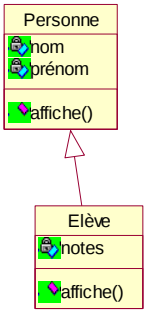
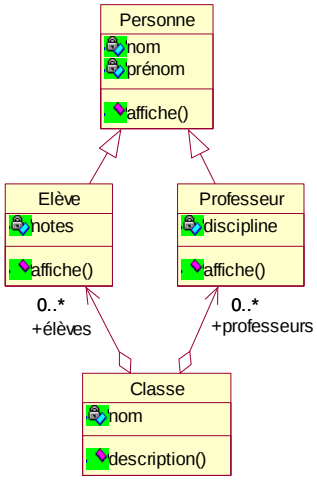


Le terme polymorphisme décrit la caractéristique d'un élément qui peut prendre plusieurs formes, comme l'eau qui se trouve à l'état solide, liquide ou gazeux.

En informatique, le polymorphisme désigne un concept de la théorie des types, selon lequel un nom d'objet peut désigner des instances de classes différentes issues d'une même arborescence. Effectivement, nous avons découvert que les classes issues d'une même hiérarchie sont compatibles. Ainsi, un objet de la classe « *Personne* » peut faire référence à un objet de la classe « *Elève* » (puisqu'un élève est aussi une personne). Nous pouvons donc écrire :

Personne &p = Elève(« Lagafe », « Gaston ») ;

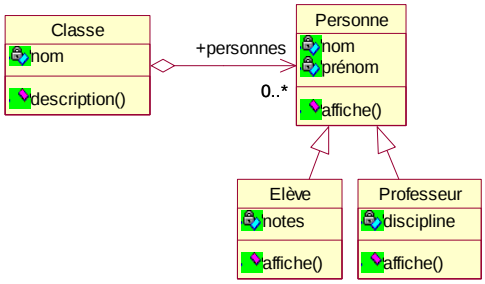


L'objet « *p* » peut aussi bien faire référence à une personne qu'à un élève ou à tout autre classe créée ultérieurement (comme la classe « *Professeur* ») faisant partie de cette hiérarchie. C'est ce principe là qui offre une grande richesse à la programmation orientée objet.

Visualisons ce concept en prenant l'exemple de la gestion d'une classe d'élèves. Nous créons donc une classe dénommée « *Classe* ». Cette classe dispose d'une méthode *description()* qui doit permettre de visualiser les caractéristiques de chacune des personnes constituant la classe.

1. Nous pouvons faire une première approche sans utiliser le polymorphisme. Nous sommes alors obligés de proposer une composition sur chacune des classes dérivées, ce qui, on l'imagine, demande pas mal d'écriture et pose des problèmes dans le cas où nous devons proposer de nouvelles classes dérivées dans cette hiérarchie. En effet, dans ce cas de figure, il serait alors nécessaire de reconstruire la classe « *Classe* » pour permettre l'intégration de la nouvelle classe dérivée.

2. Nous pouvons faire une approche totalement différente en proposant une conception basée sur le polymorphisme. Cette fois-ci, la composition est proposée directement sur la classe de base. En fait, la classe « *Classe* » s'occupe de toutes les personnes sans faire de distinctions. Le mécanisme du polymorphisme permet de retrouver automatiquement de quelle personne il s'agit, afin d'afficher la bonne description.



Principe général du polymorphisme en informatique

Les interactions entre objets sont écrites selon les termes des spécifications définies, non pas dans les classes dérivées des objets, mais dans leurs classes de base. Cela permet d'écrire un code détaché des particularités de chaque classe, et d'obtenir des mécanismes suffisamment généraux pour être valides dans le futur, quand seront créées de nouvelles classes.

Le terme polymorphisme désigne en fait le polymorphisme du comportement, c'est-à-dire la possibilité de déclencher les méthodes différentes en réponse d'un même message. Chaque classe dérivée hérite de la spécification des méthodes de ses classes de base, mais a aussi la possibilité de modifier localement le comportement de ces méthodes, afin de mieux prendre en compte les particularités de chacun. C'est le principe même de la redéfinition des méthodes comme *affiche()*.

