Algorithmique et Programmation 2 : Généralisation et Héritage

Définitions

- ***Héritage :***

  mécanisme de ***transmission de propriétés*** (méthode, attributs, etc ...) d'une classe vers une sous-classe (ou classe ***dérivée***)

Héritage

- ***L'héritage*** permet d'exprimer une relation entre classes du type ***« est une sorte de »***.

- Exemple :
  Si on considère la classe Personne, on peut définir une classe dérivée Etudiant

Hiérarchie d'héritage

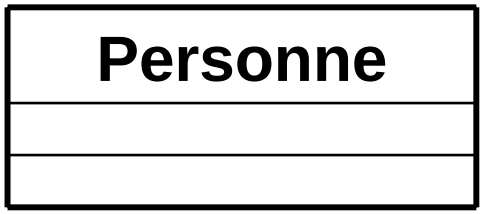
- ***L'héritage*** permet ainsi de déclarer des ***classes très générales***, puis progressivement de les ***« spécialiser ».***
- Cette ***spécialisation*** est aussi une ***« dérivation »*** de la classe de base.
- On parle alors de ***hiérarchie d'héritage***

Hiérarchie d'héritage

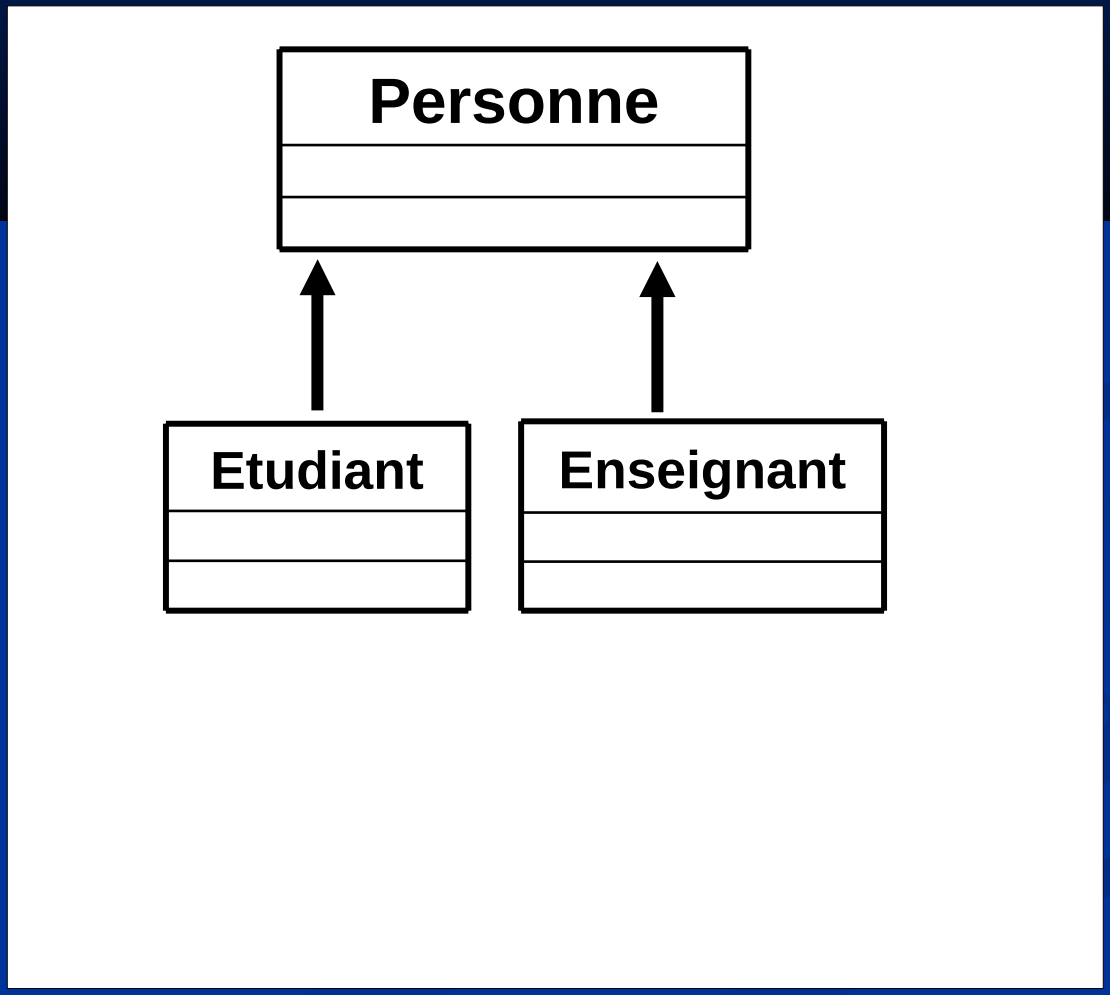
- Une classe dérivée hérite des méthodes de la classe dont elle dérive (mais pas toujours)
- Une classe dérivée peut redéfinir une méthode
- Si une classe dérivée redéfinit une méthode, c'est cette méthode redéfinie qui sera appelée pour un objet de cette classe, et non pas la méthode originale de la classe supérieure

