EECE.3220: Data Structures

Spring 2019

Programming Assignment #2: Strings and Classes

Due **Thursday, 3/21/19**, 11:59:59 PM

1. Introduction

This assignment provides an introduction to programming with classes and with composition, an object-oriented programming technique in which one object contains one or more instances of a different type of object. You will also gain practice programming with string objects.

In this program, you will model a dictionary—a collection of words and their definitions. Your main program will process a series of string-based commands that allow you to interact with a dictionary object and the smaller objects the dictionary contains.

2. Deliverables

You should submit five files for this assignment:

- ***prog2_main.cpp:*** Source file containing your main function.
- ***DEntry.h:*** Header file containing the `DEntry` class definition.
- ***DEntry.cpp:*** Source file containing definitions of `DEntry` member functions.
- ***Dictionary.h:*** Header file containing the `Dictionary` class definition.
- ***Dictionary.cpp:*** Source file containing definitions of `Dictionary` member functions.

Submit a single archive file (.zip only) containing all five .h and .cpp files to the "Program 2" assignment on Blackboard.

3. Specifications

Class Design: This assignment contains two classes:

- `DEntry`: Holds a single entry in the dictionary, which contains the following data
  - A single word
  - The part of speech that word represents (noun, verb, etc.)
  - The definition of the word
- `Dictionary`: A collection of up to 100 `DEntry` objects, with words stored in alphabetical order, as well as an integer to track the number of entries stored

Section 6 of this document contains my definitions for these classes, as well as a brief explanation of each member function and some implementation hints. My .h files will also be available on the course website.

You are welcome to use my definitions as written, modify them, or ignore them entirely! Any solution that properly implements the required commands is a correct solution.

3. Specifications (continued)

Command Line Input and Corresponding Output: Your main program should repeatedly prompt the user to enter one of the following commands. Each command should be read as a string or group of strings (I used one string per word):

- **add word:** Add a single word to the dictionary after prompting the user to enter the word, part of speech, and definition
- **add file:** Prompt the user to enter a filename, open that file, and read its contents into the dictionary.
    - You may assume the file is formatted in groups of three lines, with each word, part of speech, and definition on its own line
    - Sample files will be posted on the course websites shortly
- **print all:** Print the contents of every entry in the dictionary
    - If the dictionary is empty, print "`Dictionary is empty`"
- **print letter <*L*>:** Print the contents of every entry containing a word that starts with the letter <*L*>. (For example, “print letter b” prints all words starting with b.)
    - If the dictionary is empty, print "`Dictionary is empty`"
    - If the dictionary contains no words starting with the specified letter, print "`No words beginning with `" followed by the letter (for example, "`No words beginning with b`")
- **find <*word*>:** Search the dictionary for an entry containing the specified word.
    - If the word is found, print the corresponding entry
    - If the word is not found, print "*<word>* `not found`" (for example, "`test not found`")
- **exit:** Exit the program

If the user enters any other command, print "`Invalid command `" followed by the incorrect input.

Detailed test cases will be added to Section 5 of this assignment shortly.

4. Hints

Design process: We suggest the following approach to this assignment:

- First and foremost, plan your solution before writing any code, whether that involves writing a flowchart, an outline, some pseudocode, or any other form of high-level design.
- The main function should strictly handle the command input—most of the work will be done in your class member functions. You may want to start with a very simple main function that helps you test each member function as you write it, then modify `main()` to actually handle the commands.
- When implementing your classes:
    - Don't "reinvent the wheel" and replicate code that exists elsewhere. If one member function can be called to do some of the work required in another member function, use it!
    - You need to write your `DEntry` class first—you can't have a `Dictionary` object without a correct `DEntry` definition.
    - When writing `Dictionary`, I started with the functions to add a single entry and to print the entire dictionary—without those basics, it's hard to test anything else.
        - At first, you don't have to worry about ordering the dictionary entries—a simple implementation adds each new word at the end.
    - I found it easiest to keep the dictionary in order by ensuring each word is added in the proper position. Two possible approaches (and I'm sure there are others):
        - Find the position in which the new word belongs, then shift all words that come after it over one spot before writing the new entry
        - Add the new word at the end of the dictionary, then sort the list by repeatedly swapping the new word with each entry that should come after it.

