Practical project: Traveling merchant

420-D03-SU
Introduction to object-oriented programming
AEC Programming, Networks and Security (LEA.5F)
AEC Programming and Web Technologies (LEA.5G)
AEC Video Game Programming (LEA.CU)

Evaluation weight:	25% of final grade
Due date:	Morning of the final exam

Submission specifications

- Project name: MerchantProject + name 1 + name 2 [+ name 3]
- Your submission must Include all of your source code (**.cpp** and **.h** files)
- Important: Double check that the **.cpp** and **.h** files actually contain your code, all of your code, and the most up-to-date version of your code
- Your submission should include the Visual Studio project files (**.vcxproj**…) files
- Your submission should include a documentation text file (a readme file), describing the application's features. In particular, describe the additional features your team designed and developed, as well as any other relevant information, including any remaining bugs or problems with your code
- Important: Delete any hidden **.vs** folder
- Delete any **Debug** or **Release** folders
- Finally, compress your entire Visual Studio project folder into a **.zip** file
- Submit this **.zip** file in the correct directory on the Dropbox drive

Description

Create an application that simulates a merchant or salesperson, controlled by the user. The merchant is traveling from one city to the next, selling and buying products on the black market. Sometimes, when traveling between two cities, the merchant is caught by the authorities, who apply penalties to the merchant.

You could think of it as a game where you play as the merchant. The goal would be to avoid losing all your money, to keep business flowing as long as you can, and to make as much profit as you can, all while hoping to avoid capture and penalties.

Requirements

- Your application should support 5 types of products
- All products, of all types, must have a name
- All products, of all types, must have a price

- Using a console menu, the user can view all of the following information:

Merchant inventory:	A list of all of the products the merchant owns now
Merchant money:	How much money the merchant has now
City name:	The name of the city the merchant is in now
City inventory:	A list of all of the products this city owns now
City selling prices:	The selling price for each product type when sold in this city
City buying prices:	The buying price for each product type when bought in this city

  (One bullet point does not necessarily correspond to one menu option. Group them together where doing so makes sense.)

- Using a console menu, the user can perform the following actions:
  - Sell product
    - The user chooses a single product to sell from their inventory
    - The merchant sells the product, at its selling price within the city
    - A console message shows the action and the merchant's new state
    - Or, if the action is invalid, show a clear, user-friendly error message to the user
  - Buy product
    - The user chooses a single product to buy from the city's inventory
    - The merchant buys the product, at its buying price within the city
    - A console message shows the action and the merchant's new state
    - Or, if the action is invalid, show a clear, user-friendly error message to the user
  - Travel to a new city
    - The user chooses a city for the merchant to travel to
    - The merchant travels to that city
    - Every time the merchant travels, there is a percentage probability that the authorities catch the merchant
    - The more products the merchant has in their inventory when traveling, the more likely they are to be caught
    - If caught:
      - The merchant is moved to a random city
      - Out of all of the merchant's products, all those products of exactly one particular type are removed (this one particular type can be chosen randomly)
      - The merchant pays a fine of $100, which increases by $50 every subsequent time they are caught
        (Adjust these values to more reasonable amounts if appropriate)
      - If the merchant does not have enough money to pay the fine when caught, then the user loses the game
    - A console message shows each action/change and each new state

Requirements (continued)

- When the merchant arrives in any city:
    - The console shows the city name
    - The console shows the city's inventory
    - The console shows the selling prices in the city
    - The console shows the buying prices in the city

- Selling and buying
    - Each city has a different set of <u>selling prices</u>
      These are the prices that apply when the <u>merchant is selling products</u> to this city
    - The merchant can only sell products in their inventory

    - Each city has a different set of <u>buying prices</u>
      These are the prices that apply when the <u>merchant is buying products</u> from this city
    - The merchant can only buy products in the city's inventory

    - When a new city object is created, it should initialize each selling price and buying price to be a random value within a predetermined range
    - The range can be different for each product type

    - However, the buying price for one product type must always be higher than the same type's selling price in the same city
    - In other words, the merchant must always pay more money to buy a product from a city than the merchant can receive by selling the same product to the same city
    - After generating the random values for each city, be sure to validate that this condition applies:
        - For each product type, if the random buying price is not higher than the random selling price, then swap the two values

Additional features

In addition to these requirements, you are encouraged to add some features or functionalities to your application.

Every additional feature added <u>must</u> be documented and described in a readme file contained in the project submission.

Evaluation

This is a team project, and it must be produced by the entire team working together as a group. All work must be original, and must not be copied from any external sources, including any students in other teams or classes. All members in a team are expected to write a roughly equal amount of the code.

The team must present the project to the professor (not to the entire class). Each team member is expected to be familiar with all of the code, even in sections they did not write. Questions will be asked of team members to ensure that they are familiar with the code, understand what it does, and know why it is necessary. Points will be deducted from a student's grade if they cannot answer the questions adequately.

For example, if a student is shown to only understand 50% of the code, and the project as a whole receives a 90% grade, then the student's final grade for the project will be 45%.

The project will be graded according to the following criteria:

Exactitude

- Does the program do what it is supposed to do, and meet all requirements?

Input validation

- The application should correctly validate all user input
- The application should handle any errors that can occur during input, including displaying user-friendly error messages to the user

User experience and visual presentation

- All text displayed to the screen should be well-arranged and written in proper English
- The user interface should be user-friendly: it should be easy to use and behave predictably
- The user interface should show the right information to the user at the right time

Structure

- The code should be divided into functions, classes, and methods, according to the relevant needs
- Each function or method should perform a single simple task (usually a few lines of code)
- Each class should only contain fields and methods that are closely related to each other
- If a class contains members that are not conceptually or logically related to each other, then the class should probably be split up in to two or more classes
- Unchanging values should be stored in constants, rather than hard-coded as literal values

Format and conventions

- The code should contain relevant comments
- The code should be properly indented
- The code should respect naming conventions
    - The names of variables and functions/methods should be written in camelCase
    - The names of classes should be written in UpperCamelCase (Pascal case)
    - The names of constants should be written in UPPER_CASE