Hiérarchie d'héritage : Exemple

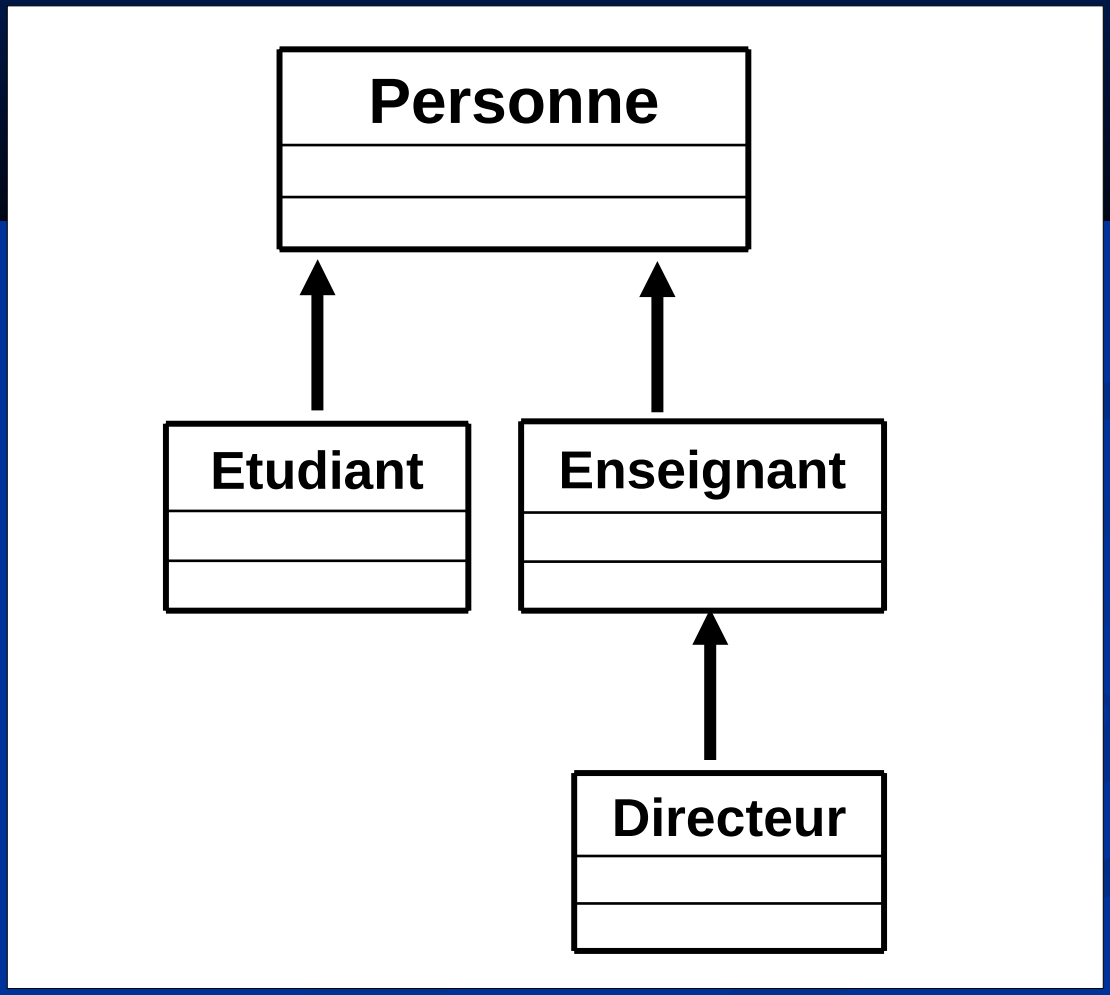
Hiérarchie d'héritage : Exemple



Hiérarchie d'héritage : Exemple



Hiérarchie d'héritage : Exemple



Héritage VS. Agrégation

- ***L'héritage*** permet d'exprimer une relation entre classes du type ***« est une sorte de »***.

  Exemple : un Etudiant est une Personne
- ***L'agrégation*** indique une relation du type ***« fait partie de »*** :

  Exemple : un Etudiant fait partie d'un Groupe

Héritage VS. Agrégation

- Il ne faut pas confondre l'héritage avec l'agrégation (ou la composition)
- Il pourrait être tentant d'utiliser l'agrégation (ou la composition) au lieu de l'héritage
- Du point de vue technique, le résultat peut paraître équivalent, mais il s'agit de deux concepts tout à fait différents

Héritage Implémentation : Exemple

```
class A {
  public:
     A();
     ~A();
     void f();
     void g();
     int n;
  private:
     int p;
};
```

```
// La classe B herite de la classe A
class B : public A {  // heritage public
  public:
     B();
     ~B();
     void h();
  private:
     int q;
};
```

Héritage Implémentation : Exemple

```
class A {
  public:
     A();
     ~A();
     void f();
     void g();
     int n;
  private:
     int p;
};

// La classe B herite de la classe A
class B : public A {  // heritage public
  public:
     B();
     ~B();
     void h();
  private:
     int q;
};

int main() {
  B b;
  b.h();     // OK, fct membre de B
  b.f();     // OK, fct membre (public) heritee de A
  b.n = 3;   // OK, donnee membre (public) heritee de A
  b.p = 5;   // ERREUR, donnee membre (private) heritee de A
             // interdit aussi dans une fonction membre de B...
}
```

Héritage : Exercice

- ***Exercice :*** Indiquez les lignes qui poseront problème dans le code ci-dessous :

```
// Fichier B.cxx
void B::h() {
  f();
  A::f();
}
```

```
// Fichier A.cxx
void A::g() {
  h();
  B::h();
  this->f();
}
```

```
class A {
  public:
     A();
     ~A();
     void f();
     void g();
     int n;
  private:
     int p;
};
```

```
// La classe B herite de la classe A
class B : public A {  // heritage public
  public:
     B();
     ~B();
     void h();
  private:
     int q;
};
```

Héritage : Exercice

- ***Exercice :*** définir les interfaces Employe et gérant

- Ecrire l'interface de la Classe employée :
  - Attributs : un nom et un salaire
  - Constructeurs : défaut, 2 paramètres
  - Méthodes : modifier le salaire, obtenir le salaire et obtenir le nom
- Ecrire l'interface de la classe Gérant (qui hérite d'employé)
  - Attribut : nombre d'employés qu'il supervise
  - Constructeur : défaut;
  - Méthode : ajouter un employé à superviser , obtenir le nombre d'employés que le gérant supervise

Héritage : Exercice

- ***Exercice :*** définir les corps Employe et gérant

- Depuis le main il faut pouvoir faire :
  - Employe e;
  - Gerant a;
  - Employe e2("nicholas",4000);
- Faire divers essais sur l'appel des méthodes (obtenirSalaire, obtenirNom...)
  - Est-il possible de faire Gerant a2 ("Marc",2000) ?
  - Modifiez le code en conséquence

