CS 202 - Computer Science II
Project 1

Due date: Wednesday, 1/29/2020, 11:59 pm

Objectives: The two main objectives of this project is to test your ability to: a) use C++ I/O (console and file), and b) design, implement and test a solution to a given problem. In addition, a review of your knowledge of Cstrings and multi-dimensional arrays will be necessary. This project will also test your ability to create, compile and run a program in a Linux environment and establish your ability to upload your program and supporting documentation to WebCampus.

Description:
For this project you will write a program to: a) **read-in** the **10 first names** from a **file** (the file is a priori <u>known to have exactly 10 entries</u>, of a <u>maximum length of 8 letters</u> each) into a **2-dimensional character array**, b) **output** the names **to the terminal** with each one preceded by a number indicating its **original order** in the list, c) **sort** the list of names, and then d) **output the sorted** names both **to the terminal and to a file**, again with each one preceded by its corresponding **original order** in the list. Although an example input file (Names.txt) is provided, for grading purposes your project will be tested against a file that we will supply but will not be provided to you beforehand. Our test file will be in the same format as the example input file.

The following minimum functionality and structure is required for your program:
Note: To store, maintain, and manipulate the names list you *have* to use a two-dimensional char array (i.e., an array of Cstrings).
Make a program that sequentially executes the following steps:

- **1: Asks the user** for the input and output **file names**.
- **2: Reads** in the list of names from the desired **input file**.
  To do so you have to declare and implement a function **`readNames()`** that takes in at least the following parameters (but also any others that you may need):
  a) **`inputFile`**, a Cstring that corresponds to the input filename.
  b) **`namesArray`**, a 2-dimensional character array (i.e., an array of Cstrings) to be updated with the names read from the file.
- **3: Prints** out the **unsorted** list of names to the **terminal**, preceded by their **original order** in the input file.
  To do so you have to declare and implement a function **`printNamesToTerminal()`** that takes in at least the following parameters (but also any others that you may need):
  a) **`namesArray`**, a 2-dimensional character array (i.e., an array of Cstrings) to be used for printing out the names.
- **4: Sorts** the list of names **alphabetically**.
  To do so you have to declare and implement a function **`sortNamesAlphabetic()`** that takes in at least the following parameters (but also any others that you may need):
  a) **`namesArray`**, a 2-dimensional character array (i.e., an array of Cstrings) to be sorted in alphabetical order (see sample output below).

- **5: Prints** the list of **alphabetically-sorted** names to the **terminal**, preceded by their **original order** in the input file.
  To do so you have to use the same function as before: **`printNamesToTerminal()`**, so you have to make sure the logic behind it holds across all its use-cases.
- **6: Sorts** the list of names **by-length first** and **alphabetically secondly**.
  To do so you have to declare and implement a function **`sortNamesByLength()`** that takes in at least the following parameters (but also any others that you may need):
  a) **`namesArray`**, a 2-dimensional character array (i.e., an array of Cstrings) to be sorted by name length. *Tie-breaks:* If 2 or more names have the same length, they should be sorted alphabetically (see sample output below).
- **7: Prints** the list of **sorted-by-length –and– alphabetically** names to the **terminal**, preceded by their **original order** in the input file.
  To do so you have to use the same function as before: **`printNamesToTerminal()`**, so you have to make sure the logic behind it holds across all its use-cases.
- **8: Writes** the list of **sorted-by-length –and– alphabetically** names to an **output file** (e.g. SortedByLength.txt), preceded by their **original order** in the input file.
  To do so you have to declare and implement a function **`printNamesToFile()`** that takes in at least the following parameters (but also any others that you may need):
  a) **`outputFile`**, a Cstring that corresponds to the output filename.
  b) **`namesArray`**, a 2-dimensional character array (i.e., an array of Cstrings) to be used for printing out the names.

- Note: For all of the above, you are not allowed to modify in any way the structure of your original 2-dimensional `char` array to achieve the purpose of maintaining the original order (i.e., do not try to prepend the order to the each name Cstring).

