CSI2372

Project

Fall 2016

A Cardgame

In this project, you are asked to program a card game in which two players place cards in chains, trade the cards and sell the chains. The cards have 8 different faces corresponding to semi-precious and precious gemstones. The goal of the game is to chain-up the cards of equal gemstones to gain coins. The player with the most coins at the end wins. The chains for the gemstone cards are formed by each player for all to see and there are a maximum two chains per player (or three chains if the player purchases a third). Each chain can only be formed with a single type of gemstone. The game proceeds with the players taking turns. Each player starts with five cards in their hand and the remaining cards form a draw deck. The cards in a player's hand need to be kept sorted. Cards will be placed on a discard pile during the game.

Each turn of a player proceeds as follows:

1. If the other player has left gemstone cards in the trade area from the previous turn (in Step 5), the player may add these gemstone cards to his/her chains or discard them.
1. The player then plays the topmost card from his/her hand. The card must be added to a chain with the same stones. If the player has currently no such chain on the table, an old chain on the table will have to be tied and sold, and a new chain must be started. If the chain has not reached a sufficient length, a player may receive 0 coins.
2. The player has the option to repeat Step 2.
3. Then, the player has the option of discarding one arbitrary card (not necessarily in order) from his/her hand on the discard pile face up.
4. The player draws three cards from the draw deck and places them in the trade area. Next, the player draws cards from the discard pile as long as the card matches one of the stones in the trade area. Once the top card does not match (or the pile is empty), the player can either chain the cards or leave them in the trade area for the next player. As in Step 2, if the player has currently no such chain matching the stone of the card, an old chain on the table will have to be tied and sold, and then a new chain is started.
5. The turn ends with the player drawing always two cards from the deck and placing them at the back of his/her hand.

Whenever a player ties a chain and sells it (Steps 2,3 or 5), the player receives coins in trade for the chain. Chains of different length and different semi-precious and precious gemstones have different value as follows:

Gemstone and Chain length	1 Coin	2 Coins	3 Coins	4 Coins
Quartz	4	6	8	10
Hematite	3	6	8	9
Obsidian	3	5	7	8
Malachite	3	5	6	7
Turquoise	2	4	6	7
Ruby	2	4	5	6
Amethyst	2	3	4	5
Emerald	-	2	3	-

Note that chains shorter than these length earn the lower number of coins, e.g. 1-3 Quartz cards are worth 0 coins, 4 and 5 Quartz cards 1 coin, 6 and 7 cards 2 coins, 8 and 9 cards 3 coins and 10 or more cards are worth 4 coins.

However, it is important to note that there a different number of cards in the game for the different gemstones:

Gemstone	Quartz	Hematite	Obsidian	Malachite	Turquoise	Ruby	Amethyst	Emerald
# cards	20	18	16	14	12	10	8	6

A player can purchase the right to form a third chain for three coins. No more than three chains can be formed simultaneously by a player during the game.

The game ends when the deck becomes empty.

The game idea is inspired by Bohnanza by Uwe Rosenberg, published in English

Program Design

The implementation of the game as a console game will follow an object-oriented design where each component of the game is represented by its corresponding class:

`Card, Deck, DiscardPile, Chain, Table, TradeArea, Hand, Players.`

We note that `Deck`, `DiscardPile`, `TradeArea`, `Hand` and `Chain` are all containers holding cards. `Deck` will initially hold all the cards which will have to be shuffled to produce a randomized order, then players' hands are dealt and during game play players draw cards from the `Deck`. There is no insertion into the `Deck`. `Deck` can therefore usefully extend a `std::vector`. `DiscardPile` must support insertion and removal but not at random locations but all at the end. Again a `std::vector` will work fine but here we can use simple aggregation. `TradeArea` needs random access as the cards are removed from the area depending on which cards the Player wants to chain. We will use `std::list` to support random access by gemstone type. `Hand` is well mapped by a queue since players have to keep their hand in order and the first card drawn is the first card played. The only derivation from this pattern is if players discard a card from the middle in Step 4 in the above description of a player's turn. Therefore, we can use a `std::vector` to remove at an arbitrary location with a `std::queue` adapter. This will make removal somewhat inefficient, i.e., O(n) but it is rare and we can accept it. A `Chain` is also a container and will have to grow and shrink as the game progresses. Again insertion is only to one end of the chain and a `std::vector` is fine. Please see below for more details below.

