LAHAMION Djamaël

SEC 2

TP2 CPP

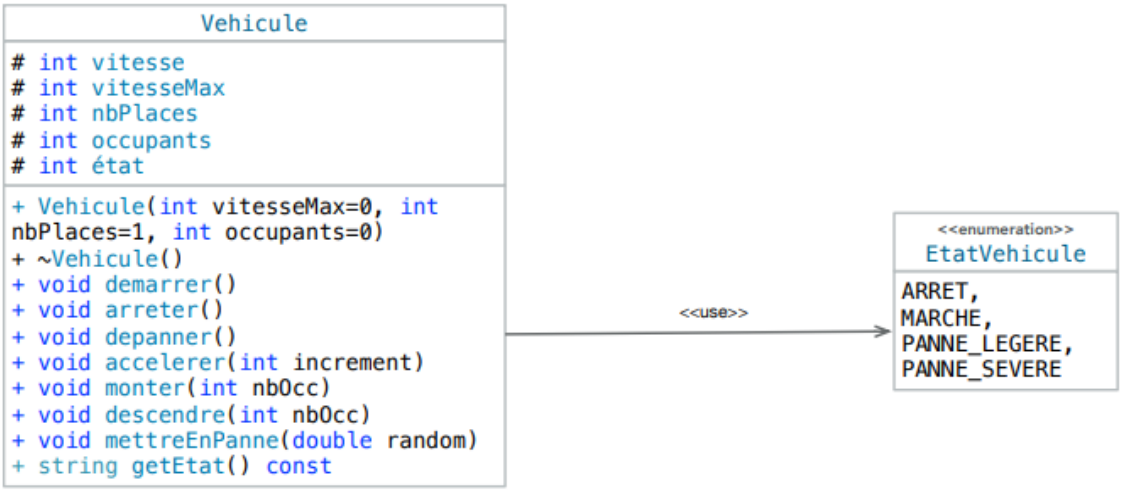
But :

L'objectif de ce TP est d'appliquer les principes de l'héritage multiple et de la création d'exceptions.

I. Réalisations des programmes

1. Création d'un vehicule:

On définit une classe Vehicule qui est décrite par le diagramme de classe ci-dessous :



L'ensemble des méthodes de la classe Vehicule sont déclarées en virtual afin de signaler qu'elles pourront être redéfinies dans ses classes filles.

Nous réalisons les méthodes de la classe Vehicule en gérer les exceptions suivantes :

- si la vitesse devient négative
- si la vitesse dépasse la vitesse maximum.
- si on cherche à démarrer un véhicule déjà en marche ou en panne.
- si on cherche à ajouter un occupant mais que le véhicule est déjà plein.
- si on cherche à faire descendre plus d'occupants qu'il y en avait dans le véhicule.

La classe Lexique comporte différentes méthodes notamment des méthodes qui permettent de surcharger les opérateurs += et -= afin de respectivement permettre de fusionner deux lexiques et de faire la différence de deux lexiques.

Nous obtenons alors, par exemple pour la méthode demarrer, le programme suivant :

Fonction demarrer

Problème : Mettre un vehicule en marche

Spécification

Fonction aucun ← demarrer()

Paramètre

Aucun

Résultat : Aucun

Algorithme

Début :

Si this->etat == MARCHE alors

Jeter l'exception("Le véhicule est déjà en marche !")

Sinon si this->etat == PANNE_SEVERE OU this->etat==PANNE_LEGERE alors

Jeter l'exception("Le véhicule est en panne et ne peut pas démarrer !")

Sinon

this->etat =MARCHE

Fin

```
void Vehicule::demarrer()
{
    if (etat_ == MARCHE)
        throw std::invalid_argument("Le véhicule est déjà en marche !");
    if (etat_ == PANNE_LEGERE || etat_ == PANNE_SEVERE)
        throw std::invalid_argument("Le véhicule est en panne et ne peut pas démarrer !");
    etat_ = MARCHE;
}
```