- You have to write your **own Cstring copy, compare, length** functions. Their prototypes shall have the form (use the prototypes as provided, with **`char []`** –i.e. character array–parameters):

```
// copies characters from source to destination until a NULL-
character '\0' is found in source, then it NULL-terminates
destination too, and returns
void myStringCopy(char destination [], const char source []);

// counts characters in array str until a NULL-character '\0' is
found, then it returns that number excluding the '\0' one
int myStringLength(const char str []);

// returns 0 when the strings match, i.e. their characters are
equal one-by-one until a NULL-character '\0' is found in both
strings and at the same position as well
// returns a value <= -1 if the first character that does not
match has a lower value in str1 than in str2
// returns a value >= 1 if the first character that does not
match has a higher value in str1 than in str2
int myStringCompare(const char str1 [], const char str2 []);
```

The following are a list of restrictions:

- Compile your code using either the C++98 or C+03 standard but no higher (**`g++ -std=c++98 ...`** -or- **`g++ -std=c++03 ...`**).
- No usage of external libraries for C-string manipulation is allowed (e.g. **`<cstring>`** **`<string.h>`**), or any **`std::string`** libraries and data types.
- No libraries except **`<iostream>`** and **`<fstream>`** are allowed.
- No global variables or constants except: a) the fixed number of names, and b) the maximum C-string size.
- No usage of pointers or dynamic memory.
- You are expected to employ code abstraction and reuse by implementing and using functions. Copy-pasting code segments (that each performs a specific functionality which can be wrapped within a function) throughout your program *will be penalized*.
  Declare, Implement, and Use functions to achieve your objectives. The already provided code structure in the project description will be considered sufficient.

Example Input File (Names.txt) **Contents:**

```
Victor
Eve
Juliet
Hector
Danielle
Romeo
Oscar
June
Ares
Dannae
```

Example Output (to Terminal and/or File)**:**

```
Unsorted Data (Original Input Order and Name)
==============================
0 Victor
1 Eve
2 Juliet
3 Hector
4 Danielle
5 Romeo
6 Oscar
7 June
8 Ares
9 Dannae

Alphabetcially Sorted Data (Original Input Order and Name)
============================
8 Ares
4 Danielle
9 Dannae
1 Eve
3 Hector
2 Juliet
7 June
6 Oscar
5 Romeo
0 Victor

Sorted-by-Length –and– Alphabetically Data (Original Input Order and Name)
============================
1 Eve
8 Ares
7 June
6 Oscar
5 Romeo
9 Dannae
3 Hector
2 Juliet
0 Victor
4 Danielle
```

The completed project should have the following properties:

- Written, compiled and tested using Linux.
- It must compile successfully using the g++ compiler on department machines.
  Instructions how to remotely connect to department machines are included in the Projects folder in WebCampus.
- The code must be commented and indented properly.
  Header comments are required on all files and recommended for the rest of the program.
  Descriptions of functions commented properly.
- A one page (minimum) typed sheet documenting your code. This should include the overall purpose of the program, your design, problems (if any), and any changes you would make given more time.

Turn in: Compressed .cpp file and project documentation.

Submission Instructions:

- You will submit your work via WebCampus
- Name your code file proj1.cpp
- If you have header file, name it proj1.h
- Compress your:
  1. Source code
  2. Documentation

  Do not include executable
- Name the compressed folder:
  PA#_Lastname_Firstname.zip
  ([PA] stands for [ProjectAssignment], [#] is the Project number)
  Example: PA1_Smith_John.zip

Verify: After you upload your .zip file, re-download it from WebCampus. Extract it, compile it and verify that it compiles and runs on the NoMachine virtual machines or directly on the ECC systems.

- Code that does not compile will be heavily penalized –may even cost you your *entire* grade–. Executables that do not work 100% will receive partial grade points.
- It is better to hand in code that compiles and performs partial functionality, rather than broken code. You may use your Documentation file to mention what you could not get to work exactly as you wanted in the given timeframe of the Project.

Late Submission:
A project submission is "late" if any of the submitted files are time-stamped after the due date and time. Projects will be accepted up to 24 hours late, with 20% penalty.