SortedVector Class

Background Information

Static arrays in C++ pose a very common problem—the need to know exactly how big the array should be at compile time. Frequently, we do not know this information 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 values stored to be determined at run time.

A vector serves as a wrapper for a dynamic array—a wrapper in that it surrounds the array with a class, creating an interface that provides certain functionality, such as accessing, removing, or counting the underlying elements. A further benefit is that vectors handle the dynamic memory allocation and resizing of arrays as necessary under the hood.

In this assignment, you will be creating your very own vector class, except with a twist. Your vector must maintain a condition of **sorted-ness**. At any given moment, all of the elements in the vector must be in ascending sorted order.

Requirements

You must implement a class called **SortedVector** using dynamic arrays. You may not use the STL vector as an 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 data where necessary. The class interface should provide the following functionality:

High Priority:

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. `void clear()` — Clears the vector by resetting the vector's size and capacity.
4. `bool empty()` — Returns whether the vector is empty (contains 0 elements).
5. `T at(unsigned int pos)` — Returns the element in position **pos** in the array. If **pos** is not a valid position, write **throw -1;** in the code.
6. `void insert(const T& data)` — Adds a new element, **data**, to the vector and maintains the sorted condition. This increases the container size by one. If the underlying array is full, create a new array with double the capacity and copy all of the elements over.
7. `T remove(unsigned int pos)` — Removes and returns the element in position **pos**. This reduces the container size by one and leaves the capacity unchanged. If **pos** is not a valid position, write **throw -1;** in the code.

Low Priority:

1. `unsigned int count(const T& data)` — Returns the number of times **data** exists in the vector.
2. Overload the `==` operator, which determines if one vector is exactly the same as another—in other words, the vectors have elements with the same values in the same order.
3. `T pop_back()` — Removes and returns the last element in the vector. This reduces the container size by one and leaves the capacity unchanged. If the vector is empty, write **throw -1;** in the code.
4. `void display()` — Prints out all of the elements of the vector in order, separated by commas.

Things to Think About

- Some requirements are unsaid in the aforementioned functionality set; think about what else you need to ensure that this class is properly written and correctly implemented.
- 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 sortedness in this class?
- The provided tester files run tests to ensure valid functionality. Use them to your advantage as you write code. Do not attempt to do everything at once. Code small pieces and test iteratively.

Submission Details

Your submission must include all the header (*.h) and source files (*.cpp) required for your program to properly compile. You must submit the provided **main.cpp** and **SortedVectorTester.h/cpp**; these files contain tests that you (and I) will use to confirm valid functionality. Therefore, do not remove them. If you wish to write your own main file for your own purposes, you may do so, but do not submit it with the assignment. Do not submit any generated executables or object files (*.o).

You must use and submit the provided makefile—updated only to include the executable file name and the sources to be compiled. You must properly document your code. You must also update the README file with any guidance that someone looking at and/or running your code needs to (and might like to) know; this could include compilation instructions, interactions between classes, problems overcome, etc.

See the **GitHub Submission Guide** on Piazza for 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

This is a required assignment for the course. However, it will not be included in the **Programming Assignments** portion of your grade (see syllabus); it will instead be worth five points out of the **Homework & Lab Assignments'** fifteen points. You must complete all of the project specifications in order for this assignment to be considered complete.

Due Date

The due date is Wednesday, March 9th, before the start of our class (7pm). A sample solution will be provided after this date.