Sujet 5 : Autour d'une carte de restaurant !

Buts : Héritage simple, polymorphisme, méthodes virtuelles, classes abstraites

Dans ce sujet on s'intéressera à la gestion de la `Carte` d'un restaurant. Une `Carte` est composée d'un ensemble d'`Items`. La solution proposée devra permettre une manipulation facile des `Items` qui composent la carte. Une `Carte` contiendra plusieurs catégories d'`Items`. On se limitera ici à deux catégories spécifiques : les `Items` de type `Boisson` et les items de type `Mets`. Chaque `Item` sera par ailleurs caractérisé par sa `Description`. Afin d'apporter une solution à ce problème, on se propose de travailler de modéliser les classes `Description`, `Item`, `Boisson` et `Mets`.

Attention : dans le cadre du polymorphisme certaines méthodes seront virtuelles.

Autour des Items

L'objectif de cette partie est de modéliser la gestion d'un `Item` (`Boisson` ou `Mets`). Ces `Items` seront ensuite regroupés thématiquement dans une `Carte`.

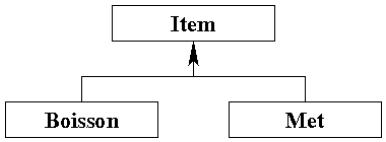

FIGURE 1 – Diagramme de classes : Item, Boisson et Mets.

La classe `Item` est une classe abstraite dont dérivent les classes `Boisson` et `Mets`.

La classe `Item` est caractérisée par son nom, son prix ainsi que ses ingrédients (`Description`). La classe `Boisson` contient en plus l'information sur son caractère alcoolisé. La classe `Mets` est caractérisée en plus par le type du mets (Entree, Plat ou Dessert).
Une `Description` d'un `Item` étant caractérisée par un ensemble d'ingrédients (chaînes de caractères) qui le composent on propose de gérer les problèmes liés à celles-ci dans une classe `Description`.

La classe Description

Une `Description` est composée d'un ensemble d'ingrédients contenus dans un mets (`elements`). L'ensemble des ingrédients pourront être stockés dans un tableau dynamique de `string`.
Ainsi on se propose de définir une classe `Description` comme suit :

```
Description
    unsigned int taille
    string* elements
```

La définition de la classe `Description` est accessible en ligne.

A-t-on besoin de définir le constructeur par copie, le destructeur et l'opérateur = dans la classe `Descritpion` ? Si oui, donnez en la définition.

La classe Item

Un `Item` est caractérisé par ses `ingrédients`, son `nom` et son `prix`. On définit la classe `Item` comme suit :

`Item`

`string nom`
`double prix`
`Description ingredients`

La définition *incomplète* de la classe `Item` est accessible en ligne. **Attention :** la classe `Item` est polymorphe.

0n souhaite définir un constructeur avec arguments dans la classe `Item`. Son prototype est :
`Item(const string &,double,const string & = "")throw (invalid_argument);`
Ce constructeur prend en paramètres le nom, le prix ainsi que la chaîne de caractères permettant d'initialiser les ingrédients initialisée par défaut à la chaîne vide. Définissez ce constructeur.

On souhaite compléter la classe `Item` avec une méthode `affiche` utilisée dans l'opérateur $<<$. Cette méthode pourra être surchargée dans le cadre de l'héritage. Pour un `Item` dont les caractéristiques sont "Pave de saumon roti", 16 et "Pave de saumon roti/beurre de ciboulette/risotto au pesto de roquette" on obtiendra l'affichage suivant :

```
Pave de saumon roti     16
        Pave de saumon roti, beurre de ciboulette, risotto au pesto de roquette
```

Définissez la méthode `affiche`.

Afin de pouvoir comparer des objets polymorphes de type `Mets` ou `Boisson` on souhaite définir un opérateur ==. Cet opérateur prendra en paramètre un objet de type `Item *` et sera surchargé dans les classes dérivées. Au niveau de la classe `Item` cet opérateur testera l'égalité des noms, prix et ingrédients. Définissez l'opérateur ==.

