UniqueVector and Classroom

Background Information

Static arrays in C++ pose a very common problem—the need to know exactly how much data you need to store at compile time. Frequently, we do not know this information ahead of time because the values come from a variable input source, such as a file of unknown size or keyboard user input. Dynamic arrays solve this problem by allowing the number of stored values to be determined at run time. Despite this benefit, dynamic arrays can be cumbersome to deal with directly.

A vector serves as a wrapper for a dynamic array—a wrapper in that it surrounds the array with a class and creates an interface that provides well-defined functionality, such as accessing, removing, or counting the array elements. A further benefit is that the vector handles the dynamic memory allocation and resizing of its underlying array as necessary under the hood, without the user of the vector needing to know anything about it. Abstraction, for the win!

In this assignment, you will be creating your very own vector class, except with a twist. Your vector class must maintain a condition of **unique-ness**. At any given moment, all of the elements must be unique. That is to say, the vector cannot contain any duplicates.

UniqueVector Requirements

You must implement a class called **UniqueVector** that uses a dynamic array as its underlying data structure. You may not use the STL vector as the underlying data structure. Your class must be able to store any type, meaning that it must be templated. All instances initially start with a dynamically created array of size three. Make sure to deallocate your array when necessary. The **UniqueVector** class interface must provide the following functionality:

1. `unsigned int capacity()` — Returns the size of the space currently allocated for the vector.
2. `unsigned int size()` — Returns the current number of elements in the vector.
3. `bool empty()` — If the vector contains zero elements, returns **true**; otherwise, returns **false**.
4. `bool contains(const T& data)` — If the vector contains **data**, returns **true**; otherwise, returns **false**.
5. `bool at(unsigned int pos, T& data)` — If **pos** is a valid position, retrieves the element in position **pos** in the array, stores it in **data**, and returns **true;** otherwise, returns **false**.
6. `bool insert(const T& data)` — If the vector does not already contain **data**, adds a new element, **data**, to the back of the vector, increases the container size by one, and returns **true**; otherwise, returns **false**. If the underlying array is full, creates a new array with double the capacity and copies all of the elements over.
7. `bool insert(const T& data, unsigned int pos)` — If the vector does not already contain **data**, adds a new element, **data**, to the vector at position **pos**, increases the container size by one, returns **true**; otherwise, returns **false**. If the underlying array is full, creates a new array with double the capacity and copies all of the elements over.
8. `bool push_front(const T& data)` — If the vector does not already contain **data**, adds a new element, **data**, to the front of the vector, increases the container size by one, and returns **true**;

otherwise, returns **false**. If the underlying array is full, creates a new array with double the capacity and copies all of the elements over.

9. `bool remove(const T& data)` — If the vector contains **data**, removes **data** from the vector, reduces the container size by one, leaves the capacity unchanged, and returns **true**; otherwise, returns **false.**
10. `bool remove(unsigned int pos, T& data)` — If **pos** is a valid position, removes the element in position **pos**, stores it in **data**, reduces the container size by one, leaves the capacity unchanged, and returns **true**; otherwise, returns **false**.
11. `bool pop_back(T& data)` — If the vector is not empty, removes the last element in the vector, stores it in **data**, reduces the container size by one, leaves the capacity unchanged, and returns **true**; otherwise, returns **false**.
12. `void clear()` — Empties the vector of its elements and resets the capacity to 3.
13. `Overload operator==` — If the vector on the left hand side has the same elements in the same order as the vector on the right hand side, returns **true**; otherwise, returns **false**.

Classroom Requirements

You must also implement a class called **Classroom**. This class will utilize a **UniqueVector** as its underlying data structure in order to maintain a unique roster of student names, stored as strings. The **Classroom** class interface must provide the following functionality:

1. `bool addStudent(const string& name)` — If a student named **name** is not already on the classroom roster, adds a new student named **name** to the classroom roster and returns **true**; otherwise, returns **false**.
2. `bool removeStudent(const string& name)` — If a student named **name** is on the classroom roster, removes the student named **name** from the classroom roster and returns **true**; otherwise, returns **false**.
3. `bool containsStudent(const string& name)` — If a student named **name** is on the classroom roster, returns **true**; otherwise, returns **false**.
4. `string listAllStudents()` — Returns a string containing the names of the students in the classroom, separated by commas.

Extra Credit

For extra credit, you can implement extra pieces of functionality. Each function is worth up to 2.5% extra points towards your project's final grade. Add the following functions to the **Classroom** interface:

1. `string removeAlphabeticallyFirst()` — Removes and returns the student whose name comes lexicographically first on the classroom roster.
2. `string removeAlphabeticallyLast()` — Removes and returns the student whose name comes lexicographically last on the classroom roster.
3. `void combineClasses(Classroom& otherClass)` — Adds all of the student names on **otherClass**' roster onto the roster of the Classroom calling this function and leaves **otherClass** unchanged.

