CSCI 104 - Fall 2015 Data Structures and Object Oriented Design

- Due: Wednesday, October 7, 11:59pm
- Directory name for this homework (case sensitive): `hw4`
  - This directory should be in your `hw_username` repository
  - This directory needs its own `README.md` file
  - You should provide a `Makefile` to compile the code.
- Warning: Start early!

Skeleton Code

Some skeleton code has been provided for you in the `hw4` folder and has been pushed to the Github repository `homework-resources`. If you already have this repository locally cloned, just perform a `git pull`. Otherwise you'll need to clone it.

```
$ git clone git@github.com:usc-csci104-fall2015/homework-resources
```

Copy the contents of `hw4` (and its subdirectories) over to a `hw4` folder under your `hw_usc-username` repository.

Overview

Our project for the semester will be to build a simple web search engine from scratch. As you will see, it will require using quite a lot of the data structures you are learning about. At a high level, a search engine is based on the following components:

1. A crawler, which tries to retrieve all the pages from the web. You will write a simplified crawler as a later part of the assignment.
2. A program that parses the web pages to extract the relevant information, such as text, links, etc. You will do this here; not for HTML or another language (that would be a bit too much work), but for now just an extremely stripped-down version of Markdown.
3. A program that has access to a local copy of all stored and parsed pages, and provides quick lookup of all the pages that contain the words of the query.
4. A ranking algorithm that takes all results and puts them in an order of relevance to the given query. You will do that in a later assignment, once you have learned more about sorting.
5. Optionally, a way for users to log in so that the engine can learn about the preferences of an individual and output more relevant results. You might do that in a later assignment as well.

For this (first) part of the project, we focus on parsing and simply returning all answers to a query, without worrying about ranking or user customization. Notice, however, that you want to keep an eye on making your code well documented and **extensible** by using good object-oriented principles (encapsulation, loose-coupling, appropriate distribution of responsibilities, etc.) as you will be adding to it later. (That said, you will also be allowed to rewrite your code later.)

Step 1 (Extend sets, 15%)

The C++ STL `set` class does not provide a `set_intersection` or `set_union` operation. You should create a new class `myset` that includes these new operations by inheriting publicly from `std::set`. This will allow you to use all the normal `std::set` functionality from your new `myset` and you can just add the `set_intersection` and `set_union` functions.

You will need two different kinds of sets: sets which contain `string`s and sets which contain `WebPage`s (a class which you will implement yourself). The proper way to do this is to create a templated class `myset<T>`, which can be instantiated as `myset<std::string>` or `myset<WebPage*>` or any other data type. Since you do not know how to create your own templated class (yet), you will instead implement two different classes: `MySetString` and `MySetWebPage`. The differences between these implementations should be very minor. You will likely revisit this and create a proper templated `myset<T>` class at a later time.

We have started this process for you in `myset.h` and `myset.cpp`

For everything that you do, you should now also switch to the use of the correct C++ iterator syntax, and use the STL provided iterators.

Step 2 (Parse Web Pages, 25%)

Your first challenge is to write a simplified MD parser. We want our search engine to be able to support alternate file formats (MD, HTML, etc.) so we created an abstract `PageParser` class with a parse method.

```
virtual void parse(std::string filename,
           MySetString& allWords,
           MySetString& allLinks) = 0;
```

You should create a derived class in your own files that implements this function to parse MarkDown. We will only support normal text and links in our Markdown format and parser. The text will consist of letters, numbers, and special characters. The interpretation is that any special character (other than letters or numbers) can be used to separate words, but numbers and letters together form words. For instance, the string `Computer-Science# 104 is really,really5times,really#great?I don't_know!` should be parsed into the words "Computer", "Science", "104", "is", "really", "really5times", "really", "great", "I", "don", "t", "know". In addition to text, you should be able to parse MD links of the form `[anchor text](link_to_file)` where `anchor text` is any text that should actually be displayed on the webpage and `(link_to_file)` is a hyperlink (or just file path) to the desired webpage file. A few notes about these links:

- The anchor text inside the `[]` could be anything, except it will not contain any `[`, `]`, `(`, or `)`. It should be parsed like normal text described in the previous paragraph
- The `link_to_file` text will not have any spaces and should be read as one single string (don't split on any special characters).
- There may be text immediately after the closing `)`. You should just treat it as a new word.

The goal of the parser is to extract all unique text words and identify all the links (i.e. all the `link_to_files` found in the `(...)` part of a link and return them in the `allWords` and `allLinks` sets that were passed-by-reference to the function.

If the contents of a file are...

```
Hello world [hi](data2.txt). Table chair desK, t-bone steak.
```

...then `allWords` should contain: `Hello`, `world`, `hi`, `Table`, `chair`, `desK`, `t`, `bone`, `steak`. In addition, `allLinks` should contain just `data2.txt`.

Step 3 (Write a WebPage Class, 15%)

After parsing a webpage we need to store the data and prepare it for either search and/or display. You should create a `WebPage` class for this. The start of its header file is provided for you in `webpage.h`. You'll want to store the filename, the set of all unique words used in the webpage as well as the incoming and outgoing links.

The function `allWords()` should return all individual text words that the parser found.

