Project 3
Over-Dosing on the C++ Standard Library:
Almost the Spell-Checker we *would* have Written

Due: 11:59 PM, Friday, March 6, 2009
Note: The Corrections & Clarifications posted on the project web site are officially part of these project specifications. You should check them at least at the beginning of each work session.

Purpose

This project is the last in the spelling-checker series. In this project, you will practice the following:

- Using some Standard Library containers and the `std::string` class.
- Using the Standard Library algorithms and the associated iterators, binders, inserters, and adapters together with your own function objects to practice these techniques.
- Using `std::string` with the algorithms as well as a fully-equipped class for performing string processing.
- Making a more complete use of exceptions for error handling.
- Solving a simple class design problem based on a division of responsibilities.

Overview

This project is a very close "clone" of Project 2; it is essentially how we would have done the Spell Checker if we had the Standard Library right from the beginning and were *determined to use it wherever possible*. In fact, in this project you will be "over-using" the Standard Library by writing code that might actually be simpler in a few places without it. This is to provide lots of practice with the algorithms in the Standard Library.

In particular, in this project, a key restriction is that the code that you write is not permitted to deal with any strings on the character-by-character basis that you probably retained from Project 1. All of your string processing will have to be done either with the STL algorithms that use the iterator interface that `std::string` supports, or by using the member functions of `std::string` itself. To put it concretely, your code may not use the `[]` operator to get at a single character in any string, nor can you write code that dereferences an iterator to do the same thing (or any other trick that circumvents the goal of this project).

A second concrete restriction is that you are not allowed to write any explicit loops (`for`, `while`, `do-while` or `if-goto` statements) except in a few specified places (details below). Instead of explicit loops, you must use the Standard Library algorithms.

The purpose of these restrictions, which will sometimes be inconvenient and clumsy, is to ensure that you have a chance to practice these facilities, which often work very well to produce simpler and clearer code. Even if using a Standard algorithm or other Library facility seems hardly worth the trouble, the purpose of this project is to become familiar with them, and so you should use them even if the savings in code is little, zero, or negative. However, in the instructor's solution, appropriate use of the Library generally resulted in fewer and simpler lines of code; yours should too.

There are two new chunks of functionality. The first is that the program will collect and output some statistics on word lengths and related values. These are output whenever the dictionary is displayed, and is always supplied for the document that is spell-checked or used to build a dictionary. This is close to trivial to supply with a function object class and an algorithm from the Standard Library.

The second is that the program will collect and output some cross-reference information when spell-checking. For example, upon request after a spell-checking session, the program will output a list showing for each word that was skipped, the line numbers in the input document where it appeared. An extreme case is that the program can show for each word the line numbers on which it appeared. This information will always be collected during spell-checking, but will be output only on request. Because of the large amount of output that might be produced, the user can control which items are listed. Details are below. The most important aspect of this new functionality is that *you are completely responsible for the design of it*.

Since this project involves several distinct design problems, they are presented in their own section below, after the behavior of the program has been specified.

Please read these specifications closely, and consult Stroustrup, Josuttis, the lecture notes, and *especially* the examples on the course website about function objects and use of the Standard Library containers and algorithms. The examples in particular will answer many questions - be sure to study them.

Program Behavior

Important

The behavior of the Project 3 program is specified by the Project 1 & 2 documents and their posted Corrections & Clarifications, together with the specifications in this document and the Project 3 Corrections & Clarifications. Project 3 has some additional features and somewhat different behavior than Project 2. In case of conflict, this Project 3 document and the Project 3 Corrections & Clarifications take precedence over those for Project 2 and 1.

