Alex Rueb
Oregon State University
CS162-400
4/30/2017

Project 2: Reflection and Testing Summary

Problem Description

The goal for this project is to create a grocery shopping list program that allows the user to display and manage a list of items. We must create an *Item* class that contains the item's name, unit of measure, quantity to buy, and unit price, as well as the ability to calculate an extended price and display the item details to the user. A *List* class must be utilized to create and hold the item objects in a dynamic array that automatically resizes as needed. The dynamic array should start empty with an array capacity of four item objects.

The program must have the ability to create a list, add items, remove items, and display the list contents to the user. The program should prompt the user to enter the details of each item that they want to buy and check the list to make sure the item doesn't already exist. If the item is found, the user should then have the option to update the existing item or cancel. To delete an item, the user needs to be able to enter the item name to located in the item list or simply remove the last item from the list. To test if an item already exists, the program needs to make use of an overloaded == operator. How this overloaded operator is used and implemented is at the discretion of the designer.

Requirements Summary

- Program must maintain and display a list of items
- Must Include *Item* class with data members for item name, unit of measure, qty to purchase, and price
- Store items in a dynamic array
- Create a *List* class class that includes a dynamic array to store *Item* objects.
- List array must start empty with a capacity to hold 4 items and resize accordingly as items are added
- Prompt user to enter name, unit, qty, and price to add an item
- Program must be able to create a list
- Program must be able to add items
- Program must be able to remove items (either by searching for an item name or simply removing the last item from the list)
- Program must be able to display the shopping list with each item name, unit, qty to buy, price, extended price

- Test if an item is on a list using an overloaded == operator
- If an item already exists on the list, display a message to the user and ask it they want to update/replace the item.
- Display the total list cost to the user.

Design Solution

My design solution is primarily focused around the *Item* and *List* classes, as stated in the project specifications. The *Item* class is relatively straight forward and contains fields for all of the necessary information about the item, along with functions to display the item data and calculate the extended price. Set methods are also included in the case where an existing item needs to be updated from outside the class.

The list class does most of the heavy lifting, since it is responsible for dynamically creating items via an array of pointers to Items and resizing the array as items are added. Separate variables are included to keep track of the list size, which is initialized at size four, and the item count, which initializes at 0. A *totalPrice* accumulator variable is also included to track the total price of all the list items.

Adding and deleting items are two key areas of functionality for managing the list. The *addItem* method first collects information from the user about the item name, unit of measure, quantity, and price, and then must check the array to see if the item already exists. To test if the item exists, the == operator of the item class is overloaded so that an object can be compared against a string entered by the user. Within this overloaded function, the name of the item and the string are both converted to lowercase characters to eliminate any case sensitivity. An item to be added can then be compared against existing items by looping through the array and using the overloaded == operator. If an item match is found in the array, the program gives the user the option to update the existing item or cancel. If the user updates, the item, set methods are called on the existing item to update the properties with the values collected from the user. In either case, the *itemCount* is adjusted as well as the *totalPrice* variable.

If the item to be added is not found in the array, the item must be dynamically created and the pointer must be added to the array. Before this can occur, the array size must be compared to the list count to ensure the array has the capacity to add the item. If the capacity exists, the item is created and added to the array. If the array is full, the contents of the array is copied to a temporary array while the original array is deleted and recreated at a capacity large enough to hold the new item The temporary contents is then transferred back to the resized array and the temporary array can be deleted from memory. At this point, the new object can be created and stored in the array.

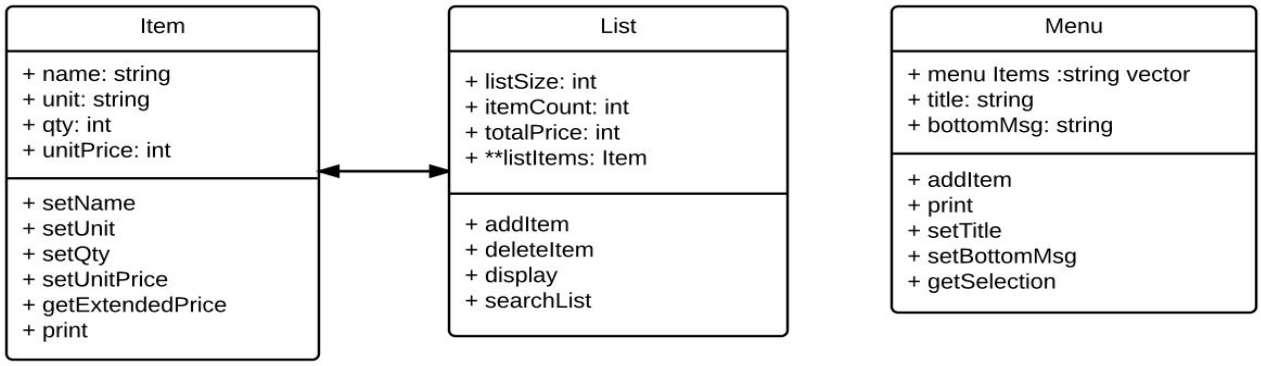
Deleting an item is similar to adding an item, except the user only needs to provide the name of the item to be deleted. The same method of searching the array as described above is

applied to find an existing item with the same name. If an item with a matching name is found, its subscript is returned so that it can be used to delete the item. Once the item is deleted, the subsequent elements must be shifted in the array so that all of the remaining items fall within the subscript of the item count variable. If the item is not found within the array, the user is notified.

