Université de Paris **Langages à objets avancés**
M1 **Année 2020 – 2021**

TP n° 6

fin du TP précédent

Terminez cet exercice du TP précédent qui reprend les difficultés que vous pouvez rencontrer dans le cas de définitions de classes croisées (l'ordre de leur définition), pose la question de l'usage de pointeurs ou de références, s'assure de la destruction des objets construits, illustre les amitiés de classes.

Exercice 1 On souhaite modéliser un scrutin pour des élections. Pour un scrutin donné, on gère plusieurs bureaux de vote (avec dans chacun une urne). Le nombre d'options de vote possibles change à chaque scrutin : par exemple, pour un référendum, il y a 3 options "oui", "non" et "vote nul ou blanc".

On va avoir une classe `Scrutin` qui contiendra le nombre de bureaux de Vote (et donc d'urnes), le nombre d'options de vote et un tableau de pointeurs sur les urnes. On fera aussi une classe `Urne` qui contiendra une référence sur `Scrutin`, un entier représentant le numéro du bureau de vote (utilisez un compteur « static ») et un tableau d'entier comptabilisant les votes pour chaque option.

De plus, `Urne` aura une méthode `bool voter(int choix)`, qui retournera `false` si l'option est impossible. Vous ajouterez les méthodes nécessaires pour pouvoir obtenir les résultats d'un bureau de vote, celui du scrutin entier et afficher ces résultats.

Indication : Vous avez dû remarquer qu'une urne contient une référence à un scrutin qui lui contient un tableau d'urnes. Si vous essayez de mettre `#include "Scrutin.hpp"` dans le fichier Urne.hpp, et vice-versa, le compilateur refusera.

Pour résoudre le problème, dans le fichier Urne.hpp, on déclare `class Scrutin;` avant la déclaration de la classe `Urne` et on fait un `#include "Urne.hpp"` dans Scrutin.hpp. La déclaration `class Scrutin;` suffit car, dans la déclaration de la classe `Urne`, on n'utilise pas d'autre information que le fait que cette classe existe.

Exercice 2 Écrire les destructeurs des classes `Urne` et `Scrutin`. À la fin d'un scrutin, on détruit les urnes (dans la réalité leur contenu). Vérifiez avec des sorties écran appropriées que l'on détruit bien les urnes.

Exercice 3 Pour éviter qu'on puisse fabriquer des urnes et les rattacher à un scrutin indûment. On va rendre les constructeurs et destructeurs d'`Urne` privées. Pour que `Scrutin` puisse construire des Urnes et les détruire, on va déclarer la classe `Scrutin` amie de `Urne` en écrivant dans la déclaration de classe de `Urne` : `friend class Scrutin;` Cette déclaration d'amitié va permettre à `Scrutin` d'utiliser les membres de `Urne` qui ne sont pas publiques.

Exercice 4 Ajoutez des **const** partout où c'est possible : attributs, méthodes, ...

Exercice 5 Comment éviter que l'on puisse voter après la fin du scrutin et que l'on puisse afficher les résultats avant la fin du scrutin ?

Héritage

Ne faites cette partie que si vous avez bien terminé et maîtrisé ce qui a été vu auparavant.

Rappels de cours :

— Syntaxe pour déclarer une classe B héritant d'une classe A :

```
class B : public A {}
```

— Le constructeur de B doit alors faire appel au constructeur de A :

```
B::B(...) : A {...}, ... {
        ...
}
```

— Les champs/méthodes privés de `A` sont invisibles dans `B`. Les champs/méthodes **protected** de `A` sont visibles dans `B` et toutes les autres sous-classes de `A`.

— La liaison n'est pas dynamique par défaut : la méthode appelée correspond au type déclaré. Pour obtenir une liaison dynamique : la méthode redéfinie doit avoir été déclarée **virtual** dans la classe mère. Dans ce cas n'hésitez pas à ajouter le mot clé **override** à la définition de votre fonction pour obtenir de l'aide du compilateur.

— Si `B` **redéfinit** une méthode `f` de `A` alors il est possible d'accéder à la méthode `f` de `A` via : `A::f()`

— Les caractères private ... et virtual peuvent être modifiés dans les classes filles

— Attention au constructeur de copie. Dans `A a; B b; a=b`, `a` n'est pas un `B`, car il est obtenu par appel du constructeur de copie.

Exercice 6 Implémentez les classes suivantes.

