CSE 332 Lab 1: Basic C++ Program Structure and Data Movement

Due by: Thu. September 19th, at 11:59 pm

Final grade percentage: 8 percent

Objective:

This lab is intended to familiarize you with basic C++ program structure, data movement and execution control concepts, including:

- C++ header files and C++ source files;
- C++ STL string, input, and output streams;
- C++ precompiler directives; and
- basic C++ data types (including enumerations and C++ style strings).

To do this, you will implement a simple C++ program that can
    (1) read in and parse command line arguments,
    (2) open a file and read in strings from it,
    (3) differentiate numeric values from other input strings, and
    (4) produce output based on the strings that it read in, to the standard output stream.

We will use (scoped or unscoped) enumerated types in a couple of places in this assignment: for program and function return values (to indicate either their success or different kinds of failure that may occur) and for different values relevant to the array of program arguments (for example its size and different positions within it). This is one good way to follow the programming guideline that instead of using "hard coded" numbers like 0, 1, or 2 in a program, for readability and ease of maintenance you should instead use descriptively named symbols like success, file_open_failed, no_file_name_given, etc.

Assignment:

Part I - Readings:

1. **The following readings in the textbook may be useful while working on this lab assignment, and selectively looking up topics and/or reading ahead (on demand as you encounter issues while working on the lab) in the textbook, lecture slides, and studio exercises is also encouraged:**
    - C++ program structure and development environment: LLM Chapter 1
    - C++ variables and basic data types: LLM Chapter 2
    - Strings, vectors, arrays: LLM Chapter 3
    - The IO Library: LLM Chapter 8

   - Enumerations: LLM Chapter 19.3
2. The output stream, C++ string class, and input and output stringstream examples in the C++_strings_arrays_vectors_io.ppt slides also may be helpful when implementing parts this lab.

3. **Please read the following CSE 332 Programming Guidelines which are relevant to this lab, and please follow them as you implement your solution:** A.1-A.7, A.10-A.17, B.1-B.3, B.8, B.13-B.17, B.20-B.24, B.29, B.31-B.33, and C.1-C.8.

Part II - Program Design and Implementation:

4. Accept the lab1 assignment from GitHub Classroom here.

5. Open up Visual Studio 2019, connect to GitHub, and clone your newly created repository to create a local working copy on your **h:** drive. Create a new Visual C++ Console App within your repository, make sure to name it something descriptive such as *Lab1*.

6. In the header file and source files for your project, you will declare and define functions, enumerations and other features of your program as you implement this assignment. **As you develop your code, please use good modularity:** for example, if a sequence of logic has several separable parts (or is long enough to fill the editor's window) you *may* want to break it up into other smaller functions.

Note: even at this early stage the details of this assignment are intentionally slightly under-specified, leaving you some room to choose what you think is the best way to implement them, as long as what you do is reasonable and you explain your design decisions in comments in the code and in your file.

7. Add a new C++ header (**.h**) file to your program and in it please declare an enumeration for the different array indices that are relevant to your program (e.g., the program name which is at position 0 in **argv**, the input file name which should be given at position 1 in **argv**, and the expected number of command line arguments including the program name, which is given by **argc**).
   In that same header file, declare an enumeration for the different success and failure values your program (and functions called within it) can return (i.e., 0 for success and different non-zero values for all the different kinds of failures the program may encounter).

8. Include that header file in your main program source file and in any other source file that has functions with non-void return values, or that needs to use indices into the array of command line arguments to the program, and throughout your program please use the labels you declared in the enumerations, instead of hard-coding numeric values like 0, 1, 2, etc.

9. Declare and define a function for parsing an input file (for example **input_file.txt**) containing text (character) strings and different forms of whitespace (spaces, tabs, line breaks, etc.). The file parsing function should take two parameters: a reference to a Standard Template library (STL) vector of C++ style strings (e.g., of type **vector<string>**

&), and a C-style string (of type **char ***); This function should use the C-style string given in the second function parameter as the name of a file, open the file (using an input file stream), and then extract one string at a time from that file until there are no more strings to read (**hint:** the input file stream **>>** extraction operator will return false when it cannot extract another string from the file).
Each time a non-empty string is extracted (any empty strings should be simply ignored), the function should push back the string into the vector that was passed as the first parameter. If the file cannot be opened or the function encounters any other problems during its execution it should print out a helpful error message indicating the problem and return the appropriate enumeration label (that has a non-zero integer value) for failing to open an input file; otherwise it should return the enumeration label (that has integer value 0) that indicates success.