Changes from Projects 1 and 2

1. When the program prints out a line of the input document during spell-checking, the line is preceded by the line number. The first line in the input document is number 1 (not zero). The cross-reference functionality will use these line numbers to refer to where words appear in the input document. There is no blank line between the document line and the command prompt.
2. The memory allocations command (**m**) and its code must be removed.
3. Instead of showing only the number of words in the dictionary whenever the dictionary is printed out with the **d** command, the program computes some statistics for all of the words in the current dictionary (such as average length) and prints the results. See the sample output for specifics of what it is computed and printed. This will be done with a component that will also be used to calculate word statistics while processing an input document (see #4 below). This component is the *Word Statistics Function Object Class* described on page 4. One of the statistics that this function object will compute is the number of words, so your program should not have the separate counters that it had in Projects 1 and 2.
4. After an input document has been processed to build a dictionary or to be spell-checked (the **b** and **c** commands), instead of the word count, the program outputs the same statistics in the same format as in the **d** command, but the statistics are for the words that were found in the input document. Only words found in the input document are included in the statistics; automatically replaced words, or replacement words entered by the user are not.
5. If a file cannot be opened, the program simply prints an error message and prompts for a new top-level command. This is to conform to the more uniform error handling approach in this project.
6. If a command both clears containers and asks for filenames, the program must ask for the filenames first, and then clear the containers only if the files can be successfully opened.

New Functionality - Cross-reference Feature

To the user, the output from the cross-reference feature consists of one or more words followed by a list of the input document *line numbers* on which that word appeared in a way that depends on the type of the cross reference. Each line number appears only once per word, and the length of this document line number list can vary widely - e.g. "the" will appear on a lot of lines, while "syzygy" might appear on only one line. Because of the wide range of number of line numbers, the user can control what range to output for some of the cross-reference output.

There is a new command, **x**, that is followed by a code letter for the type of cross-reference. Depending on the type of cross-reference, there will be additional parameters as described below. All of the cross references of the type are printed out, in alphabetical order by the word, and with the line numbers in order by their value, appearing only once for each listed word. The cross reference information (and the action item information) is collected only during a spell-checking command (**c**) and is retained until the next spell-checking (**c**), build (**b**), load (**l**), or empty (**e**) command, whereupon it is discarded. Thus the user can issue multiple **x** or **a** commands after spell check to view the cross reference and action list information in a variety of ways. See the samples for details in what follows.

The command has the following syntax and produces the following results. A couple of the cross reference types are very simple:

- **x a** - The program outputs the lines on which each added word was added. By definition, each word can be added only once during spell-checking, so each added word can appear on only one line. These words are the lower-case version of all words found in the input document that were added by the user. Only the line in which they were added appears. If no words were added, “None” appears - see the sample output for details.
- **x w <word>** - The parameter <word> is a whitespace-delimited string. The program outputs the lines on which the supplied word, converted to lower-case form, was found in the input document. As in the **x i** command (see below), the words involved are the lower-case forms of the words that appeared in the input document before any replacements, skips, or adds. In other words, these are all the words that were looked up in the dictionary prior to checking for add, replace, or skip. If the word was not encountered, “None” is output.

In the remaining commands, the code letter is followed by two integer values that allow the user to control the amount of output. The first, called <min> here, specifies the minimum number of input document lines for the cross-references we want to see. The second, <max>, is the maximum number of lines. Only those words are printed out whose number of lines is greater than or equal to <min> and less than or equal to <max>.

- **x i <min> <max>** - The program outputs the lines on which every word found in the input document appeared. The words involved are the lower-case forms of the words that appeared in the input document before any replacements, skips, or adds. In other words, these are all the words that were looked up in the dictionary prior to being replaced or skipped.

