Projet

CSI2772A- Automne 2021

Université d'Ottawa

/25

Dû le lundi 29 Novembre 2021 à 8am (par équipe de deux étudiants et une seule soumission par équipe est exigée)

Note : Certaines parties du projet ne sont pas encore couvertes en cours, vous commencez par ce que vous savez faire et compléter au fur et à mesure qu'on avance dans le cours. En attendant, vous pouvez aussi jeter un coup d'œil sur la bibliothèque standard du C++ si vous désirez.

Aussi, pour vous aider avec votre projet, les séances des laboratoires continueront jusqu'à la fin du semestre.

Projet : Jeu de cartes

Dans ce projet, on vous demande de programmer (en C++) un jeu de cartes dans lequel deux joueurs placent les cartes en chaînes, échangent les cartes et vendent les chaînes.
Les cartes ont 8 faces différentes correspondant à différents types de Haricots (voir Tableau ci-dessous). Le but du jeu est de chainer les cartes du même type d'haricot pour gagner des pièces. Le joueur avec le plus de pièces à la fin gagne. Les chaînes pour les cartes sont formées par chaque joueur et sont visibles à tout le monde. Il y a un maximum de deux ou trois chaînes en tout par joueur. Chaque chaîne peut seulement être formée d'un seul type de haricot.
Le jeu commence avec les joueurs à tour de rôle. Chaque joueur reçoit 5 cartes haricots en main et les cartes restantes forment une plate-forme de tirage (une pioche (deck)). Les cartes dans la main d'un joueur doivent être maintenus triées. Les cartes seront placées dans une pile au rebut au cours du jeu.

Chaque tour d'un joueur se déroule comme suit :

1. Si l'autre joueur a laissé des cartes haricots en échange dans le tour précédent (à l'étape 5), le joueur peut ajouter ces cartes haricots à sa/ses chaînes ou les jeter.
2. Le joueur joue alors la première carte de sa main. La carte doit être ajoutée à une chaîne du même haricot. Si le joueur n'a actuellement pas de telle chaîne sur la table, une vieille chaîne sur la table devra être liée et vendue, et une nouvelle chaîne doit commencer. Si la chaîne n'a pas atteint une longueur suffisante, un joueur peut recevoir 0 pièce.
3. Le joueur a la possibilité de répéter l'étape 2.

4. Puis, le joueur a la possibilité de se débarrasser d'une carte arbitraire (pas nécessairement dans l'ordre) de sa main dans la pile au rebut de face visible.
5. Le joueur tire 3 cartes de la pioche (deck), et les place en échange. Ensuite, le joueur tire des cartes de la pile au rebut (discard pile) aussi longtemps que la carte corresponde à l'une des haricots en échange (trade area). Une fois que la carte supérieure (top) ne correspond pas (ou la pile est vide), le joueur peut soit chaîner les cartes ou les laisser en échange pour le joueur suivant. Comme à l'étape 2, si le joueur n'a actuellement pas de telle chaîne correspondante à l'haricot de la carte, une ancienne chaîne sur la table devra être liée et vendue, et puis une nouvelle chaîne est lancée.
6. Le joueur achève son tour en piochant 2 nouvelles cartes, qui seront bien sûr placées derrière la dernière carte de sa main.

À chaque fois qu'un joueur lie une chaîne et la vend (étapes 2, 3 ou 5), le joueur reçoit des crédits en échange pour la chaîne.

Les chaînes de différentes longueurs et différents Haricots ont des valeurs différentes comme sur le tableau ci-dessous.

Cependant, il est important de noter qu'il y a un nombre différent de cartes dans le jeu pour les différentes Haricots.

		Longueur de chaîne			
Haricots (Nom de Carte)	**Total de cartes**	Une pièce de monnaie	2 pièces de monnaie	3 pièces de monnaie	4 pièces de monnaie
Blue	20	4	6	8	10
Chili	18	3	6	8	9
Stink	16	3	5	7	8
Green	14	3	5	6	7
soy	12	2	4	6	7
black	10	2	4	5	6
Red	8	2	3	4	5
garden	6	-	2	3	-

Un joueur peut acheter la bonne pour former une troisième chaîne pour trois pièces de monnaie. Pas plus de trois chaînes peuvent être formées simultanément par un joueur pendant le jeu.

Le jeu se termine quand la plate-forme (deck) est vide.

L'idée du jeu est inspirée par **Bohnanza** par Uwe Rosenberg, publié en Anglais.

