Project 2, EuroRails Game Simulator

Due: Please check due date on BlackBoard

Objectives

Use of STL Vectors, Stacks, Queues and Maps will all be covered in this project. This second Data Structures project may not use the structures that are best for printing the state of the data, but it will provide you with experience using STL and the features particular to each data structure in a way that reflects how they may be used in broader applications.

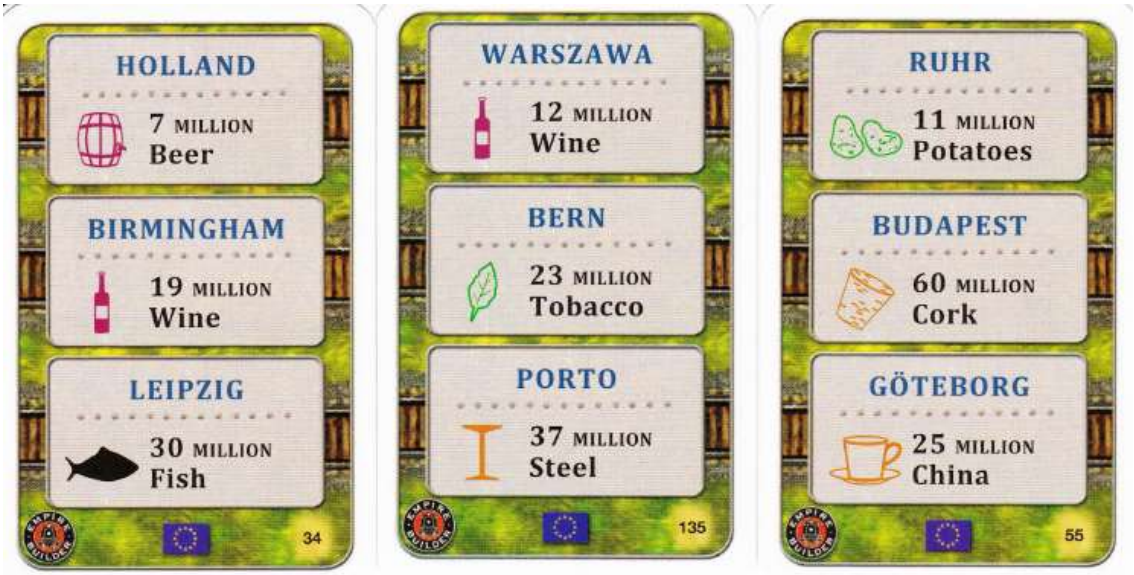
Introduction

This project will be adapting the cards from Mayfair's EuroRails to create a simple game simulation of my own design that will be run by this project. Each card has three potential objectives which include the destination city, the commodity and the payoff for reaching the objective. A stack of these cards will be provided, then dealt to each player. Using one of a selection of strategies the simulation will print out the objectives selected by each player and then their final score. The simulation will then determine the winner and print that winner to the screen.

Your program will be given two data files (examples given) that you will need to use to build your initial stack of cards and the commodities store. On the command line you will also take in two numbers, the number of players and the index of the strategy to use to play the game. From that your program will run its simulation.

