COMP 345: Advanced Program Design with C++
Winter 2021-**Assignment 1**

Concordia University
Engineering and Computer Science

Deadline:	February 19th, 2021
Evaluation:	10% of final mark
Late submission:	not accepted
Teams:	this is a team assignment

Problem statement

This is a team assignment. It is divided into distinct parts. Each part is about the development of a part of the topic presented as the team project. Even though it is about the development of a part of your team project, each assignment is to be developed/presented/tested separately. The description of each part describes what are the features that the part should implement, and what you should demonstrate. Note that the following descriptions describe the baseline of the assignment, and are related to the project description. See the course web page for a full description of the team project, as well as links to the details of the game rules to be implemented.

Specific design requirements

1. All data members of user-defined class type must be of pointer type.
2. All file names and the content of the files must be according to what is given in the description below.
3. All different parts must be implemented in their own separate .cpp/.h file duo. All functions' implementation must be provided in the .cpp file (i.e. no inline functions are allowed).
4. Though each part needs to have its main driver, you need to provide one main function to run all the parts.
5. Your classes must implement a correct copy constructor, assignment operator, and stream insertion operator.
6. Memory leaks must be avoided.
7. Code must be documented using comments (user-defined classes, methods, free functions, operators).
8. No (third-party) libraries that are platform dependent are allowed.
9. Only C++ 17 and up is allowed.

Parts to be implemented

Part 1: Map

Implement a group of C++ classes that implements the structure and operation of a map for the *eight-minute empire Legends* game. The map must be implemented as a connected graph, where each node represents country/region. Edges between nodes represent adjacency between countries/regions. Each region can have any number of adjacent regions. Continents must also be connected subgraphs, where each country belongs to one and only one continent. Each region is owned by a player and can contain a certain number of armies. The map component can be used to represent any map configuration. All the classes/functions that you implement for this component

must all reside in a single **.cpp/.h** file duo named **Map.cpp/Map.h**. You must deliver a file named **MapDriver.cpp** file that contains a main function that creates a map and demonstrates that the component implements the following verifications: 1) the map is a connected graph of adjacent countries/regions, 2) continents are connected subgraphs and 3) each region belongs to one and only one continent. The driver must provide test cases for various valid/invalid maps.

The **Map** class is implemented as a connected graph. The graph's nodes represents a territory (implemented as a **Territory** class). Edges between nodes represent adjacency between territories.
Continents are connected subgraphs. Each territory belongs to one and only one continent.
A territory is owned by a player and contain a number of armies.
The **Map** class can be used to represent any map graph configuration.
The Map class includes a `validate()` method that makes the following checks: 1) the map is a connected graph, 2) continents are connected subgraphs and 3) each country belongs to one and only one continent.
Driver that creates different **Map** objects and checks whether they are valid or not.

Part 2: Map loader

Implement a group of C++ classes that reads and loads a map file, that contains the four game boards of the *eight-minute empire Legends* game. The map loader must be able to read any map file, that is any file characterizing any of the four game's board. The map loader should store the map as a graph data structure (see Part 1). The map loader should be able to read any map file format (even ones that do not constitute a valid map). All the classes/functions that you implement for this component must all reside in a single **.cpp/.h** file duo named **MapLoader.cpp/MapLoader.h**. You must deliver a file named **MapLoaderDriver.cpp** file that contains a main function that reads various files and successfully creates a map object for valid map files, and rejects invalid map files of different kinds.

Map loader can read the *eight-minute empire Legends* map file.
Map loader creates a map object as a graph data structure (see Part 1).
Map loader should be able to read any map file format (e.g text) (even invalid ones).
Driver reads many different map files, creates a graph object for the valid ones and rejects the invalid ones.

Part 3: Player

Implement a group of C++ classes that implements the *eight-minute empire: Legends* game player using the following design: A player owns a collection of region/countries (see Part 1). A player owns the cubes, disks, and tokens armies and a card (see Part 4). A player has his/her own biding facility object (see Part 5). A player must implement the following methods, which are eventually going to get called by the game driver: `PayCoin()`, `PlaceNewArmies()`, `MoveArmies()`, `MoveOverLand()`. `BuildCity()`, and `DestroyArmy()`

All the classes/functions that you implement for this component must all reside in a single **.cpp/.h** file duo named **Player.cpp/Player.h.** You must deliver a file named **PlayerDriver.cpp** file that

creates player objects and demonstrates that the player objects indeed have the above-mentioned features.

