Project 2: All of the Words

Read everything carefully!

Introduction

This project will provide practice in using pointers, dynamic allocation, classes and multidimensional arrays. You will implement several functions that will create, manipulate, and release WordLists. A WordList is a C++ class that has the following member variables:

```
class WordList {
      //...
      unsigned int m_count;      // Number of words currently in list
      unsigned int m_max;  // The total size of the list.
      char**       m_list;       // The list storing the words
};
```

The *m_list* will be dynamically allocated through the functions you will implement. The *m_list* will be essentially an array of cstrings. Recall that cstrings are character arrays that are null terminated ('\0'), see the end of the specification for more details on cstrings. The member variable *m_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 *wordlist* that has been allocated enough space for 5 words but only stores the following three: **harry ron hermione**. Suppose we created *wordlist* with the following:

```
WordList *wordlist = new WordList(5);
wordlist->add("harry");
wordlist->add("ron");
wordlist->add("hermione");
```

We can imagine *wordlist* 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**	**i**	**o**	**n**	**e**	**\0**											
3																				
4																				

With *m_max* = 5, and *m_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. For the sake of simplicity let's make our member variables public (something you can always do when you're developing so it is easier to test, just make sure you make them private again before you submit), we can then do something like this:

```
cout << wordlist->m_list[0][2];
```

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

```
cout << wordlist->m_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 *m_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**	**i**	**o**	**n**	**e**	**\0**											
3																				
4																				

In order to be valid, hermoine would have to be moved to the second row and the *m_count* would be 2 words (*m_max* 5).

IMPORTANT: For this and every class you create, the member variables must be consistent with the intended state of the class. That is to say for this project, that *m_max* of the WordList must always be consistent with the amount of memory allocated to store that many words; if you allocated enough space to store ten words, then *m_max* must be 10. If you change the size of the WordList then you must adjust *m_max* accordingly. Further, the *m_count* must reflect the intended number of stored words. Any inconsistencies in this regard will result in buggy code. 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. This means you cannot use the string data type. However, you are provided with and may use the functions in the cstring library, e.g. strlen, strcpy, strcmp, etc; you will want to spend time getting comfortable with their usage, discussed in further detail at the end of this specification.

A bit about nullptr

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

