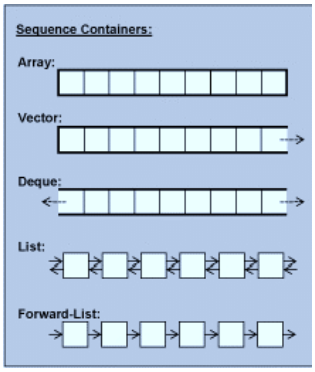
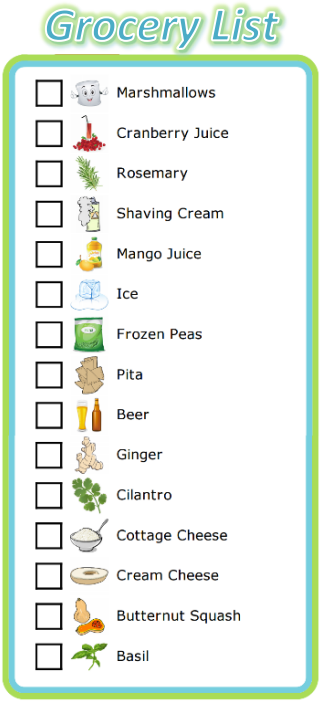
CPSC 131, Data Structures – Fall 2022
Homework 2: Sequence Containers

Learning Goals:



- Familiarization with arrays, extendable vectors, and doubly and singly linked lists
- Understanding the sequence container's insertion, deletion, traversal, and search concepts.
- 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
- Reinforce the software development cycle and practice building error free solutions on Linux

Description:

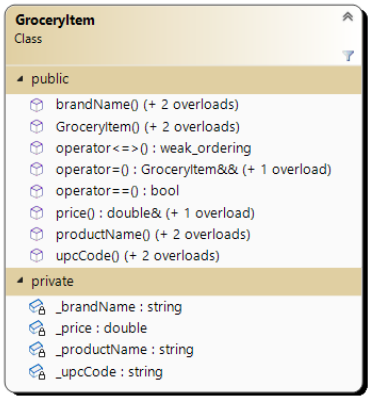


This Grocery List assignment builds on the GroceryItem class from the previous assignment. Here you create and maintain a collection of grocery items to form a grocery list. Grocery items are placed on the list, removed from the list, and reordered within the list. A complete `GroceryList` class interface and partial implementation have been provided. You are to complete the implementation.

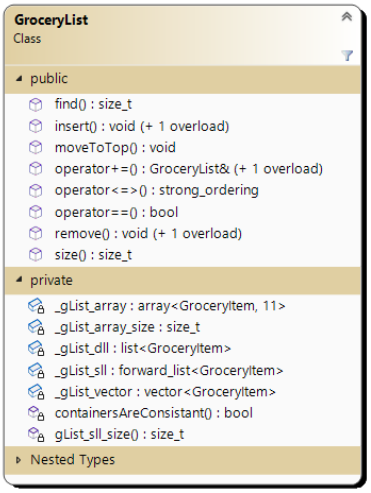
To reinforce the four data structure concepts (arrays, extendable vectors, doubly linked lists, and singly linked lists) discussed in class, your grocery list implementation mirrors grocery item insertion, removal, and reordering requests to each of four STL containers (`std::array`, `std::vector`, `std::list`,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

The following UML class diagrams should help you visualize the `GroceryList` interface, and to remind you what the `GroceryItem` interface looks like.

