CSCI 2270 – Data Structures (Fall 2021)

Instructor: Asa Ashraf

Assignment 2

Due Monday, September 13, 11:59 PM MDT

DYNAMICALLY ALLOCATED ARRAYS

OBJECTIVES

1. Read a file with unknown size and store its contents in a dynamically allocated array
2. Store, search and iterate through data in an array of structs
3. Use array doubling via dynamic memory to increase the size of the array

Instructions

In this assignment, we will write a program to analyze the word frequency of a plaintext document. As the number of words in the document may not be known a priori, we will perform array doubling on a dynamically allocated array to store the information as it arrives.

Please read all the directions *before* writing code, as this write-up contains specific requirements on how the code must be written.

Problem

Overview:

There are two files on Canvas:

(1) ***greatgatsby.txt*** - contains text to be read and analyzed by your program. The file contains the text from The Great Gatsby by F. Scott Fitzgerald. For your convenience, all the punctuation has been removed and words have been converted to lowercase.

(2) ***commonWords.txt*** contains 50 of the most common words in the English language, which your program will ignore during analysis.

Your program must take four command-line arguments in the following order:

(1) ***the name of the text file to be read and analyzed***
(2) ***the name of the file containing common words***
(3) ***an integer M: this is the index at which we begin printing***
(4) ***an integer N: this is the number of words to be printed***

Your program will read the text from the first file while ignoring the words from the second file and store all the unique words encountered in a dynamically doubling array. After necessary calculation(s), the program must print the following information:

- The number of times array doubling was required to store all the unique words
- The number of unique “uncommon” words in the file
- The total word count of the file (excluding the common words)
- After calculating the probability of occurrence of each word and storing it in an array in the decreasing order of probability, print the **N** most frequent words along with their probability (**up to 5 decimal places**) starting from index **M** of the resultant array.

  For example, running your program with the command:

  ```
  ./Assignment2 greatgatsby.txt commonWords.txt 12 5
  ```

  would print the 5 words starting from *index* 12 (of the sorted-by-count array), i.e. your program must print the 13th-17th most frequent words, along with their respective probabilities. Keep in mind that these words must **not** be any of the words from the ***common words*** file.

A sample run will look as follows:

```
Array doubled: 7
Distinct uncommon words: 6424
Total uncommon words: 29717
Probability of next 5 words from rank 12
---------------------------------------
0.00404 - like
0.00394 - down
0.00390 - is
0.00374 - some
0.00367 - back
```

Specifications:

1. Use an array of structs to store the words and their counts

You will store each unique word and its count (the number of times it occurs in the document) in an **array of structs**. As the number of unique words is not known ahead of time, the array of structs must be **dynamically sized**. The struct must be defined as follows:

```
struct wordRecord
{
        string word;
        int count;
};
```

2. Use the array-doubling algorithm to increase the size of your array

Your array will need to grow to fit the number of words in the file. **Start with an array size of 100**, and double the size whenever the array runs out of free space. You will need to allocate your array dynamically and copy values from the old array to the new array. (Array-doubling algorithm must be implemented in the *main()* function).

Note: Don't use the built-in **std::vector** class. This will result in a loss of points. You're writing code that emulates the vector container's dynamic resizing functionality.

3. Ignore the top 50 most common words that are read from the *common words* text file

To get useful information about word frequency, we will be ignoring the 50 most common words in the English language as noted in the common words file. ***commonWords.txt*** presents one such example.

4. Take four command-line arguments

Your program must take four command-line arguments

1. the name of the text file to be read and analyzed
2. the name of the file containing common words
3. an integer M: this is the index at which we begin printing
4. an integer N: this is the number of words to be printed

5. Output the *N* most frequent words starting from index *M*

Your program must print out **N** most frequent words - not including the common words - starting index **M** in the text where ***M,N*** are passed as command-line arguments. This assumes that the array you are indexing on is sorted by count (descending). **If two words have the same frequency, list them alphabetically.**

You may assume that M and N will always be valid values for a given example, provided that array doubling was performed correctly.