10. Declare and define a helpful "usage message" function that (1) takes a C style string or a reference to a const C++ style string as its only parameter (to pass the name of the program, which is always found in position 0 of **argv**), (2) uses that parameter to print out (to the program's standard output stream) a helpful usage message telling the user how to run the program correctly (with the name of the program and the name of a file to read, e.g., **usage: lab0 <input_file_name>**), and (3) returns the enumeration label corresponding to the failure return value for having the wrong number of command line arguments.

11. For your project's main function:

    - Modify the signature of your main function so that it matches the one shown in the second example given in the C++_programs.ppt slides (the one that uses the variable names **argc** and **argv**).
    - The program should first check that exactly one argument has been passed to it (in addition to the program's name). If not, the program should call the usage message function and return the value returned from that call.
    - Declare a vector of C++ style strings (of type **vector<string>**), and pass that vector and the first program argument (after the program name) to the function that parses the input file.
      If that function returns an enumeration value other than success the program should return that value. Otherwise the program should continue to the next step.
    - Declare a vector of integers (of type **vector<int>**). Iterate through the vector of strings, and test each one to see if it contains only numeric digit characters (you can test each character using the C++ library's **isdigit** function). For each string that contains only numeric digit characters:
      (1) wrap the string in an input string stream,
      (2) use the stream's extraction (**>>**) operator to convert it to an integer, and
      (3) push that integer back into the vector of integers.

      For each string that contains any other (non-digit) characters, print the string to the standard output stream.

    - Then, iterate through the vector of integers and print out each integer in it.
    - Upon completion, if no errors have occurred, the program should return an

enumeration label (that has value 0) to indicate success.

12. Build your project, and fix any errors or warnings that occur. Please be sure to note all of the different kinds of errors or warnings you ran into (though you don't have to list every instance of each kind) in your **ReadMe.txt** file. **If you were fortunate enough not to run into any errors or warnings**, please note that instead, where you would have listed errors or warnings in your ReadMe.txt file.

13. Open up a Windows console (terminal) window and change to the directory where your program's executable file was created by Visual Studio, and run the executable program through a series of trials that test it with good coverage of cases involving both well formed and badly formed inputs.

    For this lab one especially useful test is to check whether the output printed by your program

    (1) consistently represents the contents of the input file whose name it was given, and

    (2) lists all the strings with non-numeric characters and then all the integer values corresponding to strings that had only numeric digit characters.

    Another useful test is to check the return code produced by the program, which you can get by running the program in the terminal window and then immediately after it finishes running the command ***echo %errorlevel%*** which prints out the value of the errorlevel environment variable where the result returned by the most recently run program is always kept.

    In your **ReadMe.txt** file please document which test cases you ran (i.e., what the command lines were) and summarize what your program did and whether that is correct behavior, in each case.

14. In your **ReadMe.txt** file, make sure that your name and the lab number (lab 1) are at the top of the file, and that you've documented whether you ran into any errors or warnings while developing your solution (and if so what they were) and what the executable program did for each of the trials you ran. Be sure to save your changes to your **ReadMe.txt** file, as well as those to your source (and header) file(s) before committing your final solution.
15. Stage and Commit your changes with a useful commit message. Once committed, push your commit to your remote repository on GitHub. Log on to GitHub and ensure your code was pushed successfully. **If you forget to push your code, it cannot be graded!**

Note: I have initialized each repository with a .gitignore file. This file specifies which files git should track and which files it should not track. Files that should be ignored (changes to the files should not be committed or pushed) are generally Visual Studio generated files like the Debug directory, which contains the .exe file and many logging files created during the build process. Please do not manually upload these files to your Github repository.

