Homework 3

See Canvas for deadlines

Total points - 150 points

Objectives

- Develop familiarity with the concept of “static”
- Implement and use the Factory design pattern
- Design and implement a multi-object program, dealing with the intricacies of objects communicating with one another

Credit

- **Deliverable 1:** hw3_design.pdf
- **Deliverable 2:** Makefile, outline of all header & implementation files, main.cpp, csv files
- **Deliverable 3:** Makefile, complete header & implementation files, main.cpp, output.txt, csv files, peer evaluation, schedule interview

Instructions

You will write a program that simulates eBay in CLI program. We'll call ours **BidToBuy**. You have to use inheritance to define the different types of users and products in this application. The idea is to have buyers bid on products posted by the sellers and the highest bidder is determined once the bid has been closed. You have the creative freedom to design the solution for this assignment. Feel free to post as many clarifying questions as you need to about the homework expectations.

You will work in a pair to complete this assignment. You are welcome to pick your partner and are required to collaborate with them on the homework on GitHub. Ensure that your repository is kept private and add Chaitanya Bhangale - '**chaitanyab2311**' as a collaborator to your repository. If your repository is found to be public, the pair will receive 0 for the homework.

For each deliverable, make sure to identify both students in the pair in the design document, main.cpp and on Gradescope.

Part1 (Design Document & Getting Started) - 40 points

Design Document (Deliverable 1):

Your document should include clear indications of your class design and the interaction between the classes. You do not need to follow formal diagramming language, but your diagram needs to be legible and communicate the information described above.

Your submission for this deliverable must include:

1. A pdf document that details:
   a. the interfaces for the classes you choose to use - methods they will have, the methods they will call/use
   b. the data members that your classes will have
   c. pseudocode for your main function
2. 2 csv files with initialization data - you can decide the structure and fields in these files.
   a. User information: This would include information about sellers and buyers.
   b. Product/Bid information: Consider this to be historical data. Plan to have product information, buyer information and seller information at the least. This file will help with features where the seller can look at historical data to decide a price for their product. You do not have to normalize this data.

Program Flow:-

1. Initialize application from the csv files.
2. First, the user should choose a role, that of a Buyer or Seller

3. If Seller, they should be able to:
   a. post a product for sale - should also have the option to view prices of similar products that were historically sold.
   b. read and respond to messages from buyers,
   c. check their account balance,
   d. update their user information,
   e. get an overview of their products that have been sold or are yet to be sold
   f. open/close bids on their products
4. If Buyer, they should be able to:
   a. view products for sale,
   b. place a bid on a product of their choice
   c. read, respond, send to new messages from sellers,
   d. check their account balance,
   e. update their user information,
   f. get an overview of the bids they have placed
   g. view the history of products they have bought
5. The driver class should be Singleton and should be able to track all users, products and bids that are currently active and the ones that have been completed. This class will control the application interface and the flow of the program. This class will also write the new bids, that have been closed, to the product/bid information csv as these will be considered historical bids.

Bidding

Each product that is placed for sale will be open to accepting bids until the seller closes the bid. The buyers may place a bid only products are active for bidding. A buyer should not be able to place a bid that exceeds their account balance. You may assume they have a fictional balance amount in their account at the start of the run. A buyer may place a bid on more than one product. Once a bid has been closed, the buyer with the highest bid will own the product. The buyers should be notified through messages if they won or lost the bid. Once the selling process is complete, the account balances of the buyer and the seller are updated. At the time of sale, if the buyer refuses to buy the product or if the account balance of the buyer is less than the bid, the sale to that buyer is cancelled and the bid is won by the buyer with next best bid. The product is then taken off the market.

The seller should be able to re-open a new bid for an unsold product whose bid was previously closed but should not be able to resell a previously sold product.

Hint: use maps to track information for this purpose.

Messaging between Buyer and Seller

A buyer may message a seller if they win the bid for the seller's product. Similarly, a seller may also message only a buyer who has won a bid for their product. Both parties can share messages. The users should be able to respond to the messages. You may use any underlying data structure to do this. Using one that is time and storage efficient, will grant you extra credit points.

