Flashcards

Background Information

Static arrays in C++ pose a 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 a 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 own vector class with a twist. Your vector class must maintain a condition of unique-ness. In other words, the vector cannot ever contain any duplicates. Moreover, you will use this specialized vector to implement flashcards, an age-old method of studying. A flashcard is a card with a term/phrase on the front and a description/definition on the back. You will implement three classes: **UniqueVector**, **Flashcard**, and **FlashcardDeck**. **FlashcardDeck** will essentially utilize a **UnqiueVector** as its underlying data structure to hold a collection of **Flashcard**s.

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()` — Return the size of the space currently allocated for the vector.
2. `unsigned int size()` — Return the current number of elements in the vector.
3. `bool empty()` — If the vector contains zero elements, return **true**; otherwise, return **false**.
4. `bool contains(const T& data)` — If the vector contains **data**, return **true**; otherwise, return **false**.
5. `bool at(unsigned int pos, T& data)` — If **pos** is a valid position, retrieve the element in position **pos** in the array, store it in **data**, and return **true;** otherwise, return **false**.
6. `bool insert(const T& data)` — If the vector does not already contain **data**, add a new element, **data**, to the back of the vector, increase the container size by one, and return **true**; otherwise, return **false**. If the underlying array is full, create a new array with double the capacity and copy all of the elements over.
7. `bool insert(const T& data, unsigned int pos)` — If the vector does not already contain **data**, add a new element, **data**, to the vector at position **pos**, increase the container size by one, return

true; otherwise, return **false**. If the underlying array is full, create a new array with double the capacity and copy all of the elements over.

8. `bool push_front(const T& data)` — If the vector does not already contain **data**, add a new element, **data**, to the front of the vector, increase the container size by one, and return **true**; otherwise, return **false**. If the underlying array is full, create a new array with double the capacity and copy all of the elements over.
9. `bool remove(const T& data)` — If the vector contains **data**, remove **data** from the vector, reduce the container size by one, leave the capacity unchanged, and return **true**; otherwise, return **false.**
10. `bool remove(unsigned int pos, T& data)` — If **pos** is a valid position, remove the element in position **pos**, store it in **data**, reduce the container size by one, leave the capacity unchanged, and return **true**; otherwise, return **false**.
11. `bool pop_back(T& data)` — If the vector is not empty, remove the last element in the vector, store it in **data**, reduce the container size by one, leave the capacity unchanged, and return **true**; otherwise, return **false**.
12. `void clear()` — Empty the vector of its elements and reset 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, return **true**; otherwise, return **false**.

Flashcard Requirements

You must implement a struct/class called **Flashcard** that stores the data of a single flashcard, i.e., the front (term/phrase) and the back (definition/description). In addition, the **Flashcard** interface must provide the following functionality:

1. `Overload operator==` — If the flashcard on the left hand side has the same front value as the flashcard on the right hand side, return **true**; otherwise, return **false**.

FlashcardDeck Requirements

You must implement a class called **FlashcardDeck**. This class will utilize a **UniqueVector** as its underlying data structure in order to maintain a unique pack of **Flashcard**s. The **FlashcardDeck** class interface must provide the following functionality:

1. `bool addCard(const string& front, const string& back)` — If a card with the same **front** value is not already in the deck, add a new card with the supplied **front** and **back** and return **true**; otherwise, return **false**.
2. `bool removeCard(const string& front)` — If a card with the provided **front** value is in the pack, remove the card from the deck and return **true**; otherwise, return **false**.
3. `bool containsCard(const string& front)` — If a card with the provided **front** value is in the deck, return **true**; otherwise, return **false**.
4. `string listFlashcards()` — Return a string containing all of the flashcards, where the format for each card is the front value, followed by a newline, followed by the respective back value, followed by two newlines.

Extra Credit

For extra credit, you can implement more functionality. Each function is worth up to 0.25 extra points towards your project's final grade.

Add the following function to the **Flashcard** interface:

1. `Overload operator<<` — Output the card to the stream, where the format is the front value, followed by a newline, followed by the respective back value, followed by two newlines.