Things to Think About

- What data members are needed for a vector to maintain information about its current state?
- Why is the insert function the key to maintaining the condition of uniqueness in UniqueVector?
- Technically, a function can only return one value. However, many of the UniqueVector functions make use of pass-by-reference parameters; the functions usually return a `bool` to indicate success or failure and also "fill in" the value of the reference parameter. This effectively allows a function to "return" multiple values. Take the time to understand why this is useful.
- You'll notice that some parameters are passed as constant references (`const string&`) as opposed to just the bare type (`string`). Do some research to figure out why this is useful.
- Before implementing the extra credit alphabetical functions in Classroom, find out what happens when you compare two strings normally in C++. Check the STL.
- All classes are `friend`s of themselves. This means that within the implementation of `combineClasses()`, the private variables of **otherClass** are accessible.

Testing Instructions

For this assignment, the files **main.cpp, UniqueVectorTester.cpp,** and **ClassroomTester.cpp** are provided for you. These files are designed to test your classes' functions, as per the aforementioned interfaces, as you implement them. These tests make use of Catch, an automated test framework for C++. They compare the expected output of the functions with the output generated by your code. If the outputs are the same, the test passes and the program continues executing normally. Otherwise, the program ends and prints a message detailing the test failure and its line number. In the tester files, above each test, you'll find a comment with a more detailed explanation of the aspect of your code being tested, which should give you a good idea about what needs fixing. Finally, even if your code passes all of the tests, that does not necessarily mean that it does not contain errors. These tests are provided to aid your debugging, not replace it.

First, let's understand how to run all of the tests. Before compiling your code, make sure to first update the SOURCES variable in your makefile with all of the required source files. Compile your code with:

```
$ make
```

If it compiles successfully, then you can run all of the tests by running the generated executable:

```
$ ./UniqueVector
```

You can automatically get rid of the generated object files (*.o) and executable by running:

```
$ make clean
```

Chances are, you are going to code UniqueVector first (hint: you should). If you want to test only your UniqueVector code, update the makefile SOURCES accordingly by removing any Classroom related source files and then compiling/executing as above.

Similarly, you may want to test individual functions as you code to ensure that they work before moving on. The tester files contain a number of TEST_CASE's that allow you to test individual functions before you fully finish the assignment. Use these test cases to your advantage! Do not attempt to do everything at once. Code small pieces and test iteratively. You can run the compiled executable followed by the test case 'tag' to run just that series of tests.

```
$ ./UniqueVector [capacity]
```

The above command will proceed to run just the first set of tests described in UniqueVectorTester.cpp. However, note that each test case has certain function dependencies—you must be certain that those functions exist before that test case can be run. All of the test case tags are listed below, followed by a list of each one's dependencies.

The **UniqueVectorTester** test cases are:

1. [capacity] — `insert(), capacity()`
2. [size] — `insert(), size()`
3. [empty] — `insert(), empty()`
4. [at] — `insert(), at()`
5. [clear] — `insert(), size(), capacity(), clear()`
6. [contains] — `insert(), contains(), remove()`
7. [remove] — `insert(), contains(), size(), at(), remove()`
8. [pop_back] — `insert(), empty(), size(), pop_back()`
9. [push_front] — `insert(), at(), push_front()`
10. [equality] — `insert(), remove(), clear(), operator==`

The **ClassroomTester** test cases are:

1. [addStudent] — `addStudent(), containsStudent(), removeStudent()`
2. [alphabetical] — `addStudent(), removeStudent(), removeAlphabeticallyFirst(), removeAlphabeticallyLast()`
3. [combine] — `addStudent(), combineClasses(), listAllStudents()`

Submission Details

Your submission must include all of the header/source files (*.h/*.cpp) required for your program to properly compile and run. You must include the provided makefile and update it to include the sources to be compiled. Do not submit any generated executables or object files (*.o). Confirm that your code successfully compiles and runs on the UNIX lab machines. Do not modify any of the provided starter code; it contains tests that you (and I) will use to confirm valid functionality. If you wish to write your own main file for your own testing purposes, you may do so, but do not submit it with the assignment. You must properly document your code. You must also update the README file with any guidance that someone looking at your project needs to and might like to know; this could include compilation instructions, interface specifications, interactions between classes, problems overcome, etc. See the **GitHub Submission Guide** on Piazza for step-by-step information about how to submit your assignment via git and GitHub. Be sure to ask questions on the Q&A board if you have any issues.

Grading

The assignment grade is broken down as follows:

Components	Percentage	Relevant Questions
Correctness	50%	Do your classes exhibit functionality as expected?
Documentation	35%	Does your README provide users with adequate information about your program? Have you adequately commented your class interface and implementation files (when necessary)?
Style	15%	Is your code readable, with proper whitespace? Is your coding style consistent? Do you follow the modified Google Style Guide (to the best of your ability)?

If the program does not compile, you will receive no points for the **Correctness** component. Be sure to follow the Academic Integrity requirements stipulated in the syllabus. Failure to do so will result in a failing grade and disciplinary action.

Due Date

The due date is Monday, October 3rd, by 11:59pm. A sample solution will be provided some time after this date. For every day that you miss the deadline, I will deduct 10% from the project's final grade. If you have a reasonable excuse and an extension past the deadline would benefit you, I would be glad to provide one. Issues with submitting via git/GitHub are not valid excuses.