Store Backend

Points

Points	
85	Working Classes
15	Code Review
100	**TOTAL**

Objectives

- Work with classes.
- Use UML diagram as outline for creating a class.
- Create datatypes that identify but do not handle errors (e.g. throw an exception but not catch it).
- Use incremental programming.

Submission

1. **Design**: Submit a PDF.
2. **Source Code**: Submit a zip file containing the source code (**Customer.cpp , Customer.h , Product.cpp, Product.h, Store.cpp and Store.h** files) to Mimir.
   Note: you do not have to upload the file with the main() function. However, if you do, it will not affect grading. We will use one with our test cases when we compile the classes you provide.

Program

You will create a backend for organizing data for a store. The program will span two homeworks. This week's homework will be based on this UML Diagram. You will create the classes. Rather than testing a program, the autograder will use your classes

in a program used for grading. However, to develop code you will need to create a "driver" program(s) that you can run to develop and test your code. In the driver program, you can create sequences of statements to test your classes. See the end of the document for some guidance on creating driver programs.

Specifications

Based on the UML Diagram, create the Customer, Product and Store classes.

- Include all attributes / data members as indicated in the UML Diagram.
- Implement the constructors and the methods / member functions listed below.

Design (Optional)

A lot of the design has been done for you in the UML Class Diagram provided.

1. Outline your steps for creating a driver program for testing the classes.
2. Identify the dependencies, which functions rely on other functions to work correctly. You won't want to try and debug a function that relies on another function that is not working yet.
3. Additional Components: Create test cases to ensure the class behaves correctly when created and or updated.

Separate Files

Each class should be in a separate file with its own header file. By convention, the class name is capitalized and matches the name of the source (.cpp) and header (.h) files. **Do not forget to use header guards.**

const

Think about which member functions should not *change* the objects. Those member function declarations should be modified and marked as `const` for all classes. This will be important when creating overloaded output operators.

Product Class

- Datatypes
    - id, inventory, and numSold should be ints
    - name and description should be C++ strings
    - totalPaid should be double
- `Product(int productID, std::string productName);`
    - *Name cannot be an empty string, i.e. “”. If it is empty, then throw a runtime_error exception.*
- `int getID();`
- `std::string getName();`
- `void setName(std::string productName);`
    - *Name cannot be an empty string, i.e. “”. If it is empty, then throw a runtime_error exception.*
- `std::string getDescription();`
- `void setDescription(std::string description);`
- `int getNumberSold();`
- `double getTotalPaid();`

  Returns the total of all shipment costs over time. See addShipment() function.
- `int getInventoryCount();`
- `void addShipment(int shipmentQuantity, double shipmentCost);`

  Add shipmentQuantity to inventory and increase totalPaid by shipmentCost. Do not replace totalPaid, just increase its value. If you get a negative shipmentQuantity or a negative shipmentCost, throw a runtime_error exception.

- `void reduceInventory(int purchaseQuantity);`

  If there is not enough inventory, throw a runtime_error exception. Otherwise, decrease inventory by purchaseQuantity and increase numSold by purchaseQuantity. If the purchaseQuantity is negative, throw a runtime_error exception.
- `double getPrice();`

  This function will calculate the current price based on the average cost per item over time plus a 25% markup.
  - *price = (totalPaid / (inventory + numSold)) * 1.25.*
    - ***Warning: avoid integer division!***
  - *If you can't calculate a price, throw a runtime_error exception.*

Customer Class

- Datatypes
  - id should be int
  - name should be a C++ string
  - credit should be a boolean
  - balance should be a double
  - productsPurchased should be a vector of type Product

- `Customer( int customerID, std::string name, bool credit=false);`
    - *Credit means that the customer's balance is allowed to become negative. If they have credit and they make a purchase with insufficient funds in their balance, the purchase is allowed. Otherwise, they are limited to purchases that can be paid by their balance.*
    - *Name cannot be an empty string, i.e. “”. If it is empty, then throw a runtime_error exception.*
    - *If an argument is not provided for the credit parameter, set it false by default.*
- `int getID();`
- `std::string getName();`
- `void setName(std::string name);`
    - *Name cannot be an empty string, i.e. “”. If it is empty, then throw a runtime_error exception.*
- `bool getCredit();`
- `void setCredit(bool hasCredit);`
- `double getBalance();`
- `void processPayment(double amount);`

  Add amount to balance. If amount is negative, throw a runtime_error exception.

