Project 1
A Spelling Checker Using an Encapsulated Generic Container in C
Due: Friday, January 30, 2009, 1 minute after 11:59 PM

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.

Without requirements or design, programming is the art of adding bugs to an empty text file.
- Louis Srygley

Introduction

Purpose

In this project, you will practice the following in the context of programming in C:

- Designing, writing, and debugging a somewhat complex procedure hierarchy.
- Designing well-structured, modular procedural code.
- Working with C-strings, structs, and pointers.
- Using dynamically allocated memory to store string data and an array of pointers.
- Doing C-style I/O, both with the console and files.
- Using opaque types, function pointers, and void * pointers to define an encapsulated generic container in C.
- Getting introduced to a form of object-based programming in C, in which data objects are created, used, and destroyed in a consistent fashion.
- Practice in using global variables in an appropriate way, and setting up the linkage for them.

This is a very challenging project in that you are doing some complex operations using nothing more than the basic operations and library available in Standard C. Your goal, however, is not just to get the program to work, but to structure the code well with a good design of the procedure hierarchy. This project is an excellent opportunity to practice good basic code design because a poor design will result in at least twice as much code to write, or code that is painfully difficult to debug or contains messy duplications. In contrast, a good design, with well-chosen functions, will make the code relatively easy to write, debug, and test. Much of this project code can be "recycled" for use in the next C++ projects, so well-designed code here will save a ton of time later.

Don't procrastinate on this project! Maybe you can code and debug in your sleep, but good design takes prime time when you are awake and alert!

Before starting on the project, reread the Syllabus on how your project will be evaluated. Note that you must get the threshold of 80% of the autograder points for your project to qualify for human grading for Code Quality which will count for 70% of the project score. Then go to course home page and read the documents *How to do Well on the Projects* and *C Coding Standards*. Also take a look at the Sample Code Quality Checklist on the project main page.

Background

Using computers to process string data is very common; probably at any point in time more computers are doing some form of text processing than are doing "number crunching." Moreover, one of the characteristics of string processing is that the programs have to be prepared to deal with data of any amount and variable sizes, making dynamically allocated memory very handy. While simple forms of spell-checking are easy to program, providing the spell checking function in an easily used and reasonably powerful form involves getting a lot of details right. Although this project requires that you deal with only a few of these details, it will give you an opportunity to see what is involved in creating a "real" spelling checker.

Abstract data types, containers

A key concept illustrated in this program is the notion of an "abstract data type" - a characterization of what a set of data is like, and how it can be operated on, that is independent of the details of how it is actually implemented. This project uses an abstract type that keeps items in order and searches for them. This type can be used both for the dictionary - the list of words used for spell-checking, and also for a list of "actions" - skipping or replacing words throughout the document. Using some C techniques, we can code this abstract data type in a way that enables the same set of functions to be used for both purposes. This is an Ordered_array that stores `void *` pointers that can point to values of any type; the items are automatically kept in order as specified by a user-supplied ordering function. Each of the n items in the array can be referred to with an index, whose range is $\{0, ..., n-1\}$, just like an ordinary array. The operations that can be performed are:

- Create, clear, and destroy an Ordered_array.
- Insert an item into the Ordered_array, automatically keeping the items in order; there is no limit to how many items can be inserted into the array.
- Find out whether an item is present in the Ordered_array, and if so, what its index (subscript) is.
- Retrieve an item from the Ordered_array given its index.
- Obtain the number of items in the Ordered_array, so that they can be retrieved in order by simply iterating through the possible index values.

Such a type is also called a "container" in the context of object-oriented programming, because you can put things into it, and take things out, and the container code keeps track of all the details for you. Due to the limitations of C, this container is rather limited in what kinds of things can be in it - we make it as generic as possible by making it a container of `void *` pointers. We can store any type of data in that space, as long as it will fit; for example, integers will usually fit into the space required by a void pointer. However, for any type that will not fit in the space occupied by a `void *`, the user must arrange to store the objects elsewhere, and then store pointers to them in the container. A `void *` pointer is not committed to point to any particular type of object, so the container code will work to store pointers to any kind of data.

Furthermore, by using an *incomplete type declaration* in the header file for the container module, and allocating the container object only on the heap (with `malloc`), we can make the container an "opaque type" - the rest of the program code can not "see" into the container; the guts of the container are shielded from outside interference, producing encapsulation similar to using "private" with in a C++ class. This is an important C technique.

Finally, in terms of the implementation, we will take advantage of the fact that Ordered_array contains an array; the search for an item will be done with fast binary search, giving *O(log n)* for finding an item. However, inserting an item into the array will necessarily require linear time, *O(n)*, even if the location for the new item can be found more quickly (e.g. a modification of the binary-search code in Kernigan & Ritchie will give you the location to insert an item that was not found). A second efficiency feature is that if we have a set of items that are already in order, we can insert each one at the end of the Ordered_array, making it possible to fill the container much more quickly.

Basic design of the spell checker

The dictionary will be an Ordered_array containing pointers to C-strings that are in dynamically allocated memory. The actions will be in another Ordered_array as pointers to `Action_item structs`, which also will be in dynamically allocated memory.

The job of the main module of the program is to process the C-strings making up the input and output of the program, and the individual words to be checked and stored in the dictionary, and the actions in the action list. While the behavior of the program is well-specified, the design of this main body of the program is left almost completely unspecified to give you practice in designing a nontrivial function hierarchy - the fundamental and essential programming design skill.