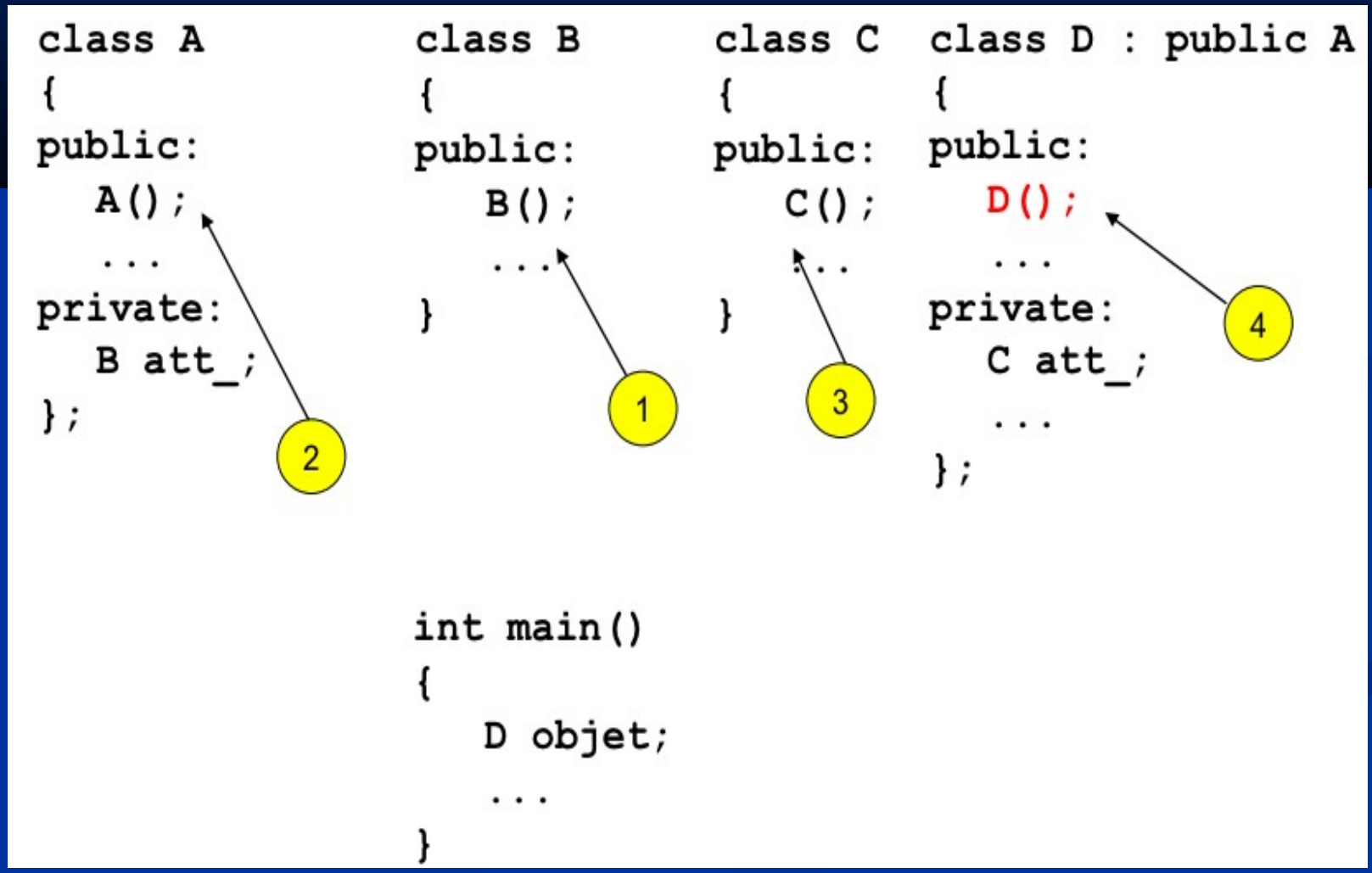
Constructeur de la classe dérivée

- Il est important de noter qu'avant d'appeler le constructeur par défaut d'une classe dérivée, le constructeur par défaut de la classe de base est d'abord appelé
- Si on veut capter le constructeur de la classe de base pour appeler plutôt un constructeur par paramètre, il faut l'appeler dans la liste d'initialisation

Constructeur de la classe dérivée

- Lors de la création d'un objet d'une classe dérivée, les constructeurs sont appelés dans l'ordre suivant:
  - Les constructeurs des attributs de la classe de base
  - Le constructeur par défaut de la classe de base (si aucun constructeur de la base n'est appelé dans la liste d'initialisation du constructeur de la classe dérivée)
  - Les constructeurs des attributs de la classe dérivée
  - Le constructeur de la classe dérivée
- Les destructeurs sont appelés en ordre inverse des constructeurs

Ordre d'appel des constructeurs



Ordre d'appel des constructeurs

```
class A
{
public:
   A();
   A(int x);
   ...
private:
   B att_;
};

A::A(int x)
   : att_(x)
{
}


int main()
{
   D objet(3,2);
   ...
}
```

On construit d'abord cet attribut, mais en utilisant son constructeur qui prend un paramètre entier

```
class D : public A
{
public:
   D();
   D(int p, int q);
   ...
private:
   C att_;
   ...
};

D::D(int p, int q)
   : att_(p), A(q)
{
   ...
}
```

Ordre d'appel des constructeurs

```
class A
{
public:
   A();
   A(int x);
   ...
private:
   B att_;
};

A::A(int x)
   : att_(x)
{
}


int main()
{
   D objet(3,2);
   ...
}
```

On exécute ensuite le constructeur de A

```
class D : public A
{
public:
   D();
   D(int p, int q);
   ...
private:
   C att_;
   ...
};

D::D(int p, int q)
   : att_(p), A(q)
{
   ...
}
```

Ordre d'appel des constructeurs

```
class A
{
public:
   A();
   A(int x);
   ...
private:
   B att_;
};

A::A(int x)
   : att_(x)
{
}


int main()
{
   D objet(3,2);
   ...
}
```

On construit cet attribut, mais en utilisant son constructeur prenant un entier en paramètre

```
class D : public A
{
public:
   D();
   D(int p, int q);
   ...
private:
   C att_;
   ...
};

D::D(int p, int q)
   : att_(p), A(q)
{
   ...
}
```

Ordre d'appel des constructeurs

```
class A
{
public:
   A();
   A(int x);
   ...
private:
   B att_;
};

A::A(int x)
   : att_(x)
{
}


int main()
{
   D objet(3,2);
   ...
}
```

On exécute finalement le constructeur de D

```
class D : public A
{
public:
   D();
   D(int p, int q);
   ...
private:
   C att_;
   ...
};

D::D(int p, int q)
   : att_(p), A(q)
{
   ...
}
```

Destructeur dans une hiérarchie

Supposons qu'une classe B hérite d'une classe A

- Lorsqu'une instance de B est détruite, l'instance de A correspondante est aussi détruite automatiquement
- Ordre de destruction : classe dérivé puis la super-classe

Appel des méthodes de la classe de base

- Soit B une sous-classe de A
- Si on considère que tout objet de type B est aussi de type A, une méthode de B devrait pouvoir appeler une méthode de la classe A
- On fait cela en préfixant par « A:: » la méthode appelée dans la classe B

Appel des méthodes de la classe de base (exercice)

- Reprenez les classes Employe et Gerant
- Redéfinissez dans Gerant la méthode
  - Double obtenirSalaire() const;
  - Cet obtenirSalaire ajoutera systématiquement une prime de 1000 euros au gérant appelant la méthode obtenirSalaire() de la classe Employe.

Le code suivant doit afficher 4000 et 3000

```
Employe e("nicholas",4000);
 Gerant a("Marc",2000);
 std::cout << e.obtenirSalaire() << std::endl;
 std::cout << a.obtenirSalaire() << std::endl;
```

Appel des méthodes de la classe de base

Soit B une classe de base et B::fct() une de ses méthodes

- Pour une classe D dérivée de B, il y a trois possibilités pour la méthode D::fct() :
  - D::fct() étend B::fct() en appelant la méthode B::fct() et en réalisant de nouveaux calculs à partir du résultat obtenu
  - D::fct() est une implémentation complètement indépendante de B::fct()
  - D::fct() n'est pas définie dans D, ce qui implique qu'on appellera directement la méthode B::fct() lorsqu'on voudra exécuter la méthode fct() sur un objet de classe D (on dit que la méthode fct() est héritée)

Visibilité des membres d'une classe

- Attribut/méthode **private** : n'est accessible que par les méthodes de la classe qui les définies.
- Attribut/méthode **protected** : accessible depuis les classes dérivées.
- Attribut/méthode **public** : accessible à tout le monde.

Visibilité des membres d'une classe

La classe B peut hériter de la classe A de trois façons différentes :

