

Lab 2

Word Lengths



CNT 252
C++ Algorithms and Data Structures
Winter 2003
7 Feb 2003

In this Lab, you will demonstrate your understanding of pointers and functions by writing a program that calculates the word-length statistics for a text file into a ragged array and then returns information about the file based on a command-line argument. The project will use function pointers and pointer arithmetic to accomplish its tasks.

2.1 Specifications

The program will take 2,3, or 4 command-line arguments:

1. A single `char` which will determine the task to perform. This argument should be case-insensitive (so `T` and `t` should provide the same result).
2. A filename and path to read.
3. (Optional, dependant on the first argument) an `int` specifying the length to search for or the line number to check, or a string specifying the output file.
4. (Optional, dependant on the first argument) an `int` specifying the word number to check.

The first argument will produce one of the following modes:

1. `T`, `S`, `L` or `A` - the program will print the *total number of words*, *length of the shortest word*, *length of the longest word*, or *average word length* of the file provided as argument 2, respectively.
2. `0` - (Letter, not number) - the program will print the location (line and word numbers) of the first word in the file provided as argument 2 that is the length specified as argument 3. If that length does not exist, it will print a 'not found' message.
3. `E` - the program will print the length of the word in the file provided as argument 2, as specified by the third argument (line number) and fourth argument (word number). If no such word exists, a 'not found' message will be printed.
4. Any other character - the program will print out the array of word lengths in the file provided as argument 2 to the filename provided as argument 3. If there is no argument 3, then it will print to the console.

2.2 Constraints

1. Because we do not yet know DMA, you will have to predeclare a size for your buffer arrays and initialize them. You may use a value of 10 for the maximum number of lines in the file (the `int*` array) and a value of 255 for the maximum number of words in a row (the `int` array). Note that you should declare the 10 `int` arrays first and then assign them to the `int*` array. Note that, because the `int` arrays will be zero terminated, you will need to have a size of 256.
2. The constraint above means that you will only need to read the first 10 lines of any file, and the first 255 words of any line. You may also limit yourselves to only reading 65,535 characters per line (so, if the first 255 words take more than 65,535 characters, you may truncate the line. This is unlikely). However, bear in mind that you should still be able to handle files that have more than 10 lines, lines with more than 255 words, and blank lines in the middle (or at the end) of a file.
3. As discussed below, an `int` array (array of word lengths) with a first item value of zero will indicate a *blank* line. An `int` array with a *negative* first item value will indicate a line that does not exist (for example, if a file only has 8 lines, the last two arrays would begin with (say) -1. Consequently, you should probably initialize your `int` arrays to begin with a negative number.
4. Because you are reading line-by-line, the `getline()` command will probably be more useful than the extraction operator. To parse a string into words, consider the `isalnum()` and `ispunct()` functions, or (if you want to get fancy) `strtok()`. For the purposes of this lab, a word will be a consecutive series of alphanumeric characters, but not punctuation or white space. The functions listed above should be documented in MSDN.
5. The functions that return line numbers and word numbers will use one-base counting. In other words, the first line will be line 1, not line 0, and the first word of the line will be word 1, not word 0. Return values of zero will typically indicate that the word or line requested does not exist.
6. This program will serve as the foundation for your next lab as well, so it is suggested that you try to get it working (well) as quickly as possible.

2.3 Function Specifications

This lab project will consist of the files and functions (with signatures) specified in the prelab (subject to updates and modifications by your instructor). The required specifications for the functions is detailed below.



Tip: It is strongly recommended that you implement these functions one at a time and test them individually. It will be much easier to troubleshoot the program if you know that individual functions are properly implemented.

2.3.1 Utility

The functions that (at a minimum) must be present in the Utility files are as follows:

- `void PrintMessage(char const * const szMessage)` - Which will print out a neatly formatted message to the console.

- `void PrintError(char const * const szLocation, char const * const szErrorMessage)` - Which will print out a properly formatted error/usage message, per the course standards. The location is the routine in which the error occurred, and the message is the error message. The function will also print a properly formatted usage message.