Warning: A poor design in this project will double or triple the amount of code you have to write and debug. Design is not easy - be sure to give it some prime time!

Overview of this document

The next section. *Summary of Functionality*, describes the capabilities of the spell checker program, followed by a section of *Detailed Specifications*. whose purpose is to tell you exactly what the program's behavior is supposed to be. Then comes a section on *Required Components*, that describe each module of the program in terms of the source files that are supposed to be present. Together with the supplied "starter" files on the course web site, this information describes the basics of how your program needs to be structured and organized, especially the interface for each module. The next section, *Programming Requirements* describes the specific programming practices, concepts, and techniques you must use in the project. This is a lot of information, but constructing real software also involves a lot of information - so learning how to deal with it is an important goal. A penultimate section briefly summarizes how your project will be evaluated. The final section, *How to Build this Project Easily and Well*, provides useful advice - take it seriously!

Summary of Functionality

In summary, your spell checker must perform the following functions:

1. It uses a list of correctly spelled words (the *dictionary*) kept in a text file between runs of the program. The user must either load an existing dictionary or build a dictionary in order to spell-check a document. No duplicate words are allowed in the dictionary.
2. To help the user get a dictionary started, the user can specify that the program run in a special *build-dictionary* mode in which it starts with an empty dictionary and then automatically adds all the words found in a specified *dictionary creation document* to the dictionary.
3. To spell-check a file (the *input document*), the program finds each word in the input document, and compares it to the dictionary. If it finds a matching dictionary entry, the word is considered correctly spelled.
4. If the word is not in the dictionary, the program checks the *action list* to see if the user had previously specified that the word was to be skipped or replaced with another word. If so, the word is skipped or replaced accordingly.

5. If the word is neither in the dictionary, nor in the action list, the program displays the line of the document that the *questionable word* appeared in. The program asks the user whether the word should be *added* (added to the dictionary), *skipped* (not added to the dictionary), or *replaced* by another word. If the word is added, the user should not be asked about it again for the rest of the document. If the word is skipped, it is ignored on this occurrence and not added to the dictionary. If the user's choice is to replace the word, the program obtains the new word from the user and replaces the original word with the new word. For simplicity in this project, it does not check the new word against the dictionary, but just accepts it as-is from the user.
6. For words that are skipped or replaced, the program asks the user if he or she wants this action to be done for the rest of the document. If so, the program stores the skip or replace action in the action list for future use in step 4 above. No duplicate words are allowed in the action list; that is, a word can be skipped, or replaced, but not both, and a word can be replaced with only one other word. Given the specifications in #5, it should be impossible to be in a situation where a word already in the action list could be added again to the action list.
7. If the user replaces a word, then the replacement must appear in the line if it is displayed again. See the sample output.
8. The program creates an *output document* file that is a corrected copy of the input document, containing the replacements for any changed words. The output document retains all other aspects of the original, that is, it has the same linebreaks, whitespace, and punctuation as the original.
9. The program keeps track of the number of new words added to the dictionary, and reports it after completing a spell-check.
10. The user can save the current dictionary to a specified file for future use.
11. After either building or loading a dictionary, the user can spell-check multiple documents. The dictionary is cleared (emptied) whenever a dictionary is loaded, and the list of actions is cleared at the beginning of spell-checking a document (that is, the actions produced from one spell-checking are not available for another spell-checking).
12. There are no limits to how big the dictionary can be (except available system memory), and there are no limits on how large an input or output document can be (except for disk space).
13. The program will keep track of the amount of memory allocated for string data, the dictionary, and action list, and will print out this information in response to a command.

Detailed Specifications

Definitions of words, action items, and ordering

1. For purposes of spell-checking, a word is defined as any string of alphabetic characters that is delimited by whitespace or punctuation and is at least two characters long but no more than 31 characters long. A word can also be delimited by the beginning or end of a line, or as discussed later, a partial line, of the input document. Apostrophes (the single-quote character, written as a character in C with the escape character as `'\''`) that are embedded in a word will be considered part of the word, as described more below.
   - For what follows, see the handout on `<ctype.h>` *C Library functions for character types* on the course website. Alphabetic characters are the letters A through Z, either upper or lower case, as defined by the `<ctype.h>` function `isalpha`. The whitespace characters are defined by the function `isspace`, and punctuation by `ispunct`. Thus, words cannot contain numbers, whitespace, or characters like `'!'`, and to be considered a word, a string of characters must be preceded and followed by a whitespace or punctuation, or the beginning or end of a line.
     - Example: The string: `" carbon-14"` contains the word `"carbon"` because a space comes before the “c” and a punctuation character follows the `"n"`; the substring `"14"` is not a word since it contains only digits. In contrast, the string `" carbon14 "` does not contain a word: `"14"` cannot be part of a word, so the string delimited by spaces does not count, and the substring `"carbon"` is not considered a word because it is terminated by a digit; to be a word, it must be terminated (and preceded) by whitespace or punctuation.
   - Apostrophes are a special case because they can appear in the middle of words, unlike other punctuation. In this project, apostrophes will be accepted if they appear anywhere after the first and before the last letter of the word, but *possessive forms* are a special case: `"'s"` at the end of a string, and the apostrophe at the end of `"s'"`, are removed to get the actual word to be considered for spell checking. Only one level of removal is done, so the program does not attempt to "look past" typographical errors involving duplicated apostrophes or apostrophe-s sequences. The removed characters are treated like punctuation in that they are retained in the corrected output.
     - Example: "O'Malley" and "can't" are considered a single word in their entirety, while "frog's" is converted to "frog", and "students'" to "students" for the purpose of spell-checking.
     - Example of typographical errors: "Tom's's" is looked up as "Tom's", and "students'" with two apostrophes at the end is looked up as "students'" with a single apostrophe. "O''Malley" is looked up as-is.
2. Words typed in different case (upper/lower) must be recognized as the same word in the dictionary. In this project, the words in the dictionary must be all lower-case, so you must convert words to lower case for use in the dictionary and to compare against the dictionary. The output document, however, should preserve the original case of correctly spelled words.
   - Example: The program checks the spelling of "David Kieras" as if it were "david kieras" but if correct, the output document contains "David Kieras".
3. In contrast, the words in the action list must be kept in the exact case, which means that the program allows more than one action to be specified for the same word if they have different cases.
     - For example, if "Kieras" is to be skipped, then "Kieras" is specified by the skip action item, not "kieras". Similarly, the action list can show both that "hTe" is to be replaced by "The", and "hte" by "the". It is possible to both skip the word "Pig" and replace "PIG" with "DOG”.