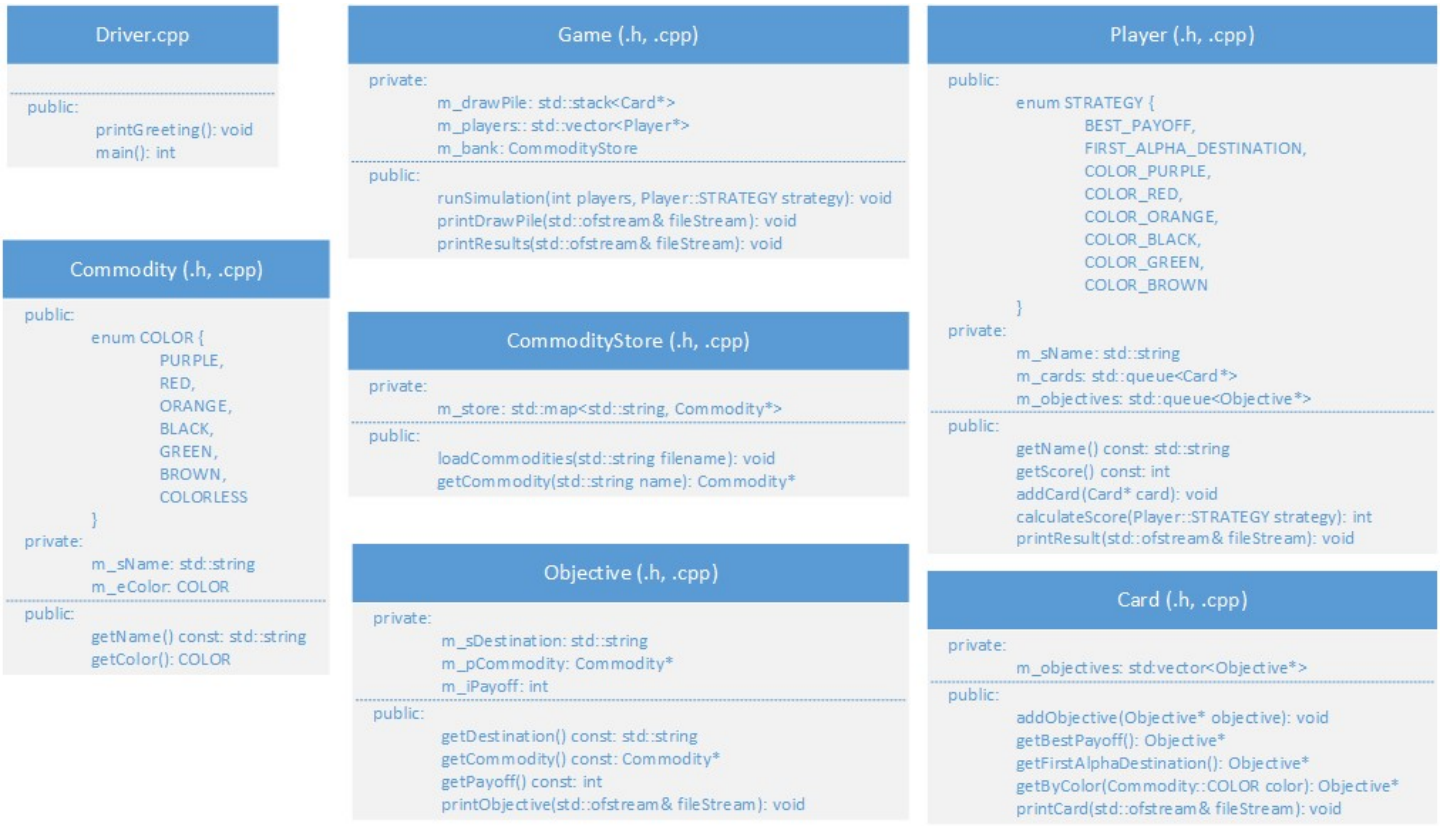
Your Driver should include a greeting function named printGreeting(), that will print YOUR NAME AND SECTION.

More details will be described below.



Project UML Diagram

This diagram contains all the classes, functions and variables that are required for your project. But this is not a comprehensive list of the classes, functions and variables that you can create. You are not limited to just the elements in this diagram.



Required Strategy Enum format (in class Player.h):

```
enum STRATEGY {
      BEST_PAYOFF, // 0
      FIRST_DESINATION, // 1
      COLOR_PURPLE, // 2
      COLOR_RED, // 3
      COLOR_ORANGE, // 4
      COLOR_BLACK, // 5,
      COLOR_GREEN, // 6
      COLOR_BROWN, // 7
};
```

Game Descriptions

The game simulates a card game where each player uses the same given strategy to calculate their score and then the highest scoring player is declared the winner. The driver for your project should perform the following operations in the order given:

Driver Operations:

1. Setup the game – Read in the commodities and cards, the cards should be put into a stack in the order they are read in, this mean that the first card you read in will be at the bottom of the deck. Each commodity read in should be stored in a single dynamic instance of a commodity object and be pointed to by the cards that use them.
2. Print the greeting – Print a greeting at the top of your output with your name and class section to the results file.
3. Print the deck – Once the setup is complete the program should print the deck to the results file using the print format described below. The first objective printed for a card should be the first objective for that card that was read in from cards.txt. This function should not change the order of the original stack in any way
4. Run the simulation – Simulates the game play for the number of players passed into the program using the deck created at setup. The simulation should employ the following steps (though it is not required that all steps happen in the order described):
    a. Create the players for the game and give them the names 'Player #' where # is the number of that player starting at 1.
    b. Deal the cards out to the players. The players should all end up with the same number of cards so if the number of cards does not equal divide among the number of players then there will remain some cards undealt in the deck.
    c. Use the provided strategy to calculate the score for each player (all players will use the same strategy). For each card in a player's hand the player will select one of the three provided objectives and add that objective's payoff to their final score. The strategy is used to determine which objective on a card the player will select. The available strategies are detailed in the STRATEGY enum. The follow explains how each strategy is expected to work.
        i. BEST_PAYOFF: The players should select the objective on each of their cards that has the highest available payoff. If there is a tie on the card for the highest payoff the first objective on the card (the one read in first from cards.txt) should be selected
        ii. FIRST_DESTINATION: The player should select the objective on each of their cards where the destination is alphabetically first. If there are two objectives with the same destination, select the first objective on the card (the one read in first from the cards.txt)
        iii. COLOR_<COLOR-NAME>: The player should select the objective whose commodity is of the color specified by the strategy. If the card has more than one objective of that color the objective of that color with the highest payoff should be selected. If the card has no commodities of that color then the objective with the highest payoff should be selected.
    d. Determine the winner of the simulation by comparing the final score of each player and selecting the highest score
5. Print the results – Once the simulation is complete, print out to the results file the results in the format provided below. Each player's results should be printed with the objective the selected for each card printed in sentence format "<commodity> to <destination> for <payoff>". After the results for each player have been printed the winner of the simulation should be declared along with their final winning score.

