj or exe files. DO NOT handin dictionary.txt**. You need to follow these directions or you may lose style points.

You need to submit your solution through AFS. Your TA's will show you how to do this in recitation or ask. The zip file needs to be submitted to

/afs/andrew.cmu.edu/course/15/123/handin/lab1/yourid

The handin folder will not be accessible after 48 hours from the due date/time.