UCLA CS31, Summer 2013
Project 4: All of the Words

Introduction

This project will provide practice in using pointers, dynamic allocation, structs and multidimensional arrays. You will implement several functions that will create, manipulate, and release WordLists. A WordList is a C++ structure defined as:

```
struct WordList {
       unsigned int count;     // Number of words currently in list
       unsigned int max_words; // The max number words allocated to the list.
       char **list;             // The list storing the words
};
```

The *list*, as well as the WordList, will be dynamically allocated through the functions you will implement. The *list* will be essentially an array of cstrings. Recall that cstrings are character arrays that are null terminated ('\0'). The member variable *list* can also be viewed as a 2-dimensional array of characters, or a matrix of characters. For example, let's suppose we have a WordList named *w_list* that has been allocated enough space for 5 words but only stores the following three: **harry ron hermoine**. We can imagine *w_list* looking like:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	**h**	**a**	**r**	**r**	**y**	**\0**														
1	**r**	**o**	**n**	**\0**																
2	**h**	**e**	**r**	**m**	**o**	**i**	**n**	**e**	**\0**											
3																				
4																				

With *max_words* = 5, and *count* = 3. Observe that the rows in this matrix are the words and the columns are the characters of the words. You may assume that words will be no more than 19 characters in length (+1 for the null character) and that words stored in the list contain no white space. The following call,

```
cout << w_list->list[0][2];
```

will output the third character in the first word, or the first 'r' in harry. Also,

```
cout<<w_list->list[1];
```

will output the entire second word or "ron". Note that empty cstrings (cstrings with just the null character) will not be considered words for this project. Following this, there should be no empty slots between words in the list. For example, the following is an invalid *list* and would be considered incorrect.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	**h**	**a**	**r**	**r**	**y**	**\0**														
1																				
2	**h**	**e**	**r**	**m**	**o**	**i**	**n**	**e**	**\0**											
3																				
4																				

In order to be valid, hermoine would have to be moved to the second row and the *count* would be 2 words (*max_words* 5). You may assume that *max_words* of the WordList will always be consistent with the amount of memory allocated to store that many words; you don't have to actually test if this is the case. You may also assume that the *count* will reflect the intended number of stored words. Note that a quick way to "clear" the WordList is simply to set count to 0, that way when words are entered into the list they overwrite the contents starting from the beginning of the list.

You may not use or include the string library, or any other libraries not already included. However, you are provided with and may use the functions in the cstring library, e.g. strlen, strtok, strcmp, etc. You may create your own helpers functions and are encouraged to do so. We will only test the functions described in this specification.

A bit about NULL

`NULL` in most programming languages refers to the value zero. We've seen the null character, which has the has the value 0, when we discussed cstrings. With regards to pointers, a `NULL` pointer is one that points to nothing. We can explicitly do this by calling

```
int *ptr;
ptr = NULL; // or ptr = 0;
```

Where `NULL` is defined by the language. Recall that when we declare a variable, we have no guarantees about what may be at the memory location, unless we explicitly set its value. With pointers, it can be more dangerous since we have no guarantees about what address is stored in that pointer, and if we accidentally access that address it will most likely crash our program. So, it is often necessary to either initialize or set our pointers to point to nothing until we have use for them.

What you get

You will be provided with three files in addition to this specification, all three files need to be present in your project/working directory in order to compile. Of the three, you will only submit **project_4.cpp**.

project_4.cpp: This file will contain your function implementations, along with any helper functions prototype/definitions, global constants, anything your implementation needs to compile and run. You cannot change the function signatures for the functions described in this specification. You will submit this file. You should rename this file to the standard we have been using this session, e.g. **project_4_<sid>.cpp**, where <**sid**> is your UCLA student ID.

main.cpp: This file will have your main function. You will be testing your functions using main, so you may make and changes you want. We will be using our own main to test. Do not submit this file.

project_4.h: This file contains the function prototypes for this project and WordList definition. Do not make any changes to this file. Do not submit this file.

When you setup your environment with these files, make sure everything compiles without error before making any changes.

Meet the functions

Each function will have a difficulty rating from 1 to 4, 1 being easiest. The number of points each function is worth is twice that of its rating. Those points will be divided among the different test cases we will use to validate your implementations. Implementing every function correctly will net a total of 60 points. Note that the project will be scored up 52 points. This means it is possible to ignore one rating 4 (or 2 rating 2's, or a rating 1 and a rating 3) function and still obtain full marks for the project; assuming all other functions are correct. Any points earned past 52 will be extra credit.

An easy function may take anywhere from 10 to 30 minutes to fully implement and a hard function may take anywhere from 1 to 3 hours. You will want to implement the *newList* and *printList* functions first since they will be your primary means to create and view WordLists. Below is a brief table of functions to be implemented, detailed specifications follow.

