University of Toronto
APS105H1F — Computer Fundamentals
Lab 6: Sorting & C-Strings

Introduction

This lab will give you practice with sorting, searching, and using the C-string functions in the `cstring` function library. You will be writing a program to read in a list of words from a file, sort them, remove duplicates and then allow the user to search for words in the list.
This Lab assumes understanding of material presented in the text up to and including section 8.1. If you have not already done so, you should read about the `cctype` and `cstring` function libraries.
You should attempt to complete the assignment *before* your scheduled lab period, as many of you will find it will take more time than will be available in the lab. Please keep in mind that the marking process will begin approximately sixty minutes before the end of the lab.

Part I—Reading the Words

Write a program named `lab6` to read text from a file, storing each *word* in an array of C-strings. You may assume no word will be longer than 25 characters, and that there will be no more than 10,000 words.
An array of C-strings can be declared as

```
char words[MAX_WORDS][MAX_WORD_LENGTH+1];
```

When you do this, you can access an individual word in the array as in the following example:

```
cout << "The length of word " << i << " is " << strlen(words[i]) << endl ;
```

so it is easy to deal with entire C-strings instead of individual characters. This will be important when sorting the words later on. **Note:** not all files will contain the full 10,000 words, so in addition to keeping track of `MAX_WORDS`, you must also keep track of how many words are actually stored in the array. You will find it helpful if you declare global named constants for the sizes of the array dimensions. This will simplify writing the declarations and implementations of any functions that you pass the arrays into.
The name of the file containing the words will be passed to your program using a command-line parameter. To do this, declare your `main` function as follows:

```
int main(int argc, char *argv[])
```

Then, if you execute the program with the command

```
$ lab6 myFile.txt
```

then `argv[1]` is the name of a C-string array containing the string "`myFile.txt`", and you can open the file with

```
ifstream inFile ;
inFile.open(argv[1])
```

Since you only want to read the words, and not the spaces in between them, using the extraction operator `>>` is sufficient. You may find that some words have punctuation at the end (*e.g.*periods or close quotes), and you may find some words have punctuation at the beginning (*e.g.*start quotes)—remove any punctuation at the start or end of words. If punctuation occurs in the middle of a word, leave it (*e.g.*`can't`). For the purpose of this lab, you may consider any character to be punctuation if it is not a letter, a digit or a white-space character. Also, since you will be sorting the words in the next part, convert everything to lower case as you read it.

Part II—Sorting

You will use selection sort, as taught in class, to sort the words. A few comments are useful here:

- To pass your array of C-strings to your sorting function, use the convention for passing multi-dimensional arrays to functions.
- Since the relation operators do not work on C-strings, use the `strncmp` function to determine the alphabetic ordering of pairs of strings.
- Since the assignment operator does not work for C-strings, your swap function will need to use `strncpy` instead.

You will sort the array into alphabetical order, from `a` to `z`. Since all strings will contain only lowercase characters, you need not worry about the how character case affects the sort.

Part III—Removing Repeated Words

In most text files, many words will appear more than once. Your sorting function, if implemented correctly, will cause all duplicates of a given word to be stored in consecutive array elements. A simple way to remove duplicates then is to scan the entire array, and any time two identical words are found in adjacent array locations, overwrite the second occurrence by moving all remaining words in the array forward by one.

Part IV—Output

Once the words have been read, sorted and duplicates removed, you are to output the list of unique words (one per line), followed by the number of unique words and also the total number of words read from the file.

Part V—Searching

Implement a function to do binary search on the array of words. After outputting the list of unique words and their count, your program should repeatedly prompt the user for words to search for, until the user enters a word that starts with a non-letter. You will need to make the words entered lowercase and remove any leading/trailing punctuation, but you can re-use the functions you wrote to do this earlier.
If a search word is found, print out the word and its index in the array. If a search word is not found, print the word in double quotes followed by `NOT FOUND`.

Part VI—Debugging

If you wish to use DDD to help debug your program, you will find it necessary to type

```
set args <fileName>
```

where `<fileName>` is replaced with the name of the file you are reading your words from, into the command area at the bottom of the DDD window. This will cause DDD to pass the filename via the `argv[]` parameter in your `main` function when it runs your program.

Part VII—Marking

You must be able to compile the program and demonstrate it for the TA who marks your lab.

Submission

Before your lab is over you **must** submit your files using the unix `submitaps105f` command. For example, the command `submitaps105f 6 Lab6.c` may be used to submit your file. The manual page for this command may be read by executing the unix `man submit` command.
The command `submitaps105f -l 6` may be used to confirm that you have successfully submitted your files.