class B : <*controle d'accès*> A {

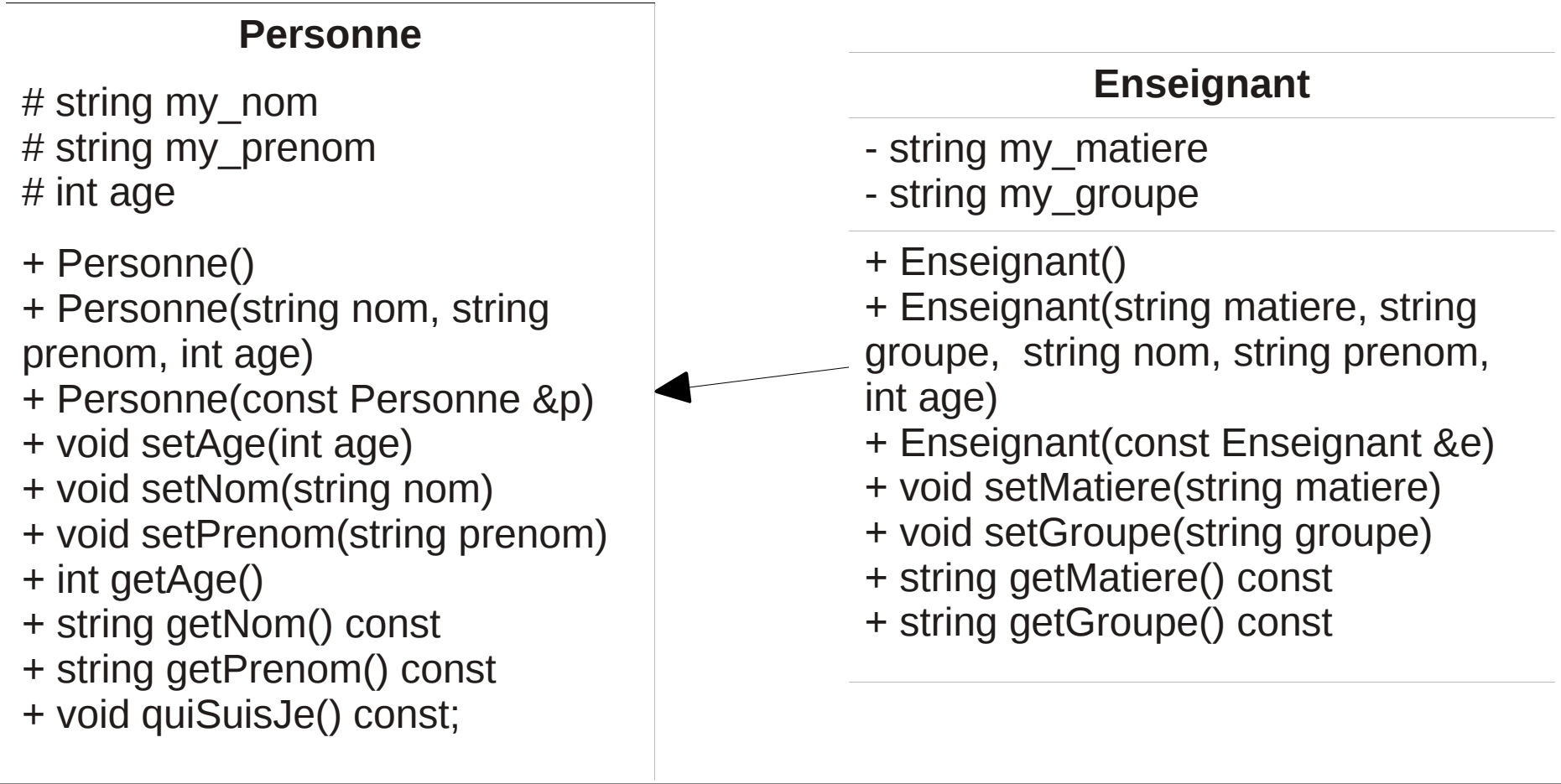
...

}

- <*controle d'accès*> = **public** : contrôle d'accès sans changement
- <*controle d'accès*> = **protected** : les données public deviennent protected
- <*controle d'accès*> = **private** : les données public et protected deviennent private

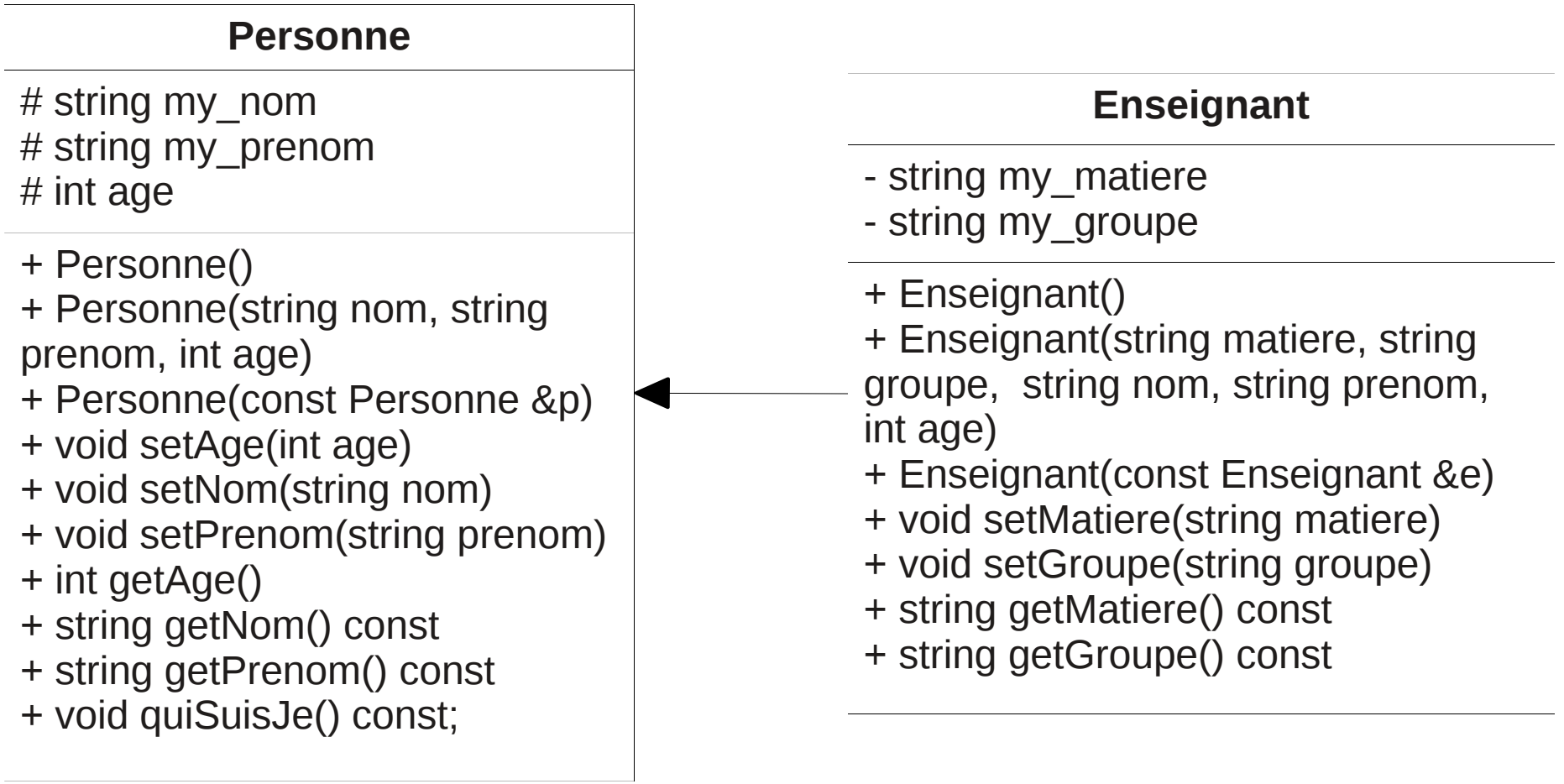
Hiérarchie d'héritage : Exercice

Exercice : Ecrivez l'interface de la classe Enseignant respectant le diagramme ci-dessous