Function	Rating	Function	Rating
newList	2	findWord	1
newList	3	findWordSequence	2
deleteList	1	removeWord	2
printList	1	removeDuplicates	3
appendList	4	sortList	3
appendList	4	pigLatin	4

Function 1: WordList * newList(const unsigned int max_words);

Dificulty: 2

Specification: Will return a `NULL` pointer if *max_words* is less than one. Otherwise, will allocate a new WordList with zero word *count*, and allocates enough memory to store *max_words*, sets the member variable *max_words* appropriatley.

Usage: `WordList *w_list = newList(5); // new WordList can store 5 words`

Function 2: `WordList * newList(const char words[]);`

Dificulty: 3

Specification: The parameter `const char words []` is will be a cstring with zero or more words. Each word will be separated by a space character, for example:

```
char words[] = "groovy egg why hat"; // 4 words
char words[] = "harry ron hermione"; // 3 words
char words[] = ""; // zero words
```

Are possible *words[]*. If there are no words *newList* returns a `NULL` pointer. Otherwise, it will create a new WordList and allocate enough space to fit the number of words being passed in and store those words in the WordList in the order they appear in *words[]*. The member variables *count* and *max_words* should be set appropriately.

Usage: `WordList *w_list = newList("harry ron hermione");` will create the WordList similar to that at the beginning of this specification, except with *max_words = 3*.

Function 3: `int deleteList(WordList *w_list);`

Dificulty: 1

Specification: Takes a dynamically allocated WordList and releases the *list* and the WordList itself, sets the WordList to null, returns 0. If the WordList is already `NULL` return -1.

Usage:
```
WordList *w_list = newList(5);
int retval = deleteList(w_list);
```

Function 4: `int printList(WordList *w_list);`

Dificulty: 1

Specification: If the WordList is `NULL` or if the *list*'s word *count* is zero, *printList* does nothing and returns -1. Otherwise, it prints the *list* in WordList to the console with a space between each word and a newline after the last; there is no space after the last word.

Usage:
```
WordList *w_list = newList("tom jerry nibbles");
int retval = printList(w_list); // print “tom jerry nibbles\n” to console
```

Function 5: `int appendList(WordList *w_list, const char words[]);`

Dificulty: 4

Specification: If there are no words in *`words[]`* or if the *`w_list`* is `NULL` *`appendList`* does nothing and returns -1. Otherwise, it will append the words into the *list*, without changing the current contents. If *`w_list`* does not have sufficient space to store the additional *`words`*, *`appendList`* will resize *`list`* with enough space to contain the additional words, returns 0 on success.

Usage:
```
WordList *w_list = newList("snape”); //count 1, max_words 1
int retval = appendList(w_list, "kills joffrey"); // count 3, max_words 3
printList(w_list); // print “snape kills joffrey\n” to console
```

Function 6: `int appendList(WordList *dst_list, const WordList *src_list);`

Dificulty: 4

Specification: If either WordLists are `NULL`, *appendList* does nothing and returns -1. Otherwise, it will append the contents of *scr_list* to *dst_list*. If *dst_list* does not have enough capacity to store the additional words, then *appendList* will resize *list* of *dst_list* with enough space to contain words from both lists, without changing the contents originally in *dst_list*. Returns 0 on success.

Usage:
```
WordList *w_list1 = newList("susannah mia"); // count 2, max_words 2
WordList *w_list2 = newList("odetta holmes dean"); // count 3, max_words 3
int retval = appendList(w_list1, w_list2); // append w_list2 to w_list1
printList(w_list); // print “susannah mia odetta holmes dean\n” to console
cout<<w_list2->count<<" "<<w_list->max_words; //prints 5 5
```

Function 7: `char* findWord(const WordList *w_list, const char word[]);`

Dificulty: 1

Specification: Finds the first occurrence of the *word[]* in *w_list* returns a pointer to that word in the list. Otherwise returns `NULL`. If *w_list* is `NULL` return `NULL`.

Usage:
```
WordList *w_list = newList("Where is Waldo");
char * here = findWord(w_list,"Waldo");
if(here == w_list->list[2]) // what is this tomfoolery?
        cout<<"Found "<<here<<endl;
```

Function 8:
```
int findWordSequence(const WordList *w_list, const char word[], char *&s_ptr, char *&e_ptr);
```

Dificulty: 2

Specification: Find the earliest occurrence in *w_list* of one or more consecutive words in the list that are equal to word; set *s_ptr* to point to the first word in the sequence, set *e_ptr* to point to the last occurrence in the sequence. Return the number of words in the sequence. If the *word[]* is not in the list then return 0 and do not change *s_ptr* or *e*_ptr. If *w_list* is `NULL` return -1. Note that prior to passing in the character pointers into this function you may have to initialize them to `NULL` otherwise you may encounter a runtime error.

Usage:
```
WordList *list = newList("Dera Walda Walda Shirei Serra Walda Walda Walda");
char *start= NULL, *end= NULL; // Must init to null prior to passing
int count = findWordSequence(list,"Walda",start, end);
if((count == 2) && (start == list->list[1]) && (end == &list->list[2][0]))
        cout<<"That's a whole lotta Walda"<<endl;
```

Function 9: `int removeWord(WordList *w_list, const char word[]);`

Dificulty: 2

Specification: If *w_list* is `NULL`, returns -1. Otherwise, searches for every occurrence of *word[]*, and removes that word of the *list*, returns the number of words removed. *removeWord* should preserve the order of the remaining words.

