CSI2772

Projet

Automne 2018

Jeu de mémoire

Pour votre projet, vous devez implémenter une version console du jeu Memoarr! Le jeu est décrit ci-bas mais il ya plusieurs resources disponibles sur le Web -- comme, par exemple, les sites “The Opinionated Gamer” et “Board Game Geek” -- pour plus d’information sur les règles du jeu. Les spécifications suivantes sont pour le jeu de base mais vous devez également programmer deux variantes plus avancées.

Les 25 cartes-mémoires sont composées d’un animal et d’une couleur arrière-scène (un paysage). Il y a cinq animaux différents (crabe, pingouin, pieuvre, tortue et morse) et cinq couleurs (rouge, vert, mauve, bleu et jaune) pour un total de 25 combinaisons. C’est un jeu pour 2 à 4 joueurs.

Dans le vrai jeu, les cartes-mémoires sont disposées, faces cachées, sur une grille 5 par 5 avec la position du centre laissée vide pour les cartes volcans et les cartes trésors. (Noter que ceci veut dire qu’une des carte-mémoire n’est pas utilisée). Dans notre adaptation, nous n’utiliserons pas de cartes volcans ni trésors et donc laisserons la position du centre vide.

À chaque tour, les joueurs choisissent une carte à retourner et cette carte doit faire une paire avec la couleur ou l’animal de la dernière carte dévoilée. La carte dévoilée reste visible pour le reste du manche. Si un joueur dévoile une carte qui ne correspond pas à la dernière carte, ce joueur est éliminé pour le reste de la manche. Une manche se termine lorsqu’il ne reste qu’un seul joueur, qui reçoit de 1 à 4 rubis au hasard. S’il n’y a plus de cartes à retourner les joueurs encore dans le jeu doivent prendre une carte volcan du centre (et donc perdre) jusqu’à ce qu’il ne reste qu’un seul joueur (dans notre version, la manche s’arrête automatiquement lorsque la dernière carte est retournée). À la fin de chaque manche, les cartes restent en place mais sont retournées face cachée. Après sept manches, le jeu se termine et le joueur avec le plus de rubis gagne. Dans la version physique du jeu, il y a 3 cartes avec 1 rubis, 2 cartes avec 2 rubis, une carte avec 3 rubis et une carte avec 4 rubis.

Le jeu de base

Les règles du jeu de base sont décrites ci-haut et nous représenterons les cartes avec une matrice 3 x 3 de caractères avec un espace entre chaque carte et rangée. Donc, en tout il faut 19 rangées et 19 caractères pour représenter la grille du jeu. Les animaux et les couleurs d’arrière-scènes sont identifés par leurs premières lettres en majuscules et minuscules respectivement. Par exemple, voici la carte pour un morse (**W**alrus) sur une arrière-scène jaune (**y**ellow):

```
yyy
yWy
yyy
```

Les cartes cachées sont représentées par des z minuscules. Un exemple du jeu avec quatre cartes dévoilées suit. La position d'une carte est indiquée par une lettre pour la rangée et un nombre pour la colonne. Le jeu suivant pourrait s'être déroulé dans l'ordre A1 D1 B4 D3.

```
  yyy zzz zzz zzz zzz
A yWy zzz zzz zzz zzz
  yyy zzz zzz zzz zzz

  zzz zzz zzz bbb zzz
B zzz zzz zzz bPb zzz
  zzz zzz zzz bbb zzz

  zzz zzz     zzz zzz
C zzz zzz     zzz zzz
  zzz zzz     zzz zzz

  yyy zzz bbb zzz zzz
D yPy zzz bTb zzz zzz
  yyy zzz bbb zzz zzz

  zzz zzz zzz zzz zzz
E zzz zzz zzz zzz zzz
  zzz zzz zzz zzz zzz

   1   2   3   4   5
```

Il peut y avoir de 2 à 4 joueurs. Le nombre de rubis de chaque joueur n'est dévoilé qu'à la fin des sept manches.