```
int *ptr;
ptr = nullptr; // ptr points to nothing
```

Where nullptr 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 the behavior is undefined, but we would hope that our program would crash. 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 four files in addition to this specification, all four files need to be present in your project/working directory in order to compile. The first thing you should (always) do is to make sure the provided code compiles. Of the four, you will only submit **wordlist.cpp** and **studentinfo.h.**

wordlist.cpp: This file will contain your function implementations. You cannot change the function signatures for the functions described in this specification or include any libraries not already included. Currently, there is little to no body to these functions, it is your job to fill them out. Note that the return values are there so the project can compile, you will eventually have these functions return what is specified. You will submit this file.

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

wordlist.h: This file contains the class definition for WordList. Do not make any changes to this file. Do not submit this file.

studentinfo.h: This file just has two functions that return strings, one for a name and an id. You will modify these functions to return your full name and smc id. You will submit this file.

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

Meet the functions

In addition to the specific instructions for each function, none of your code should result in memory leaks or unintended aliasing of dynamically allocated variables. We've seen examples in lecture on how these situations can cause severe bugs in any program.

Function 1: `WordList(const int max_words);`

Specification: Constructs an empty WordList that is allocated enough space to store *max_words* many words. If max_words is less than 1, set *m_list* to nullptr and the other member variables should be made consistent with this state. Otherwise, will allocate a new *m_list* with zero word m_*count*, and allocates enough memory to store m_max, sets the member variable *max_words* appropriately.

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

Function 2: `WordList(const WordList &other);`

Specification: Copy constructor for WordList, constructs a new WordList from an already existing WordList. The newly constructed WordList should have the same attributes as the existing one, but it should own its own dynamically allocated memory.

Usage:
```
WordList existinglist(5);   // WordList can store 5 words
//... Some WordList operations
if (...) {
       WordList newList(existinglist); // Copy Constructor
} // Should not results in undefined behavior
```

Function 3: `~WordList();`

Specification: Destructor for WordList. Releases any dynamically allocated memory.

Usage: Called when a WordList leaves scope, like in Function 2 Usage, or when delete is called on a dynamically allocated WordList: `delete wordlist; // From Function 1 Spec`

Function 4: `int print() const;`

Specification: Prints all the words in WordList in a single line with spaces between each word, then followed by a newline after the last word. Returns the number of words printed. If m_list is nullptr there is nothing to print, return -1.

Usage:
```
WordList *wordlist = new WordList(5);
//... add harry ron hermione into list
int retval = print(wordlist); // print “harry ron hermione\n” to console
```

Function 5: `char* get(const int index) const;`

Specification: Returns the word at the index position in the WordList. If the index is out of bounds return nullptr.

Usage:
```
//Assuming wordlist is the example in the beginning of the spec
char* word = wordlist->get(1);
if (word != nullptr)
	cout << word << endl; //ron
```

Function 6: `int count() const;`

Specification: Returns the number of words currently stored in the WordList.

Usage:
```
//Assuming wordlist is setup to the example in the beginning of the spec
for (int i = 0; i < wordlist->count(); i++) {
	cout << wordlist->get(i) << " ";
} //pritns “harry ron Hermione”
```

Function 7: `int add(const char word[]);`

Specification: Adds the word into WordList (words have no spaces). If WordList does not have enough space to add word, add will resize with just enough space to allow for the addition of word. If add needed to resize then return 1, otherwise if there already enough space to add word without resizing, return 0. If word is empty do nothing return -1. If m_list was nullptr, everything above still holds true except return -2.

Usage:
```
WordList *wordlist = new WordList(5);     // Dynamically allocate
                                          // WordList can store 5 words
wordlist->add("harry");
wordlist->add("ron");
wordlist->add("hermione"); //wordlist is identical to
                           //example at beginning of spec
```

Function 8: `int remove(const char word[]);`

Specification: If m_list is nullptr, returns -1. Otherwise, searches for every occurrence of word[], and removes that word of the list, returns the number of words removed. Make sure the resulting `WordList` follows the rules for a valid `WordList` outlined in the spec.

Usage:
```
WordList serenity(10);
serenity.add("Mal");
serenity.add("Inara");
serenity.add("Wash");
serenity.add("Jayne");
serenity.add("Kaylee");
serenity.add("Simon");
serenity.add("River");
serenity.add("Book");
serenity.add("Wash");
serenity.print();
// prints "Mal Inara Wash Jayne Kaylee Simon River Wash\n"

serenity.remove("Wash"); // :(
serenity.print();
// prints "Mal Inara Jayne Kaylee Simon River\n"
```

Function 9: `int append(const WordList* src_list);`

Specification: Appends the contents of *src_list to the `WordList`. If `WordList` does not have enough space appendList should dynamically allocate enough space to append the contents of *src_list to WordList, returns number of words appended. If *src_list is nullptr or empty appendList does nothing and returns -1. If this `WordList`::m_list is nullptr everything above still holds but returns -2.

Usage:
```
WordList wordlist1(0);
wordlist1.add("susannah");
wordlist1.add("mia"); // count 2, max_words 2

WordList wordlist2(4);
wordlist2.add("odetta");
wordlist2.add("holmes");
wordlist2.add("dean"); // count 3, max_words 4

int retval = wordlist1.append(&wordlist2);
wordlist1.print(); // print “susannah mia odetta holmes dean\n”
// Assuming member variables are public
cout << retval<< " "<< wordlist1.m_count << " " << wordlist1.m_max;
//prints 3 5 5
```

Function 10: `int find(const char word[]) const;`

Specification: Finds the first occurrence of the word in the WordList returns the position of that word in the list. Otherwise, if the word is not in the WordList or if m_list is nullptr return -1.

Usage:
```
WordList wordlist(4);
wordlist.add("Where");
wordlist.add("Is");
wordlist.add("Waldo");