See the next page for a few points on file I/O.

4. Hints (continued)

File input: A few points on handling file input:

- Input text files should be stored in the same directory as either your source code or your executable.
    - Xcode users must place input files in the same directory as the executable. The steps for adding input files to an Xcode project are described in a separate document.
- You cannot assume the file contents will be in alphabetical order. However, a good implementation won't have to do anything special in the `addFile()` function to ensure words are added in alphabetical order.
- Recall that input files are handled using `ifstream` objects, included in the `<fstream>` library. Details on basic input file usage are in Lecture 3, slides 7-8.
    - Any function we've discussed that you can call on `cin` (`get()`, `ignore()`, `getline()`) can be called on an `ifstream` object.
- When reading input using the >> operator, a read operation will return false if it reaches the end of a file. So, for example, given `ifstream in1` and `int x`, if `in1` accesses a file containing several integers, you could read one integer at a time from that file until you reach the end of file using the following loop:

```
while (in1 >> x) {        // Stops at end of file
   …
}
```

- The version of `getline()` that works with string objects is not an input stream member function (so, for example, you can't call `in1.getline(str)`). Given `ifstream in1`, to read a line of input from the associated file into `string s1`, call `getline()` as follows:

```
getline(in1, s1);
```

More hints may be added later …

5. Test Cases

Your output should match these test cases exactly for the given input values. I will use these test cases in grading of your assignments, but will also generate additional cases that will not be publicly available. Note that these test cases may not cover all possible program outcomes. You should create your own tests to help debug your code and ensure proper operation for all possible inputs.

I've copied and pasted the output below, rather than showing a screenshot of the output window. User input is underlined, although it won't be when you run the program. These test cases span five pages (5-9) but still may not cover all possible input cases you need to consider.

```
Enter command: print all
Dictionary is empty

Enter command: add word
Enter word: banana
Enter part of speech: noun
Enter definition: Yellow fruit

Enter command: add word
Enter word: mug
Enter part of speech: noun
Enter definition: A drinking cup, usually cylindrical in shape
and having a handle

Enter command: add word
Enter word: arm
Enter part of speech: noun
Enter definition: The upper limb from the shoulder to the elbow

Enter command: print all
arm
noun
The upper limb from the shoulder to the elbow

banana
noun
Yellow fruit

mug
noun
A drinking cup, usually cylindrical in shape and having a handle
```

```
Enter command: print letter b
banana
noun
Yellow fruit


Enter command: add file
Enter file name: wordlist1.txt

Enter command: print all
arm
noun
The upper limb from the shoulder to the elbow

banana
noun
Yellow fruit

building
noun
A relatively permanent enclosed construction over a plot of
land, having a roof and usually windows and often more than one
level, used for any of a wide variety of activities, as living,
entertaining, or manufacturing

computer
noun
A programmable electronic device designed to accept data,
perform prescribed mathematical and logical operations at high
speed, and display the results of these operations

fast
adjective
Moving or able to move, operate, function, or take effect
quickly; quick; swift; rapid

loudly
adverb
In a loud manner

mug
noun
A drinking cup, usually cylindrical in shape and having a handle
```

```
sleep
verb
To take the rest afforded by a suspension of voluntary bodily
functions and the natural suspension, complete or partial, of
consciousness


Enter command: print letter b
banana
noun
Yellow fruit

building
noun
A relatively permanent enclosed construction over a plot of
land, having a roof and usually windows and often more than one
level, used for any of a wide variety of activities, as living,
entertaining, or manufacturing


Enter command: add file
Enter file name: wordlist2.txt

Enter command: print all
accompanying
verb
To go along or in company with; join in action

action
noun
The process or state of acting or being active

and
conjunction
Along or together with

arm
noun
The upper limb from the shoulder to the elbow

banana
noun
Yellow fruit

being
noun
The fact of existing; existence
```

```
building
noun
A relatively permanent enclosed construction over a plot of
land, having a roof and usually windows and often more than one
level, used for any of a wide variety of activities, as living,
entertaining, or manufacturing

computer
noun
A programmable electronic device designed to accept data,
perform prescribed mathematical and logical operations at high
speed, and display the results of these operations

existence
noun
Continuance in being or life

fast
adjective
Moving or able to move, operate, function, or take effect
quickly; quick; swift; rapid

life
noun
The condition that distinguishes organisms from inorganic
objects and dead organisms

loudly
adverb
In a loud manner

mug
noun
A drinking cup, usually cylindrical in shape and having a handle

organism
noun
A form of life composed of mutually interdependent parts that
maintain various vital processes

sleep
verb
To take the rest afforded by a suspension of voluntary bodily
functions and the natural suspension, complete or partial, of
consciousness
```

```
with
preposition
Accompanied by; accompanying


Enter command: print letter b
banana
noun
Yellow fruit

being
noun
The fact of existing; existence

building
noun
A relatively permanent enclosed construction over a plot of
land, having a roof and usually windows and often more than one
level, used for any of a wide variety of activities, as living,
entertaining, or manufacturing


Enter command: print letter x
No words beginning with x

Enter command: add file
Enter file name: wordlist3.txt
Failed to add contents of file wordlist3.txt

Enter command: add files
Invalid command "add files"

Enter command: find banana
banana
noun
Yellow fruit

Enter command: find pineapple
pineapple not found

Enter command: quit
Invalid command quit

Enter command: exit
```

