Store Backend - Part 2

Points

Points	
10	Design
15	Code Review
75	Working Classes
100	**TOTAL**

Objectives

- Use Return by Reference.
- Use reuse by creating and using private helper functions.
- Work with a collection of a user-defined datatype.
- Deal with keeping data consistent if exceptions are encountered.
- Work with multiple classes.

Submission

1. **Design**: Submit the PDF to eCampus.
2. **Source Code**: Submit the source code (**Customer.cpp , Customer.h , Product.cpp, Product.h, Store.cpp and Store.h** files) to Vocareum
   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. The driver program will be similar to what you did in part 1.

Specifications

Based on the UML Diagram, update the Store class.

- Include all attributes / data members as indicated in the UML Diagram.
- Implement the constructors and the methods / member functions listed below.
- Also, in the Product class, be sure to update the return type of `Product::setName` from `std::string` to `void`.

Design

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. Create test cases to ensure the class behaves correctly when created and or updated. Think about how you can use and set up the different classes to test the methods in your store.

Separate Files

Each class should be in a separate file with its own headerfile. By convention, the class name 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.

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.

- `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 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.***
- `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 listProducts();`
    - *Output information about each product. (Use overloaded output operator for Product.)*
- `void listCustomers();`
    - *Output information about each customer. (Use overloaded output operator for Customer.)*