ISSAAD Samir *EISE4*
JEBALI Afif

Projet C++ : Smoke

Sommaire

I. Objectif

Créer un programme, sur le thème de la fumée tout en respectant les contraintes suivantes :

- 8 classes
- 3 niveaux de hiérarchie
- 2 fonctions virtuelles différentes et utilisées à bon escient
- 2 surcharges d'opérateurs
- 2 conteneurs différents de la STL
- diagramme de classe UML complet
- commenté le code
- Pas d'erreurs Valgrind
- Pas de méthode/fonctions de plus de 30 lignes

II. Notre projet

Notre jeu consiste en un joueur vs ordinateur. En effet, vous arborez un Personnage de type Eau ou Feu possédant des points d'attaque,des points de vie et des caractéristiques spéciales. Le but est donc de combattre une horde d'ennemis dont le nombre augmente petit à petit et la difficulté aussi (contrôlé par l'ordinateur avec augmentation des pvs, points d'attaque etc...).

En outre, vous possédez un deck de carte (une main) et vous piochez chaque tour une carte qui sera mis dans votre deck, sachant que lorsque le joueur possède 3 cartes en main il est obligé de jouer une de ses cartes qui peuvent soit le booster, soit booster la totalité de ces ennemis. Il faut donc bien choisir à quel moment jouer ses cartes pour ne pas perdre la partie.

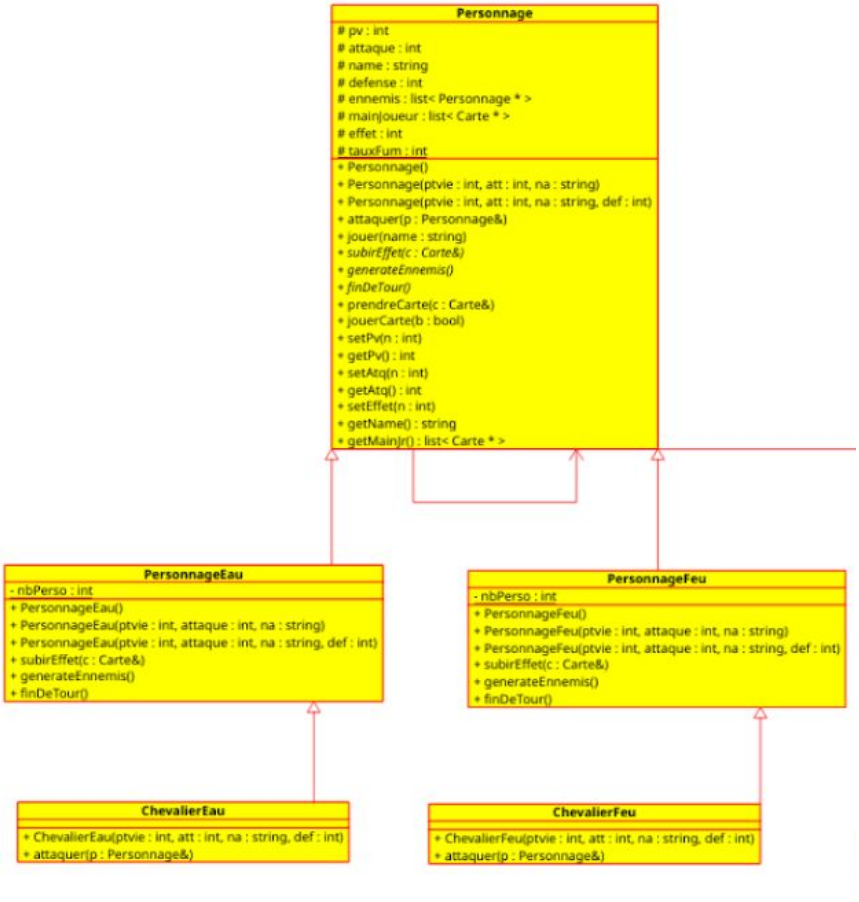
Il existe *3 types de cartes* :

- ➢ **Cartes Feu** : Si le joueur est de type Feu, il gagne un bonus sinon ce sont les ennemis qui gagnent un bonus
    - Basier :
    - Volcano :
- ➢ **Cartes Eau** : Si le joueur est de type Eau, il gagne un bonus sinon ce sont les ennemis qui gagnent un bonus
    - Boisson :
    - Tsunami :
- ➢ **Carte de Fumée** : Augmente ou diminue la jauge de fumée.

En effet, l'élément éminent ici, est la présence d'une jauge de fumée. Elle démarre à 0, et en fonction de la carte jouée, elle peut augmenter ou diminuer. Si la jauge de fumée est de valeur négative, elle donnera des points de vie aux personnages Eau et en fera perdre aux personnages Feu à la fin de chaque tour et inversement si la jauge est positive.

