Overview: For this project, you will write a small software system that again counts occurrences of words, as in Project 1. But, its implementation will follow a strictly object oriented design that employs inheritance and polymorphism.

You will expand the inheritance hierarchy provided in the WordDataList subclass example to include two WordList subclasses, one employing a vector, and the other employing a circular linked list with iterator.

Using the provided WordList example inheritance hierarchy from class, available on acad and the webpage for this project, add new WordList subclasses named WordVector and WordCList, with both implementing WordList’s pure virtual functions. The WordVector subclass of WordList will have a vector<WordData> to hold its data and overrides the printPtrRecursively() member function as well; it uses an iterator as the pointer. WordCList’s data member will be a templated circular-linked list object that you created in Project 2. The inherited virtual member functions you implement for printing will use the list iterator wrapper class you wrote to access the linked list, printing the contents of each node using the existing stream insertion operator for an ostream and a WordData object.

Write a menu-driven test driver with options exactly as in the box. Once it’s thoroughly tested, submit it to fulfill the first part of this project (see deliverables below).

Update the provided WordList application as follows:
- Update the menu.
- Each time the user makes a selection, the WordList pointer TheList is to be instantiated to point to an object of the appropriate type.
- Use polymorphism to call the correct implemented version of the WordList print functions. These calls are already in the switch statement and may only require adding cases.
- Each time the requested action is completed, reclaim the memory pointed at by TheList.

Note: The call of parseIntoList() in app.cpp does not require updating. It should polymorphically call the correct version according to the object pointed at by TheList.

Once your program works, you are to add code to time the print functions (only). At the bottom of each table, print the time to execute in microseconds.

A program has been placed on acad and the web to demonstrate the high_resolution_clock, available in the <chrono> library. Look for this example in the Examples|Microseconds subdirectory. This demo is timing a call of frequency_of_primes(), which times how long it takes to find all primes below 100,000. You can see the call of a static method now on lines 29 and 31, and the method on line 33 to secure the value in microseconds.
Requirements, Notes and Suggestions:

- The WordDataList example is provided and can be executed as is. Use it as a guide for your subclass.
- Words in the linked list and vector should be maintained sorted ascending.
- Your program’s menu will have 8 options. The WordData list has the special recursive pointer option, the others just iterative and recursive (the pure virtual functions you implement). Option 8 is to exit.
- The program is still to be runnable from the command line. If there is a command line argument, your program will NOT prompt for a file name and will instead assume the command line argument is a file name. It will then read the file and process it in the same manner as if the input had come from the user, running each menu option and printing to the screen, labeling each output appropriately, including the menu option text. After all possibilities are exhausted, the program terminates.
- Timing program execution is imprecise on a computer that isn’t dedicated to the experiments. Test your program on the machine named *hermione*. This machine is used very lightly.
  - You must ssh to Hermione from acad if off campus.
  - Make sure that the timings of your program are realistic, as your 2nd homework will use the results of timing within your program.
- g++ requires a flag to enable a program that uses the `<chrono>` library to compile. You’ll need that when you compile (and link?) your program.
- Consider the use of worker functions to make recursion work.
- You must carefully and completely document your code and place id blocks in all files, including those provided from external sources (e.g. the course instructor).
  - Lazy, chintzy, or otherwise substandard documentation is not acceptable.
  - Programs with significant documentation issues will be penalized at least three letter grades.
- You will use the Doxygen tool to create an API for your project and post it on the web area of acad. This will be discussed in class, and there are links to information on the course instructor’s links page.
- This program will be graded using a script that runs it several times with different input files. Significant penalties will be levied against otherwise working programs that don’t work with the script. Scripts for the program will be discussed in class.

Deliverables:

⇒ All h and cpp files, and a makefile. You may want to use the makefile provided with the WordDataList example as a starting point. Writing a correct makefile is part of the assignment. It is to produce an executable named *app* (that name exactly) by default. Penalties for non-compliance.