4. Throughout this document, “alphabetical order” means in the usual alphabetical order and as defined by the `strcmp` function in `<string.h>` (see the handout *C Library C-string functions* on the course website). That is, if strcmp(s1, s2) returns a negative value, it means that s1 comes before s2 in the alphabetical order.

File formats and contents

1. The input, output, and dictionary files must be plain ASCII text files containing only printable characters (and whitespace) that have a newline character (`'\n'`) at the end of each line, including the last line. Binary-format files will not be used in this course, and processing files with word-processor formatting codes is beyond the scope of this project.
2. The file names are entered as whitespace-delimited strings with no embedded whitespace. This makes them very easy to read in from the console.
3. The output document needs to have the same punctuation and whitespace in it as the original. The only way the output document can differ from the input document is words that have been replaced.
4. The output dictionary file contains one word per line with a newline at the end of the last line. The words must appear in alphabetical order.
5. Your program should assume a maximum length of 31 characters for words, words input from the console, and filenames, and a maximum length of 127 characters for an input line from the input document. These limits should be `#defined` in `p1.c` (see below). Your program should take advantage of these limits to simplify how filenames, words, and lines are read in, but at the same time, you must ensure that when these strings are input, they cannot possibly overflow the array used for storage! This is the notorious buffer overflow error that is the cause of so many security vulnerabilities in badly written network code. By proper selection and use of input functions, and correct code for scanning the input, it is easy and convenient to ensure that overflow never happens.

Handling too-long input

A basic idiom in C programming is that arrays are fixed in length once declared. Thus you will read input into an array for a line, and then extract words for spell-checking into an array for the word. But you have to ensure that the arrays will never be overflowed if the input is too long to fit the fixed-length arrays. The maximum lengths of words and lines are reasonably large, but sooner or later, input will be encountered that is "too big" and your program will have to deal with it. The minimum solution is to just make sure that the arrays will not overflow, even if the program misses or mangles some of the words in the input. The maximum solution is that the program would somehow figure out what and where the actual words were in these too-long lines. But this would make the program much more complex in ways that most C programming tries to avoid. Correspondingly, in this project your program is required to do only the minimum solution in handling too-long input. This section describes how your program should work.

1. Your program should **not** attempt to detect and signal whether a too-long word has been encountered in the input. Instead, it should simply collect characters that can form a word (including apostrophes). If it collects 31 characters and the next character is a word terminator (like a space or newline or a period), then fine - we strip off any ending apostrophe constructions and look up the word. But if it collects 31 characters without finding a word terminator, it stops collecting, discards the collected characters, and starts collecting again; since a word has to start with a delimiter, it will skip past characters until it finds a word delimiter and starts collecting again. The concept here is that a too-long string simply ends up treated as a non-word, like a date such as "1984" would be; the program simply discards the non-word string and looks for the next word.

2. Likewise, your program should **not** attempt to detect and signal whether a too-long line has been encountered in the input; instead, it should simply process the first 127 characters of any line in the input; any remaining characters should just be left in the input stream to get processed as another word or a continuation of the line, respectively.

Hint: It should not take any additional code to achieve this effect, only the simple and straightforward use of input functions.

More specifically, when your program tries to read the next line string from the input file, it should read a maximum of 127 characters. This string will either

- end with the newline character (`'\n'` - which is considered a whitespace character and thus a word delimiter) and is a *complete* line string, or
- it doesn't end with the newline character because the input file contains a very long line, meaning that the 127 character string is only a *partial* line.

Your program should assume that a word can start after a delimiter character or at the beginning of either a complete or partial line string, and must end either at a delimiter character or at the end of a partial line string. Yes, if extremely long words or lines appear in the input, some odd “words” will be found, or parts of real words might be ignored. Your program does not have to be "smart" and try to fix this situation, it just has to muddle through in a crash-proof way. See the “Length Problems” sample.

3. The situation when replacing a word is different - a too-long line might result even if the user puts in a legal-size word. In this case, your code will have to detect that the line length will be exceeded, not perform the replacement, and print out the specified error messages. See below, in “Error Behavior”.

4. If the user supplies a replacement word (see the **r** command below), and types in a word that is longer than 31 characters then the program simply uses the first 31 characters, regardless of what they are, That is, if no terminating whitespace appears within the maximum word length, the program simply uses the first 31 characters read. Any excess characters are left in the input stream.