Conception du programme

Ou utilisera une conception orientée objet C++ pour la mise en œuvre de la partie comme un jeu console. Chaque élément du jeu est représenté par sa classe correspondante :

`Card, Deck, DiscardPile, Chain, Table, TradeArea, Coins, Hand, Player.`

On note que `Deck`, `DiscardPil`, `Hand` et `Chain` sont tous des conteneurs de cartes.

- `Deck` contiendra d'abord toutes les cartes qui devront être mélangées pour produire un ordre aléatoire, puis les mains des joueurs sont distribuées, et pendant le jeu les joueurs tirent les cartes du `Deck`. Il n'y a pas d'insertion dans `Deck`. `Deck` peut donc étendre utilement un `std::vector`.
- `DiscardPile` doit prendre en charge l'insertion et la suppression tout à la fin et non à des endroits au hasard. `std::vector` fonctionne parfaitement, mais on peut procéder autrement en utilisant de simple agrégation.
- `Hand` est bien représentée par une file (`queue`) puisque les joueurs doivent conserver leur main dans l'ordre et la première carte tirée est la première carte jouée. La seule forme de dérivation de ce modèle est que les joueurs se débarrassent d'une carte au milieu de l'étape 4 (description ci-dessus du tour d'un joueur). On peut utiliser `std::list` pour retirer efficacement à un emplacement arbitraire avec l'adaptateur `std::queue`.
- `Chain` est aussi un conteneur et devra augmenter ou diminuer à mesure que le jeu progresse. L'insertion est seulement à une extrémité de la chaîne et `std::vector` correspond bien. Voir ci-dessous pour plus de détails.
- L'héritage sera utilisé pour les différentes cartes. `Card` est une classe de base abstraite et les huit différentes cartes haricots seront dérivées de celle-ci. Tous les conteneurs tiendront des cartes de type de base. Cependant, les conteneurs standards ne fonctionnent pas bien avec les types polymorphes car ils détiennent des copies (le tranchage se produit).
- Toutes les cartes vont être générées par `CardFactory`, qui assure qu'il n'y a qu'une seule unité dans le programme qui est responsable de créer et de supprimer des cartes. Autres parties du programme utiliseront des pointeurs pour accéder aux cartes. Notez que cela signifie qu'on supprime le constructeur de copie et l'opérateur d'affectation dans `Card`.
- Un patron de classe paramétrée dans le type de `Card` sera créé pour `Chain`. Dans ce projet, nous allons instancier `Chain` pour la carte haricot correspondante.

On doit utiliser des exceptions avec des conversions forcées (*Cast*) pour établir une distinction entre les différentes cartes haricots. On utilisera également une exception `IllegalType` si un joueur tente de placer une carte illégalement dans une chaîne (c.-à-d., une chaîne qui contient différents haricots).

Nous allons également utiliser des algorithmes standards, en particulier `std::shuffle` au début pour s'assurer que les cartes du jeu sont dans un ordre aléatoire.

En plus, le jeu doit avoir une fonctionnalité de pause, c'est-à-dire le jeu doit être sauvegardé et rechargé dans un fichier. Ceci doit être fait par les opérateurs d'insertion et d'extraction de flux.

Implémentation

L'interface publique de l'application que vous aurez à réaliser est décrite ci-dessous. Vous aurez à décider sur les variables de la classe et l'interface privée et protégée. Votre **note** dépendra d'une conception et documentation raisonnables dans le code. N'oubliez pas d'utiliser `const` autant que possible. Vous pouvez prendre n'importe quelle fonction ou opérateur *const* si ça conviendrait même si ce n'est pas indiqué dans l'énoncé.

Vous pouvez ajouter d'autres constructeurs et destructeurs à une classe si nécessaire, ou d'autres fonctions.

Hiérarchie de Carte (4 POINTS)

Créer la hiérarchie de carte haricot. Une carte haricot peut être affichée à la console avec son premier caractère de son nom.

- La classe de base `Card` sera abstraite,
- Les classes derivées `Blue, Chili, Stink, Green, soy, black, Red` et `garden` devront être des classes concrètes.

La classe `Card` aura les fonctions virtuelles pures suivantes :

- `virtual int getCardsPerCoin(int coins)` implémentera dans les classes dérivées le tableau ci-dessus pour un nombre de cartes nécessaires afin de recevoir le nombre correspondant de pièces (coins).
- `virtual string getName()` renvoie le nom complet de la carte (e.g., Blue).
- `virtual void print(ostream& out)` insère le premier caractère de la carte dans le flux de sortie fourni en argument.

  Vous devriez créer également un flux global de l'opérateur d'insertion pour l'affichage des objets d'une telle classe qui implémente *"Virtual Friend Function Idiom"* avec la hiérarchie de la classe.