Part III - Linux Programming Environment: (optional, worth up to 5% extra credit)

The goal of this optional part of the assignment is to practice using the Linux environment to compile and run your program on **shell.seasad.wustl.edu (a linux server)**, or another Linux platform. For extra credit, please add a section marked "Extra Credit" at the end of the **ReadMe.txt** file you submit with your solution, and copy the results of compiling and running your code on Linux (as described below) from the secure shell window into that section. Please ask for help from your instructors or teaching assistants if you are uncertain about how to do this extra credit part or if you run into any difficulty with it:

(EC 1) To connect to a linux environment, we will use a *secure shell (SSH)* client to connect to shell.seasad.wustl.edu. To do this, open a command prompt in your Windows Remote Desktop and type the following:

```
ssh <your_wustl_key>@shell.seasad.wustl.edu
```

When prompted, supply your wustl password. If logged in correctly, you should see a command prompt that looks similar to:

```
-bash-4.2$
```

Via the ssh connection, you can now send and run commands on the linux server. You should also notice that you have access to your H: drive here as well (try typing “ls” in the command prompt). Copy and paste that command prompt string and the output of the “ls” command into the extra credit portion of your ReadMe.txt file. To do this, you can highlight the text you want to copy from the command prompt and then right-click, this will copy to highlighted text to the clipboard. You can then paste the clipboard into your Readme.txt file.

(EC 2) Use the **mkdir** command to create a **cse332** directory where you will store your linux programs during the course, i.e.
We do not want to mix our Visual Studio and linux programs, so create a new folder outside of the folder you are storing your 332 github repositories.

```
mkdir cse332
```

Change to that directory, create a lab1 subdirectory in it, and change to that subdirectory:

```
cd cse332
mkdir lab1
cd lab1
```

Note that you can also issue multiple commands in sequence (from the same directory) by placing semicolons between them:

```
mkdir cse332 ; mkdir cse332/lab1 ; cd cse332/lab1
```

On your Windows machine, open up the folder where Visual Studio put your source and header files for the project (inside your lab 1 repository) and copy them into the directory you just created above (except for Windows-generated files, which you should not modify. You should only copy header and source files). When you log on to the linux lab machines, they mount your h: drive, so the directories you just created should be visible on your Window’s machine as well.

If you are familiar with Linux, feel free to accomplish this with the copy command(cp). Download the following **Makefile** and copy it to your lab1 directory as well. If necessary, change the name of the file you just added from **Makefile.txt** to just **Makefile** (Windows sometimes "helpfully" adds the **.txt** extension even though the name **Makefile** is what the **make** command you'll use expects the file's name to be).

From your Windows machine (or using your favorite text editor from the bash prompt if you are familiar with linux), use an editor to open up the files you have just transferred into the Linux lab1 directory, and update them as follows:

- In the Makefile, fill in what you want the name of the executable program file to be, at the end of the line where it says **EXECUTABLE =**
- In the Makefile, add the name(s) of your source file(s), separated by spaces if more than one of them, to the end of the line where it says **CMPL_SRCS =**
- In the Makefile, add the name(s) of your header file(s), separated by spaces if more than one of them, to the end of the line where it says **HEADER_FILES =**

After saving the changes to your **Makefile**, exit the editor and run the **make** command (by typing "make" on the bash command prompt) to build your program. Please document any errors or warnings you encountered on Linux, or if there were none please indicate that instead, in the extra credit section of your **ReadMe.txt** file. Fix any errors or warnings you see on Linux, and please ask for help from your instructors or TAs if you have any difficulty with this.

Once your program compiles with no errors or warnings, run the same trials you ran in the Windows environment, on Linux.

Hint: many terminal window shells in the Linux environment require you to put a leading **./** (saying to find the program in the current directory) before the name of the executable program when you run it, as in:

```
./lab1.exe input_file.txt
```

Copy the results of those trials into your **ReadMe.txt** file, in the extra credit section. Be sure to commit and push any changes you make to your remote repository on GitHub so that they can be graded.

Project Designed by Chris Gill