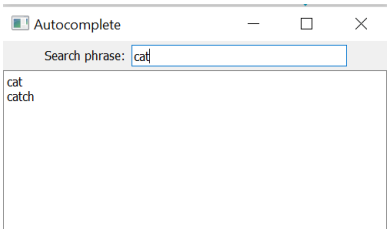
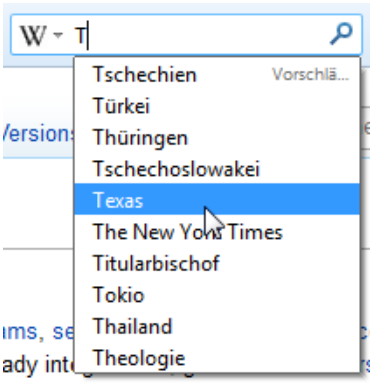
CS 132, Spring 2021
Programming Project #3: Autocomplete (20 points)
Due Thursday, April 29, 2021, 11:59 PM



This program focuses on classes and objects. Turn in five files named `WordCount.h`, `WordCount.cpp`, `CountList.h`, `CountList.cpp` and `autocompletemain.cpp`. You will also need the graphical starter project `autocomplete.zip` from the course web site; place your files in the `src` directory.

Program Description:



Many websites and other computer programs try to predict what a user is typing as they enter data and show the user a list of suggestions. In this assignment you will be writing a set of classes to generate a relevant list of suggestions.

Notice that in the screenshot at right the suggestions are not in alphabetical order. Instead, they are ordered by their popularity. A user is most likely to type in a word that they (or others) have typed in many times before. In order to figure out how popular a word or phrase is we will need to know how many times it has occurred before.

WordCount:

The **`WordCount`** class stores information about a particular word or phrase's popularity. It should have the following public member functions:

Member Function	Description
`WordCount(`**`phrase, count`**`)`	Constructor that takes a phrase and a number of times it has been seen before as arguments. The name will be a `string` and the count will be an `int`.
`WordCount(`**`phrase`**`)`	Constructor that takes a phrase as an argument. The phrase will be a `string`. Sets the count to be 1.
`getPhrase()`	Returns the phrase text.
`getCount()`	Returns the number of times the phrase has been seen before.
`getNGramLength()`	Returns the number of words in the phrase. Consider a word to be any text separated by whitespace.
`add()`	Adds 1 to the count.
`add(`**`amount`**`)`	Adds the passed in amount to the count.

You must also implement the following operators for objects of type `WordCount`:

Operator	Description
`<<`	Prints the `WordCount` to the passed in stream in the format (**`phrase : count`**). Phrases should be printed in all lowercase.
`==`	Returns `true` if the phrases are the same and the number of occurrences are the same. Returns `false` otherwise.
`<`	Compares two `WordCount` objects and returns `true` if the first should go before the second, `false` otherwise. A `WordCount` should go before if the count is larger. If the counts are exactly the same then the `WordCount` with the larger amount of words should go before. If the number of words is the same too then return `true` if the first `WordCount` is alphabetically before the second `WordCount`.

CountList:

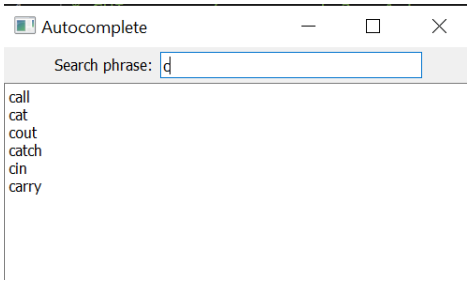
The **`CountList`** class stores information about a list of `WordCount`s. It should have the following public member functions:

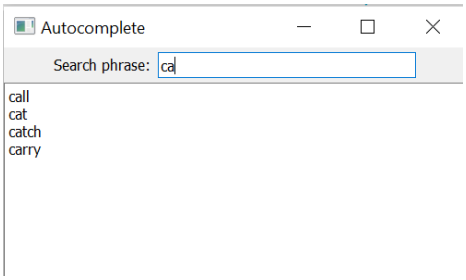
Member Function	Description
`CountList()`	Constructor that creates an empty `CountList`.
`CountList(`**`fileName`**`)`	Constructor that takes a `string` representing a file name as a parameter. Reads the contents of the file with the passed in name and stores the data in the `CountList`. Assumes each line in the file contains a number followed by a phrase. The number represents the popularity of the phrase (times it has occurred before). The phrase may be multiple words long and contain spaces.
`add(`**`wordCount`**`)`	Adds a `WordCount` to the list so that the list remains in sorted order. The list should be sorted by occurrence count with the highest count at the start. If `WordCount`s have the same count of occurrences they should be ordered by length of phrase. Longer phrases should come first. If the phrase length is also the same, the `WordCount` with the alphabetically earlier phrase should come first.
`getSize()`	Returns the number of items in this `CountList`.
`getTotalCount()`	Returns the total number of words counted (the sum of the counts of all `WordCount`s in the `CountList`).
`get(`**`index`**`)`	Returns the `WordCount` currently stored at the given index (0-based).
`getTop(`**`amount`**`)`	Returns a `vector` containing the passed in number of phrases and counts. These phrases and counts (`WordCount`s) should be the first (most popular) in the `CountList` and appear in the same order as they do in the `CountList` and be all lowercase.
`getStartingWith(`**`prefix`**`)`	Returns a `vector` containing all the phrases and counts (`WordCount`s) from the `CountList` that start with the passed in prefix string ignoring case. These should appear in the same order as they do in the `CountList` and be all lowercase.
`getStartingWith(`**`prefix, max`**`)`	Returns the same thing as the function above except that the returned `vector` will have the maximum length of the passed in number. Your code should stop looking for phrases once it has found the maximum number.

