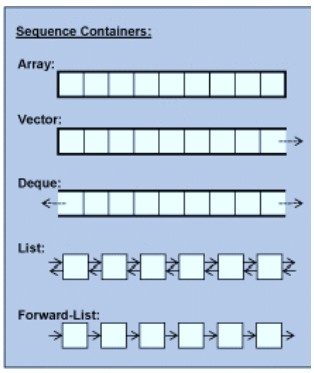
CPSC 131, Data Structures – Fall 2022
Homework 2: Sequence Containers



Learning Goals:

- Familiarization with arrays, extendable vectors, (doubly linked) lists, and (singly linked) forward lists insertion, deletion, traversals, and searches.
- Reinforce the similarities and differences between the sequence data structures and their interfaces.
- Analyze and understand the differences in the sequence container's complexity for some of the more common operations
- Familiarization and practice using the STL's sequence container interface
- Reinforce modern C++ object-oriented programming techniques



Description:

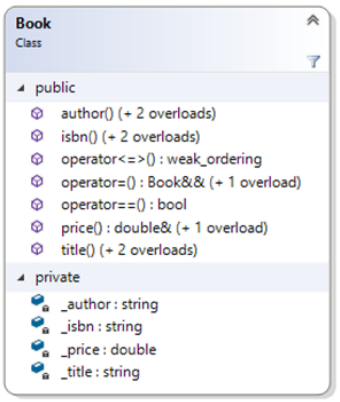
This BookList assignment builds on the Book class from the previous assignment. Here you create and maintain a collection of books to form a book list. Books are placed on the list, removed from the list, and reordered within the list. A complete `BookList` class interface and partial implementation have been provided. You are to complete the implementation.

To reinforce the four data structure concepts (arrays, extendable vectors, doubly linked lists, and singly linked lists) discussed in class, your book list implementation mirrors book insertion, removal, and reordering requests to each of four STL containers (`std::array`, `std::vector`, `std::list`, and `std::forward_list` respectively). At the end of each operation the four STL containers must be consistent. For example, when a BookList object receives a request to insert a book, your implementation of BookList::insert() will insert that

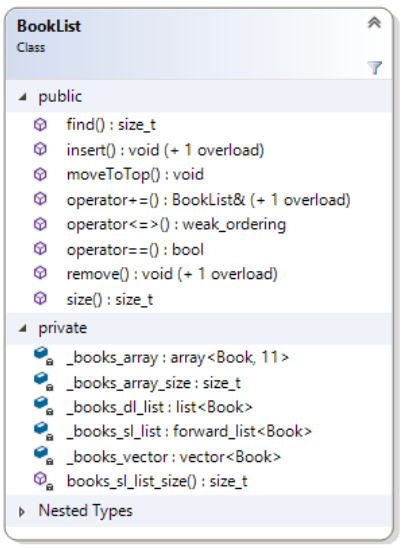
book into all four STL containers such that each container holds the same books in the same order.

The following class diagrams should help you visualize the `BookList` interface, and to remind you what the `Book` interface looks like.

class* `Book` *summary



class* `BookList` *summary



Book list function summaries:

1. `find()` takes a book as a parameter and returns the zero-based offset of that book, or the total number of books in the book list if the book was not found. For example, if your book list contains the books in the picture above, a request to find “East of Eden” returns 0, “The Good Earth” returns 5, and “Eat Pray Love” returns 48.
2. `insert()` takes a book and either a position (TOP or BOTTOM) or an offset into the list as parameters and inserts the provided book before the insertion point. For example, again if your book list contains the books in the picture above, inserting “Eat Pray Love” with an offset of 5 places “Eat Pray Love” between “Anna Karenina” and “The Good Earth”. Silently discard duplicate books from getting added to the book list.
3. `moveToTop()` takes a book as a parameter, locates and removes that book, and then places it at the top of the list. For example, a request to move “Les Mis” to the top removes “Les Mis” from its current location and places it before “East of Eden”. Of course, “Hunger Games” would then immediately follow “A Tale of 2 Cities”. The book list remains unchanged if the provided book is not in the book list.
4. `operator+=()` concatenates the provided book list to the bottom of this book list. Silently discard duplicate books during concatenation.
5. `operator<=>()` returns a negative number if this book list is less than the other book list, zero if this book list is equal to the other book list, and a positive number if this book list is greater than the other book list.
6. `operator==()` returns true if this book list is equal to the other book list, false otherwise.
7. `remove()` takes either a book or a zero-based offset from the top as a parameter and removes that book from the book list. No change occurs if the given book is not in the book list, or the offset is past the size of the book list. For example, a request to remove “Les Mis” from your above pictured book list reduces the size of the book list by one and causes “Hunger Games” to immediately follow “A Tale of 2 Cities”.