6. Format your final output this way:

```
Array doubled: <Number of times the array was doubled>
Distinct uncommon words: <Distinct uncommon words>
Total uncommon words: <Total uncommon words>
Probability of next <N> words from rank <M>
---------------------------------------
<Mth index probability> - <corresponding word>
```

```
<M+1th index probability> - <corresponding word>
...
<M+N-1th index probability> - <corresponding word>
```

For example, using the command:

```
./Assignment2 greatgatsby.txt commonWords.txt 25 10
```

Output:

```
Array doubled: 7
Distinct uncommon words: 6424
Total uncommon words: 29717
Probability of next 10 words from rank 25
---------------------------------------
0.00323 - dont
0.00320 - its
0.00316 - any
0.00310 - house
0.00306 - went
0.00299 - before
0.00296 - after
0.00293 - eyes
0.00293 - got
0.00289 - come
```

7. **You must include the following functions (they will be tested by the autograder):**
   a. **main function**
      i. If the correct number of command line arguments is not passed, print the below statement as-is and exit the program

      ```
      cout << "Usage: ./Assignment2 <inputfilename>
      <commonWordsfilename> <M> <N>" << endl;
      ```

      ii. Get common words from the common words file (ex: ***commonWords.txt***) and store them in an array (Call your `getCommonWords` function). Refer *Appendix 2.*
      iii. Array-doubling must be done in the main() function
      iv. Read words from the provided text file (ex: ***greatgatsby.txt***) and store all unique words that are not common words in an array of structs
         1. Create a dynamic `wordRecord` array of size 100

2. Add non-common words to the array (double the array size if the array is full)
3. Keep track of the number of times the **`wordRecord`** array is doubled and the number of unique uncommon words

b. getCommonWords function

```
void getCommonWords(const char *commonWordFileName, string*
commonWords);
```

This function must read the common words from the file with the name stored in **`commonWordFileName`** and store them in the **`commonWords`** array. You can assume there will always be exactly 50 common words in the file. There is no return value.

Refer *Appendix 2* on how to read words from a file.

In case the file fails to open, print the following error message and exit the function:

```
std::cout  <<  "Failed  to  open  "  <<  commonWordFileName  <<
std::endl;
```

c. isCommonWord function

```
bool isCommonWord(string word, string* commonWords);
```

This function must return whether **`word`** is in the **`commonWords`** array.

d. getTotalNumberUncommonWords function

```
int getTotalNumberUncommonWords(wordRecord* distinctWords, int
length);
```

This function must compute the total number of words in the entire document by summing up the counts of the individual unique (uncommon) words. The function must return this sum.

e. sortArray function

```
void sortArray(wordRecord* distinctWords, int length);
```

This function must sort the **`distinctWords`** array (which contains **`length`** initialized elements) in descending order by word count, such that the most frequent words are sorted to the beginning. The function does not return anything.

Additionally, if a subset of words has the same frequency then they should also be sorted alphabetically in ascending order.

> ***Note:*** You should NOT use the built-in **`sort`** function provided by the **`<algorithm>`** header. However, you may write your own implementation of a sorting algorithm, such as Bubble Sort. Feel free to refer to the pseudocode for bubble sort that was given in Assignment 1.

f. printNFromM function

```
void printNFromM(wordRecord* distinctWords, int M, int N, int
totalNumWords);
```

This function must print *N* words starting at index *M* from the **sorted** array of **`distinctWords`**. These *N* words must be printed with their probability of occurrence **up to 5 decimal places** *(refer Appendix 1)***.** The exact output format is given below. The function does not return anything.

The probability of occurrence of a word at position ***ind*** in the array is computed using the formula: ***(Don't forget to cast to float!)***

probability-of-occurrence = *(float) distinctWords[ind].count / totalNumWords;*

Outside this function, `totalNumWords` is obtained by calling the function `getTotalNumberUncommonWords()`.

```
Array doubled: 7
Distinct uncommon words: 6424
Total uncommon words: 29717
Probability of next 12 words from rank 15
---------------------------------------
0.00374 - some
0.00367 - back
0.00363 - been
0.00363 - no
0.00357 - came
0.00357 - man
```

```
0.00343 - little
0.00337 - just
0.00333 - know
0.00326 - now
0.00323 - dont
0.00320 - its
```

8. **Submitting your code:**

   Log onto Canvas and go to the **Assignment 2 Submit** link. It's set up in the quiz format. Follow the instructions on each question to submit all or parts of each assignment question.

APPENDIX 1: Printing *Z* decimal places

In order to print *Z* number of decimal places in C++, you need to include the `<iomanip>` header.

You can then format your code as follows:

```
int Z = 5;
cout << fixed << setprecision(Z);
cout << 123.45 << endl;                 // 123.45000
cout << 0.01234567 << endl;             // 0.01235
```

APPENDIX 2: Reading words from a text file

Use the following C++ snippet to read words from a file:

```
#include <fstream>

ifstream inStream;        // stream for reading in file
inStream.open(filename); // open the file

string word;
while ( inStream >> word )
{
    // process the word
}
inStream.close();         // close the file
```