You can display the messages one of 2 ways:

1. Iteratively display all the messages and have the user the option to respond to each message, or
2. You can display all messages and allow the user to choose which messages to reply to.

Inheritance for Users:

You can use this as a starter for setting up inheritance for users. You will have to fill in the missing fields and methods to meet the requirements of the project. Follow similar guidelines for products.

```
class User{
public:
  // getter and setters for all fields
  // add applicable methods and use dynamic dispatch as needed
private:
  // Add fields as applicable
};

class Seller: public User{
public:
  void addProductForSale();

private:
  // add more fields as applicable
};

class Buyer: public User{
public:
  void addBidToProduct();
```

```
private:
  // add more fields as applicable
};
```

Updating user information

The users can update their name, phone number, address.

Products

You are required to implement at least 5 concrete product category classes each having at least one level of inheritance. All products should have a base price at which the bid will start. The seller must indicate the quality of the product when posting one for sale. The quality can be one of these: New, Used-VeryGood, Used-good, Used-okay. Depending on the product, you have the freedom to decide additional behavior and data points for the classes. Plan to have detailed classes. Well-designed and detailed inheritance design will contribute to the extra credit. Here is a code snippet that may help understand the product inheritance expectation:

```
class Product{
...
};
class Product1: public Product{
..
};
class Product 2: public Product{
...
};
...
```

Getting Started (Deliverable 2):

Reminder: You may find incorporating std::map into your program, to track bids, very useful

1. Use factory design pattern when seller is adding a product for sale.
   a. You are welcome to use other design patterns as applicable in different parts of the program.
2. Write an outline for the user and product classes.

3. Write a main.cpp that allows you to instantiate the driver class and build a map of dummy bids

Using Factory Design Pattern

You need to use the factory design pattern when the seller is trying to instantiate a Product. Below is an example of a factory method you could write. Using enums will make it more robust and readable than using integers as choices in the `productFactory()`.

```
enum class ProductCategory {Vehicle, Furniture, Books,....}; // used to define all the category of products sellers can sell on this app. You may add as many categories you like. There should be at least enough of them to cover all the classes you created as part of the products inheritance hierarchy.

Product* productFactory(ProductCategory pc){
  switch(pc){
    case ProductCategory::Vehicle:
      return new Vehicle(...);
    case ProductCategory::Furniture:
      return new Furniture(...);
      .
      .
      .
      default:
        return new Product(....);
  }
}
```

Part2 (Deliverable 3) - 65 points

Implement the rest of your program. Fill out the peer evaluation form linked on the Canvas item. Peer score contributes 10 points to your grade for this homework. **Failure to fill the form by the deadline will lead to a 0 for your partner.**

Schedule an appointment for interview grading. More details below in the document.

Note: Make sure to use the same design and headers among partners to ensure smooth integration of the project.

Comments and style (15 points):

Your files and functions should have comments. We should know how to run your program and how to use your functions from your comments.

Your variables should have meaningful names. Your code should be easily readable. If you have complex sections of code, you should use inline comments to clarify.

You should follow the conventions set out in our Concise Style Guide, posted on the course github.

Interview Grading (15 points)

Each pair will schedule an interview with Chaitanya in the week of March 14. Slots will be available on Canvas by March 7th. It is MANDATORY to attend the interview grading.

Extra Credit (15 points):

Variable based on quality of design, efficiency of implementation using appropriate object design and suitable data structures for reduced time of execution.

Resources that may help you with this HW:

Maps: https://www.cplusplus.com/reference/map/map/map/
Pairs: https://www.cplusplus.com/reference/utility/pair/
Sets: https://www.cplusplus.com/reference/set/set/
Inheritance: https://www.programiz.com/cpp-programming/inheritance
Static: https://www.tutorialspoint.com/static-keyword-in-cplusplus
Design Patterns: https://refactoring.guru/design-patterns