De ce point de vue, une méthode donnée est polymorphe puisque sa réalisation peut prendre plusieurs formes. Le polymorphisme est un mécanisme de découplage qui agit dans le temps. Les bénéfices du polymorphisme sont avant tout récoltés durant la maintenance.

D'un point de vue pratique, et en reprenant l'exemple de la gestion d'une classe d'élèves, le polymorphisme consistera donc à redéfinir correctement les méthodes affiche() de chacune des classes faisant parties de la hiérarchie. Lorsqu'une nouvelle classe est créée, si elle veut s'intégrer dans le polymorphisme, c'est-à-dire, pouvoir participer à la gestion de la classe d'élève, elle aura pour contrat, de redéfinir sa propre méthode affiche().

Restriction du polymorphisme dans le langage C++

Par défaut, et contrairement au langage Java, les classes créées dans une hiérarchie dans le langage C++ n'intègrent pas le polymorphisme. Ici, pour notre exemple, nous devons modifier le comportement général de la (ou des) méthodes afin qu'effectivement le polymorphisme soit opérationnel dans notre hiérarchie. Pour cela, certains critères doivent être respectés :

1. ***Pour qu'une méthode soit désignée comme polymorphe, elle devra impérativement être virtuelle.***
2. ***Le polymorphisme est uniquement activé quand un objet de la classe dérivée est indirectement adressé via une référence ou un pointeur vers une classe de base.***

Dans la suite, nous allons découvrir pourquoi ces deux critères sont nécessaires.

Retour sur la compatibilité et la conversion entre la classe de base et la classe dérivée

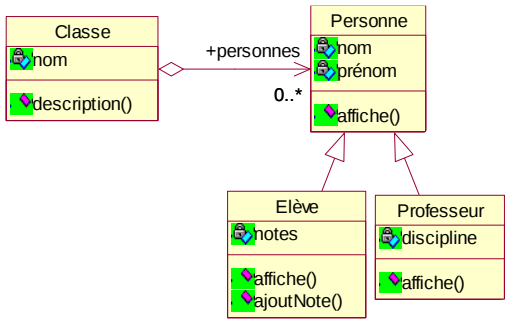
Puisqu'il existe un lien de parenté entre les classes d'une même hiérarchie, nous avons découvert qu'il existe, du coup, une certaine compatibilité. Cette compatibilité consiste, dans le cas du langage C++ en un système de conversions implicites, mises en œuvres automatiquement. Ces conversions sont les suivantes :

1. D'un objet d'un type dérivé dans un objet d'un type de base (l'inverse n'est pas possible),
2. D'un pointeur (ou d'une référence) sur une classe dérivée en un pointeur (ou une référence) sur une classe de base.

Nous nous sommes déjà penché sur le premier cas, mais nous allons y revenir pour voir la répercussion sur le polymorphisme.

Conversion d'un objet de classe dérivé vers un objet de classe de base :

```
6  class Personne {
7    string nom, prenom;
8  public:
9    Personne(string nom, string prenom) : nom(nom), prenom(prenom) { }
10   void affiche() {
11     cout << "Nom : " << nom << "\nPrénom : " << prenom << endl;
12   }
13 };
14 //-----------------------------------------------------------------
15 class Eleve : public Personne {
16   vector<double> notes;
17 public:
18   Eleve(string nom, string prenom) : Personne(nom, prenom) { }
19   void affiche() {
20     Personne::affiche();
21     cout << "Notes : ";
22     for (int i=0; i<notes.size(); i++) cout << notes[i] << ' ';
23   }
24   void ajoutNote(double note) { notes.push_back(note); }
25 };
26 //-----------------------------------------------------------------
27 class Professeur : public Personne {
28   string discipline;
29 public:
30   Professeur(string nom, string prenom, string discipline)
31     : Personne(nom, prenom), discipline(discipline) { }
32   void affiche() {
33     Personne::affiche();
34     cout << "Discipline : " << discipline;
35   }
36 };
```



Reprenons le diagramme UML qui correspond normalement à la logique du polymorphisme. Vous avez sur la partie gauche le codage correspondant. Bien entendu, seules les méthodes qui nous concernent directement sont implémentées. Les autres méthodes n'ont pas été introduites ici.