Ainsi après avoir pioché une carte, vous avez le choix de la jouer ou non (en faisant attention de ne pas être submergé par les ennemis, et de devoir jouer une carte malus), et vous devez ensuite attaquer un ennemi.

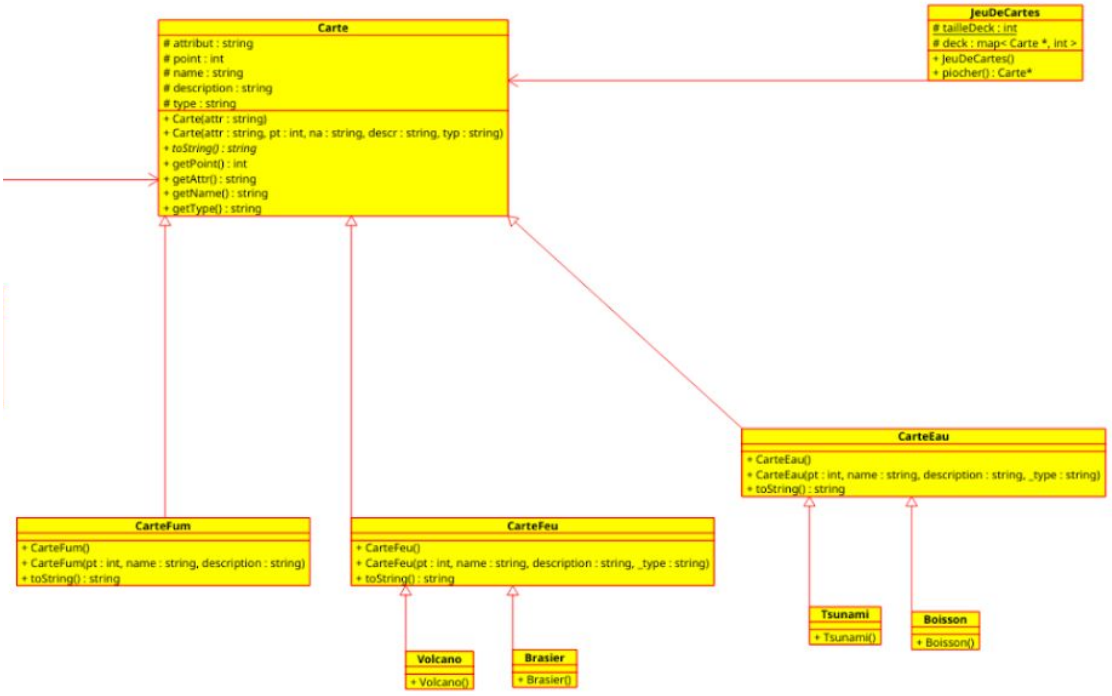
III. Diagramme UML



Notre diagramme UML est composé de 2 grandes classes qui composent le squelette

- **Personnage** : Classe permettant de caractériser le personnage que vous choisissez, ainsi que les personnages ennemis. Elle déclare différentes variables protected tel que point de vie, attaque, nom et défense. Ainsi que 2 listes, une list ennemis de type Personnage qui permet de stocker la liste des ennemis actuels du joueur et l'autre list Carte qui représente la main du joueur. Pour finir chaque joueur possède le même taux de fumée sous la forme d'une variable static qui permet soit de faire gagner des pvs au personnage soit d'en faire perdre en fonction de sa valeur.
  Cette classe compte un total de 3 hiérarchies :
  - 2ème niveau : *Personnage Eau et Feu* qui héritent de Personnage et qui redéfinissent et surchargent des constructeurs et méthodes.
  - 3ème niveau : Chevalier Eau qui hérite de Personnage Eau et Chevalier Feu de Personnage Feu.

- **Carte** : Classe caractérisant les cartes du deck. Déclare une variable attribut, qui définit si la carte est de type eau, feu ou fumée et qui permet ainsi de savoir si la carte a un effet positif ou négatif sur le Personnage choisi. Elle déclare aussi une variable nom, point et description. Cette classe compose un ensemble de 3 hiérarchies au total :
    - 2ème niveau : *Carte Feu, Fumée, Eau* avec chacun un contructeur.
    - 3ème niveau : Carte Volcano, Brasier qui hérite de Carte Feu, et Tsunami, Boisson qui hérite de Carte Eau. Carte fumée ne possède de sous classes car elle influe directement sur le taux de fumée avec un taux pouvant être négatif pour diminuer ce dernier ou positif pour l'augmenter.



