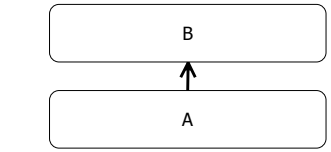
Polymorphisme

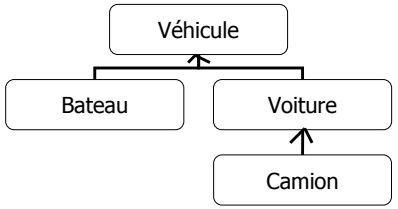
Chapitre 4

- L'héritage
- Les fonctions virtuelles
- Les classes abstraites
- Les interfaces
- Le transtypage

Technique de classification : l'héritage



- Organiser les classes, les classer en catégories
- Développements de nouvelles classes à partir de classes existantes



- A hérite de B
- Bateau hérite de Véhicule, …
- Relation de type « est_un »
  - B est une super-classe/ A est une sous-classe de B
  - B : classe de base / A : classe dérivée

Héritage en C++

class <ident> : <mode d'héritage> <nom_classe>, ... { ... }

- Héritage simple ou multiple
- Un objet de la classe dérivée contient toujours tous les membres de toutes les classes de base
- Accès aux membres de la classe de base modulé par :
  - le mode d'héritage
  - le type d'accès des données de la classe de base
- Les membres privés des classes de base sont présents mais inaccessibles

Mode d'héritage

- Héritage public
  - Relation de type « est-un »
  - Un vêtement est un article
  - Un étudiant est une personne
- Héritage privé et protégé
  - Relation : « est implémenté par »

Héritage public : héritage d'interface

- Conservation de l'interface : le comportement de la classe de base fait partie du comportement de la classe dérivée
- Le plus fréquemment utilisé
- Un objet B est une sorte de A

Statut dans la classe de base	Nouveau statut dans la classe dérivée
public	public
protégé	protégé
privé	inaccessible
inaccessible	inaccessible

Un vêtement est un Article

```
class Article {
    protected:
            string nom;
            int quantité;
            float prix;

    public:
    // constructeur avec paramètres
    Article (char *n, int q, float p)
     : nom(n), quantité(q), prix(p) {}
};
```

```
class Vêtement : public Article {
    protected:
            int taille;
            string coloris;

    public:
    Vêtement (string n, int q, float p,
    int t, string c)
            : Article(n,q,p), taille(t),
    coloris(c) {}

    void solder(float Remise)
            { prix *= (1. – remise);}
};
```


Héritage privé ≡ Héritage d'implantation

class Pile : private Tableau { …};

- Une pile n'est pas un tableau.
- On utilise la classe de base pour implanter la classe dérivée, mais on écrit une nouvelle interface pour la classe dérivée.
- Restriction de la visibilité de certains membres
- Une pile est une sorte de tableau mais les utilisateurs ne sont pas censés le savoir.
- La classe de base est accessible uniquement depuis la classe fille

Une pile n'est pas un tableau mais elle l'utilise

```
class Pile : private Tableau {
    int niveau;
public:
    Pile(int t) : Tableau(t), niveau(0) {}
    bool vide() {return niveau == 0;}
    void empiler(int x)
            { // pas d'accès à Tableau.tab, mais …
            (*this) [niveau++] = x;} //operator[] accessible
    int depiler()
            { return (*this) [--niveau] ;}
};
```

Héritage privé (suite)

```
Pile p (10);
p[i] = x;   //ERREUR: operateur [] privé
p.empiler(x);          // OUI
```

Statut dans la classe de base	Nouveau statut dans la classe dérivée
public	privé
protégé	privé
privé	inaccessible
inaccessible	inaccessible

Héritage protégé

```
class B : protected A {…};
```

- Ressemble à de l'héritage privé
- Mais les détails de l'implémentation (publics et protégés) restent accessibles aux concepteurs de classes petites-filles