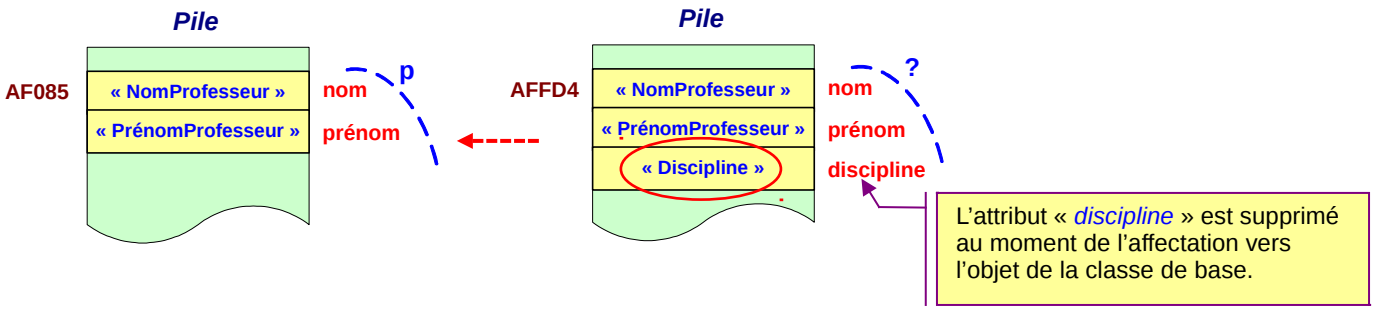
Ci-dessous se trouve un scénario qui montre la conversion d'un objet dérivé vers un objet de la classe de base.

```
36 int main( )
37 {
38   Personne p("Nom", "Prénom");
39   p = Professeur("NomProfesseur", "PrénomProfesseur", "Discipline");
40   p.affiche();
41   return 0;
42 }
```

```
Nom : NomProfesseur
Prénom : PrénomProfesseur
```

Que se passe-t-il dans ce programme ?

- *Ligne 38* : d'abord « *p* » est un objet de type « *Personne* ».
- *Ligne 39* : ensuite, nous avons une affectation entre cet objet et un objet anonyme de type « *Professeur* ». Puisque nous passons d'un objet dérivé vers un objet de classe de base, la conversion est tolérée. Seulement, il s'agit d'une copie des membres communs aux deux objets. Finalement, l'attribut « *discipline* » qui fait parti de la classe « *Professeur* » est littéralement supprimé. En fait, la portion relative à l'attribut « *discipline* » ne tient pas dans la mémoire allouée pour contenir l'objet « *p* ».



- Pour être plus précis, rappelons que lorsque nous effectuons une opération quelconque, et à fortiori, une affectation, tous les opérandes de l'opération doivent être du même type. Comme le membre gauche de l'affectation est de type « *Personne* », le membre droit doit être également de ce type. Du coup, l'objet anonyme de type « *Professeur* » est d'abord transformé en un objet anonyme de type « *Personne* » avec la perte d'une partie de ses constituants. Ensuite, nous avons une copie de l'objet anonyme de type « *Personne* » vers l'objet « *p* ». Au moment de cette copie, nous nous retrouvons donc qu'avec des objets de type « *Personne* ». Dans ce contexte, l'objet de type « *Professeur* » a totalement disparu.

- *Ligne 40* : « *p* » reste donc bien une « *Personne* ». La méthode *affiche()* qui est sollicité correspond bien à la méthode *affiche()* de la classe de base.