Mode affichage expert

Dans la version affichage experte ("expert display mode"), les règles sont les mêmes que dans le jeu de base sauf que la grille de cartes n'est pas imprimée à l'écran. Seulement les cartes dévoilées sont imprimées avec la position qu'elles occuppent. Par exemple :

```
yyy yyy bbb bbb
yWy yPy bPb bTb
yyy yyy bbb bbb

A1  D1  B4  D3
```

Mode règles expert

Dans cette version du jeu, les cartes (ou plutôt les animaux) ont une deuxième signification. Quand une pieuvre est retournée, la carte est changée de position avec une carte adjacente de la même rangée ou colonne (4 voisins possibles). La carte adjacente peut être cachée ou visible et demeure inchangée après le déplacement. Si un joueur découvre un pingouin, il peut renverser une carte visible (sauf si c'est le premier tour ou il n'y a pas d'autres cartes visibles). Le morse permet d'interdire le prochain joueur de choisir une carte particulière. Il ou elle doit donc choisir une carte différente. Si un joueur dévoile un crabe, il ou elle doit jouer encore. Si la deuxième carte ne fait pas de pair, le joueur est éliminé de la manche. Enfin, si une tortue est retournée, le prochain joueur saute son tour (et donc ne peut pas perdre).

Votre implémentation doit permettre la possibilité de jouer les deux modes présentés.

Implémentation

Les spécifications du jeu de base sont données ci-bas pour ce qui est de l'interface publique des classes. Vous pouvez ajouter n'importe quelle méthode privée ou protégée. Chaque classe doit comprendre une série de tests pour démontrer son fonctionnement sans utiliser la boucle principale du jeu. Vous devez être capable d'activer les tests pour chaque classe avec une directive du préprocesseur. Par exemple, votre classe `Board` doit contenir des tests dans `board.cpp` qui peuvent être activés avec la directive

```
#DEFINE TEST_BOARD_
```

`Pour tester, vous devez créer une version de votre projet qui n'inclut pas la fonction main mais tous les autres fichiers avec la directive préprocesseur activée.`

C'est à vous de définir les interfaces pour les modes experts. Votre implémentation sera évaluée sur la maintenabilité et l'extensibilité de votre code. Ainsi, vous devez – en autant que possible -- éviter le dédoublement de code, l'utilisation de switchs et de branches conditionnelles, favoriser l'utilisation de génériques (c-à-d, les gabarits (*templates*) et la dérivation automatique de types), la programmation orientée objet et la bibliothèque standard. Les points associés à chaque partie de votre implémentation sont indiquées dans les parenthèses carrées [] ci-bas.

Main Loop Pseudo Code [3]

```
Ask player to choose game version, number of players and names of players.
Create the corresponding players, rules, cards and board for the game.
Display game (will show board and all players)
while Rules.gameOver is false
      update status of cards in board as face down
      update status of all players in game as active
      for each player
            Temporarily reveal 3 cards directly in front of the player
      while Rules.roundOver is false
            # next active player takes a turn
            get selection of card to turn face up from active player
            update board in game
            if Rules.isValid(card) is false
                  # player is no longer part of the current round
                  current player becomes inactive
            display game
     Remaining active player receives reward (rubies)
print players with their number of rubies sorted from least to most rubies
print overall winner
```

Player [2]

Concevoir une classe `Player` qui contient le nom du joueur, son côté de la grille (haut, bas, droite ou gauche) et son nombre de rubis. L'objet doit avoir les méthodes plubliques suivantes :

- `string getName() const` retourne le nom du joueur.
- `void setActive(bool)` pour activer et désactiver le joueur
- `bool isActive()` retourne vrai si le joueur est actif.
- `int getNRubies() const` retourne le nombre de rubis qu'a gagné le joueur.
- `void addReward( const Reward& )` augmente le nombre de rubis.
- `void setDisplayMode( bool endOfGame )`

Un joueur doit être imprimable à l'écran avec l'opérateur d'insertion, e.g. `cout << player`. Si endOfGame est faux ceci donne :

```
Joe Remember Doe:  left (active)
```

Si endOfGame est vrai :

```
Joe Remember Doe:  3 rubis
```

Card [1.5]

Concevoir un objet `Card` qui prend un animal et une couleur. Une carte doit être imprimable avec un string pour chaque rangée comme dans la méthode suivante :

```
Card c(Penguin,Red); // This constructor will be private
for (int row = 0; row <c.getNRows(); ++row ) {
      std::string rowString = c(row);
      std::cout << rowString << std::endl;
}
```

À noter que `Penguin` et `Red` sont des valeurs de types énumérées `FaceAnimal` et `FaceBackground`.