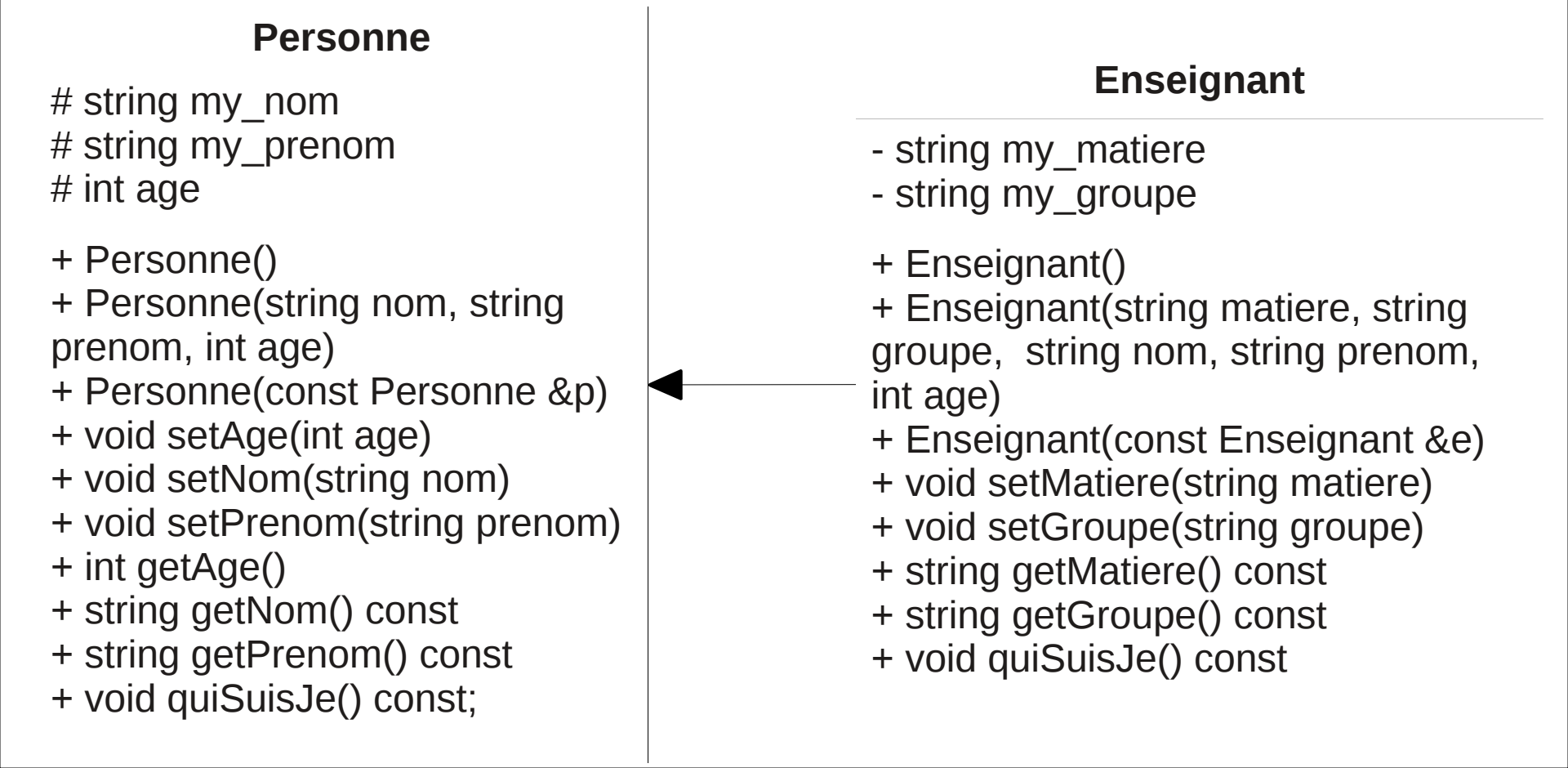
Hiérarchie d'héritage : Exercice

Exercice : Ecrivez le corps de la classe Enseignant respectant le diagramme ci-dessous



Hiérarchie d'héritage : Exercice

Exercice : Ajouter une méthode void quiSuisJe() dans la classe Enseignant



Polymorphisme

Polymorphisme = utilisation homogène d'objets distincts dans une même hiérarchie d'héritage

- le comportement de ces objets reste ***spécifique*** (au type d'instanciation).

- Puissant mécanisme ***d'abstraction*** : possibilité de traiter plusieurs formes d'une classe comme si elles ne faisaient qu'une ...

Polymorphisme

- ***Exercice*** : Si on fait:

```
Enseignant nj;
Personne p;
vector <Personne> v;
v.push_back(nj);
v.push_back(p);`
v[0].quiSuisJe();
v[1].quiSuisJe();
```

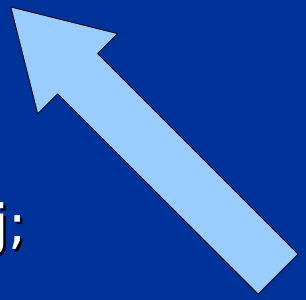
- Appelle 2 fois le quiSuisJe de Personne
- Le vecteur s'attend à recevoir un objet de la classe Personne
- On lui passe un objet Personne et un Enseignant
- Enseignant sera **converti** en oubliant tout ce qui fait de lui un Enseignant
- Donc le quiSuisJe() sera celui d'Enseignant

Polymorphisme

- ***Exercice*** : Si on fait:

```
void afficher(Personne p){
    p.quiSuisJe();
}
int main(){
  Enseignant nj;
  Personne p;
  afficher(nj);
  afficher(p);
}
```

Constructeur par copie de Personne



Passage en paramètre d'un objet dérivé...

Mais on appelle toujours la méthode de Personnes

Polymorphisme

- ***Avec des pointeurs*** :

```
Enseignant *e1=new Enseignant;
Personne *p1=new Personne;
vector <Personne *> v;
v.push_back(e1);
v.push_back(p1);
v[0]->quiSuisJe();
v[1]->quiSuisJe();
```

- Vecteur de pointeurs donc pas de conversion d'objets
- Mais se sont des Personne *
- Donc même résultats

Polymorphisme

Ce que l'on voudrait faire comprendre au compilateur :

- Je déclare un pointeur de type Personne*, mais il se pourrait que tu reçoives un pointeur sur un objet d’une classe dérivée. J’aimerais que que dans ce cas tu appelles la méthode de cette classe dérivée plutôt que celle de la classe Personne.

ou encore

- Je déclare une référence de type Personne, mais il se pourrait que tu reçoives un objet d’une classe dérivée. J’aimerais que dans ce cas tu appelles la méthode de cette classe dérivée plutôt que celle de la classe Personne

Polymorphisme

Solution :

```
virtual void quiSuisJe() const;
```

Polymorphisme : Exemple

```
int main(){
  Etudiant etd("Jonathan", "Dubois", 22);
  Enseignant ens("Romain", "Bourqui", 28);

  Personne * pers;

  pers = &etd;
  pers->quiSuisJe(); // utilise la méthode quiSuisJe() de la classe Etudiant

  pers = &ens;
  pers->quiSuisJe(); // utilise la méthode quiSuisJe() de la classe Enseignant

  return EXIT_SUCCESS;
}
```

Polymorphisme : Exemple supplémentaires

```
int main(){
  Etudiant etd("Jonathan", "Dubois", 22);
  Enseignant ens("Romain", "Bourqui", 28);

  Personne pers;

  pers = etd;
  pers.quiSuiJe(); // utilise la méthode quiSuisJe() de la classe Personne

  Personne * pers2;
  pers = &ens;
  pers2->quiSuisJe(); // utilise la méthode quiSuisJe() de la classe Enseignant

}
```

Attention

- si une méthode est déclarée virtuelle dans une classe, elle le sera automatiquement automatiquement dans toutes les classes qui dans toutes les classes qui en dérivent
- Pour éviter toute confusion, on redéclare la méthode virtuelle dans les classes dérivées
- L’avantage de cela est qu’on n’aura pas besoin d’aller consulter la classe de base pour savoir si une méthode est virtuelle

Polymorphisme et méthode héritée

- Supposons une fonction virtuelle f1() définie dans une classe de base A
- Supposons maintenant une classe B dérivée de A qui ne redéfinit pas la fonction f1()
- Selon le principe de l'héritage, la fonction f1() dans B est héritée de la classe A
- Alors, comment se comporte le polymorphisme dans ce cas?