`Chain` (2 POINTS)

Le patron `Chain` sera instancié dans le programme par les classes dérivées concrètes de carte, e.g., `Chain<Red>`. Noter que dans cet exemple `Chain` tiendra les cartes `Red` par un pointeur sur `std::vector<Red*>`. Ainsi, vous aurez besoin d'une interface de chaîne abstraite `(Chain_Base)` pour pouvoir référencer des chaînes de tout type à partir de la classe `Player`.

`Chain` aura les fonctions suivantes :

- `Chain<T>& operator+=( Card* )` ajoute une carte à `Chain`. Si le type du temps d'exécution ne correspond pas au type patron de la chaîne, l'exception de type `IllegalType` devra être lancée.
- `int sell()` compte le nombre de cartes dans la chaîne courante et renvoie le nombre de pièces correspondantes à la fonction `Card::getCardsPerCoin`.
- Ajouter un opérateur d'insertion pour afficher `Chain` dans un `std::ostream`. La main doit afficher une colonne de départ avec le nom complet du haricot, par exemple avec quatre cartes :

  `Red R R R R`
- `Chain` possède un constructeur qui accepte un `istream` et construit une chaîne (objet de type `Chain)` à partir d'un fichier lorsqu'un jeu est repris :

  `Chain(istream&, const CardFactory* );`

`Deck` (2 POINTS)

`Deck` est simplement une classe dérivée de `std::vector`.

`Deck` aura les fonctions suivantes :

- `Card* draw()` retourne et supprime la carte supérieure du `Deck`.
- Ajouter également l'opérateur d'insertion pour insérer toutes les cartes du jeu dans un fichier `std::ostream`.
- `Deck` possède un constructeur qui accepte un `istream` et construit un objet de type `Deck` à partir du fichier :

`Deck(istream&, const CardFactory* );`

`DiscardPile` (2 POINTS)

`DiscardPile` détient des cartes dans `std::vector` et est similaire à `Deck`.

`DiscardPile` aura les fonctions suivantes :

- `DiscardPile& operator+=( Card* )` jette la carte dans la pile.
- `Card* pickUp()` renvoie et supprime la carte supérieure de la pile au rebut.
- `Card* top()` renvoie mais ne supprime pas la carte supérieure de la pile au rebut.
- `void print( std::ostream& )` pour insérer toutes les cartes de `DiscardPile` dans `std::ostream`.
- Ajouter également l'opérateur d'insertion pour insérer seulement la carte supérieure de la pile au rebut à un flux `std::ostream`.
- `DiscardPile` possède un constructeur qui accepte un flux `istream` et construit un objet de type `DiscardPile` à partir du fichier :

  ```
  DiscardPile(istream&, const CardFactory* );
  ```

TradeArea (2 POINTS)

La classe `TradeArea` devra détenir ouvertement les cartes et supporte l'accès aléatoire d'insertion et de suppression. `TradeArea` détient les cartes dans `std::list`.

`TradeArea` aura les fonctions suivantes :

- `TradeArea& operator+=( Card* )` ajoute une carte à l'échange mais ne vérifie pas si c'est légal de placer la carte.
- `bool legal( Card* )` renvoie true si la carte peut être légalement ajouté à l'échange, i.e., une carte du même haricot est déjà en échange .
- `Card* trade( string )` supprime de l'échange une carte du correspondant nom.
- `int numCards()` renvoie le nombre courant de cartes en échange.
- Ajouter également l'opérateur d'insertion pour insérer toutes les cartes en échange dans `std::ostream`.
- `TradeArea` possède un constructeur qui accepte un flux `istream` et construit un objet de type `TradeArea` à partir du fichier :

```
TradeArea(istream&, const CardFactory* );
```

Hand (2 POINTS)

La classe `Hand` aura les fonctions suivantes :

- `Hand& operator+=(Card*)` ajoute une carte derrière la dernière carte de la main.
- `Card* play()` renvoie et supprime la première carte de la main du joueur .