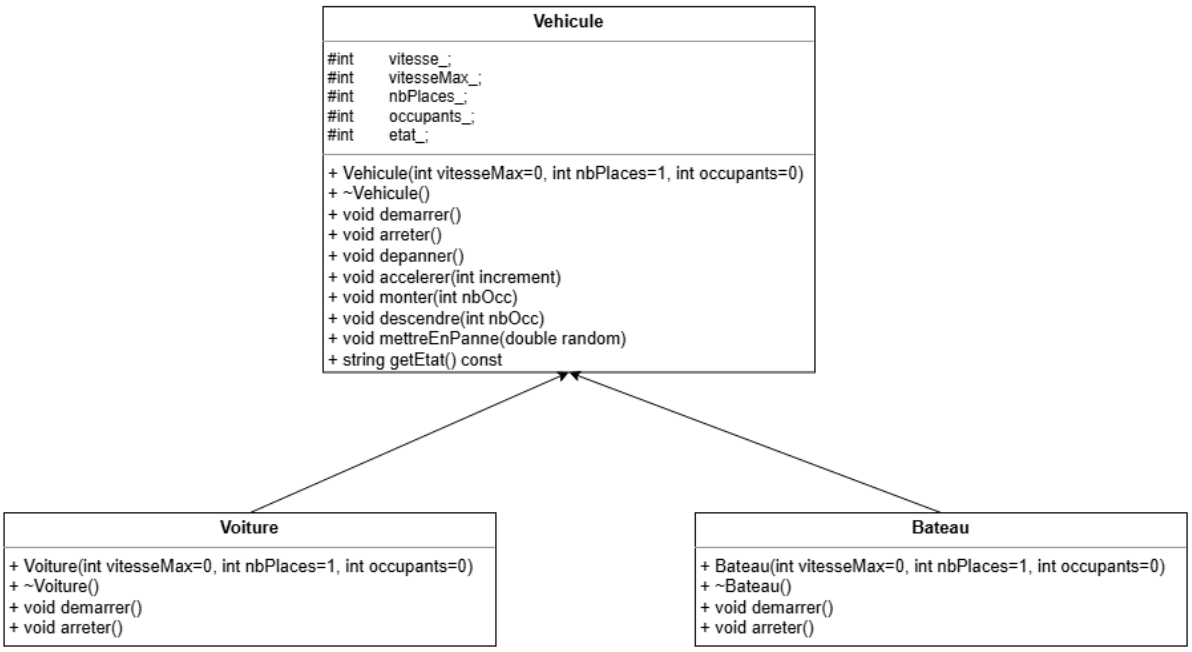
Dans ce code, une exception est jetée lorsque la fonction est lancée alors que la voiture est déjà en marche ou lorsque la voiture est en panne. Dans le cas ou aucune de ces conditions n'est respectée la fonction réalise son traitement normal qui est de mettre l'état de la voiture en marche.

Nous utilisons alors le même principe pour réaliser les méthodes monter, accelerer et descendre.

2. Création d'une voiture et un bateau

Nous réalisons deux classes filles Voiture et Bateau qui héritent de Vehicule.

Les méthodes démarrer et arrêter sont redéfinies pour afficher le type de Vehicule qui demarre ou s'arrete.