Nous avons ensuite créer le classe **JeudeCartes**

- **JeuDeCartes** est une classe qui possède en attribut une map de cartes et d'entier pour représenter la pile de pioche et qui permet de compter le nombre restants de chaque carte afin de déterminer quand un certain type de carte ne se trouve plus dans le paquet ou bien que le paquet de cartes est entièrement vide.
  Elle permet donc globalement de générer le paquet de cartes complet en début de partie, et également de renvoyer une carte au joueur quand celui-ci en pioche une, tout en s'assurant de bien décrémenter la valeur de la map concernée au fur et à mesure que le joueur pioche des cartes.

IV. Procédure d'installation et d'exécution du code :

Utilisation d'un MakeFile afin de compiler le programme, avec une régle "all" et une règle "clean", générant un exécutable nommé smoke.

Bibliothèques utilisées :

- ❏ Ostream
- ❏ IOstream
- ❏ Map
- ❏ string
- ❏ stdlib
- ❏ time
- ❏ list

V. Le code

A. Les constructeurs

Pour la classe PersonnageEau et PersonnageFeu il existe 2 types de constructeurs qui sont

- Le constructeur prenant tous les paramètres nécessaires à la création complète d'un personnage, soit son attaque, son nom… Ce constructeur sert donc à créer le personnage du joueur.
- Le deuxième constructeur quant à lui est un constructeur ne prenant pas de paramètres et qui permet donc de générer les ennemis du joueur tout en gardant le compte du nombre d'ennemi créés.

B. Actions du personnage

```
void Personnage::jouer(string name)
```

Pour commencer le joueur, en lançant la fonction jouer, va pouvoir décider de l'ennemi qu'il souhaite attaquer, puis cette fonction lancera également un appel aux 2 fonctions suivantes.

```
virtual void generateEnnemis() = 0;
```

Cette fonction est appelé après que le joueur ait fini son tour et son attaque afin de générer entre 1 et 2 ennemis qui ont plus ou moins d'attaque et de points de vie en fonction de l'avancement du jeu, ceci est géré par une variable static qui compte le nombre d'ennemis créés.

```
virtual void finDeTour() = 0;
```

La fonction finDeTour permet de retirer l'effet qui a été appliqué aux personnages durant ce tour ainsi que d'appliquer les effets du taux de fumée à tous les personnages, c'est à dire soit gagner ou perdre des pvs.

C. Actions des cartes

```
Carte* JeuDeCartes::piocher()
```

Cette fonction vérifie tout d'abord que le deck ne soit pas vide, on tire ensuite un nombre aléatoire qui permet de tirer une carte de la map représentant le jeu de cartes, on parcourt ensuite la map avec un iterator et un compteur jusqu'à arriver à la carte en question afin de la return et de pouvoir décrémenter la map.

```
void prendreCarte(Carte& c);
```

Cette fonction permet de prendre la carte retourner par la fonction piocher de la classe JeuDeCartes et d'afficher la main du joueur afin que ce dernier puisse observer toutes ses cartes et décider de son action.

```
void jouerCarte(bool b);
```

La fonction jouer carte va simplement laisser au joueur le choix de jouer une carte ou non. Le booléen en paramètre sert à déterminer si la main du joueur est pleine après avoir pris sa carte depuis le jeu de cartes, et permet donc de savoir si le joueur sera forcé de jouer une carte à ce tour ou non.

```
virtual void subirEffet(Carte& c) = 0;
```

Pour finir la fonction subirEffet permet d'appliquer l'effet de la carte jouée pas le joueur, à tous les personnages en jeu (joueur et ordinateur).

D. Les opérateurs

```
friend std::ostream& operator <<(std::ostream & out, Personnage& p);
friend std::ostream& operator <<(std::ostream & out, list<Personnage*> liste);
friend std::ostream& operator <<(std::ostream & out, list<Carte*> listeCarte);
```

Nous avons décidé de redéfinir les opérateurs << afin de faciliter les affichages en console des personnages et des cartes. En effet, nous avons donc redéfini l'opérateur pour un simple personnage afin d'afficher le personnage du joueurs mais également sur des listes de personnages et de cartes afin de pouvoir afficher respectivement les ennemis et la main du joueur.

Ces opérateurs ainsi que la fonction subirEffet() étaient très probablement les fonctions les plus dures et satisfaisantes à réaliser étant donnée qu'il a fallu gérer des conteneurs de manière assez spéciale notamment dans les opérateurs où il a fallu à la fois gérer des itérateurs pointant sur des objets abstraits tout en réalisant un affichage propre avec des retours à la ligne précédentes en affichage console.