We will be practising inheritance with the different cards. `Card` will be an abstract base class and the eight different gemstone cards will be derived from it. All containers will hold cards through the base type. However, standard containers do not work well with polymorphic types because they hold copies, i.e., slicing will occur.

All cards are going to be generated through a `CardFactory`. While here not strictly necessary, it will give us a chance to explore the factory pattern. A factory ensures that there is only a single unit in the program that is responsible to create and delete cards. Other parts of the program will only use pointers to access the cards. Note that means, we will delete the copy constructor and assignment operator in `Card`.

A template class will have to be created for `Chain` being parametric in the type of card. In this project, we will instantiate `Chain` for the corresponding gemstone card.

We are to use exceptions with downcasts to distinguish between different gemstone cards. We will also raise an exception `IllegalType` if a player attempts to place a card illegally into a chain (i.e., a chain that holds different gemstones).

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 state needs to be saved to and reloaded from file. We will use a mixture of stream insertion and extraction, dedicated constructors and function to achieve this feature (see below for details).

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. **You can (and should) make any function or operator const as you see fit, even if it is not indicated in the prototype below.**

Card Hierarchy (4 marks)

Create the gemstone card hierarchy. A gemstone card can be printed to console with its first character of its name. The base class `Card` should be abstract, derived classes `Quartz`, `Hematite`, `Obsidian`, `Malachite`, `Turquoise`, `Ruby`, `Amethyst`, and `Emerald` will have to be concrete classes.

Card will have the following pure virtual functions:

`virtual int getCardsPerCoin( int coins )` will implement in the derived classes the above table for how many cards are necessary to receive the corresponding number of coins.

`virtual string getName()` returns the full name of the card (e.g., Ruby).

`virtual void print(std::ostream& out)` inserts the first character for the card into the output stream supplied as argument. 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.

Chain (2 marks)

The template Chain will have to be instantiated in the program by the concrete derived card classes, e.g., `Chain<Ruby>.` Note that in this example `Chain` will hold the `Ruby` cards by pointer in a `std::vector<Ruby*>.`

`Chain<T>& operator+=( Card* )` adds a card to the `Chain`. If the run-time type does not match the template type of the chain and exception of type `IllegalType` needs to be raised.

`int sell()` counts the number cards in the current chain and returns the number coins according to the function `Card::getCardsPerCoin.`

Also add the insertion operator to print `Chain` on an `std::ostream`. The hand should print a start column with the full name of the gemstone, for example with four cards:

```
Ruby     R R R R
```

The `Chain` needs a constructor which accepts an `std::istream` and reconstructs the chain from file when a game is resumed.

```
Chain( std::istream&, CardFactory* )
```

Chain_Base

The template classes Chain<class T> will need a common public base class in order to design methods which can act upon any type of chain by using runtime type information.

Deck (2 marks)

Deck is a simple derived class from `std::vector`.

`Card* draw()` returns and removes the top card from the deck.

Also add the insertion operator to insert all the cards in the deck to an `std::ostream`.

The `Deck` needs a constructor which accepts an `std::istream` and reconstructs the deck from file.

```
Deck( std::istream&, CardFactory* )
```

DiscardPile (2 marks)

The `DiscardPile` holds cards in a `std::vector` and is similar to `Deck`.

`DiscardPile& operator+=( Card* )` discards the card to the pile.

`Card* pickUp()` returns and removes the top card from the discard pile.

`Card* top()` returns but does not remove the top card from the discard pile.

`void print( std::ostream& )` to insert all the cards in the `DiscardPile` to an `std::ostream`.

Also add the insertion operator to insert only the top card of the discard pile to an `std::ostream`.

The `DiscardPile` needs a constructor which accepts an `std::istream` and reconstructs the `DiscardPile` from file.

```
DiscardPile( std::istream&, CardFactory* )
```

TradeArea (3 marks)

The class `TradeArea` will have to hold cards openly and support random access insertion and removal. The `TradeArea` holds cards in a `std::list`.

`TradeArea& operator+=( Card* )` adds the card to the trade area but it does not check if it is legal to place the card.

`bool legal( Card* )` returns true if the card can be legally added to the TradeArea, i.e., a card of the same gemstone is already in the TradeArea.

`Card* trade( string )` removes a card of the corresponding gemstone name from the trade area.

`int numCards()` returns the number of cards currently in the trade area.

Also add the insertion operator to insert all the cards of the trade area to an `std::ostream`.