Add the following functions to the **FlashcardDeck** interface:

1. `void combineFlashcards(FlashcardDeck& otherFlashcardDeck)` — For any flashcards in **otherFlashcardDeck** that do not exist in *this* flashcard deck, add them to *this*. The **otherFlashcardDeck** should remain unchanged.
2. `void shuffleFlashcards(FlashcardDeck& otherFlashcardDeck)` — Combine *this* flashcard deck with **otherFlashcardDeck** in such an order that the first flashcard in *this* is followed by the first flashcard in **otherFlashcardDeck** that does not exist in *this*, followed by the second flashcard in *this*, followed by the second flashcard in **otherFlashcardDeck** that does not exist in *this*, and so on until **otherFlashcardDeck** runs out. The **otherFlashcardDeck** should remain unchanged.

Testing Instructions

For this assignment, the files **catch.h**, **main.cpp**, **UniqueVectorTester.cpp**, and **FlashcardDeckTester.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, since these unit tests are a form of blackbox testing, even if your code passes all of the tests, it does not necessarily mean that the code does not contain errors. These tests are as close a proxy to correctness as possible.

First, let's understand how to run all of the tests. Before compiling your code, make sure to open the makefile (just like a text file!) and update the SOURCES variable with a whitespace separated list of the .cpp files to compile and the EXECUTABLE_NAME variable with the name **Flashcards**. In order to compile your code, run:

```
$ make
```

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

```
$ ./Flashcards
```

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

```
$ make clean
```

Chances are, you are going to implement UniqueVector first (hint: you should). If you want to test only your UniqueVector code, update the makefile SOURCES accordingly by removing any FlashcardDeck related source files and then compiling/executing as above. Note that you might get errors when compiling that indicate missing functions. You can avoid these by providing stub implementations of the required functions (that intentionally fail tests) and then filling them in as you get to them.

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, e.g.:

```
$ ./Flashcards [capacity]
```

The above command will proceed to run just the capacity related set of tests described in UniqueVectorTester.cpp. All of the test case tags are listed below:

The **UniqueVectorTester** test cases are:

1. [capacity]
2. [size]
3. [empty]
4. [contains]
5. [at]
6. [clear]
7. [insert at position]
8. [push_front]
9. [remove]
10. [remove by position]
11. [pop_back]
12. [equality]

The **FlashcardDeckTester** test cases are:

1. [card]
2. [combine] (Extra credit! Uncomment this out in FlashcardDeckTester.cpp)
3. [shuffle] (Extra credit! Uncomment this out in FlashcardDeckTester.cpp)
4. [ostream] (Extra credit! Uncomment this out in FlashcardDeckTester.cpp)

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.
- All classes are friends of themselves. This means that within the implementation of a function like `combineFlashcards()`, the private variables of **otherFlashcards** are accessible.
- Don't forget about memory management!

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 update the makefile to include the sources to be compiled. The name of the generated executable must be **Flashcards**. Do not submit any generated executables or object files (*.o). You must also update the README file with any guidance that someone looking at your project needs to and might like to know; this includes compilation instructions, interface specifications, interactions between classes, problems overcome, etc. Confirm that your code successfully compiles and runs on the Linux lab machines. See the **Assignment Guide Using Git & GitHub** for step-by-step information about how to submit your assignment via git and GitHub.

Grading

Your grade is comprised of the following:

Components	Percentage	Relevant Questions
Correctness	50%	Do the unit tests pass for each of your class functions?
Documentation	35%	Does your README document provide users with adequate information about your program? Do you adequately comment your class interface files and class implementation files (when necessary)?
Style	15%	Is your coding style readable and consistent? Do you follow (to the best of your ability) the modified Google Style Guide?

If your program does not compile on the Linux machines, you will receive no points for the **Correctness** component.

Due Date

The due date is Monday, October 2nd, 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.

Final Words

Don't leave this assignment for the last minute! Be sure to give yourself plenty of time to plan before coding and debugging. Before submitting, ensure that your code compiles and runs successfully on the lab machines. Moreover, ensure that your code passes all of the provided tests. Good luck!