Class Descriptions

This is a list of required variables and functions and a description of their purpose. This is not a comprehensive list of functions or variables that could be written for this project. There should be an attempt to minimize the amount of memory allocated and not make unnecessary allocations. There should also be no memory leaks or segmentation faults during the running of the simulation.

Game:
- Variables:
  - m_drawPile: the data structure to be used to store the deck of cards
  - m_players: the data structure to be used to store the list of players participating
  - m_back: the data structure storing all the know commodities in the game
- Functions:
  - runSimulation: runs through the simulations of playing the game using the given strategy with the given number of players
  - printDrawPile: prints the current state of the drawPile to a given output stream
  - printResults: prints the result of the simulation to a given output stream

Player:
- Variables:
  - m_sName: the player's name to be printed in the output file, format "Player #"
  - m_cards: the data structure for holding the cards in the players hand in the order the cards were dealt to the player
  - m_objectives: the data structure for holding the selected objectives for each card in the players hand during the simulation in the order the cards were dealt to the player
- Functions:
  - getName: should return the string of the name of the player in format "Player #"
  - getScore: should return the score received by the player after running the simulations, defaults to zero.
  - addCard: adds a given card to the players hand
  - calculateScore: processes the players hand using the given strategy, storing the resulting objectives selected in m_objectives and the resulting score in an internal variable of your choosing. This function should be able to be run on the player's hand multiple times each time the selected objective should be reset before processing the hand
  - printResults: prints the resulting objectives selected by the player from each card and the player's final score

CommodityStore:
- Variables:
  - m_store: A data structure container that maps an instance of each commodity to their respective string name.
- Functions:
  - loadCommodities: Reads in the provided commodities text file and creates an instance for each commodity which it stores in m_store
  - getCommodity: Returns a commodity instance give the commodities string name

Commodity:
- Variables:
  - m_sName: The string name of the commodity
  - m_eColor: a Color enum index of the color of that commodity
- Function:
  - getName: returns a copy of the string name of the commodity

getColor: returns the Color index of the commodity

Card:

Variables:

m_objectives: the data structure containing all the objectives available on a single card in the order that they appear on the card (the order their read in from the file)

Functions:

addObjective: inserts an objective into the card

getBestPayoff: Returns a pointer to the objective that has the best payoff on the card

getFirstAlphaDestination: Returns a pointer to the objective that has the first alphabetic destination name

getByColor: Returns a pointer to the objective with a commodity whose color matches the requested color. If more than one objective has that color, returns the one of those with the highest payoff. If the no commodities on the card have that color return the objective with the highest payoff

printCard: prints the card with its objectives in the order they were read in form the file to the given output stream

Objective:

Variables:

m_sDestination: the name of the destination to which the commodity should be delivered

m_pCommodity: a pointer to the instance in the commodity store of the commodity being requested in this objective.

m_iPayoff: the amount the player will receive for delivering this commodity to its destination

Functions:

getDestination: returns a copy of the name of the destination

getCommodity: returns a reference to the commodity requested

getPayoff: returns a copy of the value received for delivering this commodity to its destination

printObjective: prints an objective as selected by a player in a simulation, "<commodity> to <destination> for <payoff>" to the given output stream.