Polymorphisme et méthode héritée

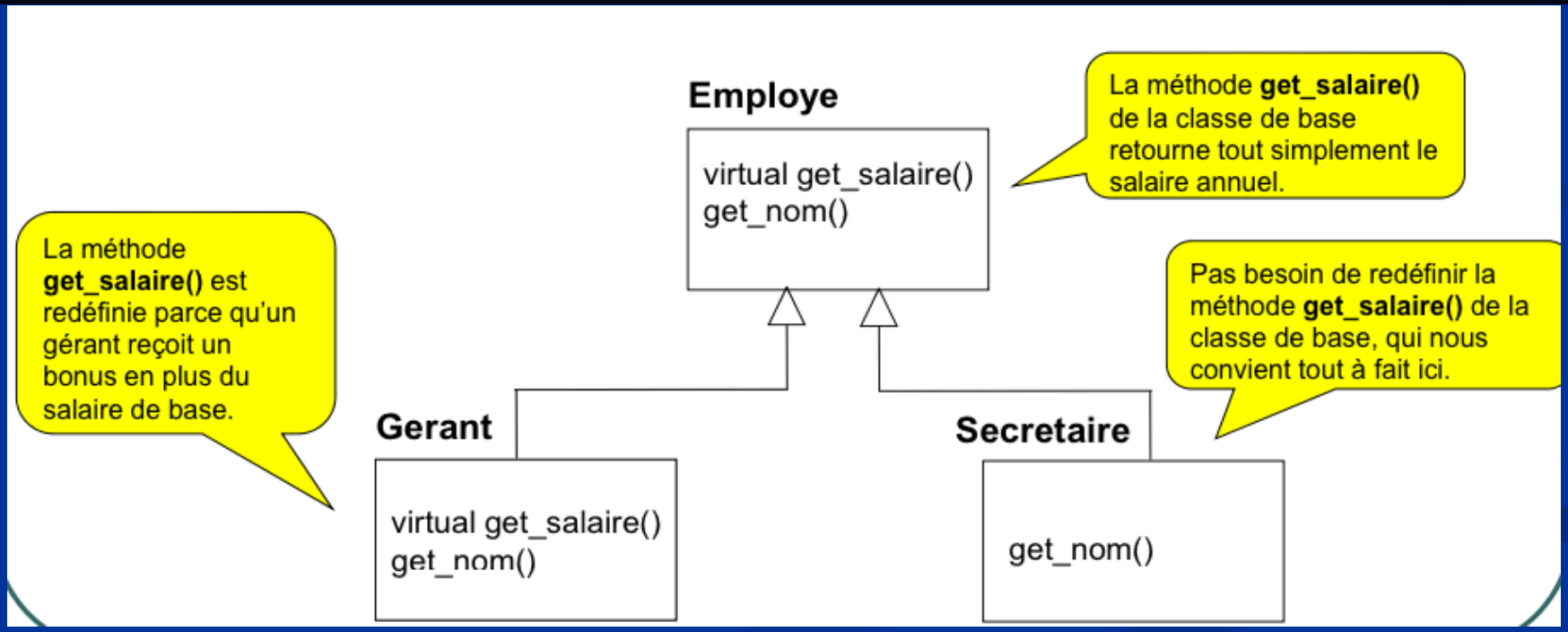
```
int main()
{
   vector< A* > v;
   v.push_back(new A());
   v.push_back(new B());
   ...

   v[1]->f1();
}
```

On ajoute deux pointeurs dans le vecteur, dont un qui pointe vers un objet de la classe dérivée.

`f1()` étant virtuelle, on doit appeler la méthode de la classe **B**. Mais comme la méthode n'est pas définie dans **B**, on appelle la methode `f1()` de la classe **A** dont elle hérite.

Polymorphisme et méthode héritée

- même si une fonction est virtuelle, on est libre de la redéfinir ou non dans une classe dérivée



Polymorphisme et méthode héritée

```
class Employe
{
 public:
    Employe(string nom = "", double salaire = 0.0);
    virtual double get_salaire() const;
    string get_nom() const;
    ...
 private:
     string nom_;
     double salaire_;
};

Employe::Employe(string nom, double salaire)
   : nom_(nom), salaire_(salaire)
{
}
string Employe::get_nom() const
{
   return nom_;
}
double Employe::get_salaire() const
{
   return salaire_;
}
```

Polymorphisme et méthode héritée

```
class Gerant : public Employe
{
 public:
    Gerant(string nom = "", double salaire = 0.0, double bonus = 0);
    virtual double get_salaire() const;
    string get_nom() const;
    ...
 private:
    double bonus_;
};

Gerant::Gerant(string nom, double salaire, double bonus)
   : Employe(nom,salaire), bonus_(bonus)
{
}
string Gerant:: get_nom() const
{
return ("Le nom du (de la) gérant(e) est" + Employe::get_nom());
}
double Gerant::get_salaire() const
{
   return Employe::get_salaire()*(1 + bonus_/100);
}
```

Polymorphisme et méthode héritée

```
class Secretaire : public Employe
{
 public:
    Secretaire(string nom = "", double salaire = 0.0);
    string get_nom() const;
    ...
 private:
    ...
};

Secretaire::Secretaire(string nom, double salaire)
   : Employe(nom,salaire)
{
}
string Secretaire::get_nom() const
{
   return ("Le nom du (de la) secrétaire est" +
             Employe::get_nom());
}
```

Polymorphisme et méthode héritée

```
bool bien_paye(const Employe& employe)
{
    return (employe.get_salaire() > 75000.0);
}

int main()
{
    vector< Gerant > v;
    Gerant gerant;

    ...
    gerant = Gerant("Juliette", 40000.0, 10);

    if (bien_paye(gerant)){
       v.push_back(gerant);
    }
    ...
}
```

Passage par référence: on exécutera donc méthode **get_salaire()** de la classe réelle, puisque cette méthode est déclarée virtuelle.

Lorsqu'on traitera un objet de la classe **Gerant**, c'est la méthode de la classe **Gerant** qui sera appelée.

Polymorphisme et méthode héritée

```
int main()
{
  vector< Employe* > v;
  v.push_back(new Secretaire("Anatole", 35000.0));
  v.push_back(new Gerant("Bernadette", 80000.0, 10));
  ...

  v[1]->get_salaire();
}
```

Il s'agit ici du type **statique** du pointeur.

Le type réel de l'objet pointé par ce pointeur n'est pas de la classe **Employe**, mais plutôt de la classe **Gerant**, qui est le type **dynamique**.