Thus, in no case may data be allowed to overflow the space set aside for it, and *the program must be stable and well-behaved and not crash or run amok no matter what it finds in the inputs*. Your code will be assessed in this regard both in correctness testing and in the code quality evaluation.

Other input and output requirements

1. When the action list is output, it must appear in alphabetical order by the word that is to be skipped or replaced.
2. When a command, filename, or replacement word is read from the input stream, any additional characters in the stream at that point are left in the stream ready for subsequent input operations to read. In other words, your program reads one character or whitespace delimited string from the console input at a time (not whole lines) and does not routinely "flush" or empty the input line. This allows the user to "type ahead", and allows you to write very simple input code using idiomatic forms of `scanf` calls. The rest of the input line will be skipped if the input is in error (see below). Please ask if you don't understand how to do this.
   - Some of you may be in the habit of always reading a whole line of input from the keyboard and always skipping the unused part of the line. This is not how console input will be done in this course! It is more idiomatic and simpler to read the input stream an item at a time - which gives you more understanding of how streams work.
3. If a dictionary file or document input file is empty, your program does not output any special-purpose message for the occasion. It simply processes the input files in the normal way, and outputs the normal messages. In other words, your program should handle this boundary case "automatically" with its "normal" code. For example, if the user asks to check a document, and the current dictionary is empty, the program simply goes ahead with the spell-check, and of course, the words in the input document will not be found in the empty dictionary.

Top-level commands and behavior

The sample outputs show the behavior of the program and the exact output messages produced. The strings used for these messages are available on the project web page so that you can simply copy-paste them into your code to avoid typing errors.

The top level of the program creates an empty dictionary and action list, then prompts the user for a command, does it, and then prompts for a new command. An unrecognized command results in an error message and a new prompt. The commands consist of single letters that can be preceded or followed by any amount of whitespace (including none, if the result is unambiguous). The commands are:

e - empty (or clear) the current dictionary and action list. The current dictionary is emptied: all its memory and that of the contained words is freed, and similarly for the action list. A message is printed (see the samples).

b - build a dictionary. The program empties the current dictionary and action list, prompts for an input document to be used to build a dictionary from, processes it, and outputs the number of words processed in the input document and the number of words added to the dictionary. The resulting dictionary is then kept in memory as the current dictionary.

l - (lower-case L) load a dictionary. The program empties the current dictionary and action list, prompts for a dictionary file name, and then loads it, and outputs the number of words present. The dictionary becomes the current dictionary. The program assumes that the specified file is the result of a previous save-dictionary command; if it is not, the results are undefined.

- ***Hint:*** If your build and load commands use the same code or approach to fill the dictionary, you have a serious case of egregious inefficiency, and rotten code quality as a result. Think this through!

s - save the current dictionary. The program prompts for an output dictionary file name, and then writes the dictionary contents to the file, and outputs the number of words saved. The dictionary remains in memory as the current dictionary.

d - display the current dictionary. The program outputs the current contents of the dictionary to the display. If the dictionary is empty, a special message is printed.

a - display the current action list. The program outputs the action list resulting from the last spell check to the display. If the action list is empty, a special message is printed.

m - memory allocations. The program outputs the current memory allocations. This consists of the total number of bytes allocated for all C-strings in the dictionary and action lists, the number of cells (void *'s) in use, the total number of cells allocated, and the total number of bytes for the allocated cells, separately for the dictionary and the action list. See the sample output for details.

c - spell-check a document. The action list is emptied. The current dictionary is used even if if is empty. The program prompts the user for an input document filename, and an output document filename, and then goes into "spell-checking mode" to process the document. When finished, it outputs the number of words processed in the input document, the number of words added to the dictionary and the number of items in the action list.

- ***Hint:*** If your spell-check command uses entirely separate code from your build command, you have wasted a lot of coding effort, and produced poor quality code as well. Think this through!

q - quit the program. All allocated memory is deallocated, a message is printed, and the program terminates.

Spell-checking commands and behavior

When spell-checking a document, if a questionable word is found that is not in the dictionary, and also not in the action list, the program outputs the line, outputs a message that the word is not in the dictionary, and prompts for a command. If the program reaches the end of the input file, it closes the input and output files and returns to the top level (see the spell-check command). The possible commands in spell-checking mode are:

a - add the word to the dictionary.

s - skip the word - it is not added to the dictionary. The user is asked whether to skip this word in the rest of the document; if so, a confirming message is printed.

r - replace the word. The user is prompted for a replacement word. The replacement word is simply the first whitespace-delimited string in the input, and thus can be *anything at all*. If no terminating whitespace appears within the maximum word length, the program simply uses the first 31 characters read. Any excess characters are left in the input stream. The questionable word is removed from the line and the replacement word is inserted in its place. The replacement word is used exactly as it is supplied by the user. The replacement word may be shorter, the same length, or longer than the original word. If the original word was followed by an apostrophe or apostrophe-s, those characters remain in the input line following the replacement word. If the same line is displayed again (for example, a second questionable word is found in the line), the replacement word must appear in the displayed line. The user is asked whether to perform the same replacement in the rest of the document; if so, a confirming message is printed.

These commands can be preceded or followed by any amount of whitespace. If the user puts in some other character than these commands, the result is an unrecognized command error; the program should behave the same way as for the top-level commands. When the program asks for whether the word should be skipped or replaced in the rest of the document, the program should read a single non-whitespace character; if it is `'y'`, it means "yes"; any other character means "no". Any excess characters are left in the stream. Notice that in this version of the program, there is no command to cancel or terminate the spell-check.