> ***Conclusion***
> Dans ces conditions, il n'est absolument pas possible d'intégrer le polymorphisme. Tout ce que nous faisons, c'est une copie des membres d'une classe dérivée vers sa classe de base. L'objet lui-même n'a pas changé de statut. Il faut plutôt que l'objet de la classe de base fasse « *référence* » à un autre objet ; soit également à un objet de la classe de base, soit à un objet de la classe dérivée. Nous devons donc utiliser les *références* (*accès direct*) ou les *pointeurs* (*accès indirect*).

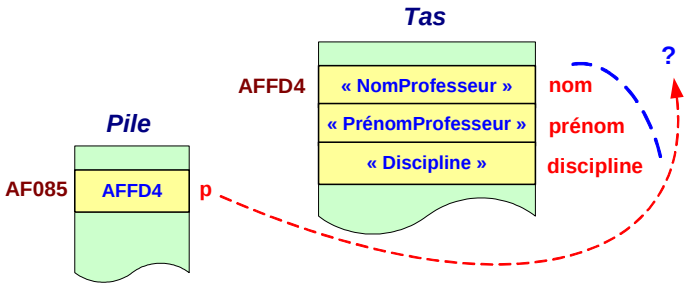
Conversion de pointeurs et de références dans une hiérarchie de classes :
Deux nouvelles conversions standard (automatiques) sont prévues entre les classes de base et leurs classes dérivées :
1. Une référence de classe dérivée pourra implicitement être convertie en une référence de classe de base publique.
2. Un pointeur sur une classe dérivée pourra implicitement être converti en un pointeur sur la classe de base publique.

Retentons l'expérience précédente, en utilisant cette fois-ci, par exemple, un pointeur sur un objet de type « *Personne* ».

```
36 int main( )
37 {
38   Personne *p;
39   p = new Professeur("NomProfesseur", "PrénomProfesseur", "Discipline");
40   p->affiche();
41   delete p;
42   return 0;
43 }
```

```
Nom : NomProfesseur
Prénom : PrénomProfesseur
```

Que se passe-t-il dans ce programme ?
- *Ligne 38* : d'abord « *p* » est un pointeur vers un objet de type « *Personne* ».
- *Ligne 39* : ensuite, nous initialisons ce pointeur par rapport à un objet dynamique de type « *Professeur* ». A droite du symbole d'affectation, nous avons un pointeur vers un objet de la classe « *Professeur* ». L'affectation propose donc une conversion implicite d'un pointeur d'un objet de type « *Professeur* » vers un pointeur d'un objet de type « *Personne* ». Sur cette ligne, nous obtenons bien l'effet souhaité. Alors que « *p* » est normalement un pointeur vers un objet de type « *Personne* », il est également capable de ce connecté vers un objet d'un autre type. Le tout, c'est que cet objet fasse partie de la descendance. Par ailleurs, l'objet connecté par ce pointeur conserve tous ces attributs.
- *Ligne 40* : lorsque nous exécutons cette ligne, nous retrouvons le même résultat. C'est encore une fois la méthode issue de la classe « *Personne* » qui est effectuée.



> ***Conclusion***
> Le problème, c'est que le choix de la méthode appelée est réalisée par le compilateur, ce qui signifie qu'elle est définie une fois pour toutes avant même que le programme ne s'exécute (la phase de compilation se fait avant la phase d'exécution du programme). Bien entendu, dans ces conditions, on comprend que le compilateur ne peut que décider de mettre en place l'appel de la méthode correspondant au type défini par le pointeur. Ainsi à la *ligne 38*, « *p* » est un pointeur de type « *Personne* », donc à la *ligne 40*, le compilateur prend automatiquement la méthode *Personne ::affiche()*.