8. `size()` takes no parameters and returns the number of books in the book list. For example, the size of your above pictured book list is 48. The size of an empty book list is zero.

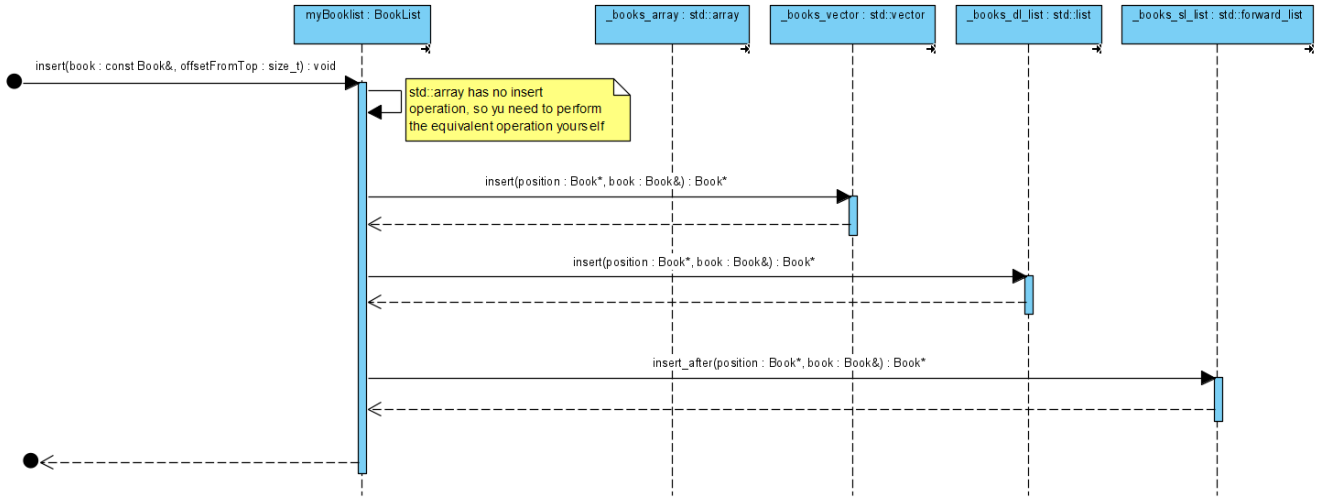
How to Proceed:

The following sequence of steps are recommended to get started and eventually complete this assignment.

1. Review the solution to the last homework assignment. Use the posted solution to fix your solution and verify it now works. Your Book class needs to be working well before continuing with this assignment.

   Book's constructor's parameter order is important for this assignment. You must allow books to be constructed with at least zero, one, or two arguments. If one argument is given, it must be the title. If two arguments are given, the first one must be the title, and the second one must be the author. Take a close look at last assignment's posted solution and double check your books constructor's parameter order.

   When you're ready, replace the Book.cpp file packaged with this assignment with your (potentially updated) Book.cpp file from last assignment.
2. Compile your program using Build.sh. There will likely be warnings, after all it's only partial solution at this point. If there are errors, solve those first. For example, implement books_sl_list_size() first to remove the "must return a value" error. Your program should now execute.
3. Once you have an executable program, start implementing functions with the fewest dependencies and work up. Consider this order: `books_sl_list_size()`, `size()`, `find()`, `insert()`, `remove()`, `moveToTop()`, and finally `opertor+=()`.
4. Implementing insert() and remove() is really implementing insert and remove on each of the four STL containers. For example, upon receipt of an "insert" request, BookList::insert() inserts the book into the `_books_array`, then into the `_books_vector`, then the `_books_dl_list`, and finally into the `_books_sl_list`.