Statut dans la classe de base	Nouveau statut dans la classe dérivée
public	protégé
protégé	protégé
privé	inaccessible
inaccessible	inaccessible

Héritage et constructeurs

- Lors de la déclaration d'un objet de la classe dérivée, il y a:
  - appel du constructeur par défaut de la classe de base (ou des constructeurs par défaut selon l'ordre d'héritage)
  - appel du constructeur de la classe dérivée
- Pour éviter l'appel au constructeur par défaut, il faut utiliser des listes d'initialisations

Exemple

```
class Etudiant {
    string nom; …};
class Doctorant : public Etudiant {
    string these;
    public :
Doctorant () { …}            // appel implicite de Etudiant()

Doctorant (string n, string t)
        : Etudiant(n)        //appel explicite du constructeur
        { these = t; }
    };
```

Héritage et destructeurs

- Lors de la destruction d'un objet de la classe dérivée, il y a:
  - Appel du destructeur de la classe dérivée
  - Appel dans l 'ordre inverse d'héritage des destructeurs des classes de bases

Héritage et copie

- Lors de la copie d'un objet de la classe dérivée, il y a:
  - appel du constructeur copie par défaut de la classe ancêtre (ou des constructeurs copie par défaut selon l'ordre d'héritage)
  - appel du constructeur copie de la classe dérivée
- Pour éviter l'appel au constructeur copie par défaut de la classe ancêtre, il faut utiliser des listes d'initialisations pour appeler le constructeur copie de la classe ancêtre

Exemple

```
class Etudiant {
      string nom; …};
class Doctorant : public Etudiant {
      string these;
      public :
Doctorant (const Doctorant & copie) { …}     // appel implicite de
    Etudiant(const Etudiant&)

Doctorant (const Doctorant & copie)
            : Etudiant(copie)  //appel explicite du constructeur copie
            { these = t; }
      };
```


Redéfinition d'une méthode héritée

- Une fonction d'une classe dérivée peut porter le même nom et les mêmes paramètres qu'une fonction de la classe de base
- La fonction appelée dépend de l'objet appelant déterminé statiquement à la compilation
- La méthode héritée est alors masquée, mais l'accès est encore possible

```
void Vêtement::affiche() {
        Article:: affiche(); // accès à la méthode masquée
       cout << coloris; cout << taille; }
```

Redéfinition

```
// Vêtement hérite d'Article
int main() {
    Article a;
    Vêtement v;
    a.affiche();  // appel de Article::affiche()

    v.affiche();   // appel de Vêtement::affiche()
}
```

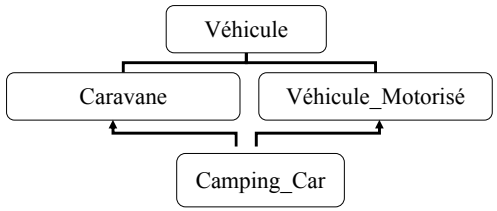

Redéfinition / Surcharge

```
class Article {
    public :
        boolean compare(Article a);
};
class Vêtement : public Article {
    public :
        boolean compare(Article a);    // redéfinition
        boolean compare(Vêtement v);    // surcharge
};
```


Attention !

```
class Base {
    public:
        int f (int i);
        int f (float r); };
class Derivee : public Base {
    public:
        int f (int i); };
int main() {
    Derivee d;
    int j = d.f(3.1415); // ≡ à int j = d.Derivee::f(3);
};
// Le fonction f est redéfinie dans la classe dérivée, sa nouvelle définition masque
    toutes les définitions héritées de la classe de base
```


Héritage multiple



Ordre d'appel des constructeurs

- dépend de l'ordre de déclaration de l'héritage

```
class CampingCar : public Caravane, public VéhiculeMotorisé{
    public :
        CampingCar (int P, int S, int PUI) : VéhiculeMotorisé(PUI),
                                             Caravane(P, S) {}
};
```


