Projet CSI2772A- Automne 2025
Université d'Ottawa
/25

Extension de l'échéance au mercredi 3 décembre 2025 à **minuit** (par groupe d'au plus deux étudiants et une seule soumission par groupe 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++ en ligne si vous désirez.

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

Projet : Jeu de mémoire

Dans ce projet, on vous demande de programmer en C++ une version console du jeu Memoarr! (Jeu de mémoire). Pour plus d'information sur les règles du jeu, consulter les sites Web tels que "The Opinionated Gamer" et "Board Game Geek".

Le jeu de base est décrit ci-bas mais vous devez également programmer deux variantes plus avancées. C'est un jeu pour 2 à 4 joueurs.

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.
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 cas, 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 de la 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.

Nombre de cartes avec 1 rubis	3
Nombre de cartes avec 2 rubis	2
Nombre de cartes avec 3 rubis	1
Nombre de cartes avec 4 rubis	1

Le jeu de base

Les règles du jeu de base sont décrites ci-haut et nous représenterons les cartes avec une matrice 3 × 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 identifiés 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 expert (“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 occupent. 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 du jeu doit permettre la possibilité de jouer les deux modes présentés.

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 de base est représenté par sa classe correspondante :

`Player, Card, Rubis, DeckFactory<C>, CardDeck, RubisDeck, Board, Game, Rules.`

Implémentation

L'interface publique du jeu de base 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 des constructeurs et destructeurs publiques quand ce n’est pas interdit ainsi que des setters/getters. Sinon, Vous pouvez ajouter n’importe quelle méthode privée ou protégée à une classe si nécessaire.

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, 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 indiqués dans les parenthèses ci-bas.

Player (2 POINTS)

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 publiques 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`** `addRubis`**`(const Rubis&);`** augmente le nombre de rubis du joueur avec un nombre donné de rubis.
- **`void`** `setDisplayMode(`**`bool`** `endOfGame);`
- **`Side`** `Player::getSide()` and **`void`** `Player::setSide(`**`Side`**`)` où `Side` est une énumération de classes de {top, bottom, left, right}

Un joueur doit être imprimable à l'écran avec l'opérateur d'insertion, e.g. cout << player. Un exemple d'impression avec `endOfGame` faux pourrait ressembler à ceci :

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

Si `endOfGame` est vrai, ce serait :

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

Card (1.5 POINTS)

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` 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.

L'interface publique de `Card` doit inclure des opérateurs de conversion de type `FaceAnimal` et `FaceBackground`.

Rubis(1 POINT)

Créer un objet de type `Rubis` qui prends une valeur de 1 à 4 rubis. Un `Rubis` doit aussi être affichable par l'instruction cout << rubis.
Un objet de type `Rubis` doit avoir un constructeur privé et donne accès à la classe `RubisDeck` (plus bas) par déclaration **`friend`**.
L'interface publique de `Rubis` doit inclure des opérateurs de conversion de type `int` renvoyant le nombre de rubis.

DeckFactory<C> (2 POINTS)

Concevoir un patron de classe `DeckFactory<C>` comme une classe de 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 (`Rubis`). Le paramètre type <C> est destiné à être un de {Card|Rubis}. Cette classe aura besoin des méthodes suivantes :

- ***void*** *shuffle();* mélange le paquet de cartes. Vous devez utiliser la fonction `std::random_shuffle` de la bibliothèque standard.
- ***C**** *getNext();* retourne la prochaine carte ou le prochain rubis 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 POINTS)

Concervoir une classe `CardDeck` dérivée de `DeckFactory<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` n'a PAS de constructeur public.

RubisDeck (2 POINTS)

Concevoir une classe `RubisDeck` dérivée de `DeckFactory<Rubis>` ayant les mêmes carcatéristiques que `CardDeck`.

Board (2 POINTS)

Concervoir une classe `Board` qui contient un tableau de strings pour afficher le jeu à l'écran. Cette classe aura besoin des méthodes suivantes :

- ***bool*** *isFaceUp(* ***const Letter&, const Number&) const*** ; retourne vrai si la carte à la position donnée est visible. `Letter` et `Number` sont des énumérations. Lancer 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.
- ***Card**** *getCard(****const Letter& , const Number&)*** ; renvoie un pointeur vers la carte à un emplacement donné. Génère une exception de type `OutOfRange` si une combinaison de lettres et de chiffres non valide a été fournie.
- ***void*** *setCard(* ***const Letter&, const Number&, Card*)*** ; met à jour le pointeur vers la carte à un emplacement donné. Génère une exception de type `OutOfRange` si une combinaison de lettres et de chiffres non valide a été fournie.
- ***void*** *allFacesDown();* remets toutes les cartes à l'état caché.

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

Le constructeur de `Board` doit lancer une exception de type `NoMoreCards` s'il n'y a plus de cartes disponibles pour construire un plateau.

Game (2.5 POINTS)

Concevoir une classe `Game` qui encapsule l'état courant du jeu et qui contient une variable d'instance de type `Board`. Cet objet est responsable pour imprimer le jeu.

Cette classe aura besoin des méthodes suivantes :

- ***int*** *getRound();* retourne un numéro entre 0 et 7 correspondant à la manche courante.
- ***void*** *addPlayer(* ***const Player&*** *);* ajoute un joueur à la partie.
- ***Player&*** *getPlayer(****Side****);*
- ***const Card**** *getPreviousCard();*
- ***const Card**** *getCurrentCard();*
- ***void*** *setCurrentCard(* ***const Card**** *);*
- ***Card**** *getCard(****const Letter&, const Number&)*** ; qui appelle la méthode correspondante dans `Board`.
- ***void*** *setCard(****const Letter&, const Number&, Card*);* qui appelle la méthode correspondante dans `Board`.

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

Rules (2 POINTS)

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 courante 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.
- ***const Player&*** *Rules :: getNextPlayer(**const Game&**) ;*

Pseudo Code (3 POINTS pour la boucle du jeu)

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

```
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 rubies

print players with their number of rubies sorted from least to
most rubies

Print overall winner
```

Les points restants sont pour les modes avancés :

- **affichage expert (Expert Display)** **(2 POINTS)**
- **règles expert (Expert Rules)** **(3 POINTS).**

Soumettre votre travail en LIGNE (un seul fichier zip) avant mercredi 26 novembre 2025 à minuit

Directives

- Créez un répertoire que vous nommerez *Projet_GroupeNum comme d'habitude*, où vous remplacerez Num par votre numéro de groupe.
- ✓ Mettez tous les fichiers (Player.cpp, Player.h, … etc) dans votre répertoire compressé *Projet_GroupeNum.zip* pour soumission dans le campus virtuel Brightspace.
- Dans le répertoire *Projet_GroupeNum*, créez un fichier texte nommé *README.txt*, qui devra contenir **les noms des deux étudiant(e)s**, ainsi qu'une brève description du contenu :

Nom étudiant :
Numéro d'étudiant :
Code du cours : CSI2772A

Fraude scolaire :

Cette partie du devoir 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/etudiants-actuels/reglements-academiques-expliques/integrite-inconduite-academique

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

En soumettant ce devoir :

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

2. vous comprenez les conséquences de la fraude scolaire.