Polymorphisme et méthode héritée

- Le type statique est évidemment déterminé lors de la compilation
- Le type dynamique, lui, est déterminé seulement lors de l'exécution (c'est une liaison dynamique)
- Par exemple, dans la boucle suivante, on ne peut pas savoir avant l'exécution le type réel de l'objet pointé par v[i]:

```
vector <Employe*> v
…
Cout << v[0]->getSalaire(); //LIAISON DYNAMIQUE
Cout << v[0]->getNom(); //LIAISON statique (méthode non
                                                virtuelle)
```

Polymorphisme et méthode héritée

- Par défaut, la liaison est statique, c'est-à-dire qu'on appelle toujours la méthode de la classe indiquée dans la déclaration de l'objet (cette méthode pouvant bien sûr être héritée)
- Dans le cas d'un pointeur ou d'une référence, cela signifie qu'on appelle la méthode de la classe qui correspond au type du pointeur ou de la référence
- Le seul cas où on effectue une liaison dynamique, c'est lorsqu'on a un pointeur (ou une référence) sur un objet d'une classe dérivée, alors que ce pointeur (ou référence) a été déclaré du type de la classe de base
- Dans ce cas, si la méthode a été déclarée virtuelle, on appellera la méthode de la classe dérivée et non pas celle du pointeur ou de la référence

Polymorphisme et méthode héritée

- Supposons que la classe Employe a une méthode print() définie de la manière suivante

```
void Employe::print(ostream& out) const
{

...
cout << get_nom() << endl;//Méthode non virtuelle.
cout << "Salaire: " << get_salaire() << endl;;//Méthode  virtuelle.
}
```

Polymorphisme et méthode héritée

```
void Employe::print(ostream& out) const
{

   out << get_nom() << endl;
   out << "Salaire: " << get_salaire() << endl;
}
int main()
{
   Employe employe("Michel", 20000.0);
   employe.print(cout);
}
```

C'est la méthode de la classe **Employe** qui est appelée.

C'est la méthode de la classe **Employe** qui est appelée.

Polymorphisme et méthode héritée

```
void Employe::print(ostream& out) const
{

   out << get_nom() << endl;
   out << "Salaire: " << get_salaire() << endl;
}
int main()
{
   Gerant employe("Michel",20000.0, 10);
   employe.print(cout);
}
```

C'est encore la méthode de la classe **Employe** qui est appelée.

C'est la méthode de la classe **Gerant** qui est appelée.

Polymorphisme et méthode héritée

```
void Employe::print(ostream& out) const
{

   out << get_nom() << endl;
   out << "Salaire: ""<< get_salaire() << endl;
}
int main()
{
   Secretaire employe("Michel", 20000.0);
   employe.print(cout);
}
```

On appelle la méthode de la classe **Employe**.

C'est la méthode de la classe **Employe** qui est appelée, puisqu'elle est héritée par la classe **Secretaire**.
Si la méthode virtuelle avait été redéfinie dans la classe **Secretaire**, c'est celle-ci qu'on aurait utilisée.

Destructeurs virtuels

Soit les classes suivantes :

```
class Vehicule
{
 public:
   Vehicule();
   ~Vehicule();
   virtual void avancer();
   ...
};

class VehiculeMotorise : public Vehicule
{
 public:
   VehiculeMotorise();
   ~VehiculeMotorise();
 private:
   Moteur* moteur_;
};
```

Destructeurs virtuels

Soit les classes suivantes :

```
VehiculeMotorise::VehiculeMotorise()
{
    moteur_ = new Moteur();
    ...
}

VehiculeMotorise::~VehiculeMotorise()
{
    delete moteur_;
    ...
}
```

Cette classe doit avoir un destructeur puisqu'il faut désallouer le pointeur qui a été alloué par le constructeur.

Destructeurs virtuels

Quel destructeur sera appelé lors de la désallocation de v?

```
int main()
{
   Vehicule* v = new VehiculeMotorise();
   ...
   delete v;
   ...
}
```

Destructeurs virtuels

- Le destructeur de la classe Vehicule n'a pas été déclaré virtuel
- Ce n'est donc pas le destructeur de la classe VehiculeMotorise qui sera appelé, même si l'objet appartient en fait à cette adresse
- Le pointeur moteur_ ne sera donc pas désalloué et on aura une fuite mémoire
- Il est donc important de déclarer virtuel le destructeur de Vehicule

Destructeurs virtuels

```
class Vehicule
{
 public:
    Vehicule();
    virtual ~Vehicule();
    virtual void avancer();
    ...
};

class VehiculeMotorise : public Vehicule
{
 public:
   VehiculeMotorise();
    ~VehiculeMotorise();
 private:
    Moteur* moteur_;
};
```

Destructeurs virtuels

•A partir du moment où une classe peut être dérivée et que vous ne savez pas à l'avance comment elle sera ensuite utilisée, vous devez traiter ce problème. Or il n'existe que deux solutions :

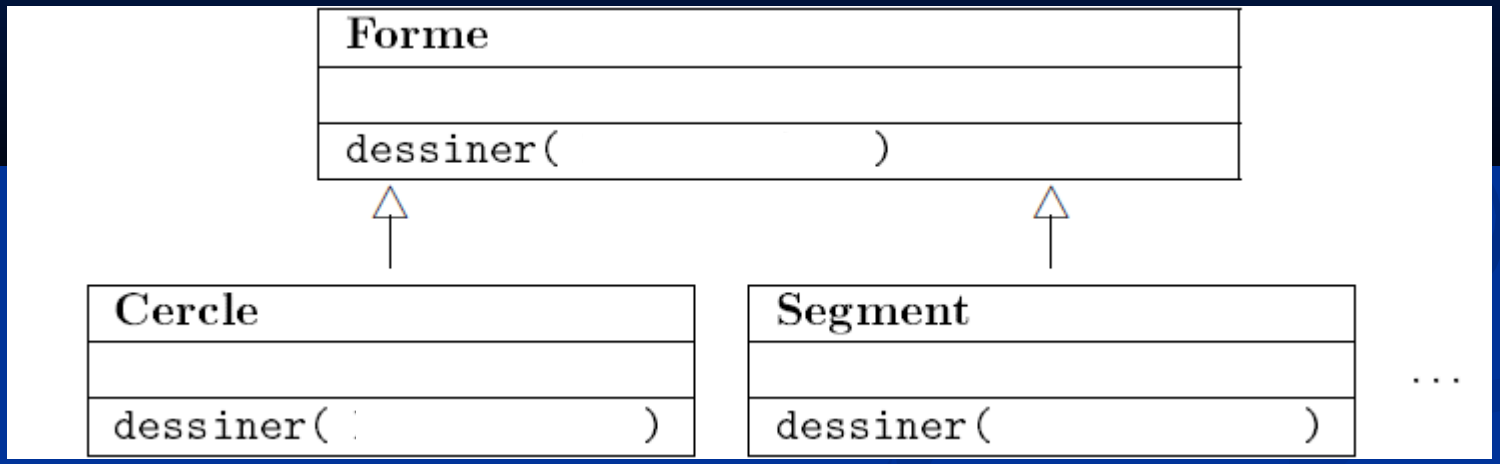
- le destructeur est public et virtuel : vous autorisez sans risque qu'un objet dérivé soit détruit à partir d'une variable de type statique de l'objet de base.

- le destructeur est protégé et non virtuel : le destructeur d'un objet de type de base ne peut être appelé que par le destructeur du type dérivé. Plus de risque qu'un objet de type statique A tente de détruire un objet de type dynamique B.