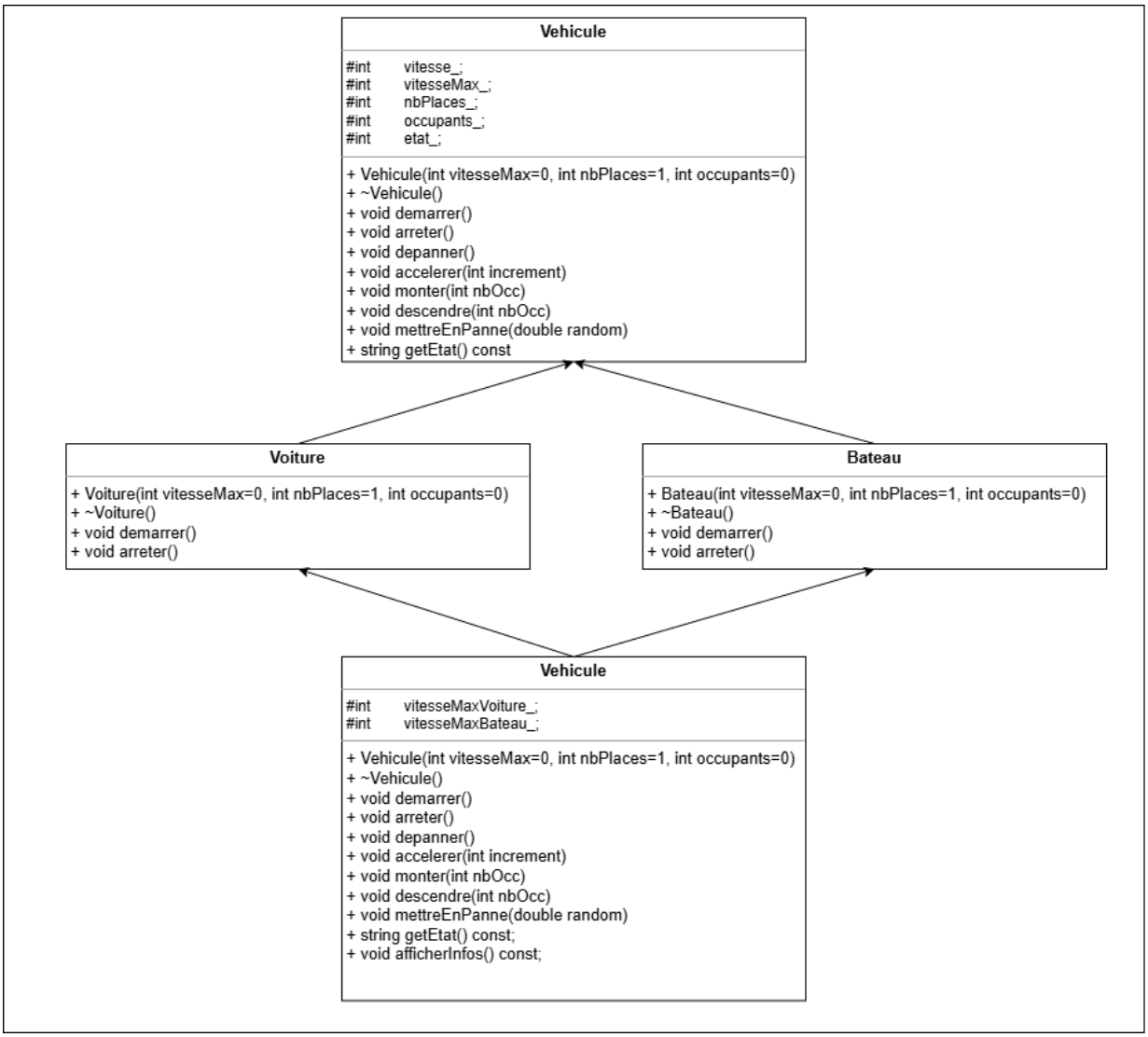
3. Création d'une voiture amphibie

Nous réalisons une classe VoitureAmphibie qui hérite à la fois de Voiture et de Bateau.

Elle implémente en plus une méthode void afficherInfos() qui affiche les infos de la voiture amphibie :

Une vitesse, un nombre d'occupants, un nombre de place et une vitesse max en mode voiture et une vitesse max en mode bateau.

Le diagramme de classe est le suivant :


Vehicule
#int vitesse_;
#int vitesseMax_;
#int nbPlaces_;
#int occupants_;
#int etat_;
+ Vehicule(int vitesseMax=0, int nbPlaces=1, int occupants=0)
+ ~Vehicule()
+ void demarrer()
+ void arreter()
+ void depanner()
+ void accelerer(int increment)
+ void monter(int nbOcc)
+ void descendre(int nbOcc)
+ void mettreEnPanne(double random)
+ string getEtat() const
Voiture
+ Voiture(int vitesseMax=0, int nbPlaces=1, int occupants=0)
+ ~Voiture()
+ void demarrer()
+ void arreter()
Bateau
+ Bateau(int vitesseMax=0, int nbPlaces=1, int occupants=0)
+ ~Bateau()
+ void demarrer()
+ void arreter()
Vehicule
#int vitesseMaxVoiture_;
#int vitesseMaxBateau_;
+ Vehicule(int vitesseMax=0, int nbPlaces=1, int occupants=0)
+ ~Vehicule()
+ void demarrer()
+ void arreter()
+ void depanner()
+ void accelerer(int increment)
+ void monter(int nbOcc)
+ void descendre(int nbOcc)
+ void mettreEnPanne(double random)
+ string getEtat() const;
+ void afficherInfos() const;

II. Test : Jeux d'essais :

I. Vehicule :

Nous testons les différentes méthodes de la classe véhicule et vérifions la gestion des exceptions :

Nous créons une instance de la classe véhicule qui possède 4 places, 2 occupants et une vitesse max de 120.

Nous faisons accélerer le vehicule une première fois de 50 puis une seconde fois de 100.

```
Valeur al├®atoire pour la panne: 0.468734
- Vitesse : 0/120
- Places : 4
- Occupants : 2
- ├ëtat : Arr├¬t

- Vitesse : 50/120
- Places : 4
- Occupants : 2
- ├ëtat : Marche

Erreur : Vitesse maximale atteinte !
```

Lors du second appeler d'accélerer à 100, on dépasse la vitesse max ce qui génère une erreur comme prévu.

