Tyler Forrester

CS162

Assignment 1

design documeNt

**Contents**

1 Overview 3

**Class-1: List** 6

**Class-2: Item** 9

**Class-3: Main** 11

**Class-4: InputValid** 12

# Overview

This Program grocery list program is a program that will allow users to add, update, view, and delete items in a list. The entries will consist of a Name, a numbered Amount, a Type of Amount, and a Price. These entries will adhere to certain input restrictions, Name and a Type of Amount will be limited to alphanumeric characters and punctuation. This string validation allows for unusual server characters to be disregarded while still providing the end-user with flexibility to write descriptive names. The Amount will always be a positive whole number and Price will always be a positive number which ends in two decimal places. Type of amount could always be renamed to a different unit of measurement, which allows amount to only be an int. If end-user needs a fractional number, the end-user can vary the type of amount to a new size. Prices in standard U.S. format are only to the second decimal place.

The program will consist of 3 classes and a main file. These three classes will be called “InputValid”, “List”, “Item”. List and Item classes are both specified in the assignment.

The InputValid class was my addition for portability to additional assignments. It validates chars, strings, prices, and positive integers in the aforementioned format.

The Item class will have store individual items on the lists values. Its private data members include itemName, unit, numberToBuy and price. Its public data members include an override constructor for item to aid in testing, set and get functions for the mentioned variables, and an override to the equality operator to test for duplicate items. The equality operator tests if the itemName is equal to another Item Objects name. This seemed be an efficient way to keep duplicates from occurring, since the unique identifier to the Item Object aside from its location on the array is the itemName.

The List class will hold functions that modify Items, and return a printout of the list on request. It holds the bulk of the logic. The list is designed to be a dynamic array which doubles in size after it fills with items. It is initialized to four because that is the size which is requested in the assignment document. I use a variable integer size to keep track of the amount of heap memory allocated for the dynamic array. Again requested in the assignment document. There are three main actions in the list with one easter egg. These actions are to add an Item to the List, to view items on the list, and to delete an item. The easter egg is the updating of quantity of items. It does not exist on the main list, but will displayed if the input function discovers a duplicate entry. Since the program does not explicitly state that there is an update function, but one is required to handle duplicates, I chose to leave it from the main menu.

The add Item functionality is implemented through an input function which flows either to the duplicate function or the addItem function. The user is asked to input the name of the item, which he would like to add to the list. This is then checked for duplication using the override equality operator in the Item class. I chose use the override equality operator because it was specified in the Week 2 overview section as a requirement to this program. If the item name is duplicate, then the program prompts a user to update the number of that item he wants to buy. This seems like an appropriate tradeoff between over engineering the assignment and providing duplication support. The end – user is allowed to the change the quantity which I assume is why he is trying to add another of the same item to the list. However, he’s not allowed to change either price or unit of sale. This can only be accomplished through the deleting the item and reading it to the list. It seems appropriate to keep the program within the scope of the assignment. If the duplicate function returns false, then the end-user is prompted to continue adding the other variables stored in the list class. This is an important step specified in the assignment. Again the input is validated through the Input Valid Object. I was hoping to be able to extend the input validation to future assignments. Once the information is validated and completely entered, the addItem function is called. I chose to force the End-User to enter all the information in the list. This makes readability of the list clearer. The addItem function then calls the function checklist, this checks to see if the current dynamic array is full. If it is full then it calls another function doubleList. This function creates a new dynamic array with double the space and the stores the information from the current list in it. The current list is then delete and its pointer is redirected toward the double array. This seems to be most efficient way to double a dynamic array. However, I am open to process improvements. Now either the new array or the old one (depending on flow) adds the item to its list and the fucntion iterates the arrayEnd data member, which keeps track of the current end of the array in the stack. This seemed like an important design choice because not knowing where the end of the array was in the stack memory was causing issues when I tried to read my array and find items in the array.

The next functional process is to display the list. The dynamic list array is iterated over. Each one of items currently loaded in the list is printed out. The seemed sensible due to the idea that it was a “read” function. Seeing all member items of the list and all data members of items was the only option. A totalCost function is called at the bottom. It multiplies the number of units by the price and then sums it. This returns the totalCost of the grocery list. Again this seems like a standard computing principle and allows for the reuse of the totalCost function outside of the Grocery List.