Comment faire ?
Nous connaissons déjà les variables ou les objets dynamiques. Ces objets ne pas sont créés au moment de la compilation, mais uniquement lorsque nous en avons besoin. Nous utilisons pour cela les opérateurs « *new* » et « *delete* », respectivement pour leur création et pour leur destruction. Ces objets sont alors créés dans le « *Tas* ». Par contre, seul l'état de l'objet est enregistré, c'est-à-dire, la valeur représentative de chacun des attributs.

Ce qu'il faudrait, c'est qu'il existe le même principe pour les méthodes. Il faudrait que le choix de la méthode correspondant au type de l'objet pointé soit déterminé uniquement au moment de son exécution et non pas, pendant la phase de compilation. Il faudrait donc un système de méthodes dynamiques. Ces méthodes existent, il s'agit des méthodes virtuelles.

Les méthodes virtuelles

Une méthode virtuelle est une méthode particulière invoquée au *moyen d'un pointeur ou d'une référence* sur une classe de base ; elle est liée dynamiquement au moment de l'exécution. L'instance invoquée est déterminée par le type de classe de l'objet adressé par le pointeur ou la référence. La résolution d'une méthode virtuelle est transparente à l'utilisateur.

Il suffit de placer le mot réservé « *virtual* » devant la méthode que nous désirons rendre virtuelle et le tour est joué. Par ailleurs, il n'est pas nécessaire de déclarer virtuelles les méthodes redéfinies dans les classes dérivées, elles le sont automatiquement.

```
6  class Personne {
7    string nom, prenom;
8  public:
9    Personne(string nom, string prenom) : nom(nom), prenom(prenom) { }
10   virtual void affiche() {
11     cout << "Nom : " << nom << "\nPrénom : " << prenom << endl;
12   }
13 };
```

Reprenons donc le scénario précédent en rajoutant juste « *virtual* » devant la méthode *affiche()* de la classe de base.

Cette fois-ci, l'affichage est celui prévu.

```
36 int main( )
37 {
38   Personne *p;
39   p = new Professeur("NomProfesseur", "PrénomProfesseur", "Discipline");
40   p->affiche();
41   delete p;
42   return 0;
43 }
```

```
Nom : NomProfesseur
Prénom : PrénomProfesseur
Discipline : Discipline
```

virtual void affiche() ;

Cette instruction indique au compilateur que les éventuels appels de la méthode *affiche* doivent utiliser une ligature dynamique et non plus une ligature statique

p->affiche() ;

Du coup, lorsque le compilateur rencontre cette instruction, il ne décidera pas de la méthode à appeler. Il se contentera de mettre en place un dispositif permettant de n'effectuer le choix de la méthode qu'au moment de l'exécution de cette instruction, ce choix étant basé sur le type exact de l'objet ayant effectué l'appel. Plusieurs exécutions de cette même instruction pouvant appeler des méthodes différentes.

```
39 int main( )
40 {
41   Personne *p;
42   p = new Professeur("NomProfesseur", "PrénomProfesseur", "Discipline");
43   p->affiche();
44   cout << endl;
45   delete p;
46   Eleve *e = new Eleve("NomElève", "PrénomElève");
47   e->ajoutNote(15); e->ajoutNote(8); e->ajoutNote(10);
48   p = e;
49   p->affiche();
50   delete p;
51   return 0;
52 }
```

```
Nom : NomProfesseur
Prénom : PrénomProfesseur
Discipline : Discipline
Nom : NomElève
Prénom : PrénomElève
Notes : 15 8 10
```

Conclusion

Nous voyons que nous pouvons intégrer le polymorphisme vraiment très simplement. Il suffit de déclarer la ou les méthodes voulues de la classe de base comme virtuelles. La seule difficulté finalement, se situe au moment de la phase de conception durant l'élaboration des diagrammes UML. C'est effectivement à ce moment là qu'il faut décider si une hiérarchie de classes propose le polymorphisme ou pas. Dans l'affirmative, il est en effet souvent nécessaire de rajouter de nouvelles méthodes dans la classe ancêtre, alors que ce n'était pas spécialement prévu au départ.