- **x s <min> <max>** - The program outputs the lines on which every word was skipped by the user or skipped by the program using its action list. The skipped words are the mixed-case actual words that appeared in the input document that were skipped. (That is, if an action item is created, this would be the match_str for the action item.) Every line where they were skipped, either by the user or by an action item, is included. If no words were skipped, “None” is output.
- **x r <min> <max>** - The program outputs the lines on which every replacement word appears. The replacement words are the mixed-case words that were supplied by the user, or were inserted by the program using its action list. (That is, if an action item is created, this would be the replace_str for the action item.) Every line where the replacement was done, either by the user or by an action item, is included. *Note:* The words in this set of cross references might not appear in any of the other cross-references, because the replace_str word might not be present in the input document. Since the input and output document are required to have the same number of lines and contents of the lines except for replacements, the user can apply these cross-reference line numbers to the output document to see where they appear.
- **x ' <min> <max>** - The program outputs the lines on which every *apostrophe word* appears. These are words containing an apostrophe after any possessive endings were stripped off (e.g. "don't"), in the lower-case form that was looked up in the dictionary.

Some examples:

- x s 5 20 will output the cross-references for each skipped word that appeared in at least 5 lines, but no more than 20 lines.
- x r 1 1 will output the line number for replacements that were done exactly once, showing the replacement word (not the original word).
- x i 20 30 will output the line numbers for all words in the input documents that appeared in at least 20 lines but no more than 30 lines.

Some specifics:

- When error checking the input of the <min> and <max> value, the program first checks for whether it can read an integer value for the first parameter, and then for the second parameter. If not, the program does the usual error processing and prompts for a new command. If both integers have been successfully read, it then checks for whether the two values met the following constraints:
  <min> >= 0
  <max> >= 0
  <max> >= <min>
- If this is not true, the program must print an error message (see the supplied strings.txt file) and prompts for a new top-level command. There is no upper limit to the maximum number, so the the user can supply a large number to ensure seeing all the cross-references.
- For reasonable neatness in the output, the output is formatted as follows:
  - Each cross reference appears on a new line. The word appears first followed by a colon and a space. Each line number appears in a field whose width is 5 characters. Up to ten line numbers can appear on a line of output. If more lines of output are needed, the next line will start with a series of spaces whose length is the length of the word + 2.
  - If there are no cross reference entries that satisfy the the user's command (or none at all of that type), the program outputs a message of “None” instead of the line containing the word and the first set of line numbers.
  - It will be easier to see the formatting of both the samples and your output if you view them in a fixed-width font (e.g. Courier) - your programming editor will often do this for you very conveniently.
  - Use the stream output format manipulators to get this result (see the output formatting handout for help). Trying do this with DIY code is a *lot* harder.

Design Problems

Choice of Containers and Algorithms

Other than the command map (see below), you must choose which Standard Library containers to use in the project. Note that some containers should be used in combination with a good choice of STL algorithm, while others can be used in the project with their member functions alone. Follow the guidelines presented in lecture and make some good choices. In order to give some practice in choosing and using, the requirements for this are very general: You must use at least *three different container types* in the project (the map container for the commands is required, and so is not covered in this specification). The term "container type" here includes these classes in the Standard Library: list, vector, deque, set, map, multiset, multimap, stack, queue, priority_queue. You have to use at least three different container types in your project, not counting the required command-map container. So using a map, list, set for the rest of the project would be OK, but not set, set, list.

Suggestions: A good choice depends on the situation, and has both reasonable efficiency and involves simple, clear code. More specifically, a linear time cost is rarely justified if a constant or logarithmic one is readily available, which in fact it almost always is, thanks to the power of the Standard Library. "I don't have time to learn about it" does *not* count as a factor in the choice of containers for this project - *the goal is to learn and practice using the containers and algorithms*.

Action List

As noted in the description of Action_item.h, .cpp below, you are free to change how the list of actions is implemented in the program - you may change the Action_item class in any way you wish - even get rid of it!

Word Statistics Function Object Class

To compute the dictionary and document statistics, you must define and use a function object class, all of whose member variables must be private, has a good division of responsibilities, and which has the fewest public member functions that will do the job. The dictionary statistics are to be computed using the contents of the dictionary container and an algorithm from the Standard Library. The input document statistics are to be computed while building a dictionary or spell-checking a document. It is neither necessary nor permitted to read any input files more than once to both process the document and collect the statistics - to do so would be egregious inefficiency.