Usage:
```
WordList *serenity;
serenity = newList("Mal Inara Wash Jayne Kaylee Simon River Book Wash");
char alliance[] = "Book";
int retval = removeWord(serenity, alliance); // :(
printList(serenity);
// prints "Mal Inara Wash Jayne Kaylee Simon River Wash\n"
char reavers[] = "Wash";
retval = removeWord(serenity, reavers); // :(
printList(serenity); // prints "Mal Inara Jayne Kaylee Simon River\n"
```

Function 10: `int removeDuplicates(WordList *w_list);`

Dificulty: 3

Specification: If *w_list* is `NULL` return -1. Searches through *w_list* and removes all duplications of a word, keeping the first occurrence of the word. Returns the number of words removed. *removeDuplicates* should preserve the order of the remaining words.

Usage:
```
WordList *meville;
meville = newList("Perdido City Scar Council Scar Scar City Embassytown");
int retval = removeDuplicates(meville);
printList(meville); // prints "Perdido City Scar Council Embassytown\n"
```

Function 11: `int sortList(WordList *w_list);`

Dificulty: 3

Specification: If *w_list* is `NULL` return -1. If there is only one word in the *list* return 1. Otherwise, *sortList* sorts the words in *w_list* in ascending order. Returns 0 on success.

Usage:
```
WordList *neverwhere = newList("Richard Door Carabas Islington Abbot");
int retval = sortList(neverwhere);
printList(neverwhere); // prints "Abbot Carabas Door Islington Richard\n"
```

Function 12: `int pigLatin(WordList *w_list);`

Dificulty: 4

Specification: If *w_list* is `NULL`, *pigLatin* does nothing and returns -1. Otherwise, *pigLatin* will translate each word in *w_list* to pig latin. There are three rules for pig latin translation:

1. If the word begins with a consonant then move all the characters up to the first vowel to the end of the word, then append "ay"

   For example: groovey => oovey gr ay => ooveygray

2. If the word beings with a vowel, then append "way" to the word.

   For example: egg => egg way => eggway

3. If the word has no vowels then we make no changes to the word.

   For example: why => why

Note that we will not consider 'y' to be a vowel for this function. *pigLatin* returns the number of words which were **not** translated

Usage:
```
WordList *oink = newList("groovey yes why donut happy duck glove");
int retval = pigLatin(oink);
printList(oink); //prints "ooveygray esyay why onutday uckday oveglay\n"
```

Submission

You will submit your project_4.cpp, renamed to project_4_<sid>.cpp on Courseweb.

Tips

- Pointers offer a large amount of flexibility to our programming. However, with that flexibility comes a fair amount of complexity, which seems to depend on the context of how we're using our pointers. The key to success in using pointers will depend on your ability understand the context, i.e. how is this pointer being used? What is its type?
- Of particular interest, the majority of concepts addressed in this project directly relate to the some topics you will encounter in your final exam. You can use this project as a study tool to help you prepare.
- It is good practice to develop your test cases prior to developing your code. Identify, the different inputs to your functions that will exercise a single aspect of that function and have a known expected output. That way when you implement that particular aspect you can see if it behaves as expected, if not you make the necessary adjustments.
- Work on a single function at a time, after implement *newList* and *printList* work on the easier functions then move onto the harder functions. You should notice that many of the functions share similarities in their implementations with other functions. Take advantage of this to save yourself time.
- When submitting, Courseweb will keep your most recent submission. Every time you finish a function or two, and your code compiles; submit it. This way you're guaranteed to have something we can grade. But, be sure your code compiles first.
- While we won't be explicitly checking for memory leaks, it is good programming practice to ensure you are releasing any dynamically allocated memory when you are no longer using it (and setting the pointers to null). Generally, you should match a call to new with a call to delete.
- It is usually a good idea to compile your code using at least two different compilers. This guarantees that your code will work regardless of what machine you compile on. For this project we will be testing in a Linux environment, which uses g++ to compile C++ code. So, it behooves you to make sure your code compiles in that environment.

  For small projects using g++ is actually easier in terms of compiling, you lose the nice features that accompany a full blown IDE, but you don't have to deal with setting up projects. The Seasnet computers give you access to Linux terminals, and Macs are based Linux, so it is possible to install g++ on Mac OS.

  Compiling with g++ is straight forward, you open up a command terminal and navigate to the directory your files are stored. This is done with the change directory command, *cd* <*directory*>. You can use *ls* or *dir* to list what is in the current directory. Once in the directory with your files you just make the call to g++ with your source files, for example:

  g++ main.cpp project_4.cpp

  that's it, no need to include project_4.h, it is already included in the source files (remember #include"project_4.h"?). If there are no errors then you will only see the prompt. If there are any errors then g++ will give you the file and line number the error occurs. To execute the program you can enter ./a.out This will execute the binary created from compiling.