En voyant cette simplicité, nous pourrions nous dire que nous n'avons pas besoin de nous poser autant de questions. Nous pouvons systématiquement spécifier toutes les méthodes comme virtuelles, puisque nous rajoutons un seul mot sur chacune des méthodes de la classe de base. L'étude suivante va nous montrer que ce n'est pas aussi simple.

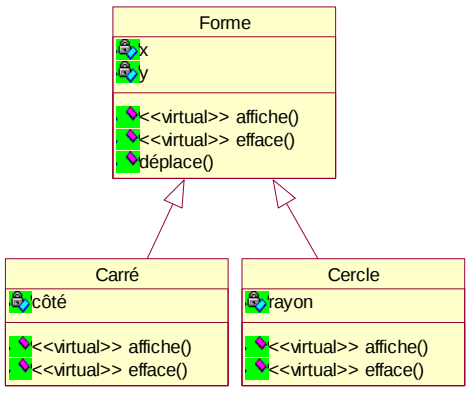
Mécanisme d'identification dynamique des objets

Nous venons de voir que le polymorphisme est très simple à implémenter dans le langage C++. Nous pouvons nous passer de toutes autres connaissances subsidiaires. Toutefois, il peut être intéressant d'avoir une compréhension plus fine du mécanisme interne, en prenant connaissance de l'implantation de la ligature dynamique.

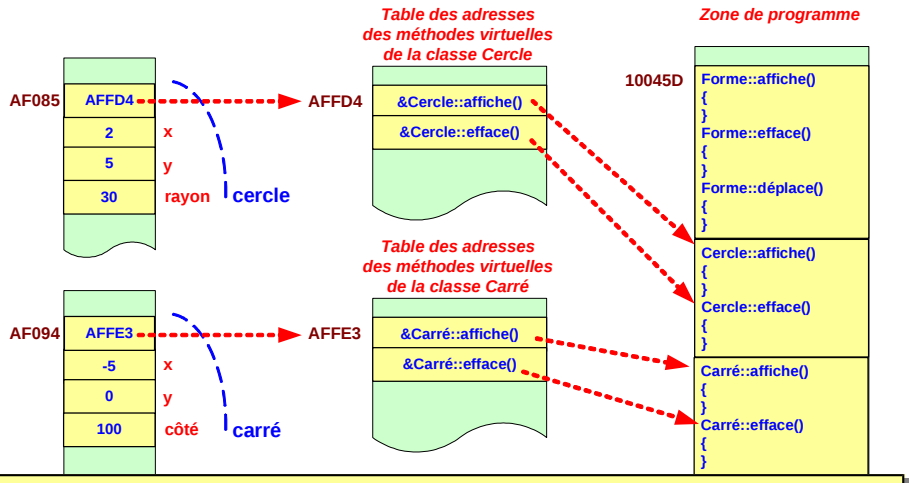
D'une manière générale, lorsqu'une classe comporte au moins une méthode virtuelle, le compilateur lui associe une table contenant les adresses des méthodes virtuelles correspondantes.

D'autre part, tout objet d'une classe comportant au moins une méthode virtuelle se voit attribuer par le compilateur, outre l'emplacement mémoire nécessaire à ses attributs, un emplacement supplémentaire de type pointeur, contenant l'adresse de la table associée à sa classe.

Nous pouvons ainsi dire que ce pointeur, introduit dans chaque objet, représente l'information permettant d'identifier la classe de l'objet. C'est effectivement cette information qui est exploitée pour mettre en œuvre la ligature dynamique. Chaque appel d'une méthode virtuelle est traduit par le compilateur de la façon suivante :



1. Prélèvement dans l'objet de l'adresse de la table correspondante,
2. Branchement à l'adresse figurant dans cette table à un rang donné. Notez bien que ce rang est parfaitement défini à la compilation. Toutes les tables des classes d'une même hiérarchie sont structurées exactement de la même façon. Ainsi, l'adresse de la méthode virtuelle *affiche()* se situe toujours en première position sur toutes les tables.
3. En revanche, c'est lors de l'exécution que sera effectué le « choix de la bonne table ».