The `TradeArea` needs a constructor which accepts an `std::istream` and reconstruct the `TradeArea` from file.

`TradeArea( std::istream&, CardFactory* )`

Hand (2 marks)

`Hand& operator+=(Card*)` adds the card to the rear of the hand

`Card* play()` returns and removes the top card from the player's hand.

`Card* top()` returns but does not remove the top card from the player's hand.

`Card* operator[](int)` returns and removes the `Card` at a given index.

Also add the insertion operator to print `Hand` on an `std::ostream`. The hand should print all the cards in order.

`Hand` needs a constructor which accepts an `std::istream` and reconstruct the `Hand` from file.

`Hand( std::istream&, CardFactory* )`

Player (3 marks)

`Player( std::string& )` constructor that creates a Player with a given name.

`std:string getName()` get the name of the player.

`int getNumCoins()` get the number of coins currently held by the player.

`Player& operator+=( int )` add a number of coins

`int getMaxNumChains()` returns either 2 or 3.

`int getNumChains()` returns the number of non-zero chains

`Chain_Base& operator[](int i)` returns the chain at position i.

`void buyThirdChain()` adds an empty third chain to the player for two coins. The functions reduces the coin count for the player by two. If the player does not have enough coins then an exception `NotEnoughCoins` is thrown.

`void printHand(std::ostream&, bool)` prints the top card of the player's hand (with argument false) or all of the player's hand (with argument true) to the supplied `std::ostream`.

Also add the insertion operator to print a `Player` to an `std::ostream.` The player's name, the number of coins in the player's possession and each of the chains (2 or 3, some possibly empty) should be printed. Note that the `Hand` is printed with a separate function. The player printout may look as follows:

```
Jane     3 coins
Ruby     R R R R
Quartz   Q
```

`Player` needs a constructor which accepts an `std::istream` and reconstruct the `Player` from file.

`Player( std::istream&, CardFactory* )`

Table (2 marks)

Table will manage all the game components. It will hold two objects of type Player, the Deck and the DiscardPile, as well as the TradeArea.

`bool win( std::string& )` returns true when a player has won. The name of the player is returned by reference (in the argument). The winning condition is that all cards from the deck must have been picked up and then the player with the most coins wins.

`void print(std::ostream&)` prints the complete table with all content. Intented for serialization to file.

Also add the insertion operator to print a `Table` to an `std::ostream.` The two players, the discard pile, the trading area should be printed. This is the top level print out. Note that a complete output with all cards for the pause functionality is printed with the separate function print above.

`Table` needs a constructor which accepts an `std::istream` and reconstruct the `Table` from file.

`Player( std::istream&, CardFactory* )`

CardFactory (2 marks)

The card factory serves as a factory for all the gemstone cards. In the constructor for `CardFactory` all the cards need to be created in the numbers needed for the game (see the above table). Ensure that no copies can be made of `CardFactory` and that there is at most one `CardFactory` object in your program.

`static CardFactory* getFactory()` returns a pointer to the only instance of `CardFactory`.

`Deck getDeck()` returns a deck with all 104 gemstone cards. Note that the 104 gemstone cards are always the same but their order in the deck needs to be different every time. Use `std::shuffle` to achieve this.

Pseudo Code (4 marks for game loop)

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

Setup:

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

`While` there are still cards on the Deck

    `if` pause save game to file and exit

    `For` each Player

        Display Table

```
            If Player has 3 coins and two chains and decides to buy extra chain
                Add chain to player
            Player draws top card from Deck
            If TradeArea is not empty
                Add gemstone cards from the TradeArea to chains or discard them.
            Play topmost card from Hand.
            If chain is ended, cards for chain are removed and player receives coin(s).
            If player decides to
                Play the now topmost card from Hand.
            If chain is ended, cards for chain are removed and player receives coin(s).
            If player decides to
                Show the player's full hand and player selects an arbitrary card
                Discard the arbitrary card from the player's hand and place it on the discard pile.
            Draw three cards from the deck and place cards in the trade area
            while  top card of discard pile matches an existing card in the trade area
                draw the top-most card from the discard pile and place it in the trade area
            end
            for all cards in the trade area
                if player wants to chain the card
                    chain the card
                    If chain is ended
                        cards for chain are removed and player receives  coin(s).
                else
                    card remains in trade area for the next player.
            end
            Draw two cards from Deck and add the cards to the player's hand (at the back).
        end
end
```