Afin de pouvoir comparer des objets polymorphes de type `Mets` ou `Boisson` on souhaite définir un opérateur <. Cet opérateur prendra en paramètre un objet de type `Item *` et sera surchargé dans les classes dérivées. Cependant, les informations nécessaires à la définition de cet opérateur sont absentes dans la classe `Item`. Déclarez l'opérateur <.

Afin de pouvoir classer les `Items` dans la `Carte` du restaurant en les regroupant par type (Entree, Plat, Dessert ou Boisson) on souhaite disposer d'une méthode `getType` permettant de retourner le type de l'`Item` stocké dans un objet de type `string`. Cependant, les informations nécessaires à la définition de cet opérateur sont absentes dans la classe `Item`. Déclarez la méthode `getType`.

A-t-on besoin d'un destructeur dans la classe `Item` ? Si oui, définissez le.

La classe Mets

La classe `Mets` hérite de la classe `Item`. Cette classe a pour caractéristiques, outre celles de la classe `Item` :

- le type du repas (Entree, Plat ou Dessert).

On propose de définir la classe `Mets` qui hérite de la classe `Item` comme suit :

`Mets`
`string type`

La définition de deux méthodes de la classe `Mets` est accessible en ligne.

Définissez la classe `Mets`.

On souhaite disposer d'un constructeur avec arguments dans la classe `Mets`. Ce constructeur prend en paramètres l'ensemble des informations nécessaires à l'initialisation des champs de l'objet et propose une initialisation par défaut similaire à celle du constructeur de la classe `Item` (*cf.* trace d'exécution disponible en ligne.) Ce constructeur lance une exception lorsque l'utilisateur demande à ce que le champ `type` soit initialisé à une chaîne de caractères différente des chaînes `Entree`, `Plat` ou `Dessert`. Définissez ce constructeur.

Pourquoi est on obligé de surcharger l'opérateur $<$ défini initialement dans la classe `Item` dans la classe `Mets` ? Définissez l'opérateur $<$, sachant que l'ordre entre les différents `Items` est défini comme suit :

- une `Boisson` est inférieure à un `Mets`
- une `Entree` est inférieure à un `Plat`
- un `Plat` est inférieur à un `Dessert`
- en cas d'égalité du `type` des `Items` c'est l'opérande gauche de l'opérateur < qui sera considérée comme inférieure. Définissez l'opérateur <.

La classe Boisson

La classe `Boisson` a pour caractéristiques, outre celles de la classe `Item` :

- l'information sur le caractère alcoolisé ou non de la boisson.

On propose de définir la classe `Boisson` qui hérite de la classe `Item` comme suit :

`Boisson`
`bool alcool`

où `alcool` sera à vrai lorsque la `Boisson` est alcoolisée, faux sinon.

Définissez la classe `Boisson`.

On souhaite disposer d'un constructeur avec arguments dans la classe `Boisson`. Ce constructeur prend en paramètres l'ensemble des informations nécessaires à l'initialisation des champs de l'objet et propose une initialisation par défaut similaire à celle du constructeur de la classe `Item`. Définissez ce constructeur.

Surchargez la méthode `affiche` de la classe mère afin qu'elle puisse afficher en plus l'information sur le caractère alcoolisé de la boisson. Pour une `Boisson` dont les caractéristiques sont : "$Americano\ maison$", 5.2 et $true$ on obtiendra l'affichage suivant :

```
Americano maison        5.2
         (alcoolisee)
```

Surchargez les opérateurs < et == ainsi que la méthode `getType` dans la classe `Boisson`.

Constitution d'une carte

L'objet de cette dernière partie sera de valider les objets préalablement implémentés. Pour ceci, dans la fonction `main` déclarez un `vector` vide contenant les différents éléments présents dans la `carte`. Ces éléments seront des objets polymorphes (`Item` *) pouvant contenir des `Boisson`s ou des `Mets`.

Créez deux objets de chaque type. Insérez les dans le `vector` en utilisant la méthode `push_back`. Affichez la `carte` obtenue.

Testez l'ensemble des méthodes préalablement définies.

Triez votre `carte`, puis affichez la à nouveau.