Un objet de type Card ne peut pas être copié ni assigné et doit avoir un constructeur privé. Ce n'est que l'objet CardDeck, qui y aura accès grâce à une déclaration friend, qui pourra créer des cartes.

Reward [1]

Créer un objet `Reward` qui prends une valeur de 1 à 4 rubis. Un reward doit aussi être affichable par l'instruction `cout << reward`.

Un objet de type Reward ne peut pas être copié ni assigné et doit avoir un constructeur privé et donne accès à la classe RewardDeck (plus bas) par déclaration friend.

Deck<C> [2]

Concevoir un objet `Deck<C>` comme une fabrique abstraite (voir https://fr.wikipedia.org/wiki/Patron_de_conception) qui sera utilisée pour créer un paquet de cartes ou un ensemble de rubis (Reward). Le paramètre type <C> est intendu d'être un de {Card|Reward}. Cette classe aura besoin des méthodes suivantes :

- `void shuffle()` mélange le paquet de cartes. Vous devez vous servir de la fonction `std::random_shuffle` de la bibliothèque standard.
- `C* getNext()` retourne la prochaine carte ou la prochaine récompense dans le paquet. Retourne nullptr s'il n'y a plus de cartes.
- `bool isEmpty() const` retourne vrai si le paquet de cartes est vide.

CardDeck [2]

Concervoir une classe `CardDeck` dérivée de `Deck<Card>`.

- `static CardDeck&  make_CardDeck()` est la seule méthode publique pour cette classe. La méthode doit toujours retourner le même objet CardDeck pendant l'éxécution du programme.

Un objet de type CardDeck ne peut pas être copié ni assigné et n'a pas de constructeur public.

RewardDeck [2]

Concevoir une classe `RewardDeck` dérivée de `Deck<Reward>` ayant les mêmes carcatéristiques que CardDeck.

Board [2]

Concervoir un objet Board qui contient un tableau de strings pour afficher le jeu à l'écran.

- `bool isFaceUp( const Letter&, const Number&) const` retourne vrai si la carte à la position donnée est visible. Letter et Number sont des énumérations. Lance une exception de type OutOfRange si la combinaison de lettre et numéro est invalide.
- `bool turnFaceUp( const Letter&, const Number& )` change l'état d'une carte et retourne faux si la carte était déjà visible. Lance une exception de type OutOfRange si la combinaison de lettre et numéro est invalide.
- `bool turnFaceDown( const Letter&, const Number& )` change l'état d'une carte et retourne faux si la carte était déjà cachée. Lance une exception de type OutOfRange si la combinaison de lettre et numéro est invalide.
- `void reset()` remets toutes les cartes à l'état caché.

Un objet Board doit être affichable avec l'opérateur d'insertion comme dans `cout << board`.

Game [2.5]

Concevoir un objet Game qui encapsule l'état courrant du jeu et qui contient une variable de type Board. Cet objet est responsable pour imprimer le jeu.

- `int getRound()` retourne un numéro entre 0 et 7 correspondant à la manche courrante.
- `void addPlayer( const  Player& )` ajoute un joueur à la partie.
- `Player& getPlayer()`
- `const Card* getPreviousCard()`

- `const Card* getCurrentCard()`
- `void setCurrentCard( const Card*)`

Un jeu doit être affichable avec l'opérateur d'insertion comme dans `cout << game.` Ceci doit imprimer la grille et tous les jouers.

Rules [2]

Concevoir une classe Rules qui contient les méthodes pour vérifier si la sélection d'un joueur est valide.

- `bool isValid(const Game&) retourne vrai si la carte précédente et courrante font paire; faux sinon.`
- `bool gameOver(const Game&) retourne vrai si le nombre de manches est rendu à 7.`
- `bool roundOver(const Game&) retourne vrai s'il ne reste qu'un seul joueur.`

Les points restants sont pour les modes avancés : affichage expert (Expert Display) [2] et règles expert (Expert Rules) [4].