1. Créez une classe `Article` qui contient 2 champs, son nom (`std::string`) et son prix (**double**), ainsi que les accesseurs utiles. Pensez à mettre **const** là où c'est utile et à proposer une méthode d'affichage `string toString()const`.
2. Testez votre classe dans un main qui devra afficher : « `Parapluie, 5e` ».
3. La classe `ArticleEnSolde` hérite d'article et contient en plus une remise (en pourcentage).
   — Ecrivez un constructeur `ArticleEnSolde(nom, prix, remise)`
   — Ecrivez un autre constructeur qui prenne en entrée un article et une remise. Testez !.
   — Ajoutez une valeur de remise par défaut au constructeur précédent. Vous avez maintenant un constructeur par copie. Que pouvez-vous faire pour le tester ?
   — Réécrivez l'accesseur `getPrix()` qui devra renvoyer le prix en tenant compte de la remise.
   — Vos articles soldés s'affichent-ils maintenant avec le bon prix ? (Si non, peut-être que votre méthode d'affichage d'article n'utilisait pas `getPrix()` ? N'oubliez pas également que `getPrix` doit être **virtual** pour que la liaison dynamique fonctionne !)

4. Surchargez les destructeurs pour qu'ils affichent "destruction d'article" (ou d'article en solde). Remplacez votre `main` par le suivant :

```
int main() {
    Article a1("Parapluie", 12);
    cout << a1.toString() << endl;

    ArticleEnSolde a2("Botte", 12, 5);
    cout << a2.toString() << endl;

    ArticleEnSolde a3(a1);
    cout << a3.toString() << endl;

    ArticleEnSolde a4 = a1;
    cout << a4.toString() << endl;

    return 0;
}
```

   Essayez de prévoir les affichages, puis d'exécuter le code. Tout se passe-t'il comme prévu ? Si non, pourquoi ?

5. Définissez la classe `Caddie`, destinée à gérer un tableau d'articles.
   — Munissez la classe `Caddie` d'un constructeur par défaut initialisant le tableau.
   — Ecrivez une méthode `void ajoute(Article &a)` qui prenne en argument une référence et ajoute l'article au Caddie et une méthode d'affichage `std::string toString()const`.
   — Écrivez un test du Caddie dans votre main.
   — Une façon de définir une politique sans fuite de création et destruction d'objet consiste à faire en sorte que la méthode `ajoute` fasse un clone de son argument. Comment faire pour préserver le polymorphisme ?

Exercice 7 On considère les classes suivantes :

```
class A{
        public:
        void f();
        void g();
        virtual void h();
        void k(int i);
        virtual void l(A *a);
        virtual void l(B *a);
};
```

```
class B: public A {
        public:
        void f();
        virtual void h();
        void k(char c);
        virtual void l(B *a);
};
```

On suppose que le code de chacune des fonctions déclarées se résume à un affichage sommaire, sur le modèle :

```
void A::k(int i){
        cout << "A::k(int)" << endl;
}
```

Avec le `main` ci-dessous :

```
int main(){
        A* a = new A;
        B* b = new B;
        A* ab = new B;

        cout << "Appels de f():" << endl;
        a->f();
        b->f();
        ab->f();

        cout<< "Appels de g()" << endl;
        a->g();
        b->g();
        ab->g();

        cout << "Appels de h()" << endl;
        a->h();
        b->h();
        ab->h();
        cout << "Appels de k(--)" << endl;
        a->k('a');
        b->k(2);
        ab->k('a');

        cout << "Appels de l(--)" << endl;
        a->l(a);
        a->l(b);
        a->l(ab);
        b->l(a);
        b->l(b);
        b->l(ab);
        ab->l(a);
        ab->l(b);
        ab->l(ab);

        return 0;
}
```

1. indiquez quelles lignes ne compilent pas et les affichages que produisent les autres ;
2. vérifiez ensuite sur machine.

Exercice 8 On suppose que dans le code suivant chaque fonction `f()` écrite pour une classe `X` affichera `X::f()` lors de son exécution.

Proposez une hiérarchie des classes pour qu'on obtienne le comportement décrit en commentaire. Ecrivez les, puis vérifiez.

```
cout << "---- 1 ----" << endl;
A *a=new A();
a->f(); // A::f()
a->g(); // A::g()
cout << "---- 2 ----" << endl;
A *b=new B();
b->f(); // B::f()
b->g(); // A::g()
cout << "---- 3 ----" << endl;
... *c=new C(); // le type de la variable est à compléter
c->f(); // B::f()
c->g(); // B::g()
cout << "---- 4 ----" << endl;
B *d=new D();
d->f(); // D::f()
d->g(); // D::g()
cout << "---- 5 ----" << endl;
A *e=new E(); // avec E hérite de C
e->f(); // B::f()
...e... -> g(); // ajoutez un cast de e vers B pour obtenir E::g()
```