Note: In computing an average, if the divisor is zero - and this is clear in the output - it is better practice for code to output zero for an average rather than let the average value be the symbol `NaN` (Not a Number) that most implementations report in such circumstances.

Cross-reference Functionality

The cross-referencing facility is up to you to design. There are two restrictions on your design:

1. The cross-reference data has to be *collected and saved during the spell-checking process*, and then output on request; it cannot be generated from a separate scan of the input document, either by reading the file or by scanning a copy of the document in memory. This means you will be using some kind of container(s) to hold the information while the document is scanned, and then access the container(s) to output the cross-reference information.
2. You must have *at least one class* in your design. Do not take this specification to mean that more than one class is better than one; it means only that you must have a class (or more than one class) in the cross-reference design instead of a design with no classes at all.

All aspects of this class or classes and the other code for this facility must be well designed, following the guidelines and concepts presented in this course, and making appropriate use of the Standard Library.

The key to a good design is careful thinking to define very clearly the responsibilities of parts of the program: what is the client responsible for, and what are the cross-reference class(es) responsible for? Then design the new class(es) to encapsulate the responsibilities and hide the details from the client. Reviewing the material on basic class design may be helpful.

Warning: *Spend some quality time on this* - even though the cross-referencing is a simple facility, the quality difference between a sloppy hack and a thoughtful design is *extreme*.

Here is a more specific recipe for how to approach this problem:

1. Forget about the code, or even pseudocode, as long as possible. Jumping into coding specifics too soon is bad for design. Don't even think about the choice of containers you will use until just before it is time to start coding.
2. Write down in simple non-technical English a problem description: What has to be done for each kind of cross-referencing - what is actually involved? What's the input, what has to be stored, what's the output?
3. Then study this description - what does each kind of cross reference have in common and how do they differ?
4. Consider the division of responsibilities - which part of the entire program knows best how to handle what information or make what decisions?
5. Then create an idea for how to do the cross-referencing, taking advantage of the commonality, minimizing how much code deals with the differences, and dividing the responsibilities between the parts of the program best suited to handle them.
6. While all design involves tradeoffs, a characteristic of many good designs is that they are good for multiple reasons - for example, not only is the division of responsibilities good, but one of the components is actually quite general and easily reused for additional kinds of cross-referencing purposes. For example, a good solution that works for a couple of the cross-reference types will be trivially applicable to the other cross-reference types. Test your design by considering what would have to be done to add yet more cross-reference types - For example, one for all words that start with an upper-case letter in the document.
7. Most design problems have relatively simple solutions, so if your solution is more conceptually complex that your English problem description, something is probably wrong; rethink from the beginning.
8. Don't try to choose the container(s) involved until you have completed all of the above design steps. Choosing the container(s) and writing the code will be fun and easy, if the design is good.
9. Finally, don't worry about getting the output to look good until you have solved everything else. Nice-looking output is a nasty coding problem, but only a coding problem, so don't let it distract you from getting a good basic design.

The instructor's solution for the cross-referencing problem is painfully simple, elegant, and easily extensible, so if yours is very complex, heavily special-cased, or contains near-duplicated code or decision-making, it's time to take a break and rethink the problem.

Programming Restrictions and Requirements

Explicit Loop Restrictions

A key goal of this project is to get practice with Standard Library programming techniques involving the "STL" components: the containers, algorithm function templates, adapters, binders, iterators, inserters, and using your own custom function objects where necessary. To make sure you practice these techniques, there are restrictions and requirements in how you write the code for this project, especially in how your code does iterations. By *explicit loop* is meant code that you write that explicitly uses the `for`, `while`, `do-while`, or `if-goto` constructs to iterate. In this project, you must avoid explicit loops when Standard Library algorithms will work well.

