Concordia University
Department of Computer Science and Software Engineering

Advanced program design with C++
COMP 345 --- Fall 2024

Team project assignment #1

Deadline:	October 4th, 2024
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. At this point, your program should be executed by a main function that simply calls each of the test functions that are specified in each of the different parts below.

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. All classes must implement a correct copy constructor, assignment operator, and stream insertion operator.
5. Memory leaks must be avoided.
6. Code must be documented using comments (user-defined classes, methods, free functions, operators).
7. If you use third-party libraries that are not available in the labs and require setup/installation, you may not assume to have help using them and you are entirely responsible for their proper installation for grading purposes.

Parts to be implemented

Part 1: Map

Implement a group of C++ classes that implement the structure and operation of a map for the game Warzone (https://www.warzone.com/). The map must be implemented as a connected graph, where each node represents a territory. Edges between nodes represent adjacency between territories. Each territory can have any number of adjacent territories. Continents must also be connected subgraphs, where each territory belongs to one and only one continent. Each territory is owned by a player and contain a number of armies. The **`Map`** class can be used to represent any map configuration. The map objects should be created by loading a map file in the `.map` text file format as found in the "Conquest" game source files, available at the following web page: http://www.windowsgames.co.uk/conquest_maps.html using a **`MapLoader`** class. The map loader must be able to read any map file available on the above-mentioned web page. The map loader should store the map as a **`Map`** object as described above. The map loader should be able to read any text file (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 **Map.cpp/Map.h**. You must deliver a file named **MapDriver.cpp** file that contains a free function named **testLoadMaps()** that successively creates a map by reading a set of conquest map files and successfully creates a map object for valid map files, and rejects various different invalid map files. After a map object has been successfully created, the driver should demonstrate that the **Map** class implements the following verifications: 1) the map is a connected graph, 2) continents are connected subgraphs, and 3) each country belongs to one and only one continent.

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.
Map loader can read any Conquest map file.
Map loader creates a map object as a graph data structure.
Map loader should be able to read any text file (even invalid ones).
A driver named **testLoadMaps()** that reads many different map files, creates a **Map** object for valid files and rejects the invalid ones.

Part 2: Player

Implement a group of C++ classes that implement a Warzone player using the following design: A player owns a collection of territories (see Part 1). A player owns a hand of Cards (see Part 5). A player has their own list of orders to be created and executed in the current turn (see Part 3). A player must implement the following methods, which are eventually going to get called when decisions have to be made: The player should contain a method named **toDefend()** that return a list of territories that are to be defended, as well as a method named **toAttack()** that returns a list of territories that are to be attacked. For now, all that these methods should do is to establish an arbitrary list of territories to be defended, and an arbitrary list of territories that are to be attacked. The class must also have a **issueOrder()** method that creates an Order object and puts it in the player's list of orders. 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 contains a free function named **testPlayers()** that creates player objects and demonstrates that the player objects indeed have the above-mentioned features.

Player owns a collection of territories (see Part 1)
Player owns a hand of Warzone cards (see Part 5)
Player contains methods **toDefend()** and **toAttack()** that return a list of territories that are to be defended and to be attacked, respectively.
Player contains a **issueOrder()** method that creates an order object and adds it to the list of orders.
A driver named **testPlayers()** that creates players and demonstrates that the above features are available.

Part 3: Orders list

Implement a group of C++ classes that implement a Warzone player orders using the following design: Orders are created by the player during their turn and placed into the player's list of orders. By default, each order is placed in the list sequentially. After orders are put in the list, the player can move them around in the list (using the **`move()`** method) or delete them (using the **`remove()`** method). The different kinds of orders are: deploy, advance, bomb, blockade, airlift, and negotiate. All orders must have a **`validate()`** method that verifies if the order is valid. All orders must have an **`execute()`** method that will result in some game action being implemented (see the project description document). Note that the orders' actions do not need to be implemented at this point. Invalid orders can be created and put in the list, but their execution will not result in any action. All the classes/functions that you implement for this component must all reside in a single **`.cpp/.h`** file duo named **`Orders.cpp/Orders.h`**. You must deliver a file named **`OrdersDriver.cpp`** file that contains a free function named **`testOrdersLists()`** that creates a list of orders and demonstrates that the **`OrdersList`** implemented according to the following specifications:

The **`OrdersList`** class contains a list of **`Order`** objects.
Each kind of order is implemented as a subclass of the **`Order`** class.
The **`Order`** class implements a stream insertion operator that outputs a string describing the order. If the order has been executed, it should also output the effect of the order, stored as a string.
Every order subclass must implement the **`validate()`** method that is used to validate if the order is valid.
Every order subclass must implement the **`execute()`** method that first validates the order, and executes its action if it is valid, according to the order's meaning and the player's state.
The **`OrdersList`** class implements a **`remove()`** method that deletes an order from the list.
The **`OrdersList`** class implements a **`move()`** method to move an order in the list of orders.
A free function named **`testOrdersLists()`** that creates orders of every kind, places them in an **`OrdersList`** object, and demonstrates that the above features are available.

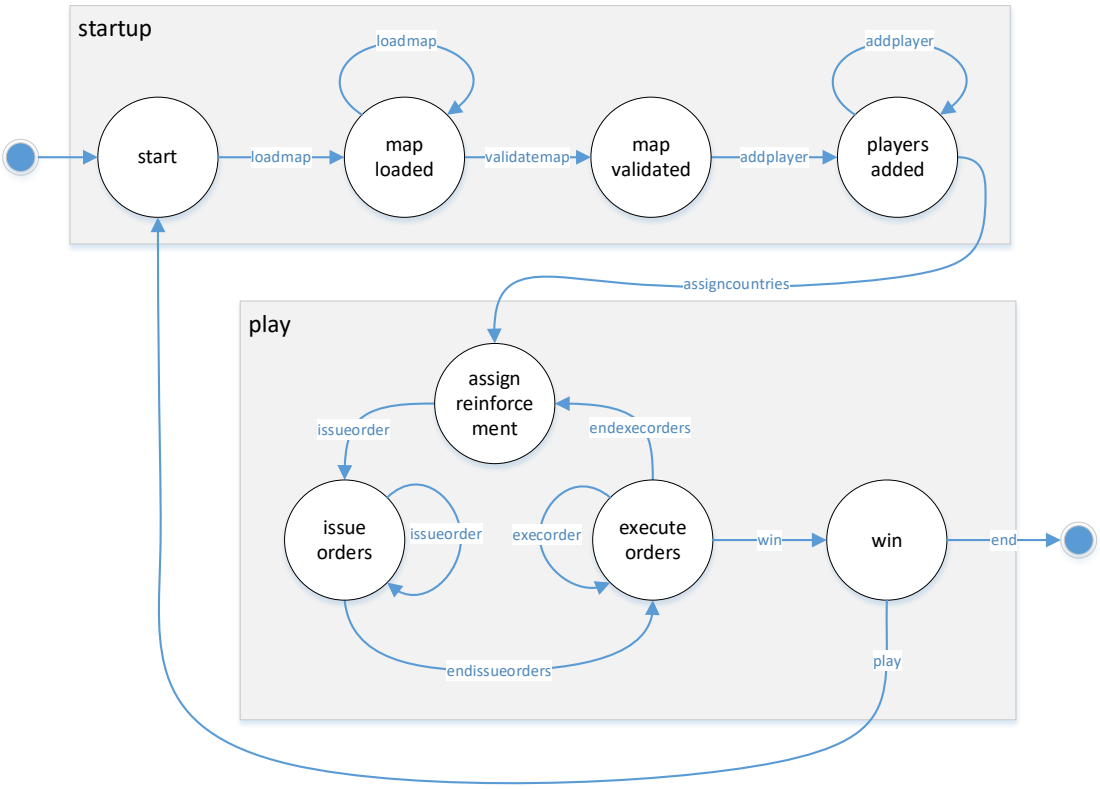
Part 4: Cards deck/hand

Implement a group of C++ classes that implements a deck and hand of Warzone cards. Each card has a type from: bomb, reinforcement, blockade, airlift and diplomacy. The **`Deck`** class must have a **`draw()`** method that allows a player to draw a card at random from the cards remaining in the deck and place it in their hand of cards. The objects of type **`Card`** must have a **`play()`** method that is called to play the card, which creates an order and adds it to the player's list of orders and then returns the card to the deck. 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 contains a free function named **`testCards()`** that creates a deck of Warzone cards, then create a hand object that is filled by drawing cards from the deck.

A **`Deck`** object contains a finite collection of Warzone cards.
A **`Hand`** object is a finite collection of Warzone cards.
Each card has a type from: bomb, reinforcement, blockade, airlift, and diplomacy.
The **`Deck`** has a **`draw()`** method that allows a player to draw a card at random from the cards remaining in the deck and place it in their hand.
Each card has a **`play()`** method that enables a player to use it during game play by creating special orders. Once a card has been played, it is removed from the hand and put back into the deck.
A free function named **`testCards()`** that creates a deck of cards of all different kinds, then creates a hand object that is filled by drawing cards from the deck by repeatedly calling its **`draw()`** method, then calls the **`play()`** method of all cards in the hand, resulting in the cards being put back in the deck.

Part 5: Game Engine