Outgoing links are those actually seen in the current webpage (i.e., they point from "this" webpage to some other page). These are all known immediately after parsing. However you should also store and track (for future assignments) the incoming links which are the page (file) names that link to "this" webpage. You'll likely need to add these little by little as you parse more pages. **Note:** We won't use the incoming and outgoing links directly in this assignment but we will test your code to ensure you are identifying and adding them correctly. We just wanted you to parse and store them now so you don't have to add that later on in the next assignment.

The operator << should be defined for this class and print (to the provided `ostream`) the page's text as seen in the file except when you encounter a link. For links you should just print the anchor text in brackets, but not the parentheses or the file name. So for the example file above you'd display:

```
Hello world [hi]. Table chair desK, t-bone steak.
```

To generate this display text you do not have to compute and store it ahead of time but are welcome to open the file and read/parse the file contents each time `operator<<` is called. Just as a web browser would not cache and store all webpages in advance but instead go read the information from the specified site (i.e., link location) when a user wants to display or see the page.

You may add any additional member data and functions you desire to this class but do not change the given functions' interface.

Step 4(Create a Search Engine and Command-line User Interface, 45%)

Your actual search engine application will need a list of all the webpages to search. This would normally be done with some kind of web crawler application but for now we will just provide a text file (called an **index** file) that will contain the names of all the webpage files you need to parse and be able to search. Your user interface and the main application will be initiated in `search.cpp` which contains `main()`. The index file will be passed via the command line to your application:

```
$ ./search data/index.txt
```

We recommend that you create a subdirectory `data` to store this index file and the other webpage files just to keep your code and data files separated. We have already done this in the `hw4` folder provided via the `homework-resources` repository.

The contents of `index.txt` are the file names of the web pages themselves, one per line. Each web page is stored in its own file. Your program should then read in all the web pages whose file name was listed in the index. There will be no format errors in the index file other than possibly blank lines, which you should just skip and continue to the next line.

In `search.cpp`, we will implement the main user interface logic which will use the terminal and perform text-based queries via `cin/cout`. In implementing your user interface, be careful to follow our instructions precisely. We will test this automatically, and if you take too many liberties in designing the interface, it may fail our tests.

The query "." (a single dot) should exit the application. A query can be of one of the following three types:

- A single word: the user wants to see all the pages that contain the given word.
- `AND word1 word2 ... wordN`: the user wants to see all the pages that contain all of the given words. The number of words here can be arbitrary. There will always be at least one whitespace between each element (`AND` and any of the search term(s)).
- `OR word1 word2 ... wordN`: the user wants to see all the pages that contain at least one of the given words. The number of words here can be arbitrary. Same rules for whitespace as for AND queries.
- Any other type of query (such as "word1 word2") should be considered an error; print an error message and return to the query prompt. Be careful that a two word query does not cause two consecutive searches.

Queries should be case-insensitive, so if the user typed "USC", and a page contained "usc", that page should be displayed. In response to the query, you should tell the user how many pages matched his/her query, and if it was more than 0, display all the pages indicating their filename before printing out their contents one at a time. We have provided this logic to you in a `display_results(...)` function in `search.cpp`. You will need to complete the rest of `search.cpp` to implement the query interface just described.

In the future we will replace this interface with a graphical user interface. To support that cleanly, we recommend separating interface logic from actual search logic. To facilitate that, we have provided a `SearchEng` (Search Engine) class that can be used to store all the webpages and other indexing data as well as actually performing the search operations and returning the appropriate `WebPages`. In order to be able to answer queries, you should use a `map` that maps single words to sets of web pages containing that word. This map should get initialized when the program starts (i.e., when you parse all the webpages) or updated anytime a webpage is added. You will need processing beyond just a lookup in the map in order to answer multi-word queries; the `set_union` and `set_intersection` functions may be useful here.

Also, in order to not run into memory problems, you probably do not want the values in your map to be actually sets of WebPage objects themselves, as that would duplicate huge amounts of text. Instead, depending on where you store the web pages, you may want to use their indices (i.e., an integer index to a list) or pointers to WebPage objects. The exact choice here is up to you.

We have started this `SearchEng` class and require two member functions:

```
void add_parse_from_index_file(std::string index_file, PageParser* parser);
void add_parse_page(std::string filename, PageParser* parser);
```

The first function will be given the filename of an index file (which in turn contains names of all the webpage as described earlier in this section). You should read each file specified in the index file one at a time and update/add it (and its information) to your data structures. To parse the files use the `PageParser` pointer provided. By allowing the PageParser to be passed in (and being a base class pointer), the client can pass in a derived MD parser for one set of files, then call again with an HTML parser if HTML files are provided, etc.

The second function just reads a single actual webpage file and adds all the data. Think about how the first function might utilize the second function.

Add other functions to your `SearchEng` object to support the desired functionality. Remember, try to keep a clean interface between the user interface logic (which should go in the main application in `search.cpp` and search logic (which should go in this `SearchEng` class)

Step 5 (Review and Test Your Code, 0%)

Go back and review your code and consider if there are ways to organize it better, separate responsibilities into alternate classes, etc. Be sure to test it thoroughly with some of your own files (write dummy webpages with links, etc.) You are welcome to share those files you create with other students. Be sure there are no memory errors (not just leaks, but no memory errors like invalid reads, etc.). You may even consider writing some unit tests (using Google Test) for your classes. That way as you make changes you can re-run those tests and ensure you didn't accidentally break some other aspect of your class. Also remember to write your `Makefile` correctly to compile the necessary code when changes are made to your source.