Specifically, you may *not* use an explicit loop for the following, but must use Standard Library *algorithms* (with the iterators, adapters, binders, or inserters), or `std::string` or container *member functions* instead, where they usually will work quite well, or at least instructively:

- Inputting or outputting the dictionary (the **l**, **d**, and **s** commands).
- Computing the dictionary statistics.
- Outputting the action list or cross reference information.
- Searching the dictionary or action list.
- Any loop that scans or processes individual characters in a string.

On the other hand, you *must* use an explicit loop for the following purposes because trying to use the Standard Library algorithms doesn't help and is a serious inconvenience:

- The top-level command loop
- The add/skip/replace command loop
- The loop that reads in each line from the input document (in performing a **b** or **c** command)
- The loop that processes each word found in the input document (in performing a **b** or **c** command)
- A loop, if desired, that repeats the word search if a non-word was found (see below)
- A loop that skips the rest of the input line after an error message is printed

Which explicit loops are allowed for finding a word. In order to make the code reasonably straightforward, an explicit loop is allowed around the process of finding a word in the line, but you may not use any explicit loops to scan the line for the characters making up a word - you have to do this with algorithms like `find_if` or `std::string` functions like `find_first_of`.

The basic logic is that first you have find something that is delimited like a word, and only then can you tell how long it is. (There might be other ways to do it, but this seems easiest to think of.) If the something isn't long enough, you have to look again. So in this concept, you first scan for something that might be a word - which has to be done with Std. Lib. techniques - and then if it isn't really a word, you look for another candidate word - where an explicit loop is allowed. Of course, if it is a word, you use it and then go find another candidate word.

To explain a bit more: In your Project 1 and 2 solutions, you probably had some kind of find-word function that scans for a possible word and reports failure or success with the found-word. In Project 3, the find-word function will be implemented differently, but you are allowed to have an explicit loop around the call to this find-word function. In addition, you are also allowed to have an additional explicit loop inside the find-word function in which, if the candidate word is too short, the scanning operation is repeated to find the next candidate word - this is the same pattern as in the instructor's solution excerpt for Project 1.

String Processing Restrictions

You may not write code in which you use the `std::string` subscript operator, nor may you use `std::string::iterator` objects to store or retrieve individual characters from any string. You must use the algorithms and other string member functions to avoid this "C-like" level of programming. You may not use any other approach that avoids using the algorithms and other string member functions to scan or manipulate the contents of a string. This practice is what the project is about.

Thus, all processing of input file lines or words must be done with the scanning and modifying member functions of `std::string` or algorithms using `std::string::iterator` to manipulate the contents of a `std::string`. For practice in using these functions and algorithms, you are not permitted to manipulate any of the characters in the strings directly using isolated iterators, subscripts, or pointers. For example, the instructor's solution for Project 2 included a function to generate the lower-case version of a String, which translated to `std::string` would look like this:

```
string tolower(string str) { // forbidden code
   for(int i = 0; i < str.size(); i++)
   str[i] = tolower(str[i]);
   return str;
}
```

This code would be forbidden this project. Also forbidden would be:

```
string tolower(string str) { // forbidden code
    for(string::iterator i = str.begin(); i != str.end(); i++)
    *i = tolower(*i);
    return str;
}
```

And so would this code:

```
string tolower(const string& str) { // forbidden code
    string result;
    for(const char * i = str.c_str(); *i; i++)
    result += tolower(*i);
    return result;
}
```

All of these are fiddling with individual characters in some way in an explicit loop. Instead, one can write one line of code using a Standard algorithm that creates a lower-case copy of a string. Learning how to do this sort of thing is the goal of the project. Similarly, instead of scanning the input line looking at individual characters in a loop to see if they are part of a word, you can do this with member functions of string or Standard algorithms in a few lines of code. Give it a try!