Plusieurs chemins d'accès à Vehicule

```
class Véhicule { public :
    int vmax; …};
class Caravane : public Véhicule {
        protected :
        int places, surface;
        … };
class VéhiculeMotorisé : public
  Véhicule {
        protected :
            int puissance;
        … };
```

```
void main() {

Camping_Car C;
C.vmax = 1;        //Erreur
C.VehiculeMotorisé::vmax = 2;
        //OK
C.Caravane::vmax = 3;      //OK
}
```


Héritage virtuel : classe virtuelle

- Pour éviter la duplication d'informations, une classe de base peut être partagée si ses classes dérivées en héritent de manière virtuelle.

```
class Caravane : virtual public Véhicule { …};
class VéhiculeMotorisé : virtual public Véhicule {…};
```

- Véhicule devient une classe de base virtuelle.

Appel du constructeur de la classe virtuelle

- avant les autres constructeurs
- dans le constructeur de Camping_Car, appel de :
  - Véhicule()
  - liste d'initialisation

```
Camping_Car Eriba(2,10,90);
// vmax =100 (valeur par défaut), nbplaces = 2,
// surface = 10, puissance = 90
```

Conflits de membre

- Deux membres compatibles avec la forme d'appel peuvent être déclarés dans deux classes ancêtres
- Solution : préfixer le nom du membre par le nom de la classe qui le contient

```
class Spectacle : public Lumière, public Son {
    public :
        float & luminosité() {return Lumière::intensité;}
        float & volume() {return Son::intensité;}
        };
```


Polymorphisme d'héritage

```
class Personne {
        protected : string nom;
        public : Personne(string);
                void afficher();
};
class Etudiant : public Personne {
        private : float moyenne;
        public : Etudiant(string, float);
                void afficher();
};
class Prof : public Personne {
        private : int nbPublication;
        public : Prof(string, int);
                void afficher();
};
```

```
int main() {
    Personne *p;
    Personne bob(« Bob »);
    p=&bob;
    p->afficher();
        //Personne::afficher()
    Etudiant tom(« Tom »,12.);
    p=&tom;
    p->afficher();
        //Personne::afficher()
    Prof pierre(« Pierre »,20);
    p=&pierre;
    p->afficher();
        //Personne::afficher()
}
```


Appel polymorphe

- Si l'on définit la fonction **Personne::afficher** comme étant une fonction virtuelle, l'appel **p->afficher** devient **polymorphe**.
- Sa signification change selon les circonstances

```
class Personne {
        protected : string nom;
        public : Personne(string);
            virtual void afficher();
};
```


Fonctions virtuelles

```
class Personne {
    protected : string nom;
    public :
    Personne(string);
    virtual void afficher(){ // virtual nécessaire
            cout << nom;
        }
};
class Etudiant : public Personne {
    private : float moyenne;
    public :
    Etudiant(string, float);
 /*virtual */ void afficher() {// virtual implicite
            cout << nom;
            cout << moyenne;
        }
};
```

- Les fonctions virtuelles spécifient dans une classe de base des fonctions qui seront redéfinies par les classes dérivées selon un même profil (types des arguments, sans type résultat)
- Chaque classe dérivée pourra donc définir son implantation de la fonction virtuelle
- La redéfinition d'une fonction virtuelle est elle-même gérée comme une fonction virtuelle

Liaison dynamique

- Le compilateur attend l'exécution du programme pour savoir quelle fonction appeler selon l'objet réel.
- Dans le cas d'un pointeur sur une classe qui appelle une méthode virtuelle, c'est la classe de l'objet pointé réellement par ce pointeur qui détermine la fonction appelée.
- Une méthode virtuelle est liée dynamiquement à un objet (donc elle ne peut être static)
- => **polymorphisme** (implanté par la virtualité)

mot clé virtual => liaison dynamique :
Etudiant::afficher() est appelée
PAS de mot clé virtual => liaison statique :
Personne::afficher() est appelée