To display the list, a function exists within the list class to loop through the array of pointers to *Item* objects and calls the print function defined within the *Item* class. At the end of the loop the total cost of all of the list items is displayed to the user.

A simple menu structure facilitates the list management. An initial menu asks the user if they would like to add an item or exit the program. Once the user adds the first item, another menu asks the user if they would like to add another item, remove an item, display the list, or exit the program. This menu repeats via a *do while* loop until the user exits the program.

Class Hierarchy



Testing Plan and Results

Test Description	**Input(s)**	**Expected Output / Rationale**	**Outputs**
Add a single item and display the list	-Choose to add an item -Enter apple, lb, 3, 2 -Choose to display the list	The item should be displayed in the list with an extended price of $6 and a total list price of $6.	Results are as expected: apple (qty 3 @ $2/lb) - $6 Total list cost: $6

Add a two items and display the list	-Choose to add an item -Enter apple, lb, 3, 2 -choose add another item -Enter pear, ea, 1, 3 -Choose to display the list	Both the apple and pear items should be displayed with extended prices 6 and 3, respectively. The list total should be $9.	Results are as expected: apple (qty 3 @ $2/lb) - $6 pear (qty 1 @ $3/ea) - $3 Total list cost: $9
Add a five items and display the list. This will make sure the array resizes. Also, items with multiple words to test method of gathering input.	-Choose to add an item -Enter apple, lb, 3, 2 -choose add another item -Enter pear, ea, 1, 3 -choose add another item -Enter red grapes, lb, 2, 4 -choose add another item -Enter beer, 6pk, 1, 9 -choose add another item -Enter chicken, lb, 2, 7 -Choose to display the list	All five items will be displayed with a total list cost of $40.	Results are as expected: apple (qty 3 @ $2/lb) - $6 pear (qty 1 @ $3/ea) - $3 red grapes (qty 2 @ $4/lb) - $8 beer (qty 1 @ $9/6pk) - $9 chicken (qty 2 @ $7/lb) - $14 Total list cost: $40
Add an item, delete the item, display the list contents. Use inconsistent cases.	-Choose to add an item -Enter apple, lb, 3, 2 -choose to delete an item -Enter Apple -Choose to display the list	The list will be empty.	Results are as expected: There are no items in your list.
Add two items, delete the item, display the list contents. Use inconsistent cases.	-Choose to add an item -Enter jam, jar, 1, 6 -Choose add another item -Enter jelly, jar, 2, 4 -choose to delete an item -Enter jam -Choose to display the list	Only jelly will be in the list with an exptended price of $8 and a list total or $8	Results are as expected: jelly (qty 2 @ $4/jar) - $8 Total list cost: $8
Add the same item twice. Use inconsistent cases, this time with the last character.	-Choose to add an item -Enter apple, lb, 3, 2 -choose add another item -Enter applE, lb, 2, 2 -Choose to update item -Choose to display the list	Will be given the option to update the item or cancel. After choosing to update and displaying the list, only one Apple item will exist with an extended price or $4 and a list total of $4.	Results are as expected: applE (qty 2 @ $2/lb) - $4 Total list cost: $4

Reflection

For the most part, I found this program to be relatively straight forward to implement. The biggest challenge was meeting the requirement of overloading the == operator. I wasn't sure initially whether I should overload the operator in the Item or List class. On my first attempt, I overloaded the operator in the List class, looping through the array and comparing the Item name with the input name argument. This could successfully identify if an item name already existed in the list, but since it only returned true or false, it wasn't very useful in identifying the item that needed to be removed or updated. I ultimately ended up overloading the == operator i in the item class and created a function within the list class that looped through the *listItems* array and used the overloaded Item operator to compare each item name with the input name. If a match was found, this function would return the subscript of the matching array element, which could then be used to delete or update the item.

Another challenge was resizing the array in the event that the item list exceeded the original capacity of four elements. My initial design resized the array only large enough to add the next item. This worked but proved to be an inefficient use of memory resources. I ended up making the array double it's capacity whenever an item needed to be added to a full array, which cut back on the number of times the array needed to be copied to a temporary array, deallocated, and recreated at a larger capacity.

Other deviations of my original design consisted of creating functions for blocks of code that needed to be called more than once. For example, I needed to search the array for both adding and deleting an item. Rather than duplicate the code in each function, it made more sense to create a method for searching the array that could be called in the add and delete methods. I also found the addItem method to be somewhat long, so broke the section of the code that resized the array down into its own function. This required passing some of the variables in as arguments, but helped keep the sections of code into smaller and more manageable blocks.