Lab 8: Arrays and Reusable Code

Overview

Large projects are usually not written as a single source code file, for various reasons:

- When more than one person works on a project, each person works on a different piece of it and it is more efficient if these pieces are in separate files, otherwise they would have to take turns editing the file.
- Putting frequently needed portions of code into distinct files makes it possible to incorporate this code into multiple projects without actually having to rewrite the code or have multiple copies of it, making that code reusable.
- If all of the code is in one large file, then each time any small part of it is changed, the entire file has to be recompiled. These days with such fast computers, compiling takes very little time so this is not an issue for small programs, but for very large programs, this can still take some time. If the program is composed of several small files, then only the changed file needs to be recompiled, and the program re-linked to the changed file. This is usually much faster than recompiling the entire program.
- Maintaining the code is easier because logically related code is all in a single small file, making it easier to find the parts of the code that need to be revised.

This exercise is designed to give you practice creating a multiple-file program whose parts can be compiled separately. It also gives you more practice with arrays, sorting, and file input and output.

Exercise

You will write a program that prompts the user to enter the names of an input file and an output file, and then reads input from the input file, processes the input, and sends its output to the output file. If the input files cannot be opened, or if the output file cannot be opened, the program returns with the error value 1. The input file will consist of lines containing single words, one per line. You can think of the words as the usernames of people who have used some online service. The service provider is interested in seeing usage patterns for its users, so they have commissioned you to do some very simple analysis of this data. Specifically, your program must read the words and store them into an array. It must sort that array using the usual comparison operation for strings (i.e., `str1` < `str2` for strings `str1` and `str2`). The data will probably have many words that are the same, and when the data is sorted, these lines will be adjacent. For example, if the data file has the names

```
haley, sue, bingo, sam, sue, bingo, cranky, mel, sam, sue
```

(one per line, but here written separated by commas to save space), then after sorting it will look like

```
bingo, bingo, cranky, haley, mel, sam, sam, sue, sue, sue
```

Your program must create a new array that contains the data from the original array but without any repeats. In other words, it must create a new array that is also sorted, but such that every word in that new array is unique. For the above example, the array would contain

```
bingo, cranky, haley, mel, sam, sue
```

It must write the contents of this array to the output file whose name the user specified.

Program Design

The program must consist of exactly five files, named as follows:

- A header file named `file_io.h`
- An implementation file for this header file, named `file_io.cpp`
- A header file named `list.h`
- An implementation file for this header file, named `list.cpp`
- A main program file named `lab08_main.cpp`

The `file_io.h` header file must contain the prototypes of the two functions needed to do file I/O: one that reads the contents of an input file stream into an array, and one that writes the contents of an array into an output file stream. The following listing uses DOxygen-style comments to describe what these functions must do and what their prototypes should look like.

```
/** readlist(inputstream, names, limit, length)
 *  @param istream& inputstream [in/out]  the stream to read from
 *                                        one string per line
 *  @param string   names[]     [out]     the array to fill with strings
 *  @param int      limit       [in]      max size of the array
 *  @param int&     length      [out]     number of names stored in names[]
 *  @pre   inputstream is open && limit > 0 && names is size limit
 *  @post  0 <= length <= limit && names[0]..[length-1] is filled with strings
 *         from inputstream
 */

/** writelist(outputstream, names, length)
 *  @param ostream& outputstream [in/out] the stream to write to
 *  @param string   names[]      [in]     the array of strings to write
 *  @param int&     length       [in]     number of names stored in names[]
 *  @pre   outputstream is open && length >= 0 && names[0]..names[length-1]
 *                                        contains words to write, one per line
 *  @post  none
 */
```

`file_io.cpp` must contain the definitions of those two functions.

The `list.h` header file must contain the prototypes of the two functions needed to do array processing: one that sorts an array of strings, and one that, given a sorted array of strings, fills a second array with the unique list of words contained in the first array. Because the size of the second array may be smaller than the first, this function must also fill a parameter with the size of the second array. The following listing describes what these functions must do and what their prototypes should look like.

```
/** sort (list, length)
 *  @param string  list[]   [in/out] the array of strings to be sorted
 *  @param int     length   [in]     number of strings stored in list
 *  @pre   list[] has 0 or more strings
 *  @post  list[0] <= list[1] <= ... <= list[length-1] and every string
 *         that was in list before is in list exactly once after sort finishes
 *         (none got deleted or replicated)
 */

/** uniq (origlist, newlist, origlength, newlength)
 *  @param string  origlist[] [in]  a sorted array of strings
 *  @param string  newlist[]  [out] an array of strings containing no duplicates
 *  @param int     origlength [in]  number of strings stored in origlist
 *  @param int&    newlength  [out] number of strings stored in newlist
 *  @pre   origlist[] is sorted (and may have duplicate adjacent strings)
 *  @post  newlist is sorted and has unique strings && newlength <= origlength
 *         && newlength is the number of elements in newlist.
 */
```

The file `list.cpp` must contain the definitions of those two functions.

The main program, `lab08_main.cpp`, will be very small. It must prompt the user to enter the names of the files, open streams for these files, and then call the functions in the other files to (1) read the input file data into the array, (2) sort it, (3) create the unique word array, and (4) print that unique word array to the output file stream, one word per line.

Advice. First create the two header files. Then create your main program. Then create the two implementation files. You can check the main program file by compiling with the `g++ -c` option. This will not attempt to do any linking, so you can see programming errors in `main()` even if you do not have the implementation files.

There are files in the lab08 directory containing a list of names that you use for testing your program. The command

```
$ sort -u filename
```

will sort the contents of the file `filename`, removing duplicates, and display the result on the standard output. You can redirect it to a file using

```
$ sort -u filename > sortedfile
```

You can check if your program is correct by comparing your output file to the file produced by the above command using the `diff` command. diff has no putput if two files are identical:

```
$ diff file1 file2
```

will produce no output if `file1` is an exact copy of `file2`.

What to Submit

Submit your program, however complete it is, by the end of today's lab, i.e., before the end of the class at 2:00 P.M. ***There is no grace period for this. Programs submitted after 2:00 PM will not be accepted.*** The instructions for submitting are:

1. Create a directory in
   `/data/biocs/b/student.accounts/cs135_sw/cs136labs/lab08/submissions`
   whose name is your *username*. For example, I would create the directory `sweiss`.

2. Copy your five files, which should be named as described above, to that directory. You will lose 5% of the grade if you misname the files! Make sure that each file has a preamble with your name and other appropriate information in it.

3. Change the permission on the directory that you created so that no one else can read or modify it. You do this with the commands

   ```
   $ cd /data/biocs/b/student.accounts/cs135_sw/cs136labs/lab08/submissions
   $ chmod 700 username
   ```

Do not submit executable files. Remember to document your code (both preamble and comments in the code) and make it easy to read. Your work will be graded based on the rubric outlined in the Programming Rules document. There are absolutely no extensions to the deadline.