Other Specific Programming Requirements

1. ***Uniform error handling with exceptions.*** User input errors in the top-level commands, such as unrecognized commands, bad filenames, numeric input read failures, or invalid numeric values, must result in an `Error` exception object (defined in Utilties.h) being thrown and caught in the top-level command loop. This is where the program prompts for a top-level command and then calls the appropriate function. This loop should enclose a `try` block followed by a `catch (Error&)` block that outputs the message carried by an `Error` exception object, skips the rest of the console input line, then allows the loop to prompt for a new command. See the strings.txt file for a list of the message strings that will be used with an `Error` exception object.
   - *Note*: The individual add, skip, and replace action commands for each unrecognized word during spell checking are *not* top level commands. Errors in these commands should be handled in whatever way is convenient, keeping the program in spell-checking mode. (If the user makes an error on a spell-checking commands, you do not want to interrupt the spell-check process!)
2. For practice in a somewhat exotic use of a map container, you must use an `std::map` container to translate from characters for commands to the functions that do the commands, using the technique presented in lecture. However, for simplicity, check for the **q** command first and act on it directly This requirement applies only to the top-level commands like **b** or **c**, not to the commands **a**, **s**, or **r** entered during spell-checking mode.
   - *Note:* In C++, `std::exit()` does not have the same effect as returning from `main()`. Your program should terminate by returning from `main()`.
3. It is not necessary for you to allocate dynamic memory yourself anywhere in this program - the Std. Lib. classes, properly used, will do it all for you.
4. You must use `std::string` everywhere your `String` class was specified in Project 2.
5. You may not declare or allocate any character arrays or use the `<cstring>` functions anywhere in this program.
6. Anywhere that a one of the STL adapters, binders, inserters, or iterators can be used directly in an algorithm, you must use it instead of your own custom function objects.
   - For example, there are places where a stream iterator can be used in an algorithm, and should be used instead of a custom function object.
7. You should still use C-string constants for purposes such as output message text, as literals or as `const char * const` variables. There is no advantage to wrapping such constants in `const std::string` objects.
   - Note that file-scope `const` variables in C++ automatically get internal linkage.

General Requirements

To practice the concepts and techniques in the course, your project must be programmed as follows:

1. The program must be coded only in Standard C++, appropriately idiomatic. Be sure to review the C++ Coding Standards while working on your code, and again before submitting your code.
2. If you wish, you can use the TR1 `bind` facility (see the handout) which is available in gcc 4.x; sometimes this is much more convenient in the STL algorithms than the Standard adapters and binders.
3. Use of inheritance, and other post-midterm concepts, is not allowed. This project, including the design problems, can be well-executed without any use of inheritance or other concepts and techniques to be covered after the midterm. In terms of the

design guidance presented, this is purely a "concrete classes" project. Thus use of inheritance in this project will be considered a serious design failure - it would be an unnecessary complication.

4. You must follow the recommendations presented in this course for using the `std` namespace and the contents of header files. Review the relevant handouts.
5. Your use of the Standard Library should be straightforward and idiomatic, along the lines presented in the books, lectures, and posted examples.
6. Your program must be reasonably efficient. It is possible to misuse the Standard Library to produce a program that is grossly inefficient. While this project is definitely not a speed contest, gross inefficiency will produce not only a loss of code quality credit, but also a loss of autograder credit, because the autograder will time your code out after what should be an ample amount of time for reasonably well-designed code.

Required Components

The files listed below must be included in your submission to the autograder, and follow the specifications in this document and any supplied "starter" files.

Since we are using the Standard Library, the previous Ordered_array and String components are not needed and must be removed - they are not part of this project.

Action_item.h, .cpp

These files and the Action_item class can be directly recycled from Project 2, and you are completely free to modify this class in any way you please to support a good design and quality code in this project. In fact, you can even eliminate this class altogether if you want (just be sure to submit empty versions of these files to keep the autograder happy). The only restriction is that whatever you do with actions must be well-designed and consistent with the concepts and guidance presented in this course thus far (e.g. you can't make use of inheritance yet).