Error behavior

If your program outputs an *error message* in response to incorrect user input, such as a bad filename or an unrecognized command, then your program *must* empty the input line - it must read and discard characters until it has read and discarded a `'\n'` character. This allows the user to have a clear-cut point of resumption of input if they had been typing ahead, and frees your program of responsibility for trying to interpret typed-ahead input in the context of a user error. The supplied output message strings that count as error messages are identified in the supplied strings. More specifics:

- The program outputs an error message in response to an unrecognized command, both top-level and spell-checking commands. See the sample output.
- The program must check for input file opening errors. If it cannot open a file for input, it prompts the user for a new filename, and tries again, and continues until the user has input a valid filename. See the sample output for the messages. Likewise for output files, though opening errors are rare for output files. Any pre-existing output file with the same name is to be overwritten.
- Before performing a replacement, your code should check whether the maximum line length will be exceeded. If it will be, your code needs to do the following: If the user has just entered the replacement string as part of an **r** command, print an error message, clear the console input stream, and prompt for a new command. If the excess line length would be a result of a replace action in the action list, print the warning message and do not perform the replacement action.
- Note that since text is being read, there are no invalid input errors possible in reading from the files, or reading user-supplied file names or replacement words. It is not required to check for "hard" I/O errors.

Required Components

This section describes the required components that you have to implement according to specifications supplied here. Each component consists of a header file (a `.h` file) and an implementation (or source) file (a `.c` file).

- For some of these components, the complete header file is supplied on the project web page or file server directory. For others, a "skeleton" header file will be supplied that you must complete to turn into the actual header file.
- *If the complete header file is supplied, you may not modify it in any way.* If it is a skeleton header file, you must follow the instructions to complete the header file.
- In general, throughout this course, you must use the exact structure or class declarations, with the specified members and specified functions, using the exact same names, types, or prototypes, as specified in the skeleton header files. You may not add any additional functions or declarations to these header files: they specify the fixed "public interfaces" of the components.
- Many details of the specifications are included in the comments in these supplied header files to avoid cluttering this document with excessive detail. Study these specification comments carefully as you read this section.

Important: You must submit a file with each specified name, and only files with these names. Missing files will cause the autograder to fail your program with a build error, and excess files will be ignored by the autograder, probably causing a failed build.

`Ordered_array.h, .c`

See the supplied header file that defines the "public interface" of this component. You will write an implementation file (`Ordered_array.c`) that will have a complete declaration of the Ordered_array structure type and definitions for the functions whose

prototypes are in the header file. You can have other functions or type declarations as you choose, but they should be declared static so that they have only internal linkage for your component implementation. The constants required (such as the initial allocation) should be `#defined` in the `Ordered_array.c` file - there is no need for the rest of the program to have access to them.

The Ordered_array stores pointers to void; this means you can store pointers to any type of object such as a C-string, an action item, or anything else. However, if these objects are created in dynamically allocated memory, you are responsible for deallocating the memory before the Ordered_array is cleared or destroyed - the Ordered_array component does not have to know what your pointers are pointing to, and it does not attempt to manage your data for you - it is responsible only for building its array of pointers, keeping them in order, and deallocating this array when told to do so.

When created, the Ordered_array should contain an array of `void *` pointers with 3 cells. When an new item needs to be inserted and these three cells are filled, your code should allocate another array that is *twice* as big, copy the previous items into it along with the new item, and then deallocate the old array. When these six cells are filled, the allocation should be doubled again, and this process repeated as often as necessary. The only limit on the size of the array is the amount of memory available - your code should always check for a successful allocation and terminate the program if allocation fails.

You use Ordered_array as follows: First, define the ordering function that determines which of two items should come before, after, or treated as the same as, the other. Call the `create` function to create an Ordered_array container, and save the returned pointer to it. Insert items into the container with the `insert` function - they will be kept in order in the internal array, which is automatically expanded as more items are inserted. Find whether an item is in the container with the `find` function; if it returns a non-negative result, this is also the index of the item in the container, which can be used to retrieve the item with the `get` function. An item can be removed with the `remove` function. The array is deallocated and reallocated to its initial allocation of 3 cells with the `clear` function. The array and the Ordered_array struct both are deallocated by the `destroy` function. The project web page has some demonstration code that illustrates how Ordered_array is supposed to be used.

If you try to insert an item into the container when there is already a matching item in it, the results are *undefined*. Your Ordered_array code is not required to do anything in particular about it. The *client* code that uses the Ordered_array container is expected to prevent this from happening. For example, write your client code so that it first checks for an item being present, and only adds an item if it is not. Notice that this is a natural way to use the container, and so this approach is a reasonable one.

The trick with using this container is the casts that are required to convert between the actual data and the generic `void *` stored in the array. To put items in, you have to cast them to `void *`; when you get items out, you get a `void *` and have to cast it to the actual type. Normally the items you are storing in the Ordered_array are pointers of different types, such as `char *` or `Action_item *`. Since pointers are the same size regardless of what they point to, you can store a pointer to any type in the space occupied by a void pointer. Since a pointer is normally something like at least 4 bytes in size, there are some data types that you can store directly in the array; for example, with the proper casts you can store an `int` in the `void *`-sized cell of the array. See the project web page for example code showing how Ordered_array can be used.