- `void processPurchase(double amount, Product product);`
    - *If the customer has credit: subtract amount from balance. Recall that balance can be negative if credit is true.*
    - *If the customer does not have credit: if the balance is greater than or equal to the amount then subtract amount from balance. Otherwise throw a runtime_error exception. Recall, balance is not allowed to be negative if credit is false.*
    - *If the purchase occurs, then add product to productsPurchased if it is not already there. (Note: this will be a copy so its values will not be updated if the original Product object is updated.)*
    - *If amount is negative, throw a runtime_error exception.*
- `std::string getProductsPurchased();`

  Put the product name and product id into a string. If name is "Coozie" and id is 32498, then the string will be (note there are end of line markers at the end of each line):

  Product Name: Coozie

  Product ID: 32498

Store Class

If invalid customerIDs or productIDs are passed into any function, it should throw an exception. Note this could be by calling getProduct() or getCustomer() with the invalid identifiers. If invalid data would cause an exception in the Product or Customer class, it should throw those exceptions as well. This could be by calling the functions in Product or Customer class that would in turn throw exceptions.

- Datatypes
    - name should be a C++ string
    - products should be a vector of type Product
    - customers should be a vector of type Customer
- `Store();`

- `Store(string name);`
- `string getName();`
- `void setName(string name);`
- `void addProduct(int productID, string productName);`

  Create a new Product and add it to products. If this productID already belongs to another product, throw an exception.
- `Product& getProduct(int productID);`
  - *Find the matching product and return it. Be sure that the product can be modified so that the changes persist in the store's vector. If the product does not exist throw an exception.*
- `void addCustomer(int customerID, string customerName, bool credit=false);`

  Create a new Customer and add it to customers. If this customerID already belongs to another customer, throw an exception. If an argument is not provided for the credit parameter, set it false by default.
- `Customer& getCustomer(int customerID);`
  - *Find the matching customer and return it. Be sure that the customer can be modified so that the changes persist in the store's vector. If the customer does not exist throw an exception.*

- `void takeShipment(int productID, int quantity, double cost);`
  - *Find matching Product. If product is not in list of products throw an exception. Otherwise, update product with the shipment quantity and cost.*
- `void takePayment(int customerID, double amount);`

    - *Find matching customer and process the payment. If the customer does not exist, you should throw an exception or let an exception propagate that is thrown when you call another function to get the customer.*
- `void sellProduct(int customerID, int productID, int quantity);`
    - *Make the sale if it is allowed, otherwise throw an exception or let an exception propagate. In this context allowed means that the product's reduceInventory() and the customer's processPurchase() functions would not throw an exception and the product and customer must exist.*
    - *Note the difference between **int quantity** (input parameter to sellProduct) and **double amount** (input parameter to processPurchase). **quantity** refers to the number of items that will be sold, while **amount** refers to the total price that the customer will pay for the purchase. Therefore in order to call processPurchase, you will need to calculate **amount** by using **quantity** and the price of the product that is being sold.*
    - ***Warning: Do not change the product or customer if both cannot be done successfully.***
- `string listProducts();`
    - *Output information about each product. (Use overloaded output operator for Product.)*
    - *Should not crash if price can't be calculated. Output "Price: Unavailable" in that case.*
- `string listCustomers();`
    - *Output information about each customer. (Use overloaded output operator for Customer.)*

Output Operators <<

Create overloaded output operators for the Product and Customer classes. Recall, these are helpers for those classes. So the function declaration should go in the header (.h) file for the corresponding class, but outside of the class definition. The function definition should go in the cpp file of the corresponding class. You should **avoid** using the friend approach for creating the overloaded output operator. Provide and endl at the end of each line.