Définition du polymorphisme

Une fonction appelée sur des objets de type différents répondra différemment (car elle est codée différemment selon le type de l'objet)

Le polymorphisme est réalisé grâce :
- aux fonctions virtuelles
- à l'héritage
- aux pointeurs : un pointeur sur une instance d'une classe de base peut également pointer sur toute instance de sous-classe

Utilisation du polymorphisme …

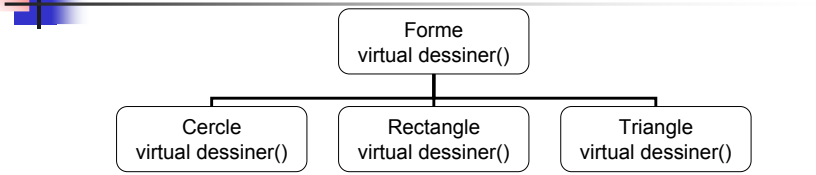
```
class Article {
    float prixHT;
    public :
    virtual float prixTTC() {
            return (prixHT * 1.196);
    }
};
class Alimentation : public Article {
    public :
    float prixTTC() {
            return (prixHT * 1.055);
    }
};
```


… dans un tableau de pointeurs

```
int main() {
Article * commandes [500];
float totalTTC;
…
commandes[13] = new Article («
    Machine », 100., 50);
commandes[14] = new Alimentation («
    Fromage », 10., 50);
…
for (int i=0; i<500; i++)
    totalTTC +=
        commandes[i]->prixTTC();
}
```

- La variable commandes [i] pointe différents types d'articles
- Ces types d'articles doivent :
  - être membres de la même hiérarchie
- La même instruction se comporte différemment, selon le type réel de l'objet et non pas selon son type statique

Dessiner une liste de différentes formes géométriques en utilisant un conteneur hétérogène



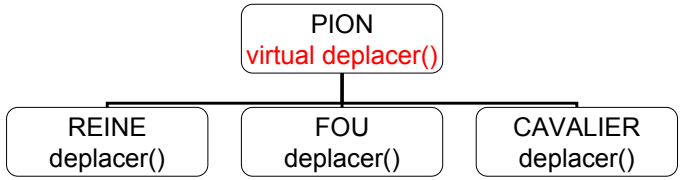

```
class UneClasse {
        list <Forme *> l;
    public :
        void dessinerTout() {
            for (list <Forme *>::iterator it = l.begin(); …)
                (*it)->dessiner();
    }
};
```


Exemple 3 : Au lieu d'un switch

```
void deplacer (int typePion) {

    switch (typePion) {
            case REINE :
                // deplacement reine
                break;
            case FOU :
                // deplacement d'un fou
                break;
            case PION :
                // deplacement d'un pion
                break;
    }
```


Exemple 3 : héritage + fonction virtuelle




```
void deplacer (Pion * typePion) {
    typePion->deplacer();
}
```


Un objet est polymorphe

- S'il a au moins une méthode virtuelle
- Il conserve des informations sur son type dynamique
- Il est manipulé uniquement à l'aide de pointeurs ou de référence

```
class Base {
        public :
        virtual ~ Base() {}
        };
```

En java tous les objets sont polymorphes.

Objet polymorphe et surcharge
Tout dépend …
du type dynamique de l'objet courant

```
class Article {
    public :
virtual bool compare(Article *a);
 };

class Vêtement : public Article {
    public :
bool compare(Article *a);
bool compare(Vêtement *v);
};
```

```
int main() {
Article *pta1 = new Article;
Article *pta2 = new Vetement;
Vetement *ptv1 = new Vetement;

pta1->compare(pta2); //???
pta2->compare(pta1);        //???
pta2->compare(pta2); //???
pta2->compare(ptv1); //???
}
```

