CSI2372

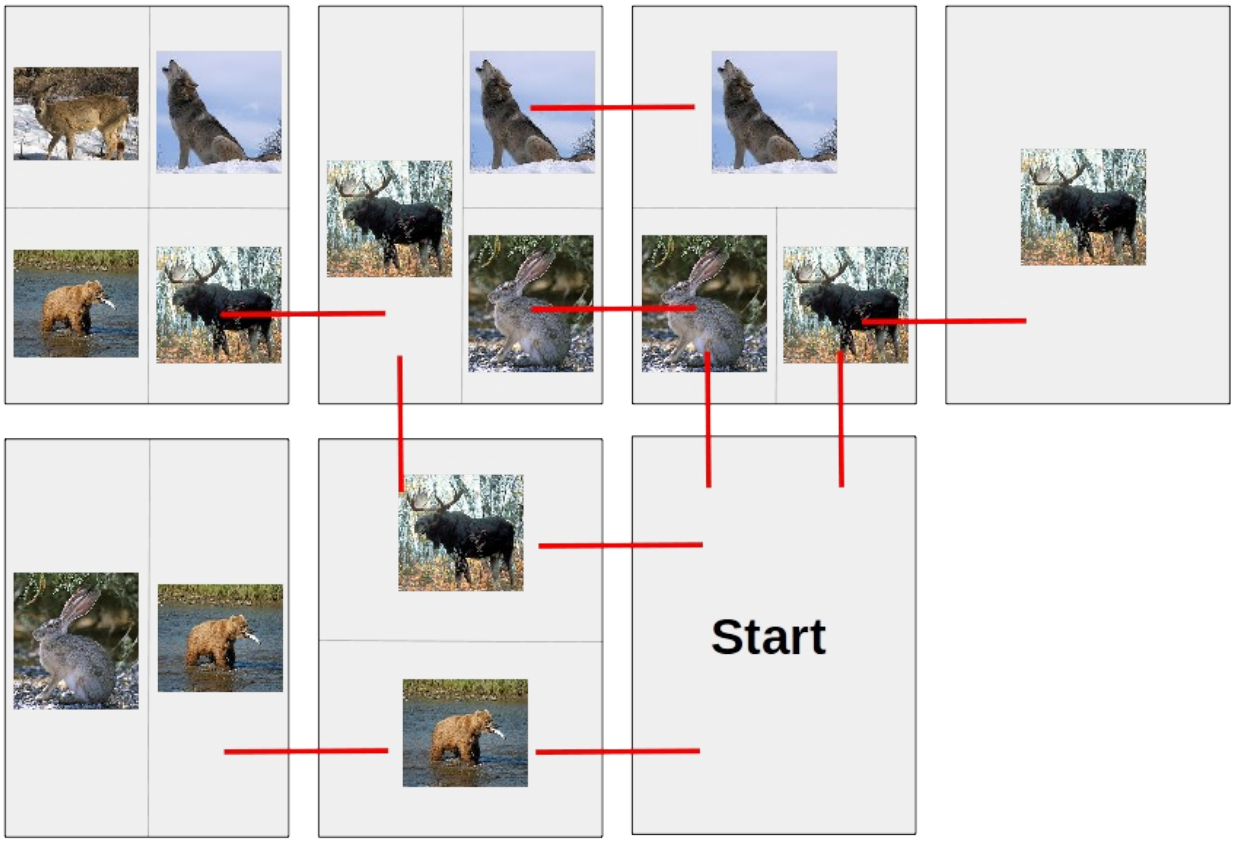
Project

Fall 2015

A Cardgame

In this project, you are asked to program a card game in which players place cards in a grid on the table. The cards show 1 to 4 different animals where each quarter of the card can have a different animal. Each player gets a secret animal. The secret animal is a player's favourite and hence the player's goal is to get seven animal cards showing this player's secret animal connected on the table. A joker and the start card can replace any animal card.

There are five different animals in total: bear, deer, hare, moose, wolf. The cards can only be placed if at least one match exist. The cards have to line up in rows and columns and no offsets are allowed. Note that cards may be turned 180 degrees but not 90 degrees. For a card to be placed legally at least one quarter has to match with one of the neighbouring cards. A valid table examples looks as follows (matches in red):



Besides the 50 animal cards in the game, there is also a joker which matches any animal card. There is also a start card (marked with *Start* above) which if not covered matches also any animal. In addition, there are 15 action cards which show a single animal but also trigger an action. Action cards can only be placed on top or the bottom of the stack at the start card.

At the start of the game, the start card is placed on the table as the only card and each player will be given a secret animal (bear, deer, hare, moose or wolf) at random, along with three animal cards (including action cards and joker). The players take turns. At each turn, a player draws a card from the deck of the remaining animal cards. Then the player places a card on the table: an animal card must be connected to the cards on the table or an action card which must be placed either on top or bottom of the stack of the start card. If the player plays an action card and places it at the bottom of the stack, the action is performed next. An animal card on the table can be matched with each of its quarters with a horizontal or vertical neighbour (diagonally matches are not allowed). If an animal card is placed such that two to four different matches with existing cards exist, then the player draws one to three extra cards from the deck, respectively, and keeps them in the player's hand. (At most four quarters can be matched because each card has at most four different quarters).