Product output example

Product Name: Coozie
Product ID: 32498
Description: A great way to keep a canned beverage cold.
Inventory: 83
Number Sold: 107
Total Paid: 2850
Price: 18.75

Customer output example

Customer Name: Miss Reveille
Customer ID: 2198123
Has Credit: true
Balance: -228.33
Products purchased --

Product Name: Coozie
Product ID: 32498

Product Name: 12th Man Towel
Product ID: 121212

Driver Program Guidance

Your driver program will help you test and debug your objects as you create them. A nice thing about using a driver program, is you do not have to validate inputs since you can ensure you are only entering the values you want to test. In many cases you can just use a literal value instead of taking in input. Remember, this program is only for testing, so you do not have to create a rock solid program for testing. However, member function may need to do validation as indicated. However, the response to that will generally be to throw an exception to let the user of the class decide how to deal with the error.

Initially, you will want to create and test the Product class since it does not rely on another class. So testing means creating and being able to see values in in the class and trying both valid and invalid inputs. We will show you some steps in how the testing evolves for the Product class. Also as you code you should be constantly compiling every time you type fresh code to catch problems as soon as possible. It is much easier to catch errors as you go rather than write tons of code and then go back and debug. Taking that approach can double, triple, or even quadruple your development time.

1. Initially we can use the UML diagram to create the class definition.
2. In this first phase we can go ahead and create the corresponding .cpp file and fill it with stubbs.
3. Finally, we can create a driver program that creates an instance of the class. Now we can compile and fix things until we compile successfully. You might do what I did initially and use “Product p;” in the driver, but this does not compile. If you provide any other constructor, it will not create a default constructor which is why it would not compile. I just passed in values for the expected parameters

for the constructor so it would compile. I didn't run it because I don't have any output that I can look at.

   - Code (v0)

4. Since we had to use a parameterized constructor to create the object let's go ahead and do the constructor. However, there is no way to check the values, so I'll go ahead and implement the getID() and getName() functions so we can output those values. Then we can update the driver to check if the constructor is working by looking at the values via calling the functions on the objects. We can also create multiple objects with different values to ensure each object has different values.
   - Code (v1)
5. We're not quite done testing the constructor. If the string is empty it should throw an exception, and I haven't added that to the code yet. I'll also need to add a try/catch block to the driver to catch the exception when it occurs.
   - Code (v2)
6. Now we might want to implement the setName function. We can use the getName function to check if everything is good. We can keep most of our existing code since it shows the values of the objects before we try to change them. However we'll comment out the part that tests an empty string with the constructor since an exception won't let us test code past that point. (We won't delete in case we need to go back and use the test code again. Now we can call setName on objects with valid and invalid values and check that both work.
   - Code (v3)
7. You would work similarly, but for this example I'm going to skip to the addShipment function to illustrate checking a slightly more complex function. I'm not going to implement getPrice yet. It really depends on addShipment working correctly to set the values used in getPrice. So let's work on addShipment. The first parameter is the number of items received so that will increase the inventory. The second is the amount paid for the shipment so that will increate to totalPaid. Let's only address this part for now. We'll

only use “correct values” here and make sure that works. Then in the next step we’ll check the values passed in and throw exceptions if needed. We’ll use the getTotalPaid and getInventoryCount functions to look at the object and see if function works as expected. You’ll see some commented out cout statements in addShipment. I had to debug to figure out why inventory was not being modified. It was, but I had not implemented the getInventoryCount function and those helped me focus and eventually resolve the issue. I also hand calculated the results so I could compare with the output and ensure things are going well.

    - Code (v4)

8. The last thing I’ll model is dealing with the exceptions for the addShipment function. They really make sense, the number of items must be positive. Plus, we can only pay positive amounts. I’ll check each separately, but in what you see in code the first test of an exception will be highlighted out so I can then test the second one. Regardless, we will check the values at the beginning of the function. We don’t want to change the state of the object and then find out there was a problem. The bad changes might hang around and mess up the values later.
    - Code (v5)

Note: as things get more complicated it is helpful to label what you are outputting. If you are methodical, develop your code incrementally, and compile and test frequently, you will produce code much faster than when you try to code everything first and then go back and fix the problems.