6. Class Definitions

The definitions for my two classes are listed below along with implementation notes. Again, remember: while you must implement these two classes, you do not have to use my specific implementation.

```
class DEntry {
public:
   void writeEntry(string w, string p, string d);
   void printEntry(ostream &out);
   string getWord();
private:
   string word;      // Word
   string part;      // Part of speech
   string defn;      // Definition
};
```

Notes:

- This class has no constructor because the data members are empty strings by default, and I saw no need for a parameterized constructor.
- `writeEntry()` is a "set" function to modify all three data members of an entry.
- The only data member I needed to directly access in my solution was the word in each entry, so that's the only "get" function I provided.
- I don't think we discussed this point in class, but any input or output function that takes a stream object as an argument should always pass the argument by reference. Printing to `cout`, for example, modifies that stream object, so your program won't work properly if you pass the object by value, thus allowing your function to work with a copy of `cout`.

The `Dictionary` definition is on the next page.

```
class Dictionary {
public:
   Dictionary();
   bool addWord(string w, string p, string d);
   bool addFile(string fname);
   bool find(DEntry &entry, string word);
   void printAll(ostream &out);
   void printLetter(ostream &out, char letter);
private:
   DEntry entries[100];     // Array of dictionary entries
   unsigned size;           // # entries actually in dictionary
   unsigned posn(string w); // Position in which word w either
                            //  exists or should be placed
};
```

Notes:

- This class should contain a default constructor, but also, like `DEntry`, does not need a parameterized one.
- Each public member function essentially implements one required command.
- My add functions return a `bool` to indicate success (true) or failure (false). Possible reasons for failure: (1) the dictionary is full, (2) issues reading the file.
- The find function may look a little odd:
  - The function returns a `bool` so I can easily test whether the desired word is actually found in the dictionary.
  - The reference argument `entry` is used to copy the `DEntry` containing the word (if found) so it can be printed outside the function. You could print the entry inside the function, making that argument useless, but you'd need to pass in an appropriate `ostream` argument in that case.
- A couple of notes on the helper function `posn()`:
  - Remember, "helper function" is another term for a private member function—it only helps as part of other member functions.
  - My implementation does a binary search to find the position in which either its input argument is stored (technically, the entry holding that word) or where a new entry with that word should be placed.
  - This function can obviously search for a particular word (find command), but I also used it to (1) find where to add a new word to the dictionary, and (2) figure out where to start printing words beginning with a specific letter.
  - One issue: depending on implementation, when adding a new word, the position may be off by one spot. You can test for cases in which that happens and adjust accordingly.