You must also implement the following operators for objects of type `CountList`:

Operator	Description
`<<`	Prints the `CountList` to the passed in stream in the format [(**`phrase : count`**), (**`phrase : count`**), ..., (**`phrase : count`**)]

AutocompleteMain:





`autocompletemain.cpp` should contain a `main` function that can run your program. This program should prompt the user for a file name and construct a `CountList` with that file's contents. It should then prompt the user for a prefix and print out the top (max of 10) recommendations. It should keep prompting the user for new prefixes and printing out recommendations until the user hits enter without typing in any text. When this happens, your program should print out the text `Goodbye!` And halt.

Graphics:

A graphical main program called **`autocompleteguimain.cpp`** is included in the provided starter zip. You do not need to alter this or to change your code files in order to run it. Just place your `CountList` and `WordCount` files in the `src` folder of the starter project, reconfigure and rebuild. When you run, you should see a prompt for a file name. Once you type one and hit the enter button, you should see a window like the screenshot to the right. As you type in the top textbox

you should see the autocomplete suggestions in the box below change.

Make sure that your classes can run without any issues before you run the graphical main! It will be much easier to debug with a text-only program.

Sample Output:

For full credit your output should match the expected output exactly. User input is displayed as bold and underlined in these samples so that it is easier to tell apart from your output.

```
Welcome to CS 132 Autocomplete!
Choose a dictionary file and then type in phrases to autocomplete.
Hit the enter key without typing anything to exit the program.

Dictionary file? small.txt

Phrase to complete? c
Suggestions: [(call : 12), (cat : 5), (cout : 3), (catch : 2), (cin : 2), (carry : 1)]

Phrase to complete? ca
Suggestions: [(call : 12), (cat : 5), (catch : 2), (carry : 1)]

Phrase to complete? cat
Suggestions: [(cat : 5), (catch : 2)]

Phrase to complete? catt
Suggestions: []

Phrase to complete?
Goodbye!
```

```
Welcome to CS 132 Autocomplete!
Choose a dictionary file and then type in phrases to autocomplete.
Hit the enter key without typing anything to exit the program.

Dictionary file? small.txt

Phrase to complete? there
Suggestions: []

Phrase to complete? D
Suggestions: [(dog : 12)]

Phrase to complete? BI
Suggestions: [(bike : 12), (bin : 1)]

Phrase to complete?
Goodbye!
```

```
Welcome to CS 132 Autocomplete!
Choose a dictionary file and then type in phrases to autocomplete.
Hit the enter key without typing anything to exit the program.

Dictionary file? medium.txt

Phrase to complete? the ot
Suggestions: [(the other : 38), (the other boys : 4), (the other pirates : 4)]

Phrase to complete? DAYS
Suggestions: [(days and : 11), (days and nights : 3)]

Phrase to complete?
Goodbye!
```

Development Strategy:

This program has a lot more files than our previous assignments so it can feel intimidating at first. However, most of the functions you need to write are a very small amount of code. We suggest working on this program in the following order:

1. Write the `WordCount` header file. This will include all the function headers for the required functions as well as any member variables.
2. Write the `WordCount` cpp file.
   a. Start with the three getter member functions. These just return the object's state.
   b. Create a mini test main for yourself that creates a `WordCount` and prints out its values. This will help you check that your getters are working.
   c. Next, add the `add` member functions to `WordCount` and test them in your mini test main.
   d. Add a constructor to `WordCount`. Test this in your mini test main.
   e. Add the << operator to `WordCount` and test this in your mini test main.
   f. Add the == operator to `WordCount` and test this in your mini test main.
   g. Add the < operator to `WordCount` and test this in your mini test main.
3. Write the `CountList` header file. This will include all the function headers for the required functions as well as any member variables.
4. Write the `CountList` cpp file.
   a. Start by initializing your `CountList` to store some fixed data so it will have some contents to use to test your other member functions on.
   b. Add the `getSize` and `get(`**`index`**`)` member functions. Create a mini main and test these, just as you did for the `WordCount` member functions.
   c. Add the << operator to `CountList` and test this in your test main. This will make your future testing easier.
   d. Add the `getTop` member function. Test this in your mini test main. You may want to add a function to print a `vector<string>` to help you debug and test more easily.
   e. Add the `getStartingWith(`**`prefix`**`)` member function. Test this in your mini test main.
   f. Add the `getStartingWith(`**`prefix, max`**`)` member function. Test this in your mini test main.
   g. Add the `add` member function. Test this in your mini test main.
   h. Add the constructors. Test them both in your mini test main.
5. Write `autocompletemain.cpp`. Make sure you use your `CountList` and `WordCount` whenever possible.
6. Test your `WordCount` and `CountList` classes with the provided graphical main.

Implementation Guidelines:

Your code should be well-structured and avoid redundancy. If you find yourself writing redundant code or with a very long function, add a `private` function. You may not introduce any other `public` members to the required classes, although you can add as many `private` member functions as you would like.

Make sure that you do not reimplement one class's functionality in another. For example, if `WordCount` has a member function to do a task, don't rewrite that logic in `CountList`. Instead, call the function on your `WordCount` variable.

Avoid unnecessary data members: Use data fields to store the important data of your objects but not to store temporary values that are only used within a single call to one member function.

In all files, you should follow general past style guidelines such as: appropriately using control structures like loops and `if/else` statements; avoiding redundancy using techniques such as functions, loops, and `if/else` factoring; properly using indentation, names, types, variables; and not having lines longer than 100 characters. You should properly comment your code with a proper heading in each file, a description on top of each function, and on any complex sections of your code. Specifically, place a comment heading at the top of each member function of `WordCount` and `CountList`, written in your own words, describing that member function's behavior, parameters, and return values if any.