Polymorphisme : exercice

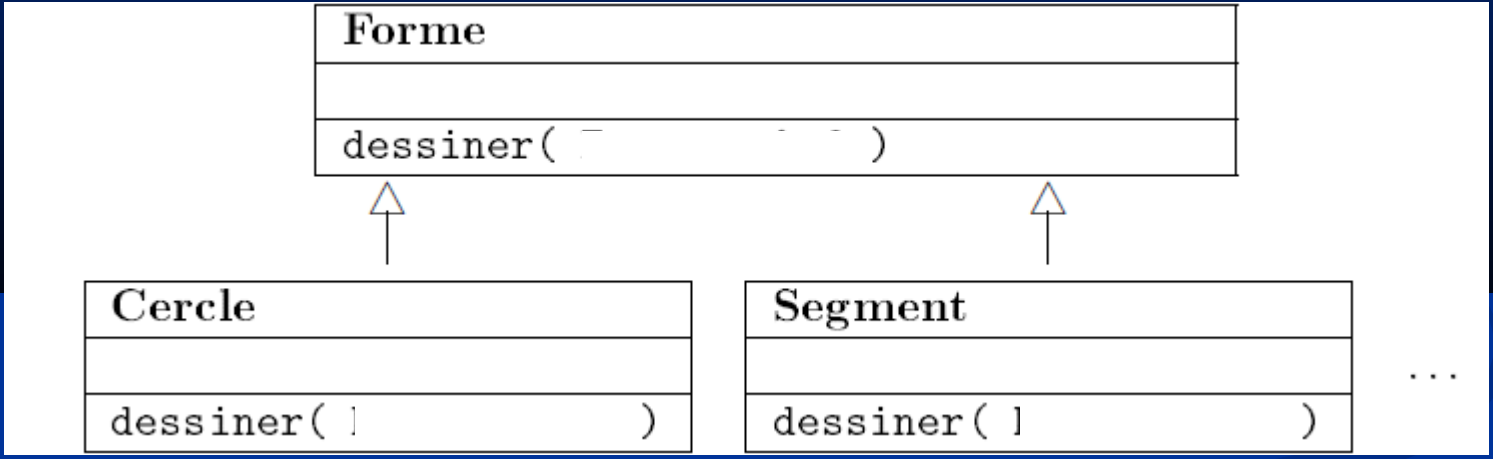
- Passez la méthode quiSuisJe() en virtuelle
- Testez depuis le main le bon fonctionnement de cette méthode virtuelle

Classe Abstraite



- Implémentation de la méthode dessiner dans la classe Cercle : aucun problème
- implémentation de la méthode dessiner dans la classe Segment : aucun problème

Classe Abstraite



- Implémentation de la méthode dessiner dans la classe Cercle : aucun problème
- implémentation de la méthode dessiner dans la classe Segment : aucun problème

Implémentation dans la classe Forme ??

Classe Abstraite
Méthodes virtuelles pures

- Une classe abstraite contient une ou plusieurs méthodes ***virtuelles pures***
- Dès qu'une classe contient un méthode virtuelle pure, on ne peut plus instancier d'objet

Classe Abstraite
Méthodes virtuelles pures

fichier essai.cxx

```
class A {
public:
  virtual void fct() = 0;
};

int main( int argc, char** argv )
{
  A a;
}
```

Un shell

```
=> g++ essai.cxx
essai.cxx: In function ‘int main (int, char **)’:
essai.cxx:8: cannot declare variable ‘a’ to be of type ‘A’
essai.cxx:8:   since the following virtual functions are abstract:
essai.cxx:3:    void A::fct ()
```

Classe Abstraite
Méthodes virtuelles pures

- Une classe abstraite contient une ou plusieurs méthodes ***virtuelles pures***
- Dès qu'une classe contient un méthode virtuelle pure, on ne peut plus instancier d'objet
- Certaines fonctions peuvent être définies dans une classe abstraite.
- Pour qu'une méthode soit virtuelle pure, on ajoutera le symbole « **=0** » à la définition de la méthode

Classe Abstraite Méthodes virtuelles pures: Exercice

fichier Forme.h

```
class Forme
{
   ...
   virtual void dessiner(                ) = 0;
   // Notez le '= 0' qui transforme 'Forme' en classe abstraite.
   ...
};
```

Classe Abstraite Méthodes virtuelles pures: Remarque

Si vous écrivez une classe héritant d'une classe abstraite, mais qui **ne définit pas** de corps à une (ou plusieurs) méthode virtuelle pure héritée, alors cette nouvelle classe est toujours une **classe abstraite**.

Obtention du type à l'exécution

- Supposons que nous ayons un vecteur de pointeurs d’employés
- Nous voudrions compter le nombre de secrétaires parmi ces employés
- Une façon de le faire serait d’ajouter une méthode méthode get type() qui retournerait le get_type() qui retournerait le type de l’objet

Obtention du type à l'exécution

```
class Employe
{
 public:
    Employe(string nom = "", double salaire = 0.0);
    virtual string get_type() const;
   ...
 private:
    ...
};

string Employe::get_type() const
{
   return "Employe";
}
```

Remarquez que la méthode doit être virtuelle. Pourquoi?

Obtention du type à l'exécution

```
class Gerant: public Employe
{
 public:
     ...
     virtual string get_type() const;
     ...
 private:
     ...
};

string Gerant::get_type() const
{
   return "Gerant";
}
```

Obtention du type à l'exécution

```
class Secretaire: public Employe
{
 public:
     ...
     virtual string get_type() const;
     ...
 private:
     ...
};

string Secretaire::get_type() const
{
   return "Secretaire";
}
```

Obtention du type à l'exécution

```
int main()
{

  vector< Employe* > v;
  ...

  short nb_secretaires = 0;
  for (size_t i = 0; i < v.size(); ++i) {
  if (v[i]->get_type() == "Secretaire")
       ++nb_secretaires;
  }
  ...
  return 0;
}
```

Obtention du type à l'exécution

- Le problème avec cette approche est qu'elle exige de programmer une méthode pour chaque classe
- En plus, il faut gérer nous-même un ensemble de descripteurs de types
- Tout ça alors qu'on sait à l'exécution à quelle classe on a affaire
- Meilleure approche: utiliser l'opérateur typeid de C++

Obtention du type à l'exécution

```
#include <typeinfo>
#include <vector>

int main()
{

  vector< Employe* > v;
  ...

  short nb_secretaires = 0;
  for (size_t i = 0; i < v.size(); ++i) {
     if (typeid(*v[i]) == typeid(Secretaire))
       ++nb_secretaires;
  }
  ...
  return 0;
}
```

Obtention du type à l'exécution : transtypage

- Le **transtypage** permet d'obtenir à partir d'un pointeur sur le type de base, un pointeur sur le type réel de l'objet.

dynamic_cast<pointeur_sur_type_désiré>(ptr_type_base);

- Ceci retourne soit :
  - Un pointeur sur le type désiré si ptr_type_base pointe réellement sur un objet de ce type
  - NULL (ou 0), sinon

Obtention du type à l'exécution : transtypage

```
#include <vector>

int main()
{
  vector< Employe* > v;
  ...

  short nb_secretaires = 0;
  for (size_t i = 0; i < v.size(); ++i) {
     Secretaire * s = dynamic_cast<Secretaire *>( v[i] );
     if(s == NULL)
        cout << "Ce n'est pas un/une secretaire" << endl;
     else // s est un pointeur valide sur un/une Secretaire
        ++nb_secretaires;
   }
   ...
   return 0;
}
```