`Action_item.h, .c`

An Action_item specifies a skip or replace action created when the user wants to skip or replace a word in the remainder of a spell-checked document. It is a simple structure type that holds an enumerated value and two pointers to C-strings, `match_str` and `replace_str`. The enumerated type specifies whether the action is skip or replace. The `match_str` pointer points to a C-string for the word that is to be skipped or replaced, while the `replace_str` pointer points to a C-string for the replacement word. For neatness, this pointer should be set to `NULL` in a skip action. The Ordered_array for action items should keep them in order by the `match_str`; thus the comparison function orders two Action_items by comparing their `match_strs`.

The purpose of specifying the Action_item is to ensure that you get some practice using the Ordered_array to store a pointer to a struct with a not-completely-trivial comparison function. Therefore, you cannot alter the declaration of the `struct Action_item` in `Action_item.h`: you must use it as-is, and cannot modify, remove, or add to, the function prototypes specified in this file. Your implementation must be done in the `Action_item.c` file.

`p1_globals.h, .c`

As an exercise in appropriate use of global variables and the linkage issues that arise with them, your project must use the variables specified in the skeleton header file to keep track of the memory allocations. These variables are incremented when the corresponding items are allocated using `malloc()`, and decremented with they are deallocated using `free()`. There may not be any other variable, function, or macro declarations in this header file, and no function definitions are allowed in this module.

`p1.c`

This source file contains function `main` and all of the other declarations and functions for the project not in some other module. The behavior of main and its subfunctions is specified in detail elsewhere in this document. Designing a good set of functions for this module is the major design exercise in this project; refer to course materials and discussion for guidance. For example, your main function should be very small, doing nothing more than setting things up and calling other functions to do the actual work of each command. Duplicated code should be refactored into a well-chosen single function.

Declare all the functions with function prototypes near the beginning of the file, and tell your compiler to require function prototypes. This will enable the compiler to catch a variety of errors that would be otherwise very hard to detect. It also enables you to put the functions in a comprehensible and easy-to-read order; put main first, followed by the functions it calls, followed by the functions they call, etc. It should be possible to read your program like a story, from the top down, and each function should be easy to find in the file. Don't put functions at the bottom of the calling hierarchy before the functions that call them.

Comments about each function should appear with the function's definition - comments on the declarations (prototypes) are not helpful to the reader, and so are not required.

`Utilities.h, .c`

To keep the autograder happy, your project must include these two files even if they are empty and do nothing. These files are for handy utility functions that good programmers develop and keep around to simplify their programming. Learning how to recognize and setting aside these functions is a valuable programming skill, so your choice of these functions is important. Here is what to include or not to include as Utility functions:

- A good utility function is one that is used in multiple `.c` files in this project, or might be used in other, possibly very different projects in the future.
- Functions that are used only from one `.c` file, or are totally project-specific, do not belong in the Utility module. Utility means "generally useful" not "miscellaneous."
  - For example, a function that determines whether a string contains a word as defined in this project is hardly likely to be useful in a different project, and it should only be called from one module in this project, meaning that it should be in that module, not in the Utility module.
- The functions in Utility must not depend on functions defined in one of the other project modules - they must be "standalone", at most relying only on the Standard C library or other functions in Utility.
  - An exception: for purposes of this project, the utility functions can manipulate the global variables declared in `p1_globals.h`.

Suggestions for good Utility functions (there are additional good choices):

- A function that it does a `malloc` call and checks for a successful allocation, printing a message and terminating the program if it fails. This makes it easy to follow the good programming practice of always checking the result of a memory allocation attempt.
- A function that given a C-string, copies it into a properly-sized newly allocated piece of memory and returns the pointer to it. Once debugged, this function makes it easy to perform this common and error-prone operation.

Programming Requirements

Specific concepts and techniques

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; at this time, you may not use any of the C99 extensions.
   - You can use any functions in the C Standard Library for dealing with characters and C-strings, including any of those in `ctype.h`, or `string.h`. See a Standard Library reference, or the online handouts. Any non-Standard functions must be written by yourself. For example, `strcmp` is in the Standard Library; it is case-sensitive. There is no Standard Library case-insensitive function for comparing C-strings; if you want one, write it yourself, or figure out how to get the results using `strcmp`, or write the desired function yourself.
   - ***Hints***: The project can be coded easily using only basic Standard functions like `scanf`, `printf`, `tolower`, `strcmp`, `strcpy`, `strlen`, and `strncpy`. You will probably just make it harder on yourself if you try to use some of the more exotic library functions for string processing. Check out `fgets` for reading the input document a line at a time.
2. You must close each file shortly after the last input or output operation on it.
3. You should not attempt to read the entire input document into memory. This limits the size of the document and it is definitely unnecessary and clunky given that the job can be done easily by reading and processing one line at a time.
4. You should not dynamically allocate memory for simple character array buffers to hold lines and C-strings as they are being worked on. This is inefficient and unnecessary, and a sign of poor quality code, and is a common source of memory-leak bugs. Remember also that dynamically allocating memory is very slow, while allocating an array on the stack is practically instantaneous. Simply declare some local character arrays using the specified maximum sizes. It is possible to build this program with only one buffer for an input line and a few buffers for the individual words or filenames in existence at a time.
5. On the other hand, since they must have a long and controlled lifetime, the strings stored in the dictionary and action list items must reside in dynamically allocated memory, and good practice calls for this memory to be allocated with the *minimum* size necessary to properly store the string.
6. When your program terminates, it must have already deallocated all dynamically allocated memory itself - this is good practice. However, if the program terminates abnormally due to a memory allocation failure, this should not be attempted. Also, there should not be any memory leaks. You can help prevent them by proper code organization. You can detect them with various

unfortunately platform-specific tools. The memory allocation command, in conjunction with the empty command, can also help detect leaks, but only if you have designed your code properly.