Utilities.h, .cpp

The supplied skeleton header file skeleton_Utility.h contains a class declaration for an `Error` class that must be used to create exception objects to throw when a user input error occurs. This class simply wraps a `const char *` pointer which the constructor initializes to a supplied text string. Thus, to signal an error, a function simply does something like.

```
if(whatever condition shows the input value is invalid)
   throw Error("Invalid input value!");
```

Otherwise, the rules for this module are the same as in previous projects, but can be made more specific as follows: Things belong in this module only if (1) they are general-purpose enough that they could be meaningfully used in some other quite different project, or (2) they are used in more than one of the other modules in this project. Misplaced code here is a serious design failure.

Xref.h, .cpp

These files will contain the declarations and implementation for your cross-referencing facility. Code specific to this program feature is expected to appear in these files. Since this facility is up to you to design, no further specifications can be made about the content of these files, other than that they need to conform to the guidelines and concepts presented in the course.

p3.cpp

This source file contains function main and all of the other declarations and functions for the project. In general, the contents of this main module should follow the same principles as in the previous projects, with the specifics for this project described below.

Important: Any special-purpose function object classes or templates needed for one or two specific functions should be declared in this file, and not in Action_items.h or Utilities.h. Cluttering these other files with unnecessary declarations is very poor design. See the C++ Coding Standards on this topic.

Suggestions

Using the `std` containers for the dictionary and action list is pretty easy; if you have to write a lot of code to do this, you are missing something - go back and read your notes and check the posted examples and Stroustrup or Josuttis.

Study the `std::string` class closely. Notice that it has an interface that works both in terms of subscript-like positions, and also in terms of iterators. The iterator interface is especially interesting, because you can then process a string using the STL algorithms to call functions, which turns out to work very well with certain parts of this project. For example, in the instructor's solution, the word-finding function collapsed from many lines of character-by-character hacking into a few fairly simple lines.

Some handy algorithms to consider are: `copy`, `find`, `find_if`, `for_each`, and `transform`. Take advantage of this good opportunity to learn to use stream iterators and the insertion adapters; they work extremely well here.

If you aren't sure how to use the Standard Library algorithms and adapters, first make sure you understand what the loop(s) need to do by sketching out the for-loop code that you would ordinarily write. Then translate it into the algorithm version. Worst case: code the for-loops completely, test and debug, then translate, and regression-test (do the same tests as before the modification).

Be prepared for astoundingly confusing error messages resulting from incorrect template-using code. Few compilers do this error reporting well. The g++ compiler messages are actually among the best, though at first glance they look like gibberish: the beginning of the message describes the context, then comes how the template parameters were instantiated, and then finally, the specific problem the compiler choked on. If you get stumped, copy-paste the offending code plus the complete and exact error message into an email to eecs381help and we will help you interpret it.

Some implementations define `tolower` and other `<cctype>` functions as macros or in other ways that might interfere with their being used as a function pointer argument for the Standard algorithms. Let us know if this seems to be happening and which compiler/platform you see it for. The workaround for this project is to define a simple wrapper, e.g. `tolower_f`, that hides the flakey implementation inside a normal-looking function.

Project Evaluation

Your program will be tested for overall output correctness by the autograder similarly to Projects 1 and 2.

Because this program will behave very much like Project 2, the autograding will have two parts: In the first part, we will test for whether your program behaves correctly using basically the "old" spell-checking functionality - the same as Project 2, with the only difference being in the document statistics and the removed **m** command. Then a separate set of tests will evaluate the correctness of your new cross-referencing functionality.

There won't be any separate component tests because the design of the cross-referencing and action list handling is up to you, so we can't specify an interface that would allow component testing.

Your final code will be human-evaluated for quality and whether it meets the specifications on how you implemented it. As in project 1, your project must get the threshold number of autograder points in order to qualify for human grading, and the human grading will count for most of the project credit - see the course syllabus for specifics.

The human-grading of your code will be for (1) conformance to the project requirements, (2) how well it follows the practices and techniques presented in this course, (3) quality of your cross-referencing design solution, and (4) the general quality of the code.