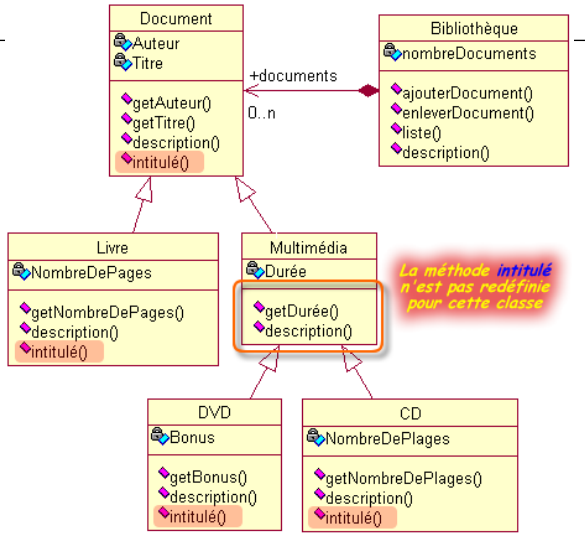


Conclusion

Si vous intégrez le polymorphisme, la structure interne se complexifie alors largement. Par ailleurs, vous remarquez que l'accès à une méthode passe par deux indirections, ce qui rallonge d'autant le temps de réponse. Pour finir, le stockage de ces différentes tables demande de la mémoire supplémentaire. Il ne faut donc pas qu'une hiérarchie soit considérée systématiquement comme polymorphique. Il faut que cela corresponde à un besoin, déterminé au moment de la phase de conception, notamment durant l'élaboration des diagrammes UML.

Propriétés des méthodes virtuelles

Dans ce chapitre, nous allons faire un certain nombre de remarques afin que les méthodes virtuelles soient correctement implémentées.



1. Les méthodes virtuelles doivent impérativement exister pour qu'elles puissent être adressées à l'aide de la table correspondante. Elles sont donc nécessairement non « *inline* » (si vous la déclarez « *inline* », le compilateur fabrique une véritable méthode).

2. Le mot « *virtual* » se place uniquement dans la déclaration de la classe. Lorsque vous définissez la méthode à l'extérieur de la classe, vous ne devez plus re-spécifier le mot « *virtual* » devant la signature de la méthode.

3. La redéfinition d'une méthode virtuelle dans une classe dérivée doit réaliser une adéquation parfaite (nom, signature, et type de retour) avec la méthode virtuelle déclarée dans la classe de base. Il n'est pas nécessaire de re-spécifier le mot « *virtual* ». Si la re-déclaration dans la classe dérivée ne réalise pas une adéquation parfaite, la méthode n'est pas gérée comme une méthode virtuelle de la classe dérivée. Dans ce cas là, il s'agira tout simplement d'une sur-définition.

4. Il n'est pas obligatoire que toutes les classes dérivées redéfinissent impérativement toutes les méthodes virtuelles données par la classe de base. C'est notamment le cas lorsque la méthode héritée de la classe de base fait déjà tout ce qu'il faut. Par contre, rien n'empêche à une classe ultérieurement dérivée de ces classes dérivées de proposer, elle, la redéfinition de la méthode virtuelle, même si son proche parent ne l'a pas fait.

