CS260 Lab: Shelf Life

Instructions

Your RPG game includes all sorts of valuable items, including different types of potions. Normally players find potions on quests or in random places in the game. However, some players just want to buy potions at a shop. In order to support this you have decided to implement an apothecary shop. The realm will support any number of apothecary shops but each one will be implemented with the same class. These shops are "self-service", non-manned kiosks throughout the realm. Players can walk up to a kiosk and either buy a potion (if there are any there) or submit a potion request. Every once in a while the magician comes by the kiosk and services the potion requests. He (or she) makes the potions from the requests and puts them all on the shelf for future players to buy. (Think of it as Redbox for potions).

Receiving orders

Apothecaries don't get to make any potion they want. Instead, they receive orders for potions. Each order is really a request to make a particular type of potion. Requests are processed in the order that they are recevied. Note, though, that just taking orders is not the same as making potions (see next section).

Making potions

From time to time the magician shows up at a kiosk and makes potions for all of the requests. As each potion is made, it is put on a shelf, at the front. Since it's put at the front of the shelf it pushes all of the other potions back. This means that the most recently made potion is at the front of the shelf.

Buying potions

Players that want to buy potions go to the shop and buy a potion. They cannot select which potion they get, it is just the potion at the front of the shelf.

Capacity limits

Finally, each shop has a size limit for the number of orders they can take (the size of the queue) and the number of potions that they can store on their shelf (the size of the stack). There is no restriction on what these limits are (e.g., one shop may be able to handle only 5 orders while another can store 5000 orders) but once a shop is created, the limits for that shop can't change.

Implementation Requirements

Before understanding the implementation requirements you should look at a couple of files: main.h and main.cpp. The header file contains a simple enumerated type of all of the different potion types that the program supports. It also contains the function prototype for a utility function for priting out the potion type as a string. The main.cpp file contains test code to "run" a shop. This code creates an

Apothecary object (with the two limits), issues various order requests, tells the shop to make potions a few times, and buys some potions. If you look at this file you will see the different classes, constructors, and methods that are needed by your program.

The Potion class

The Potion class contains the details about a potion. Your potion class should have a private data member to store the potion type for a created potion. The Potion class needs to have a GetType() method that returns the potion type for a given potion. You also need a default value constructor. You are free to add any other methods that you need.

The Apothecary class

The Apothecary class is where the majority of the work is done. As explained in the overview, each apothecary has different limitations on the number of orders and number of made potions they can store. To support this, the Apothecary class needs to have a constructor that takes in those two values. Next, this class needs to contain a queue for taking orders (order queue) and a stack for holding the completed potions (shelf stack).

In order to take orders, make potions, and allow users to buy potions, the class needs to have these three methods:

```
bool OrderPotion(PotionType potion);
int MakePotions();
bool BuyPotion(Potion& potion);
```

The OrderPotion() method takes in a potion type and adds a request for this type of potion to the order queue in the shop. This method needs to make sure that the order queue length limit isn't exceeded. If there is not room to add the request to the queue, the method needs to print a message (see the sample output for the message). The method should return true if the order was successfully added to the queue, false otherwise.

The MakePotions() method reads orders off of the request queue and creates the requested potion. As each potion is made it is put onto the shelf stack. The method returns and integer which is the number of potions made. The method will try to process all of the orders on the request queue. However, MakePotions() needs to be aware of the shelf limit and not make any potions that would put it over the limit. This means that there might be cases where not all of the orders can be processed. In that case there will still be orders left on the queue. Appropriate messages should be printed when potions are made or orders can't be fulfilled. See the sample output.

The BuyPotion() method is the method that is used to take one potion off of the shelf (popping off of the stack). The potion is passed back through a reference parameter. The boolean return value for BuyPotion() is true if a potion was bought and false otherwise.

Other requirements

The main.cpp and main.h files are attached to this assignment. You must use this exact main.cpp file without any modifications for your program. In order for this to work correctly you will need to make sure to use correct file name "Apothecary.h". You will also need to make sure to name your classes and methods as described above.

You need to implement your own stack and queue classes. You cannot use the STL queue and stack or any other pre-made stack or queue code.

When you run your program using this main function your program should generate the output shown in the attached output.txt file. You need to match the text exactly (including spaces and newlines).

To Submit This Assignment

First, "clean" your project by doing the following:

- execute **Clean Solution** in the **Build** menu
- close Visual Studio, and then delete the **.ncb** file from your solution's root folder

After cleaning your project, compress it into **lab2.zip**. Make sure you have your Zip application set to *preserve the folder structure* of your project, and make sure that your .zip file *will extract into a single folder*, the way that the .zip files from this course do.
Doing these things will ensure that your .zip file is as small as possible, which will make both your upload and my download quicker.
Submit the .zip file containing your cleaned project to Desire2Learn.