Université de Paris **Langages à objets avancés**
M1 **Héritage simple**

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.

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

— Attention aux copies/recopies. Dans la séquence `A a; B b; a=b`, `a` n'est pas un `B` : il est modifié par appel de l'opérateur d'affectation.

Exercice 1 On suppose que dans le code suivant chaque fonction `f()`,`g()`, si elle est écrite pour une classe `X`, affichera `X::f()` lors de son exécution.

Proposez sur papier une hiérarchie des classes pour qu'on obtienne le comportement décrit en commentaire. Ecrivez ensuite le code pour vérifier.

```
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(); //  cast de e vers un objet B pour obtenir E::g()
```

Exercice 2 Implémentez les classes suivantes.

1. Créez une classe `Article` qui contient 2 champs privés : un nom (`string`) et un prix (**double**), ainsi que les accesseurs utiles (pensez à mettre **const** partout où cela à du sens). Surchargez l'opérateur d'affichage mais laissez pour le moment les autres méthodes dont on parle dans la forme canonique d'une classe prendre leurs valeurs par défaut.

2. Testez votre classe dans un `main` qui devra afficher un article : `"Parapluie, 5e"`.
3. La classe `ArticleEnSolde` hérite d'article et contient en plus une remise (en pourcentage). Puisqu'on définit une hiérarchie, il se pose la question de savoir quelle accessibilité est permise pour les champs d'`Article` dans la classe fille : disons ensemble qu'ils sont considérés **private**.
   — Ecrivez un constructeur `ArticleEnSolde(nom, prix, remise)`
   — Réécrivez l'accesseur `getPrix()` pour les articles en soldes qui devra renvoyer le prix en tenant compte de la remise.
   — Modifiez l'opérateur d'affichage (n'en écrivez pas un autre) pour qu'il précise : `"Botte, prix origine : 12 , prix affiché 11.4"`. N'oubliez pas que le mot clé **virtual** joue un rôle pour que la liaison dynamique fonctionne.
   — Surchargez les destructeurs pour qu'ils affichent "destruction d'article" (et d'article en solde).
4. Ecrivez un autre constructeur qui prenne en entrée une référence constante vers une article et une valeur de remise remise. Ajoutez au constructeur précédent qu'il y a par défaut une remise de 10%. Testez le.
5. Remarquez que vous avez maintenant un constructeur avec une signature compatible avec celle qu'on utilise d'habitude pour un constructeur de copie, et que ce dernier existe probablement par défaut ... Assurez vous en. [1]
6. On souhaite définir une classe `Caddie`, destinée à gérer un ensemble d'articles.
   — Justifiez que vous ne pouvez pas utiliser `vector<Article>`. (Discutez ce que deviendrait l'ajout d'un article en solde)
   — Définissez la classe `Caddie` qui encapsule un `vector<Article *>`
   — On veut utiliser une méthode `void ajoute(const Article &a)`, mais plusieurs choses devraient vous poser problème.
      — On ne peut pas simplement utiliser et stocker `&a` à cause du **const**. Il apparaît donc qu'on met dans le caddie une version copiée de l'article souhaité. Il vous faut faire en sorte que cette copie soit fidèle (un article ou un articleSolde) : définissez pour cela une méthode de clonage.
      — L'utilisation de pointeurs pose la question de qui gère la vie/mort/copie/affectation de ces objets. Soignez votre destructeur de caddie, et poser vous la question de la copie et l'affectation entre caddies.
   — Ecrivez une méthode `VenteFlash()` qui s'applique à un caddie, et qui solde (éventuellement en plus) tous les articles de 10 pourcent. Assurez vous qu'il n'y ait pas de fuite de mémoire et ne construisez que des copies si elles sont nécessaires.

1. Naturellement, le fait que pour une notation `article1{article2}` il y ait deux sémantiques différentes arbitrées par le sous-type du second article est peut être maladroit. Notre propos était juste de vous montrer que des ambiguités peuvent être causées indirectement (ici par l'ajout d'une valeur par défaut dans une signature). Le plus simple est de renoncer à cette valeur par défaut.

Exercice 3 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. anticipez sur papier quelles lignes ne compilent pas et quels sont les affichages produit si on retire les lignes d'erreur.
2. vérifiez ensuite sur machine.