There are five different action cards (3 of each):

- The wolf action card allows a player to pick up a card from the table and place it in the player's hand.
- The bear action card allows a player to select another player to switch hands.
- The moose action card triggers a rotation of secret animals form player 1 to player 2, from 2 to 3, etc. in turn.
- The hare action card allows a player to move an animal card on the table including the joker to a different location on the table where the card is valid.
- The deer action card allows a player to select another player to trade goals with.

Program Design

The implementation of the game as a console game requires the following container classes: Table, Deck, Hand and StartStack. We will use three different design strategies to implement these containers: extending a C++ standard container, aggregation of C++ standard container, and wrapping a low-level pointer structure. `Deck` should extend a `std::vector`. `StartStack` should aggregate a double-ended queue (`std::deque`). `Hand` should aggregate a `std::list`. `Table` should be a wrapper around a four-connected graph implemented with a fixed size of array of pointers .

Inheritance will have to be used with the animal cards with an abstract base class `AnimalCard`. Four different derived classes `NoSplit`, `SplitTwo`, `SplitThree` and `SplitFour` representing the corresponding number of animals on the card. A separate `Joker` and `StartCard` class will need to be be created. The abstract `ActionCard` class will have five concrete children: `BearAction`, `DeerAction`, `HareAction`, `MooseAction` and `WolfAction`. However, standard containers do not work well with polymorphic types because they hold copies, i.e., slicing will occur. Hence, a good

approach is to work with pointers. C++11 provides std::shared_ptr which enables us to use pointers and at the same time, we do not have to worry about memory leaks.

A template class will have to created for `Deck` being parametric in the type of card. In this project, we will use it with the class `AnimalCard`.

We are to use exceptions with downcasts to distinguish between an `ActionCard` and other `AnimalCard` played. We will also raise an exception `IllegalPlacement` if a player attempts to place a card illegally (i.e., no matching animals).

We will also use standard algorithms, in particular, `std::shuffle` at the beginning to ensure the cards in the deck are in a random order.

In addition, the game needs to have a pause functionality, i.e., the game needs to be saved to and reloaded from file. This needs to be done through the stream insertion and extraction operator.

Implementation

Below describes the public interface of the implementation that you will have to realize. You will need to decide on class variables and the private and protected interface of the classes. Your mark will depend on a reasonable design (information hiding) and documentation in the code. Remember to use constness as much as possible.

Create the animal card hierarchy. An animal card can be printed to console with two characters per row over two rows. Cards have to be separated by a space character in a column and an empty row (see below). The printing will use the first character of the respective animal: **b**ear, **d**eer, **h**are, **m**oose and **w**olf. The table shown above with images would print as follows (c is for Start**C**ard):

```
  0  1  2  3
0 dw mw ww mm
  bm mh hm mm

1 hb mm cc
  hb bb cc
```

AnimalCard (4 marks)

`virtual void setOrientation( Orientation )` changes the orientation of the animal card. `Orientation` is a scoped enumeration with two values `UP` and `DOWN`, effectively rotating the card by 180 degrees.

`virtual void setRow( EvenOdd )` will change the print state for the current card. `EvenOdd` is a scoped enumeration with the three values `EVEN`, `ODD` and `DEFAULT`. `EVEN` should set the next row to be printed the top row while `ODD` means the next row to be printed will be the bottom row. `DEFAULT` will keep the state unchanged.

`virtual EvenOdd getRow()` returns the state of the next row to be printed.

`virtual void printRow( EvenOdd )` prints two characters for the specified row. An argument of `DEFAULT` will use the state of the `AnimalCard`.

You will also need to create a global stream insertion operator for printing any objects of such a class which implements the *"Virtual Friend Function Idiom"* with the class hierarchy.

The derived classes `NoSplit`, `SplitTwo`, `SplitThree` and `SplitFour` will have to be concrete classes.

The derived classes `Joker` and `StartCard` will have to be concrete classes derived from `NoSplit`.

StartStack (2 marks)

The `StartStack` has to be derived from `AnimalCard`. The behaviour as a card will be determined by which card is on top (of the aggregated `std::deque`). The class should have the following functions:

`StartStack& operator+=( std::shared_ptr<ActionCard> )` places a copy of the action card on top and implicitly changes `StartStack` behaviour as an `AnimalCard`.

`StartStack& operator-=( std::shared_ptr<ActionCard> )` places a copy of the action card on the bottom which does not change how `StartStack` behaves as an `AnimalCard`.

`std::shared_ptr<StartCard> getStartCard()` returns a shared pointer to the start card (needed for insertion into Table).

The default constructor should create a `StartStack` which holds only the `StartCard`.