Next we move to deleting an item from the list. I made the choice to allow the user to input the number of the item on the list to be deleted. This seem like a sensible and relatively straightforward way to find items in the dynamic array. The number of the item of list was always going to be one higher than the position in the array. I validate that the position of the item was on the list and that an item exists. If the position is not on the list, then the end-user is prompted for a new number. If an item does not exist, then the end user is taken back to the main menu. This take care of all three delete use cases. User enters invalid input, User tries to delete from a list with no items and a User successfully deletes an item.

The main method of the project uses a basic menu setup. I setup a displayMenu() function and a getChoice() function both of which are from our C++ book and continueOn() function which pauses the program after the exit of each switch case. This gives my program basic functionality. A user is allowed to select from a list of “Add an Item”, “Display List”, “Delete an Item”, and “Exit the Program.”. The choices are labeled 1-4 with 4 being exit. I use my input validation to validate that only positive digits are return as choices. The getChoice function also does range checking. I wanted to make my program as foolproof as possible. I initialize a List object and Input Valid object in my main. This allows me to access the appropriate methods and validate their inputs. Items are all stored in the list class so there is no need to initialize in main. I tried to keep main as brief and readable as possible. Each menu item calls the appropriate function in list to start the use the case. All item input is done in the list class. This seems to adhere closest to object oriented principles. Please classes below for more complete descriptions of individual methods and variables.

Classes

### **Class-1: List**

| Function Names | Description | Variables/ Functions Used | Parameters | Output | Position in Use Case |
| --- | --- | --- | --- | --- | --- |
| **List Constructor** | Creates a dynamic array of Items of size 4 | Item[] itemList, int arrayEnd, size | Void | none | Start of Program |
| **Input** | Prompts user for information regarding an item, checks name of item against list to prevent duplication. Also formats price to the two decimal places. | Item object and associate get and set methods, duplicate(Item, InputValid), valid.validateString(), valid.validatePrice(), valid. validateInt(), cout.precision(2), fixed | InputValid valid |  | UC-AddItem->Input->Duplicate or AddItem-> main |
| **Duplicate** | Checks list for the same item name as passed in as variable. If true prompts user to modify numberToBuy | arrayEnd, newItem, valid, itemList[].setNumberToBuy(), itemList[i].getNumberToBuy() , valid.validateInt(), Item overloaded equality operator. | InputValid valid and Item newItem | bool | UC-AddItem->Input->Duplicate true -> setNumbertoBuy -> main. If Duplicate false -> input |
| **addItem** | adds an Item to Item[] itemList | Item newItem; Item[] itemList, arrayEnd, checklist() | Item newItem | none | UC – AddItem -> input -> duplicate is false -> addItem -> checklist() true -> doubleList-> add item; checklist() is false -> add item -> main |
| **checkList** | Checks itemList for items, if full calls doubleList | arrayEnd size, doubleLst)( | Void | none | UC – AddItem -> input -> duplicate is false -> addItem -> checklist() array full-> doubleList-> add item; checklist() array not full -> add item -> main |
| **doubleList** | Doubles List Size  Copies contents of itemList to another array then double the size of itemList. Deletes item list[] then points ItemList \* to the copied array. | Item[] itemList , size, OldSize  Item \* itemList, Item \* array2 | Void | none | UC – AddItem -> input -> duplicate is false -> addItem -> checklist() array full-> doubleList-> add item -> main |
| **displayList** | Displays List in Console: Prints out | arrayEnd  Item[] itemList  totalCost() | None | Print list to console. Void return. | 1.UC – Display List main->displayList(), subfunction totalCost -main  2. UC – deleteItem -> displayList ->checkDel() -> deleteItem() -> displayList -> main |
| **totalCost** | Multiplies number to buy \* price and then adds to a toal | Int total, int arrayEnd, itemList[].getprice() and itemList[].getNumberToBuy | Void | Total Cost of Shopping Cart | displayList() subfunction |
| **checkDel** | Checks if list exits, if it does prompts user to select an item to delete. If not prints “There are no items to delete.” | displayList(), deleteItem(), valid.validateInt() – 1, InputValid valid | InputValid valid | Print test to console | UC – deleteItem -> checkDel = true -> deleteItem() = true -> main or deleteItme() = false -> loop  checkDel()= false -> main |
| **deleteItem** | Deletes Item from itemList, changes size of Array to adjust | Integer position, Item[] itemList, arrayEnd | position | The Item in position “x” has been deleted.  Bool return true or false. | UC – deleteItem -> checkDel = true -> deleteItem() = true -> main or deleteItme() = false -> loop to checkDel() |
|  |  |  |  |  |  |
| **Deconstructor** | Deletes itemList on exit. Never Used |  |  |  |  |