- `Card* top()` renvoie mais ne supprime pas la première carte de la main du joueur.
- `Card* operator[](int)` renvoie et supprime une carte à une position donnée.
- Ajouter également un opérateur d'insertion pour `Hand` dans un `std::ostream`. La main doit afficher toutes les cartes dans l'ordre.
- `Hand` possède un constructeur qui accepte un flux `istream` et construit un objet de type `Hand` à partir du fichier :

```
Hand(istream&, const CardFactory* );
```

`Player` (3 POINTS)

La classe `Player` aura les fonctions suivantes :

- `Player( std::string& )` constructeur qui crée un objet de type `Player` avec un nom donné.
- `std:string getName()` obtenir le nom du joueur.
- `int getNumCoins()` obtenir le nombre de pièces actuellement détenues par le joueur.
- `Player& operator+=(int )` ajoute un nombre de pièces
- `int getMaxNumChains()` renvoie 2 ou 3.
- `int getNumChains()` renvoie le nombre de chaînes non nulles.
- `Chain& operator[](int i)` renvoie la chaîne en position i.
- `void buyThirdChain()` ajoute une troisième chaîne vide au joueur pour deux pièces. La fonction réduit le nombre de pièces par deux pour le joueur. Si le joueur n'a pas suffisamment de pièces alors une exception `NotEnoughCoins` est soulevée.
- `void printHand(std::ostream&, bool)` affiche la première carte de la main du joueur (avec l'argument *False*) ou l'ensemble de la main du joueur (avec l'argument *True*) dans le flux correspondant `ostream`.
- Ajouter également l'opérateur d'insertion pour afficher un joueur (`Player`) dans `std::ostream`. Le nom du joueur, le nombre de pièces que possède le joueur et chacune des chaînes (2 ou 3, certaines peuvent être vides) doivent être affichés. `Hand` est affichée à l'aide d'une autre fonction. L'affichage du joueur peut se présenter comme suit :

```
Dave    3 coins
Red     R R R R
Blue    B
```

- `Player` possède un constructeur qui accepte un flux `istream` et construit un objet de type `Player` à partir du fichier :

```
Player(istream&, const CardFactory* );
```

`Table` (2 POINTS)

`Table` permettra de gérer tous les composants du jeu. Elle détiendra deux objets de type `Player`, `Deck` et `DiscardPile`, ainsi que `TradeArea`.

La classe `Table` aura les fonctions suivantes :

- `bool win( std::string&` renvoie *true* si le joueur a gagné. Le nom du joueur est transmis par référence (en argument). La condition de gagner est que toutes les cartes de `Deck` doivent avoir été reprises et alors le joueur avec le plus de pièces gagne.
- `void printHand(bool)` affiche la première carte de la main du joueur (avec l'argument *false*) ou l'ensemble de la main du joueur (avec argument *true*).
- Ajouter également l'opérateur d'insertion pour afficher `Table` dans `std::ostream`. Les deux joueurs, la pile au rebut, et `Tradearea` doivent être affichés.
  Veuillez noter qu'un résultat complet avec toutes les cartes pour la fonctionnalité *Pause* est affiché avec une fonction distincte.
- `Table` possède un constructeur qui accepte un flux `istream` et construit un objet de type `Table` à partir du fichier :

```
Player(istream&, const CardFactory* );
```

`CardFactory` (2 POINTS)

`CardFactory` sert d'usine pour toutes les cartes haricots.

La classe `CardFactory` aura les fonctions suivantes :

- Dans le constructeur de `CardFactory`, toutes les cartes doivent être crées en nombre nécessaire pour le jeu (voir le tableau ci-dessus).
- `static CardFactory* getFactory()` renvoie un pointeur à l'unique instance de `CardFactory`.
- `Deck getDeck()` renvoie un jeu avec toutes les 104 cartes haricots. Notez que les 104 cartes haricots sont toujours les mêmes mais leur ordre dans `Deck` doit être différent à chaque fois. Utiliser `std::shuffle` pour réaliser ceci.

Aussi, interdire toute copie dans `CardFactory` et s'assurer d'avoir au moins un objet `CardFactory` dans votre programme.

Pseudo Code (4 POINTS pour la boucle du jeu)

Le pseudo-code simplifié de la boucle `main` est comme suit :

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
         Player draws top card from Deck
         If TradeArea is not empty
               Add bean 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.
               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
```

Fraude scolaire :

Cette partie du projet a pour but de sensibiliser les étudiants face au problème de fraude scolaire (plagiat). Consulter les liens suivants et bien lire les deux documents:

https://www.uottawa.ca/administration-et-gouvernance/reglement-scolaire-14-autres-informations-importantes

https://www.uottawa.ca/administration-et-gouvernance/sites/www.uottawa.ca.administration-et-gouvernance/files/processus_de_traitement_des_cas_de_fraude_academique_-_nov_2019.pdf

Les règlements de l'université seront appliqués pour tout cas de plagiat.

En soumettant ce projet :

1. vous témoignez avoir lu les documents ci-haut ;

2. vous comprenez les conséquences de la fraude scolaire.