ActionCard and QueryResult (3 marks)

The abstract `ActionCard` class will have to be derived from `StartCard.` The class should add the pure virtual function:

`virtual QueryResult query()` which will display the action on the console and query the player for input if needed. Returns a QueryResult object storing the result.

`virtual void perfom( Table&, Player*, QueryResult )` which will perform the action with the user input stored in QueryResult.

The classes `BearAction`, `DeerAction`, `HareAction`, `MooseAction` and `WolfAction` will implement this function.

Action cards should print themselves in capital letters (see example in Hand).

Hand (2 marks)

`Hand& operator+=(std::shared_ptr<AnimalCard>)` adds a pointer to the AnimalCard to the hand.

`Hand& operator-=(std::shared_ptr<AnimalCard>)` removes a card equivalent to the argument from the `Hand`. If the card does not exist an exception `MissingCard` is thrown.

`std::shared_ptr<AnimalCard> operator[](int)` returns the `AnimalCard` at a given index.

`int noCards()` returns the number of cards in the hand.

Also add the insertion operator to print `Hand` on an `std::ostream`. The hand should print a header row counting the cards from 0 followed by the cards, for example:

```
0  1  2  3
mw hw ww BB
bb mh hm BB
```

Player (2 marks)

`std:string swapSecretAnimal(std::string&)` changes the current secret animal to the argument and returns old secret animal.

`std::string getSecretAnimal()` gets the current secret animal as a string.

Also add the insertion operator to print a `Player` to an `std::ostream`. The player should print the `Hand` and its secret animal. Player needs to hold a player's name. Hand should be a publicly accessible class variable of Player.

Table (5 marks)

Table implements a four-connected graph holding each `AnimalCard` with `std::shared_ptr`. The graph will be stored in a two-dimensional array of a `std::shared_ptr` to the `AnimalCard` at the location of a given row and column. The neighbouring nodes can therefore simply be found through indexing. Because the game has only 51 `AnimalCard` plus the cards of the `StartStack` a fixed size array of 103x103 will suffice.

The default table constructor will create a `StartStack` with only the start card at location 52, 52.

`int addAt(std::shared_ptr<AnimalCard>, int row, int col)` adds an `AnimalCard` at a given row, column index if it is a legal placement. It will return an integer between 1 and 4 indicating how many different animals can be matched between the current card and its neighbours. It will return 0 and not add the card to `Table` if no valid match is found.

`Table& operator+=( std::shared_ptr<ActionCard> )` places a copy of the action card on top of the `StartStack` in `Table`.

`Table& operator-=( std::shared_ptr<ActionCard> )` places a copy of the action card on the bottom of the `StartStack` in `Table`.

`std::shared_ptr<AnimalCard> pickAt(int row, int col)` removes an `AnimalCard` at a given row, column index from the table. Note, cards on the StartStack cannot be picked and the method should throw an exception `IllegalPick`.

`bool win( std::string& animal )` Returns true if the animal in the string has won. An animal wins as soon as there are seven matching animal cards (including the joker and action cards).

Also add the insertion operator to print the `Table` to an `std::ostream`. A print of the table will include a row and column index as shown above.

Deck<T> (1 mark)

Deck is simple derived class from `std::vector` and is templated by type.

`std::shared_ptr<T> draw()` returns and removes the top card from the deck.

AnimalCardFactory (4 marks)

The animal card factory serves as a factory for all the std::shared_ptr<AnimalCard> except for the `StartCard`. In the constructor for `AnimalCardFactory` a random order of the cards need to be produced. This order must be different with different executions of the program. Ensure that no copies can be made of `AnimalCardFactory` and that there is at most one `AnimalCardFactory` object in your program.

The factory should produce a deck of 50 regular animal cards plus a joker. The regular cards need to be divided into 5 `NoSplit`, 10 `SplitTwo`, 20 `SplitThree` and 15 `SplitFour`. Each animal has to be shown the same number of times in each of the four sub-classes and no two cards can be the same at 0 or 180 degree rotation.

`static AnimalFactory* getFactory()` returns a pointer to the only instance of Animal Factory.

`Deck<std::shared_ptr<AnimalCard> > getDeck()` returns a deck with all 51 animal cards. Note that the 51 animal cards should always be the same but their order in the deck needs to be different every time. Use `std::shuffle` to achieve this.

Pseudo Code (3 marks for game loop)

The simplified pseudo-code of the main loop is as follows.

Setup:

- Input the names of 2-5 players. Initialize the Deck and draw 3 cards for the Hand of each Player; or
- Load from file.

While no Player has won

    if pause save game to file and exit

    For each Player

        Display Table

```
            Player draws top card from Deck
            Display Player
            do
                Ask Player input to choose card
                Play a card
                Place card in Table
            while card is not placed legally
            if ActionCard was played and added on the bottom, perform the action
            for all players
                check if the player has won  // Note player may win even at another player's turn
            end
        end
end
```

Images from Wikipedia (Bear image is from “Driving to Alaska” by Carl Chapman, CC 2.0, the remaining images are in the public domain).

The game is inspired by Andrew Looney's Seven Dragons.