Implement a group of C++ classes that implements a game engine that controls the flow of the game by using the notion of *state*, *transition,* and *command*. The state represents a certain phase of the game and dictates what are the valid actions or user commands that take place in this phase. Some actions or commands may eventually trigger a transition to another state, which is what controls the flow of the game. All the classes/functions that you implement for this component must all reside in a single **`.cpp/.h`** file duo named **`GameEngine.cpp/GameEngine.h`**. You must deliver a file named **`GameEngineDriver.cpp`** file that contains a free function named **`testGameStates()`** that allows the user to type command strings on the console, whose result is to trigger some state transitions as depicted in the state transition diagram presented below. Any command string entered that does not correspond to an outgoing edge of the current state should be rejected.



User enters commands a string on the program console, specified on the edges of the above graph.
Control flow uses states as specified on the above graph.
The correct commands trigger state transitions, as specified on the above graph.
Incorrect commands result in an error message displayed on the console and do not result in a state transition.
A free function named **`testGameStates()`** that creates a console-driven interface that allows the user to navigate through all the states by typing commands as stated in the above graph.

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, 5). Your code must include a single main function that calls all of the five above-mentioned **`test*()`** free functions that allows the marker to observe the execution of each part during the lab demonstration. The main function should be in the file **`MainDriver.cpp`**. Each driver should simply create the components described above and demonstrate that they behave as mentioned above.

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 are allowed to use any C++ programming environment as long as you can demonstrate your assignment on zoom during the demonstration time.

Evaluation Criteria

Knowledge/correctness of game rules: *Mark deductions: during the presentation or code review it is found that the implementation does not follow the rules of the Warzone game.*	**2 pt (indicator 4.1)**
Compliance of solution with stated problem (see description above): *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.*	**12 pts (indicator 4.4)**
Modularity of the solution: *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.*	**2 pts (indicator 4.3)**
Mastery of language/tools/libraries: *Mark deductions: the program crashes during the presentation. The presenter shows lack of understanding or clarity in discussing the technical aspects of the implementation.*	**2 pts (indicator 5.1)**
Code readability: naming conventions, clarity of code, use of comments: *Mark deductions: some class/variable/function names are meaningless, code is hard to understand, comments are missing, presence of commented-out code.*	**2 pt (indicator 7.3)**
Total	**20 pts (indicator 6.4)**