| File Variable Names | Description |
| --- | --- |
| **Int Size** | Marks the current end of memory allocated to the dynamic array |
| **Int arrayEnd** | Marks the current end of items in the dynamic array |
| **Item \*itemList** | Pointer to dynamic array that stores Items |

### **Class-2: Item**

| Function Names | Description | Variables Used | Parameters | Output |
| --- | --- | --- | --- | --- |
| **Item Constructor** | Default Constructor Initializes Item Class | None | None | none |
| **Item Constructor** | Override Constructor Initializes Item Class | itemName, unit, numberToBuy, Price. | itemName, unit, numberToBuy, Price. |  |
| **setItemName** | Sets Item Name | String itemName | String from user input | none |
| **setUnit** | Sets Unit | String Unit | String from user input | none |
| **setNumberToBuy** | Sets numberToBuy | Double numberToBuy | double from user input | none |
| **setPrice** | Sets price | double price | Double from user input | none |
| **getItemName** | Gets Item Name | String itemName |  | String itemName |
| **getUnit** | Gets Unit | String Unit |  | String Unit |
| **getNumberToBuy** | Gets numberToBuy | Double numberToBuy |  | Double numberToBuy |
| **getPrice** | getsPrice | double price |  | double price |
| **Operator==** | Overrides default “==” to test if names are the same. | Item x.getName, itemName | Item |  |

| File Variable Names | Description |
| --- | --- |
| **String ItemName** | Stores the name of an item |
| **String Unit** | Stores the unit of sale |
| **Int numberToBuy** | Stores the amount of units to purchase |
| **Double price** | Stores the price per unit of sale. |

### **Class-3: Main**

| Function Names | Description | Variables/ Functions Used | Parameters | Output | Position in Use Case |
| --- | --- | --- | --- | --- | --- |
| **displayMenu** | Displays system Menu Example 6-14 in Gaddis |  |  | A printout list of menu choices | Start of program -> upon exit of all choices except exit program |
| **getChoice** | Allows user to select menu item  Example 6-14 in Gaddis | Int choice | InputValid Object | choice | displayMenu-> getChoice -> UC Add Item, UC Delete Item , UC Display List. Depends on number |
| **continueOn** | Stops input until c is entered. | Char entry, valid.validateChar(); | InputValid Object | “Printout to screen asking for c” | After each UC. |

| File Variable Names | Description |
| --- | --- |
| **Int Choice** | The number of the choice of the user |
| **List List** | Object that allows list functions to be called |
| **InputValid valid** | Object allows InputValid class functions to be called. |
|  |  |

### **Class-4: InputValid**

| Function Names | Description | Variables Used | Parameters | Output |
| --- | --- | --- | --- | --- |
| **InputValid()** | Constructor of InputValidation initializes input to “” | String input | none |  |
| **validateInt()** | Validates Positive Int by testing input for the digits. | String input  Int myNumber  Bool isNotNumber  StringStream myStream | none | Positive integer |
| **validateDouble()** | Currently tests by taking an a double and comparing via stringstream to a string. Probably not working as intende.d | String Input  Double myNumber  Stringstream myStream | none | Double |
| **validatePrice()** | Tests for a Price in with the ending .DD. Where D is a digit and . is in the third to last position in the string. | Double myNumber  Bool isNotPrice  String input  Stringstream myStream | None | Double in the form “.XX” |
| **validateString()** | Test string for “odd” characters such as backspace entered in console. | Bool isNotPrice  String input  Stringstream myStream | none | String with either alphabetic, numeric, punctuation or space characters |
| **validateChar()** | Currently unused and doesn’t have return statement. | Char myChar  input | none | none |

| File Variable Names | Description |
| --- | --- |
| **String Input** | Takes console input |