Store Backend Design (notes/layout etc)

Note: Sections 1 2 and 3 are all handled within this document.

So initially for design, I will use the skeleton code given to handle input and output of files with predetermined test cases that can be changed, and the .cpp file will contain instructions with how to handle the data.

The file of checks will be formatted in a way that it allows the program to run as many or as few tests as the user desires. To make a direct example, it will be more or less a file I/O version of the palindrome assignment with the way that input size accommodates the user.

An example format would be as follows:
<value> <variable changed> <expected return value/message> <type of test>

Which would be read into the program until the end of the file is reached using parallel vectors. Once read into the program, a for-loop would start going through each grouping of conditions. During this for loop, the following would happen:

- The program checks for the type of error to be checked by the user
    - Ex: Negative balance with/without a credit card
    - Ex: Testing getters/setters
    - Ex: Anything else that is specific to the program that is needed to be tested.
- After getting the type of error checked, a try-catch block is initiated around the test values, and the test values are sent to the function.
- After the function runs, either an error is thrown by the (non-pilot) program, or everything is able to be calculated.
- If an error is thrown, it will have an e.what attached to help the user debug.
- If an error is NOT thrown, the program then checks to see whether or not the returned value is equal to expected (for testing entire objects, you could use operator overloading. Note to self: make an input case for entire object data being passed, or figure out a format to TEST that data after each piece had been input individually).

More or less, this is how I could handle the driver in order to help with the debugging process. I could more or less code test cases as I'm going, or before I start (so that they're 100% in accordance with the requirements of the program) to debug efficiently as possible.

For an example, here is the Customer Class:

- Customer Class
    - Private
        - String name
        - Int customerID
        - Bool credit
    - Public
        - String getName()
        - Void setName(string name)
        - Int getID()
        - Bool getCredit()
        - Void setCredit(bool hasCredit)
        - Double getBalace()

So some obvious test cases would be:

- Try to use credit when credit is not available.
- Set the UID to one that is taken by another person, and check if it can be done.
- Check if adding balance adds enough
- Check if the name is proper
- Etc

UPDATED PORTION TO THE DESIGN

Things updated:

- Store class added
    - Has get/set name as a string
    - Can create objects of the sub-class product
    - Can create objects of the class customer
- Updates to Product Class
    - New method called addShipment that takes integer ‘shipmentQuantity’ and double ‘shipmentCost’.
    - New method called reduceInventory that takes integer purchaseQuantity.
    - New method called getPrice which will calculate the current cost with a 25% markup.
- Updates to Customer Class

        - New method called processPayment which adds an amount to balance. If amount is negative, an exception is thrown.
        - New method called processPurchase
            - If the customer has credit, always do transaction.
            - If the customer doesn't have credit and the purchase exceeds current total funds, throw an exception.
            - When a product is purchased, add the product to productsPurchased. If amount is negative throw an exception.
- New method called listProductsPurchased
    - Uses an ostream passed to the method to output all products purchased.

Based upon the given criteria on the explanation page, each of these new methods must be added to the current driver in order to be able to ensure that they all work correctly.

Part III updated design

Changes to Store Class

1. Customer& getCustomer(int customerID)
    a. Finds the customer by the customer ID, if the ID doesn't exist, throws an exception. Make sure it passes back by reference so the original customer's file can be altered.
    b. Test cases:
        i. Send a customer ID that does not exist, check for error.
        ii. Send a customer ID that cannot be changed, check for error.
2. void takeShipment(int productID, int quantity, double cost);
    a. Takes a shipment and updates the shipment quantity and cost. Throws an exception if the product ID does not exist
    b. Test cases:
        i. Send a product ID that does not exist.
        ii. Send negative quantity.
        iii. Send negative cost.
3. void makePurchase(int customerID, int productID, quantity)

a. Makes a purchase if it is allowed. If not allowed, throws an exception. Reduceinventory and processpurchase have to not throw an exception. To avoid this, check if they will throw an exception before they are called to avoid unwanted changing of data.
b. Test cases:
   i. Send more than inventory allows
   ii. Send a negative quantity
   iii. Send a product ID that doesn’t exist
   iv. Send a customer ID that doesn’t exist

4. void listProducts();
   a. Simply lists the products and their information using overloaded output.
   b. Test cases:
      i. Send a call to the function to see if it correctly outputs all the wanted data.
5. void listCustomers();
   a. Outputs the information about the customers.
   b. Test cases:
      i. Send a call to the function to see if it correctly outputs all the wanted data.
6. void addCustomer(int customerID, string customerName, bool credit);
   a. Change addCustomer. This is to send the credit value of the customers to the default constructor, which wasn’t being done previously.

Time to code!