Input Files Example

It can be assumed that all cities, commodities and colors will not have spaces in them and will be a single word (though those words could have special characters). All input data is separated by a single space. Each card and commodity set will be ended with a newline character.

commodities.txt

```
Bauxite Purple
Beer Red
Cars Orange
Cattle Black
Cheese Red
China Orange
Chocolate Red
...
```

cards.txt

```
Birmingham Oranges 38 Lodz Bauxite 12 Nantes Tourists 17
Lisboa Cars 30 Warszawa Tobacco 39 Leipzig Marble 22
Bruxelles Cars 12 Marseille Imports 20 Cork Machinery 30
Marseilles Tobacco 21 Warszawa Bauxite 14 Aberdeen Cork 63
...
```

Command Line Expected Format

```
make run CARDS="<card_filename>" COMMODITIES="<commodities_filename>"
PLAYERS=<num_players> STRATEGY=<strategy_id>

// Example
make run CARDS=cards.txt COMMODITIES=commodities.text PLAYERS=4 STRATEGY=3

// The lines to create the run command in your make file should be as follows:

run:
        ./Driver.out $(CARDS) $(COMMODITIES) $(PLAYERS) $(STRATEGY)
```

* Example input and output files are provided for you. Consider each set of files. The location of the input file will be in the same directory as your Driver.cpp file. Sample files may not address all cases.

Output File (results.txt)

The results of the simulation that is run should be stored in a file called results.txt. This file will be populated entirely by calls to print functions within the classes and to printGreeting in the Driver. Data should always remain in the type of data structure in which they are stored. Below is the format expected out of each print function listed in the UML Diagram.

`Game::printDrawPile()`

Format for each Card:

```
---------- CARD ----------
<commodity_name> to <destination> for <payoff>
<commodity_name> to <destination> for <payoff>
<commodity_name> to <destination> for <payoff>
--------------------------
```

Example:

```
---------- Draw Pile ----------
---------- CARD ----------
```

```
Marble to London for 31
Iron to Munchen for 26
Wine to Szczecin for 12
-------------------------
---------- CARD ----------
Oranges to Ruhr for 33
Imports to Stockholm for 28
Wood to Venezia for 15
-------------------------
```

Game::printResult()

Format for each Player:

```
--------Player 1-----------
<commodity_name> to <destination> for <payoff>
...
Score: <total_score>
```

Example:

```
---------- RESULTS ----------
--------Player 1-----------
Marble to London for 31
Hops to Holland for 16
Flowers to Kaliningrad for 25
Tobacco to Bern for 23
Score: 95
--------Player 2-----------
Oranges to Ruhr for 33
Copper to London for 25
Steel to Krakow for 20
Cork to Aberdeen for 63
Score: 141
--------Player 3-----------
Sheep to Sarajevo for 44
Machinery to Milano for 20
Oil to Kikiningrad for 27
Cars to Bruxelles for 12
Score: 103
--------Player 4-----------
Tourists to Cork for 17
Chocolate to Berlin for 13
Cork to Budapest for 60
Marble to Leipzig for 22
Score: 112
--------Player 5-----------
Wheat to Lisboa for 26
Bauxite to Kobenhavn for 30
Wine to Birmingham for 19
Oranges to Birmingham for 38
Score: 113
-------------------------
```

```
-------------------------
Winner: Player 2 Score: 141
```

Running and Compiling Requirements

Please compile and complete your work on your own directory on GL. I have had several students that are compiling on mine (or slupoli/pub/cs341...). This takes up a lot of space and my disk quota. My directory is only to be used a repository for your completed files. If this is abused, your directory will be closed automatically by GL.

What to Submit

Read the course project submission procedures. *Submission closes by script immediately after 9pm*. Submit well before the 8:59pm deadline, because 9:00:01 might already be late (the script takes only a few seconds to run).

You should copy over all of your code under the src directory. You must also supply a Makefile. Do NOT submit your own test data files. **Any unnecessary files submitted will be considered for a deduction.**

Make sure that your code is in the ~/cs341proj/proj2/src directory and not in any other subdirectory of ~/cs341proj/proj2/. In particular, the following Unix commands should work.

```
cd ~/cs341proj/proj2/src

make

make run =”<card_filename>” COMMODITIES=”<commodities_filename>” PLAYERS=<players>
STRATEGY=<strategy>

make clean
```

The command “make run” should simply run the project that compiled successfully.

Don't forget the Project Submission requirements shown online!! One hint, **after you submit**, if you type:

ls ~/cs341proj/proj2/

and you see a bunch of .cpp and .h files, this is WRONG. You should see:

src

instead. The C++ programs must be in directories under the src directory. Your submissions will be compiled by a script. The script will go to your proj2 directory and run your makefile. This is required. You will be severely penalized if you do not follow the submission instructions.