5. Lorsque nous avons une redéfinition des destructeurs dans une hiérarchie de classe qui comporte des méthodes virtuelles, il est généralement préférable que les destructeurs fassent également partie des tables des adresses des méthodes virtuelles. En effet, lorsque nous réalisons un « *delete* » sur un pointeur d'une classe de base, le bon destructeur est alors pris en compte. Vous obtenez ce comportement en déclarant « *virtuel* » le destructeur de la classe de base.

```
39 int main( )
40 {
41   Personne *p;
42   p = new Professeur("NomProfesseur", "PrénomProfesseur", "Discipline");
43   p->affiche();
44   cout << endl;
45   delete p;
46   Eleve *e = new Eleve("NomElève", "PrénomElève");
47   e->ajoutNote(15); e->ajoutNote(8); e->ajoutNote(10);
48   p = e;
49   p->affiche();
50   delete p;
51   return 0;
52 }
```

Doit faire appel au destructeur de la classe Professeur

Doit faire appel au destructeur de la classe Elève

6. Lorsque qu'une méthode virtuelle est invoquée à l'intérieur d'un des constructeurs de la hiérarchie, c'est toujours la méthode virtuelle de la classe de base qui est sollicité. En effet, puisque nous sommes en phase de création, les tables des adresses des méthodes virtuelles n'existent pas encore. Nous ne pouvons donc pas intégrer le polymorphisme sur un constructeur, il faut que l'objet soit d'abord créé. Du coup, un *constructeur* ne peut jamais être *virtuel*.

Contrôle des surcharges en C++11

La norme 2011 du C++ introduite deux nouveaux mots-clés ***override*** et ***final*** afin d'éviter des surcharges et masquages involontaires de méthodes. Ces mots réservés ne concernent que les méthodes virtuelles que nous venons de découvrir.

Le mot réservé ***override*** permet de redéfinir une méthode dans une sous-classe (classe fille) à condition qu'une méthode avec exactement la même signature existe dans la classe ancestrale (classe mère). Si la signature est différente (liste de paramètres différents), ou qu'il n'existe aucune méthode de ce nom dans la classe ancestrale, une erreur de compilation est déclenchée.

> ***Sans ce contrôle, vous provoqueriez la définition d'une nouvelle méthode dans la sous-classe alors que votre intention était de redéfinir la méthode héritée. Il devient donc très important de rajouter ce qualificatif supplémentaire afin d'éviter toute erreur involontaire.***

Exemple avec la gestion du personnel de l'éducation

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

class Personne
{
  string nom, prenom;
public:
  Personne(string n, string p) : nom(n), prenom(p) {}
  virtual void affiche()
  {
    cout << "Nom = " << nom << endl;
    cout << "Prenom = " << prenom << endl;
  }
};

class Eleve : public Personne
{
  vector<double> notes;
public:
  Eleve(string nom, string prenom) : Personne(nom, prenom) {}
  void ajout(double note) { notes.push_back(note); }
  void ajout(vector<double> notes)
  {
    for (double note : notes) this->notes.push_back(note);
  }
  void affiche() override // redéfinition de la méthode affiche()
  {
    cout << "Élève ----------------------------" << endl;
    Personne::affiche();
    cout << "Notes : ";
    if (notes.empty()) cout << "vide";
    else for (double note : notes) cout << note << ' ';
    cout << endl;
  }
};

class Professeur : public Personne
{
  string discipline;
public:
  Professeur(string n, string p, string d) : Personne(n, p), discipline(d) {}

  void affiche() override // redéfinition de la méthode affiche()
  {
    cout << "Professeur -----------------------" << endl;
    Personne::affiche();
    cout << "Discipline : " << discipline << endl;
  }
};

void connaitre(Personne &p)
{
  p.affiche();
}

int main()
{
  Eleve martin("PECHEUR", "Martin");
  Professeur prof("ALBAN", "Michel", "Informatique");
  martin.ajout(15.0);
  martin.ajout({12.0, 5.0, 8.5});
  connaitre(martin);
  connaitre(prof);
  return 0;
}
```

Dans le même ordre d'idée, il devient possible de couper court à toute tentative de redéfinir une méthode virtuelle dans une sous-classe en ajoutant cette fois-ci l'identifiant ***final*** (définitif). La méthode ainsi qualifiée devient la dernière dans la lignée de classes. Aucune sous-classe ne pourra plus redéfinir cette méthode avec exactement la même signature. C'est le comportement inverse de ***override***.