7. Your code must conform to good programming practices with regard to allocating memory - for example, you must check the result of all memory allocations to be sure they succeeded. For simplicity, your program should simply output a message of your choice and terminate if memory allocation fails.
8. The program should be "bullet proof" - it should run successfully regardless of the contents or length of the input document (limited only by disk space and available memory for the dictionary and action list), and it must not crash regardless of any gibberish commands the end user might type in.
9. Take advantage of the fact that all of the commands are single characters to practice using a handy C/C++ technique: use a **`switch`** statement to choose what function to call or code to execute depending on the command character. This is a much more expressive way to write this control structure than a block of if/else statements. Most programming languages have a "case" statement like switch, and for good reason. You should use it where appropriate.
10. Your code should use the **`assert`** facility to detect programming errors. It should not be used to detect user errors - these the program code will detect, print a message, and then ask the user to take action to correct the error. Likewise, it should not be used to detect run-time errors such as failed memory allocations or file opening failures. Assertions should be used only to state what should be the case if the program code itself is correct - it is a check for programming errors, not user errors or run-time environment errors. Putting such `assert` statements in wherever reasonable can make debugging your program much easier by preventing it from running amok, especially when pointers are involved. Check your lecture notes or ask if it isn't clear what you should be using `assert` to check.
    - For example, your code should never attempt to access an item from Ordered_array using an invalid index - it simply shouldn't happen, and there should be no way that a user can cause it to happen. But if your program is buggy, which it will be at least at first, it might happen anyway, and strange things will occur. Why risk getting confused by weird and crazy behavior? Put an `assert` statement in the `get_Ordered_array_item` function that detects the programming error and halts the program immediately with some useful debugging information.
11. Read the web page “Why and how you should comment your code” before you start the project, and follow its advice.

Global variable usage

Good procedure hierarchy design means that you should have functions that call other functions, down to very simple ones that do a specific job. Many of these functions have to be able to access the dictionary and the action list, so you will be passing these parameters down the function call hierarchy. It is tempting to make these global variables instead of parameters to save typing time and to shorten the function parameter lists.

However, global variables are normally avoided as much as possible. Beginning programmers often use a lot of global variables, believing that they make the program simpler. However, experienced programmers, who worship simplicity, avoid using global variables like the plague! Why? As soon as a program gets large and complicated, it becomes extremely difficult to keep track of the flow of information through global variables. A typical bug-hunt ends up in confusion, with questions like "who put *that* value there?" Also, it can become difficult to keep track of which variables are global and which are not. A small lapse in attention could result in defining a local variable that hides the global one, producing even more confusion. Thus, once the project gets complicated, *the supposed simplicity of global variables can result in buggy complexity*. In contrast, passing information through function parameters is very disciplined, very simple, and easy to track, especially when the use of function prototypes allows the compiler to type-check your function calls.

In certain uncommon situations, global variables can be very helpful. In fact, the standard I/O streams (e.g., `stdin`, and `cout`) are nothing more than global variables! "Good" global variables follow these rules:

- The variable is truly unique in the entire program - it corresponds to exactly one object that cannot be meaningfully duplicated.
- The variable is distinctively named, easy to keep track of and avoid hiding (e.g. its name starts with something like "g_").
- The variable is write-once, read-many, or the value of the variable is changed in only one distinctive part of the program, and the rest of the program simply reads this value.
- The variable does not contain data that is central to the specific logic of the program, so it is unlikely to be a source of bugs or other problems.
- There is a significant benefit of the global variable that would be difficult, complex, or error-prone, to achieve otherwise.

Our recommendation is not to use global variables except in similar situations; in this course you can use them only as we specifically and explicitly allow them in the project specifications. Any unauthorized use of global variables will be treated as a serious code quality failure and severely penalized.

Th global variables specified in this project will allow you to learn something about linkage, an important topic, and also illustrate a use of global variables in an appropriate way, providing of course that you organize your code correctly. Here's how to tell if you have: The main module (`p1.c`) should only read the variable values; other components (in other `.c` files) are the only ones to modify the values. This will result if your code has a good division of responsibilities between modules, so that the main module is not "micromanaging" memory allocations that are the proper business of other modules.

There is one additional place in this project where a global variable is permitted in order to simplify making the program crash-proof: you can use a global variable that is local to `p1.c` to hold a `scanf` format string constructed at run time to be used for reading input in a way that ensures that the input buffer cannot be overflowed.

Project Evaluation

Watch for an announcement that the autograder is open. Instructions and suggestions will be on the course and project web page. We plan to test your Ordered_array implementation component separately. That is, we should be able to use your `Ordered_array.c` file with our testing files, and it should work correctly. (We will also use your `Utilities.h, .c` files in this test). Likewise, we should be able to substitute our `Ordered_array.c` file for yours, and your project should compile and run successfully. This will work if your `Ordered_array.h` file is the same as the supplied one, and your program uses this component only according the specifications, and your implementation of `Ordered_array.c` conforms exactly to the specifications. Be sure to test carefully all of the Ordered_array functions, especially those not used in the main project!

Before starting on the project, reread the Syllabus on how your project will be evaluated. Note that you must get the threshold of 80% of the autograder points for your project to qualify for Quality Evaluation, and the Quality Evaluation will count for 70% of the project credit. Then go to course home page and read the documents *How to do Well on the Projects* and *C Coding Standards*. Also take a look at the Sample Code Quality Checklist on the project main page. Review your lecture notes about the reasons and concepts for high-quality code.