2.3.2 LineFuncs

- `int * Populate(int * piArray, char const * const szString)` - Which will calculate the lengths of the words in `szString`, one word at a time, in order, into the `int` array `piArray`. The `int` array will be zero-terminated - which means that, once all the words (or 255 words) have their lengths recorded, a zero will be stored to indicate that there are no more words on that line. An array with an initial value of zero will indicate a blank line. The string passed as an argument will be one line.
- `int WordCount(int const * const piArray)` - Will return the total number of word lengths (i.e. the number of words in the line) stored in the passed array.
- `int ShortestWordLength(int const * const piArray)` - Will return the smallest word length in the passed array (a line). Returns zero if no words.
- `int LongestWordLength(int const * const piArray)` - Will return the largest word length in the passed array (a line). Returns zero if no words.
- `double AverageWordLength(int const * const piArray)` - Returns the average word length from the array (a line). Zero if no words. The `WordCount` function may be of service.
- `int GetLengthByIndex(int const * const piArray, unsigned int uiIndex)` - Returns the length of the `n`th word - where `n` is specified by `uiIndex`. `uiIndex` will be one-based - so the first word will be at an index of one. Returns zero if no word there.
- `unsigned int GetIndexByLength(int const * const piArray, int iLength)` Returns the index (which word) of the first word of length `iLength`. This is also one based. Returns zero if no such word.

2.3.3 LineArrayFuncs

- `int ** LoadString(int ** ppiArray, char const * const szString)` - This routine will take the string passed into it and calculate its word lengths into the next empty `int` array (identified by a leading negative value). It will return the modified ragged array that it was passed. This will call (at least) `Populate()`.
- `int ** LoadString(int ** ppiArray, char const * const szString, unsigned int uiIndex)` - this routine is an overloaded version of the previous function, and will do the same task, but load the word lengths into array stored as the `uiIndex`th array in the ragged array, rather than simply the next empty one.
- `int TotalWordCount(int const * const * const ppiArray)` - which will call `WordCount()` and return the total number of words read from the file.
- `int ShortestWord(int const * const * const ppiArray)` - which will call `ShortestWordLength()` and return the length of the shortest word read from the file.
- `int LongestWord(int const * const * const ppiArray)` - which will call `LongestWordLength()` and return the length of the longest word read from the file.

- `double AverageWord(int const * const * const ppiArray)` - which will call `AverageWordLength()` and `WordCount()` and return the average word length of all the words in the file. Think about why you will need `WordCount()` to get an accurate number.
- `void GetLocationByLength(int const * const * const ppiArray , int iLength , unsigned int & uiLine , unsigned int & uiWord)` - which will call `GetIndexByLength()` and return (using its reference arguments) the location of the first word read with the specified length. It will return two zeros if a word of the length does not exist.
- `int GetLengthByLocation(int const * const * const ppiArray , unsigned int uiLine , unsigned int uiWord)` - which will call `GetLengthByIndex()` and return the length of the word specified by the two parameters. Zero if the word does not exist.

2.3.4 FileFuncs

- `int ** ReadFile(char const * const szFilename , int ** ppiBuffer , unsigned int & uiLineCount)` - will calculate the word lengths of the specified file into the ragged array that it has been passed. `uiLineCount` will specify how many lines to read. If fewer lines are read (i.e. - the file has fewer lines), it will modify the argument to reflect how many lines were actually read). `uiLineLength` specifies the maximum number of *characters* to read per line. Note that this routine is expected to make use of the `LoadString` functions, above.
- `bool PrintResults(char const * const szFileName, int const * const * const ppiBuffer)` - will print the array of word lengths, space separated, one line per line, to the specified file. Blank lines will still appear as blank lines, but words will be replaced by their lengths. Do not print the terminating zeros or negative 'no line' flags.
- `bool PrintResults(int const * const * const ppiBuffer)` - overloaded version of above function which prints to the console instead.



Warning: Note that the last two functions above have had their signatures modified. Make sure you modify your prelab functions to reflect this.

2.4 Requirements

Sample runs of the program, and a sample executable will be provided to you at a later date.