Et du type statique de l'objet courant pour le paramètre …

Destructeurs virtuels

- Un destructeur virtuel sert à détruire l'objet réellement pointé (pour éviter les fuites mémoire).
- Dans une classe de base, il faut définir un destructeur virtuel, même si le code est vide.
- Un destructeur peut être virtuel mais pas virtuel pur, le destructeur des classes de base étant automatiquement appelé.
- Il n'y a pas de constructeur virtuel

Fuites mémoire évitées

```
class X { int *p;
    public :
    X() { p = new int [2];}
   virtual ~X() { delete [] p;}
};
 class Y : public X { int * q;
    public :
          Y() {q = new int [1024];}
          ~Y() { delete [] q;}
};
```

```
int main() {
    for (int i = 0; i<8; i++)
          { X *r = new Y;
          delete r;
          }
}
```

Si ~X() n'est pas déclaré virtuel :

Appels de X(), Y() et ~X()

-> 1024 int restent alloués à chaque itération

Sinon

appels de X(), Y(), ~Y() et ~X()

Implémentation des fonctions virtuelles

- Tout objet d'une classe contenant une fonction virtuelle possède un pointeur **vptr** vers une table nommée **vtable** qui contient les adresses des fonctions virtuelles de la classe

```
class A { int a;
    public :
    A() ;
    virtual ~A() ;
    virtual void f();
    virtual void g();
    virtual void h();
    void k();
};
```

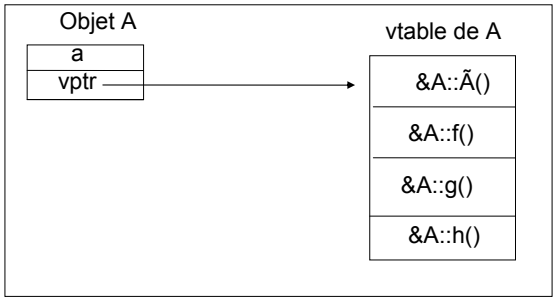
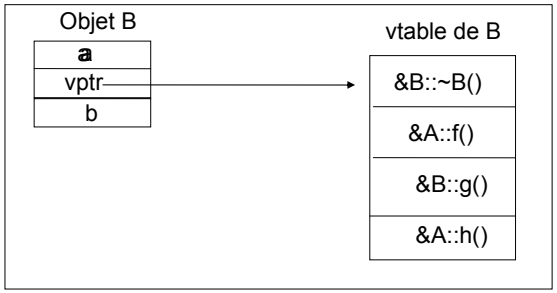


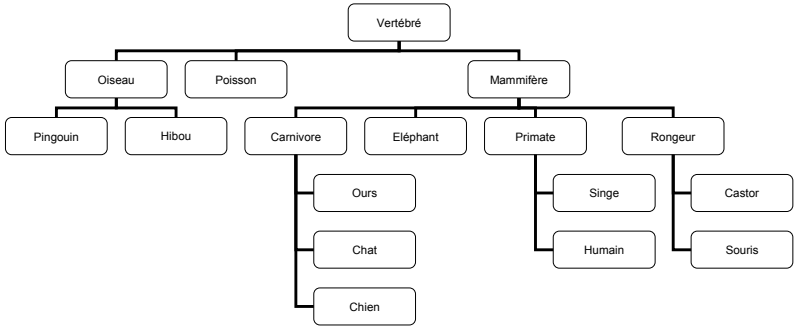
Table virtuelle de B

- Soit B une classe dérivée de A.
- B redéfinit la fonction virtuelle g

```
class B : public A {
    int b;
    public :
    B() ;
    virtual ~B() ;
    virtual void g();
    };
```



Classes abstraites et vertébrés



Caractérisation de fonctions virtuelles

Fonctions spécifiques **à une classe**

Poisson.nager(),
hibou.voler(),
chien.creuser()

Fonctions spécifiques (virtual)

à toutes les sous-classes d'une classe

vertebre.manger(),
mammifère.teter(),
primate.peler()

-> Implémentation spécifiques de ces méthodes dans les sous-classes

Fonctions virtuelles pures

1) Si une fonction virtuelle est redéfinie dans toutes ses sous-classes
2) Et on n'a pas besoin de la coder dans sa classe de base

Alors la fonction virtuelle est pure.