Player owns a collection of regions/countries (see Part 1)
Player owns a hand game cards (see Part 4)
Player has his own biding facility object (see Part 5)
Player must implement `PayCoin()`, `PlaceNewArmies()`, `MoveArmies()`, `MoveOverLand()`. `BuildCity()`, and `DestroyArmy()`.
Driver creates players and demonstrates that the above features are available.

Part 4 : Cards deck/hand

Implement a group of C++ classes that implement a deck of *eight-minute empire: Legends* game cards. Note that the cards should not be read from the file. The deck object is composed of as many cards of the *eight-minute empire: Legends* cards. Each card gives goods and action. The deck must have a *`draw()`* method that allows a player to draw a card from the cards remaining in the deck and place it in with the cards space. The hand object is a collection of cards that has an *`exchange()`* method that allows the player to select the card from its position in the row and pay the coin cost listed at the top of the board (see the game rules for details). All the classes/functions that you implement for this component must all reside in a single **.cpp/.h** file duo named **Cards.cpp/Cards.h.** You must deliver a file named **CardsDriver.cpp** file that creates a deck of *eight-minute empire: Legends* game cards.

Deck object is composed of the game's cards
Each card has a good and action
Deck has a *`draw()`* method that allows a player to draw a card from the cards remaining in the deck and place it card space.
Hand object is the collection of face-up cards with assigned coin cost.
Driver creates a deck of cards. Creates a hand object that is filled by face up cards from and that return the each card with its goods and action characteristics.

Part 5: Biding facility

Implement a group of C++ classes that implement a biding facility to be used during start of the game to see who will start first. There is only one bid per game. The biding consists of each player picking up his coins and privately chooses a number to bid. When all players are ready, all players reveal the amount they have chosen to bid at the same time. The player who bids the most coins wins the bid and puts the coins he bids in the supply. Other players do not pay coins if they lost the bid. If the bids are tied for most, the player with the alphabetical last name order wins the bid (e.g., Ng will go first vs. Noah) and pays his coins. If all bids are zero, the last name alphabetical order player goes first.
You must deliver a driver that creates the biding facility objects, with the following tests: 1) one can shows a player who bid the most coins wins, 2) that show bids that are tied and the player with an alphabetical last name order wins the bid, 3) one that if all bids are zero the player with an alphabetical last name order wins.

Enable the player object to pick up his coins and privately chooses a number to bid.
Request all players to reveal the amount they have chosen to bid at the same time.

The player who bids the most coins wins the bid and puts the coins he bids in the supply. If the bids are tied for most, or the bids are zero, the player with the alphabetical last name order wins the bid will go first, and pays his coins.
Driver that test: 1) one can shows a player who bid the most coins wins, 2) that show bids that are tied , 3) one that if all bids are zero.

Assignment submission requirements and procedure

You are expected to submit a group of C++ files implementing a solution to all the problems stated above (Part 1, 2, 3, 4, and 5). Your code must include:

a) a *driver file* for each part, that allows the marker to observe the execution of each part. Each of these drivers should simply create the components described above and demonstrate that they behave as mentioned above.
b) *one main driver function* for all the parts to observe the execution of your parts during the lab demonstration.

You have to submit your assignment before midnight on the due date on the Moodle page for the course, late assignments are not accepted. The file submitted must be a .zip file containing all your C++ code. Do not submit other files such as the project file from your IDE. You need to use C++ 17 and up as your programming environment. Each team needs to demonstrate their assignment on zoom during the scheduled demonstration time. There should be one and only one submission per team.

Evaluation Criteria

Knowledge/correctness of game rules: 2 pts (indicator 4.1)
Mark deductions: during the presentation or code review it is found that the implementation does not follow the rules of the eight-minute empire: Legends

Compliance of solution with stated problem (see description above): 12 pts (indicator 4.4)
Mark deductions: during the presentation or code review, it is found that the code does not do some of which is asked in the above description.

Modularity of the solution: 2 pts (indicator 4.3)
Mark deductions: some of the data members of user-defined class type are not of pointer type, or the above indications are not followed regarding the files/classes/methods needed for each part.

Mastery of language/tools/libraries: 2 pts (indicator 5.1)
Mark deductions: the program crashes during the presentation. The presenter shows lack of understanding or clarity in discussing the technical aspects of the implementation.

Code readability: naming conventions, clarity of code, use of comments: 2 pts (indicator 7.3)
Mark deductions: some class/variable/function names are meaningless, code is hard to understand, comments are missing, presence of commented-out code

---

Total 20 pts (indicator 6.4)