Thus the quality of your code is the most important aspect of it; having code that produces the correct output simply gets you into the game. Review the above sources and go over your code before submitting it.

Everything that makes your code have good quality also makes it easier to write and debug, so if you have to go back and do major surgery on your code to make it acceptable quality, you have missed the point and wasted a lot of time. Try to write it well from the beginning and the work will go faster and be more fun.

How to Build this Project Easily and Well

Design before coding

Ordered_array and Action_item are pretty well specified, and the functions involved are individually fairly simple. The remainder of the program is responsible for finding the words in the input documents, interacting with the user, and generating the dictionary and corrected output document. This involves standard procedural programming techniques using a function hierarchy. This project requires you to design the function hierarchy, given only some specifications about its behavior. You must design this part of the program before coding it, at least if you want to keep your sanity and finish in a reasonable amount of time. Some tricky coding is involved that will benefit considerably by working it out beforehand, and by breaking the project into small function-sized pieces, you can code and test portions of the code much more easily than trying to do it all at once. Only an inexperienced newbie programmer will attempt to write the code for this entire project before starting testing and debugging - be an instant non-newbie and don't do it!

But before you can start coding functions, you need to know what the functions will be. Start at the top of the program concept - what does the program as a whole do? What are the natural sub-parts of this process? Can each subpart be represented with a function? What does each function need to do? Are there functions that do almost the same thing, and so can be combined into a single function? Are there functions that can be used in more than one place? Work your way from the top down to the bottom, continuing until the functions at the bottom are doing something so simple that you can easily imagine writing their code. Remember that standard coding practice is to use functions to show the natural structure of the code even if the functions are only called from one place! Only beginning programmers write programs consisting of only a few large do-all functions!

You will find it useful to write out the algorithms for each function in "pseudo-code" or brief English phrases before you start coding - in fact, try writing comments that describe what each function does, and then later fill in the code. Try to figure out whether the functions will actually work as planned with the inputs and outputs you plan for the function. The goal is to work out how the program is going to be organized, and exactly what it will do, and how the code will work, before you write a single line of code. An hour spent in this way can easily save many hours of frustrating work of coding, debugging, throwing it away, recoding, redebugging, etc. in the wee hours of the morning.

But be prepared to discover that your design is flawed - it happens all the time! If you are having trouble working out the code, stop and reconsider your design - sometimes you don't appreciate the real issues until you have tried to write the code. Struggling with a bad design is a bad idea - you can often do a redesign and complete the project much faster, more easily, and get a higher-quality result. The previous design work is not wasted - even if you have to revise or correct a design along the way, you will certainly understand the problem better, and the reworking will be quite fast. In general, even if you have to modify the design, you will complete the coding a lot faster than if you try to work from no design at all.

Hints for coding, testing, and debugging

First, study the samples posted on the project web page, and refer to them frequently if there is some aspect of the specifications that appears confusing. They illustrate how the program is supposed to behave, and exactly what the output messages are supposed to be. Your program must be able to reproduce the samples exactly, so if it doesn't, you definitely have a bug. However, the samples are not a complete test of the program, so it is possible to still match the samples and fail the other autograder tests. You will need to devise your own tests to identify the problems. Testing each piece of functionality as you build it will be much easier than trying to test the entire program at once.

Building this project naturally divides into two parts. The Ordered_array component can, and should, be built and tested as a self-contained unit. The rest of the project consists of parsing the input documents and interacting with the containers. Some of the bottom-level functions in this second part can be built and tested by themselves. Nothing is more wasteful and confusing than trying to debug a low-level function in the middle of a complex project. Small simple functions are always easier to write and debug than large complex functions, so make your life better by writing and debugging small units of code as much as possible. This means taking advantage of those places where you can write and debug independent units of code. As much as possible, write and test code that depends on other units only after you have thoroughly debugged those other units.

Test the Ordered_array component with a small testing driver program. Do not attempt to get the whole program to work until you have tested this component very thoroughly with a testing driver! You'll be glad you did!

To start on the p1.c part of the project, go back and reread the section on the first page called "Design before coding." Many people find that it works best to design the code from the top down to the bottom of a function hierarchy, and then code and test from the bottom of the design up to the top. Each piece is thus built out of lower-level pieces you have already debugged.

For example, there are two tricky parts: One is finding each word in an input line and extracting it for checking or adding to the dictionary. You can design, code and test a function for doing this separately, using a hard-coded string as a test input line. Be sure to test all the possibilities of where each word is in the line relative to the ends of the line and other words! The second is replacing a word in the input line - the new word might be longer or shorter than the original word! This can also be debugged separately using a dummy line. Designing, coding, and testing these functions by themselves is orders of magnitude easier than trying to do it in the context of the entire program!

Keep in mind that the program has to show the user the entire line when it encounters a questionable word, and has to copy all whitespace characters that it finds to the output file, not just ignore them. *Hint*: Read the input document an entire line at a time using **`fgets`**, and then operate on the line as needed, and then write the line to the output file.

While you are debugging and testing your whole program, create and use a very small dictionary creation file and document file (only a few words each). This will allow you to use the debugger to step through the program in a reasonable amount of time and with a minimum of confusion! You should be able to use your normal programming editor to create, view, and modify these files - even in the IDEs like Xcode or MSVC, these editors work in terms of ASCII text files.

Don't put off the error checking, especially of file openings - it is easy to misspell a file name, or mistype a command! These checks are your friends! They may save you hours of confusion and frustration!