int retval = wordlist.find("waldo");
if (retval < 0)
cout << "Not found" << endl;
```

Function 11: `int sort();`

Specification: If WordList is nullptr return -1. If there is only one word in the *m_list* return 1. Otherwise, sort sorts the words in WordList in ascending order, returns 0.

Usage:
```
WordList neverwhere(5);
neverwhere.add("Richard");
neverwhere.add("Door");
neverwhere.add("Carabas");
neverwhere.add("Islington");
neverwhere.add("Abbot");
neverwhere.sort();
neverwhere.print(); // prints "Abbot Carabas Door Islington Richard\n"
```

Function 12: `WordList& operator=(const WordList &other);`

Specification: Copies from an existing WordList to another exiting WordList. Should not result in any memory leaks or aliasing of dynamically allocated memory.

Usage:
```
WordList ben(5);     // WordList can store 5 words
//... Some WordList operations
if (...) {
       WordList ann(4);
       ann = ben;
} // Should not results in undefined behavior
ben = ben; // should not result in undefined behavior
```

Submission

Place both wordlist.cpp and studentinfo.h in a zip file for submission. I will use a similar wordlist.h that is provided to you, this means you cannot make any changes to the header file otherwise it will conflict with the header file I will use.

Cstrings

Cstrings are the way C implements text/variable functionality, they are simply character arrays with an additional character '\0', null terminated character, to indicate the end of the cstring. C++'s string class uses character arrays as its underlying data structure, but does not include the use of the null terminated character.

Characters in programming languages are encoded, the simplest encoding on modern machines being ASCII, meaning there is a unique numerical value associated with each character. A consequence of this is that upper case characters are different from their lower case counterparts. In particular, 'A' is encoded as 65 and 'a' is 97. This also means the 'A' is less than 'a', which might seem counter intuitive but for this project it is easiest simply to utilize these character encodings. In particular, since "hAt" is less than "hat" due to the differences in the second character; "hAt" would come before "hat" in a sorted sense.

You can declare a statically defined cstring with:

```
char cstr[] = "This is a Cstring.";
```

This cstring has 18 characters +1 for the null termination character, so the total size of this character array is 19. You can also specify a fixed sized character array and initialize the first few elements to a cstring:

```
char cstr1[20] = "Hi";
```

The total size of cstr1 is 20 elements but in terms of the cstring itself, there are currently only 2 characters +1 for the null terminiation character. There are many prebuilt functions you can use in working with cstrings, details can be found in any C++ reference for the cstring library. Below is a brief description on a few of the more common ones.

You can obtain the length of a cstring by using the the strlen function from cstring library, it will return the number of characters in the cstring, not including the null termination character:

```
char cstr[] = "This is a Cstring.";
int len = strlen(cstr); // 18 characters in length
cout << "Len: " << len << endl; // 18
// The 19th character is the null terminated character
if (cstr[len] == '\0') cout << "null terminated" << endl;
```

You can compare two cstrings using the strcmp function. This will return 0 if they are identical, any other value returned indicates they are not the same, it can be positive or negative. If the return value is negative the first word is "smaller" than the second, if it is positive, the first word is "greater" than the second. For example:

```
cout << "strcmp: " << strcmp("hAt", "hat") << endl; // -1
```

You can copy from one cstring into another using the strcpy function:

```
char cstr1[20] = "Hi";
char* cstr2 = new char[20];

strcpy(cstr2, cstr1);
cout << cstr2 << endl; // Hi

strcpy(cstr2, "This is a test");
cout << cstr2 << endl; // This is a test
```

strcpy is considered unsafe since it doesn't do any bounds checking among other issues. However, as long as you ensure that all your static/dynamic character arrays are the same size and reside in different regions in memory and you make sure the cstrings have length 1 less than the array sizes there should be no issues.

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?
- 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.
- Encapsulation dictates that we make our member variables private so users of our code cannot affect the intended behavior of our code, among other reasons. However, when developing there is nothing stopping you from making member variables public to make testing easier. This is especially useful when you have not implemented any printing functions yet and have no way to see if say your constructors are doing what they should be. Of course, when it comes time to deploy make the member variables private again.
- Work on a single function at a time, start with implementing the constructors, *add* and *print* and testing those to ensure they work to spec. This will generally always be the first things to implement for classes, you need a way to create your object and some way to observe its state in order to conduct any meaningful testing of later functions.
- When submitting over Canvas, it will give me 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 I can grade. But, be sure your code compiles first.