class* `GroceryItem` *summary



class* `GroceryList` *summary



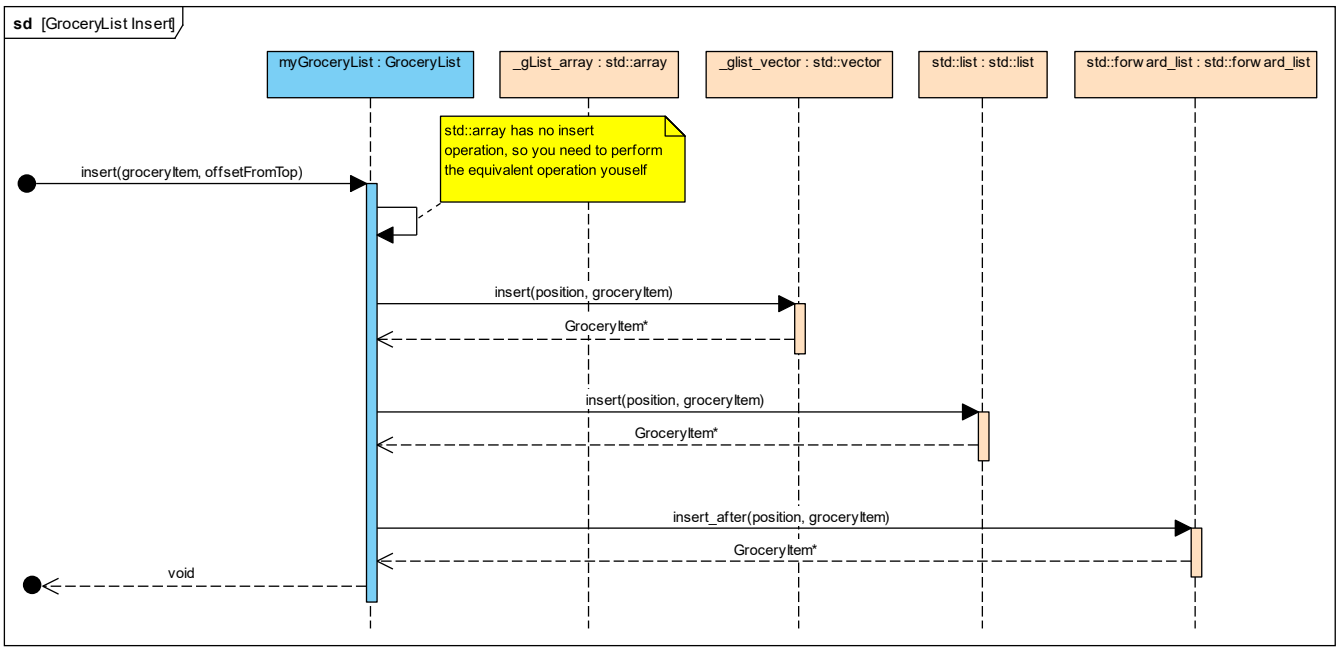
GroceryList function summaries:

1. `find()` takes a grocery item as a parameter and returns the zero-based offset of that grocery item, or the total number of grocery items in the grocery list if the grocery item was not found. For example, if your grocery list contains the grocery items in the picture above, a request to find "Marshmallows" returns 0, "Ice" returns 5, and "Peanut Butter" returns 15.
2. `insert()` takes a grocery item and either a position (TOP or BOTTOM) or a zero-based offset into the list as parameters and inserts the provided grocery item before the insertion point. For example, again if your grocery list contains the grocery items in the picture above, inserting "Peanut Butter" with an offset of 5 places "Peanut Butter" between "Mango Juice" and "Ice". Duplicate grocery items are silently discarded (not added to the grocery list).
3. `moveToTop()` takes a grocery item as a parameter, locates and removes that grocery item, and then places it at the top of the list. For example, a request to move "Beer" to the top removes "Beer" from its current location and places it before "Marshmallows". Of course, "Ginger" would then immediately follow "Pita". The grocery list remains unchanged if the provided grocery item is not in the grocery list.
4. `operator+=()` appends the provided grocery list to the bottom of this grocery list while silently discarding duplicate grocery items. This behavior is similar to `std::string`'s append (a.k.a. concatenation) operator. See std::string::operator+=().
5. `operator<=>()` returns a negative number if this grocery list is less than the other grocery list, zero if this grocery list is equal to the other grocery list, and a positive number if this grocery list is greater than the other grocery list. This behavior is similar to std::vector's three-way-comparison (a.k.a. spaceship) operator. See std::vector::operator<=>() #7
6. `operator==()` returns true if this grocery list is equal to the other grocery list, false otherwise.
7. `remove()` takes either a grocery item or a zero-based offset from the top as a parameter and removes that grocery item from the grocery list. No change occurs if the given grocery item is not in the grocery list, or the offset is past the size of the grocery list. For example, a request to remove "Frozen Peas" from your above pictured grocery list reduces the size of the grocery list by one and causes "Pita" to immediately follow "Ice".
8. `size()` takes no parameters and returns the number of grocery items in the grocery list. For example, the size of your above pictured grocery list is 15. The size of an empty grocery 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 `GroceryItem` class needs to be working well before continuing with this assignment. When you're ready, replace the GroceryItem.cpp file packaged with this assignment with your (potentially updated) GroceryItem.cpp file from last assignment.
2. Compile your program using Build.sh. There will likely be warnings, after all it's only a partial solution at this point. If there are errors, solve those first. For example, implement `gList_sll_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: `gList_sll_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, `GroceryList::insert()` inserts the provided grocery item into the `_gList_array`, then into the `_gList_vector`, then the `_gList_dll`, and finally into the `_gList_sll`. The following UML sequence diagram summarizes the flow of execution through `GroceryList::insert()`. `GroceryList::remove()` is similar.



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 19 such sections of code you are being asked to complete. 18 of them are in GroceryList.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
GroceryItem.hpp GroceryList.hpp	1. GroceryItem.hpp 2. GroceryList.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.
GroceryItem.cpp	3. GroceryItem.cpp	Replace with your (potentially updated) file from the previous assignment.
main.cpp GroceryList.cpp	4. main.cpp 5. GroceryList.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 communicate your thoughts to the grader
RegressionTests/ GroceryListTests.cpp GroceryItemTests.cpp CheckResults.hpp		These files contain code to regression test your GroceryItem and GroceryList 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.