You may want to:

   a. Work the insert() and remove() functions together for arrays, then for vectors, lists, and finally forward_lists. Insertion and removal (or as the STL calls it, erasure) are very close complements of each other.
   b. While working insert and remove for each container, you may want to temporary turn off container consistency checking by commenting out those functions. But don't forget to uncomment them before you're finished.

Rules and Constraints:

1. You are to modify only designated TO-DO sections. The grading process will detect and discard any changes made outside the designated TO-DO sections, including spacing and formatting. Designated TO-DO sections are identified with the following comments:

```
////////////////////////// TO-DO (X) ////////////////////////////////
...
////////////////////// END-TO-DO (X) ////////////////////////////////
```

   Keep and do not alter these comments. Insert your code between them. In this assignment, there are 18 such sections of code you are being asked to complete. 17 of them are in BookList.cpp, and 1 in main.cpp.

   Hint: In most cases, the requested implementation requires only a single line or two of code. Of course, finding those lines is non-trivial. Most all can be implemented with less than 5 or 6 lines of code. If you are writing significantly more than that, you may have gone astray.

Reminders:

- The C++ using directive `using namespace std;` is **never allowed** in any header or source file in any deliverable product. Being new to C++, you may have used this in the past. If you haven't done so already, it's now time to shed this crutch and fully decorate your identifiers.
- It is far better to deliver a marginally incomplete product that compiles error and warning free than to deliver a lot of work that does not compile. A delivery that does not compile clean may get filtered away before ever reaching the instructor for grading. It doesn't matter how pretty the vase was, if it's broken nobody will buy it.
- Always initialize your class's attributes, either with member initialization, within the constructor's initialization list, or both. Avoid assigning initial values within the body of constructors.
- Use Build.sh on Tuffix to compile and link your program. The grading tools use it, so if you want to know if you compile error and warning free (a prerequisite to earn credit) than you too should use it.
- Filenames are case sensitive on Linux operating systems, like Tuffix.
- You may redirect standard input from a text file, and you must redirect standard output to a text file named output.txt. Failure to include output.txt in your delivery indicates you were not able to execute your program and will be scored accordingly. A screenshot of your terminal window is not acceptable. See How to build and run your programs. Also see How to use command redirection under Linux if you are unfamiliar with command line redirection.

Deliverable Artifacts:

Provided files	Files to deliver	Comments
Book.hpp BookList.hpp	1. Book.hpp 2. BookList.hpp	You shall not modify these files. The grading process will overwrite whatever you deliver with the ones provided with this assignment. It is important your delivery is complete, so don't omit these files.
Book.cpp	3. Book.cpp	Replace with your (potentially updated) file from the previous assignment.
main.cpp BookList.cpp	4. main.cpp 5. BookList.cpp	Start with the files provided. Make your changes in the designated TO-DO sections (only). The grading process will detect and discard all other changes.
	6. output.txt	Capture your program's output to this text file using command line redirection. See command redirection. Failure to deliver this file indicates you could not get your program to execute. Screenshots or terminal window log files are not permitted.
	readme.*	Optional. Use it to share your name and/or your thoughts with the grader. Names can also be provided via contributors.txt
BookListTests.cpp BookTests.cpp CheckResults.hpp		These files contain code to regression test your Book and BookList classes. When you're far enough along and ready to have your class regression tested, then place these files somewhere in your working directory and Build.sh will find them. Simply having these files in your working directory (or sub directory) will add it to your program and run the tests – you do not need to #include anything or call any functions. These tests will be added to your delivery and executed during the grading process. The grading process expects all tests to pass.
sample_output.txt		A sample of a working program's output. Your output may vary.

References documentation by Professor Thomas Bettens