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

Classe abstraite

- contient au moins une fonction virtuelle pure
- ne peut pas produire d'objets
- interface standard pour toutes ses classes dérivées.

```
class Vertebre {
        public:
        virtual void manger() = 0;
};
```

- Les classes dérivées doivent implémenter toutes les méthodes abstraites

On ne peut pas créer d'objet abstrait

```
Vertebre e;          // est impossible
Vertebre tab[2];     // est impossible
```

- MAIS déclaration possible de pointeurs et de références vers des objets de classes dérivées.

```
Vertebre * titi = new Oiseau(" titi ");
Vertebre & roMinet = * new Chat(" roMinet ");
```


Copie d'un objet polymorphe

```
class Vertebre {
    public:
    virtual void manger() = 0;
    virtual Vertebre* clone() const = 0;
};

class Chat : public Vertebre {
    public:
    virtual void manger() ;
    virtual Vertebre* clone() const {
        return new Chat(*this);}
        };
```

- Un constructeur copie ne peut être virtuel
- Ajout d'une fonction virtuelle pure dans la classe de base pour le clonage

Les interfaces en C++

- Une interface **est une classe** comme les autres contenant uniquement des fonctions virtuelles pures

```
class IForme {
    public:
        virtual void afficher () = 0;
        virtual float surface () = 0;
        virtual ~IForme() {};
};
```

Hiérarchie d'interfaces

```
class IDeuxDim : public IForme {
    public:
    virtual float périmètre () = 0;
};
class ITroisDim : public IForme {
    public:
    virtual float volume () = 0;
};
class Carre : public IDeuxDim {
    public:
     void afficher ();
     float surface ();
    …
};
```

Les fonctions virtuelles pures comme «afficher » et « surface », doivent être définies dans les classes qui ne sont pas abstraites

=> Inutile de les coder dans les interfaces IDeuxDim et ITroisDim

RTTI –Run Time Type Information

- Accès dynamique au type d'un objet

```
#include <typeinfo>

void printClassName(Base *b) {
        cout << typeid(*b).name << endl;
 }
```

- Possible uniquement si b est polymorphe (sinon renvoie le type statique)
- Objet b doit être désigné par un pointeur ou une référence

Opérateur *typeid*

⇒ Renvoie un **const type_info&**

- Une seule instance de type_info par classe.
- On en obtient une référence par appel à

```
const type_info& typeid(expression)
const type_info& typeid(nom_type)
```


Quelques méthodes de type_info

```
class type_info {
    public:
    const char *name() const;
    int operator==(const type_info&) const;
    int operator!=(const type_info&) const;
    boolean before(const type_info&) const;
…
};
```

a.before(b) renvoie true si a est une super classe de b

Comparaison de types dynamiques

```
Vêtement v;
Article *pta= &v;

typeid(*pta) == typeid(Vêtement) // renvoie vrai
typeid(pta) != typeid(Vêtement *) // renvoie vrai
```

En effet :

```
cout << typeid(*pta).name() ;    // affiche Vêtement
cout << typeid(pta).name() ;          // affiche Article *
```


A ne pas faire

```
void move (Pion *p) {

  if (typeid(*p) == typeid(Reine))
          p->Reine::move();

  else if  (typeid(*p) == typeid(Fou))
          p->Fou::move();

  else if  (typeid(*p) == typeid(Pion))
          p->Pion::move();
}
```

Utiliser le polymorphisme

```
class Pion {
      public :
      virtual void move();
       // ou abstraite (=0)
      };
```