Nous créons ensuite une instance de la classe véhicule qui possède 2 places, 2 occupants et une vitesse max de 90.

Nous faisons descendre les deux passagers et en faisons monter 4 ceci génère une erreur car il n y a que deux places dans le vehicule

```
- Vitesse : 0/90
- Places : 2
- Occupants : 2
- ├ëtat : Arr├¬t

Descente de tous les passagers
- Vitesse : 0/90
- Places : 2
- Occupants : 0
- ├ëtat : Arr├¬t

Mont├® de 4 passagers
Erreur : Trop de passagers !
```

Nous testons ensuite la mise en panne des véhicules en appelant mettreEnPanne sur un vehicule et en verfiant que l'état du véhicule a bien changé.

```
- Vitesse : 0/100
- Places : 4
- Occupants : 1
- ├ëtat : Arr├¬t

- Vitesse : 0/100
- Places : 4
- Occupants : 1
- ├ëtat : Panne s├®v├¿re

Erreur : Le v├®hicule est en panne et ne peut pas d├®marrer !
```

Nous testons ensuite de démarrer un véhicule en panne et constatons qu'il y a une erreur car un véhicule en panne ne peut démarrer.

Les méthodes de la classe Véhicule réalisent le comportement souhaité et les exceptions sont bien gérées.

II. Voiture et Bateau

Nous créons une instance de la classe Voiture et une instance de la classe Bateau et les faisons démarrer.

```
Cr├®ation d'une voiture (4 places)
Cr├®ation d'un bateau (2 places)
> La voiture d├®marre.
> Le bateau d├®marre.
Voiture
- Vitesse : 0/100
- Places : 4
- Occupants : 1
- ├ëtat : Marche

Bateau
- Vitesse : 0/100
- Places : 2
- Occupants : 1
- ├ëtat : Marche
```

Nous observons bien les messages indiquant le type de vehicule qui demarre et la mis à jour de l'état des vehicule lors de l'affichage qui suit l'appel de démarrer.

```
> La voiture tombe en panne (Panne s├®v├¿re).
Voiture
- Vitesse : 0/100
- Places : 4
- Occupants : 1
- ├ëtat : Panne s├®v├¿re

Bateau
- Vitesse : 0/100
- Places : 2
- Occupants : 1
- ├ëtat : Marche

Erreur : Le v├®hicule est en panne et ne peut pas d├®marrer !
```

Afin de verifier la bonne gestion des exceptions nous mettons l'instance voiture en panne puis affichant les infos du bateau et de la voiture puis nous faisons démarrer la voiture. Nous observons l'affichage de l'erreur comme précédemment lors du test de la méthode en panne pour l'instance vehicule.

III. Voiture Amphibie :

Nous créons une instance de la classe Voiture amphibie qui hérite de Voiture et Bateau

```
Cr├®ation d'une voiture (2 places)
Cr├®ation d'un bateau (2 places)
Cr├®ation d'une voiture amphibie !
=== Infos Voiture Amphibie ===
Occupants : 1/2
├ëtat : Arr├¬t
Mode voiture ÔåÆ vitesse max : 120
Mode bateau  ÔåÆ vitesse max : 200
```

Nous faisons descendre 1 occupant puis un autre.

```
1 personnes descendent de la voiture amphibie
=== Infos Voiture Amphibie ===
Occupants : 0/2
├ëtat : Arr├¬t
Mode voiture ÔåÆ vitesse max : 120
Mode bateau  ÔåÆ vitesse max : 200
Erreur : Pas autant de passagers ├á descendre !
```

Lors de la première descente, il n y a aucune erreur puis lors de la deuxième descente, on constate une erreur du au fait que l'on demande à faire descendre plus de personnes qu'il n y a d'occupant.

La gestion de l'exception dans la méthode descendre est donc bien fonctionnelle.

III. Conclusion :

Ce TP a permis de mettre en pratique les concepts de structuration des données en C++ pour le traitement de fichiers textes. La réalisation des classes Lexique et LexiqueAvecLignes a montré l'importance du choix des structures de données adaptées, ici std::map pour associer efficacement mots et occurrences, et l'utilisation de tableaux dynamiques pour stocker les numéros de lignes.

Les différents tests réalisés ont confirmé le bon fonctionnement des opérations fondamentales : affichage, vérification de la présence d'un mot, suppression, fusion et différence de lexiques.

Ainsi, ce TP a consolidé les notions de manipulation de conteneur, d'héritage et de surcharge d'opérateurs en C++, tout en sensibilisant à l'importance de choisir des structures de données efficaces pour un traitement optimal des informations.