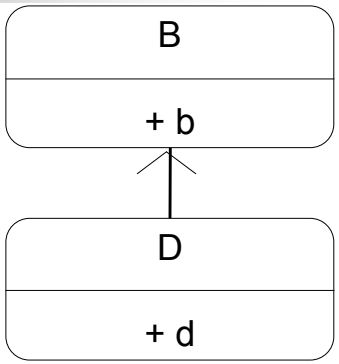

Transtypage

- Conversion d'un pointeur sur un objet en un pointeur sur un sous-objet
- Conversion de B* vers D*
  - a un sens uniquement si B* pointe un objet de classe D ou d'une classe dérivée de D
- Conversion de D* vers B* : conversion standard
  - généralisation de D* vers B*

Exemple

```
class B {
public :
void b() {cout << "debug : fonction b
    ";}
virtual ~B() {}  // b est polymorphe
};

class D : public B {
public:
void d() {cout << "debug : fonction d
    ";}
};
```



Transtypage dynamique : dynamic_cast

```
int main() {
    B* ptb = new D;

    ptb->b();  // ok B::b()

    ptb-> d();  // NON car d() n'est
    pas un service de B
}
```

```
//  opérateur de cast du C
// sans vérification
((D*)ptb)-> d();
// ok D::d()


// transtypage dynamique
// si B est polymorphe
dynamic_cast<D*>(ptb) -> d();
// ok D::d()
```


L'opérateur dynamic_cast

- Opérateur fiable car il vérifie la validité du transtypage
- Vérification via l'identification dynamique de type
  => Il faut utiliser des objets polymorphes
- Si le transtypage se passe mal, retour de :
  - NULL si le type cible est un pointeur
  - exception **bad_cast** pour une référence d'objet

Affectation des objets polymorphes

```
Article *pta , *ptv;
 pta = new Vetement ;
 ptv = new Vetement;
*pta = *ptv;          //sinon affectation entre adresses
```

appel de Vetement::operator=(Article&)
car le type dynamique de pta est un Vetement *

=> Redéfinition de l'affectation

Opérateur d'affectation dans la classe mère

```
class Article {
   protected : int prix;
public:
   virtual Article & operator= (const Article &article) {
        if (this != &article) {
                prix = article.prix;}
        return *this;
};
```

Redéfinition dans la classe dérivée

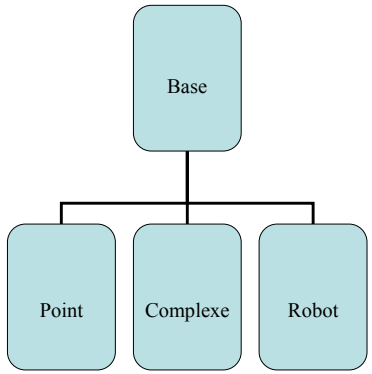
```
class Vêtement : public Article {
    int taille;
public:

virtual Article & operator= (const Article &article) {
    if (this != &article) {
            Article::operator=(article);

            // on ne peut le faire que si Article est polymorphe
            const Vetement &v = dynamic_cast<const Vetement &>(article);
            taille = v.taille;
                }
    return *this;
};
```


TD : **Ensemble d'objets hétérogènes**

- Le but de cet exercice est de créer une classe EnsHeter qui représente un ensemble d'objets hétérogènes stockés dans un tableau.
- Contrainte : les objets hétérogènes doivent dériver de la même classe de base Base.



TD (suite)

1. La classe **EnsHeter** offre les fonctions suivantes :
    1. Constructeur
    2. Copie constructeur
    3. Opérateur d'affectation
    4. Destructeur
    5. Ajout d'un objet à l'ensemble
    6. Afficher tous les objets.
    7. Fonction de test d'appartenance d'un objet à l'ensemble via la fonction virtuelle pure operator== de la classe Base et redéfinition dans les sous-classes.