




Ce chapitre et le suivant discutent des plus importantes caractéristiques de la programmation orientée objets : l'*héritage* et le *polymorphisme*. L'héritage consiste à créer des nouvelles classes à partir de classes existantes afin d'en hériter leurs fonctionnalités. Le polymorphisme permet d'écrire des programmes d'une manière générale afin de supporter une vaste gamme de classes existantes et de classes apparentées encore à spécifier.

L'héritage et le polymorphisme constituent des techniques efficaces dans le traitement de la complexité de logiciels.



## Objectifs

---


- Créer des nouvelles classes à partir de classes existantes
  - Comprendre les notions de classes de base et de classes dérivées
- Comprendre comment l'héritage favorise la réutilisation des logiciels
- Connaître les notions d'héritage multiple

2

Marco Lavoie 14728 ORD - Langage C++

L'héritage présente une forme de réutilisation de logiciel dans laquelle de nouvelles classes sont créées à partir de classes existantes, en absorbant leurs attributs et leurs comportements et en les substituant ou en les embellissant avec les caractéristiques requises par les nouvelles classes.

La réutilisation des logiciels épargne un temps considérable dans le développement de programmes, favorisant cette pratique avec des logiciels de haute qualité, éprouvés et débogués, pour ainsi réduire les problèmes une fois qu'un système est devenu fonctionnel. Il s'agit là d'une pratique aux possibilités passionnantes.



## Aperçu

---

- Héritage
  - Classes de base, et
  - Classes dérivées
- Membres protégés (**protected**)
- Transtypage de pointeurs de classe
- Surcharge de fonctions membres
- Héritage **public**, **protected** et **private**
- Constructeurs et destructeurs d'instances
- Héritage multiple

3

Marco Lavoie 14728 ORD - Langage C++

Au lieu d'écrire des fonctions et des attributs membres complètement nouveaux, le programmeur peut spécifier que, lors de la création d'une nouvelle classe, cette dernière *hérite* des membres d'une *classe de base* définie antérieurement. On fait alors référence à cette nouvelle classe par la locution de *classe dérivée*. Chaque classe dérivée devient elle-même candidate pour être une classe de base pour d'autres classes dérivées, d'où émerge une *hiérarchie de classes*. Avec l'*héritage simple*, une classe est dérivée d'une seule classe de base alors qu'avec l'*héritage multiple*, une classe dérivée hérite de multiples classes de base, non nécessairement apparentées.

Le C++ offre trois formes d'héritage : **public**, **protected** et **private**. Ces types d'héritage permettent un contrôle plus précis de l'accessibilité aux membres de la classe de base à partir des fonctions membres de la classe dérivée.



## Introduction

- Héritage : notion fondamentale de la programmation orientée objet
  - Avec le *polymorphisme* (chapitre 10)
- Objectif : créer une nouvelle classe (*classe dérivée*) à partir d'une classe existante (*classe de base*)
  - Seulement implanter dans la classe dérivée les nouvelles fonctionnalités ou celles différentes de la classe de base

Marco Lavoie

14728 ORD - Langage C++

Une classe dérivée peut ajouter des attributs et/ou fonctions membres qui lui sont propres; elle peut donc être de plus grande taille que la classe de base. Une classe dérivée est plus spécifique que sa classe de base et représente un groupe d'objets plus précis. Avec l'héritage simple, la classe dérivée débute essentiellement de la même façon que la classe de base. La véritable force de l'héritage provient de l'habileté à définir dans la classe dérivée des additions, des substitutions ou des améliorations aux caractéristiques héritées de la classe de base.



## Introduction (suite)

- Principe de réutilisation de logiciels
  - La classe dérivée exploite les fonctionnalités de la classe de base sans les réimplanter
    - Elle peut exploiter les membres hérités de la classe de base
    - Elle peut modifier les fonctions membres héritées
    - Elle peut implanter de nouveaux membres non disponibles dans la classe de base
  - En général, la classe dérivée *améliore* ou *spécialise* les fonctionnalités de la classe de base

Marco Lavoie

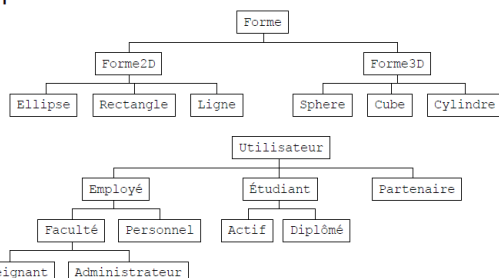
14728 ORD - Langage C++

L'expérience relative à la construction de systèmes logiciels complexes indique que des portions significatives de code traitent de cas spéciaux intimement liés. Dans de tels systèmes, il devient difficile de capter l'image d'ensemble puisque les préoccupations du concepteur et du programmeur demeurent concentrées sur les cas spéciaux. La programmation orientée objets offre plusieurs façons de « voir la forêt à travers les arbres » avec le procédé nommé *abstraction*.



## Classe de base versus dérivée

- Les classes dérivées sont généralement des spécialisation de leur classe de base



Marco Lavoie

14728 ORD - Langage C++

Dans bien des cas, un objet d'une classe constitue également un objet d'une autre classe. Un rectangle est à coup sûr une forme géométrique à deux dimensions. Par conséquent, on peut dire que la classe **Rectangle** hérite de la classe **Forme2D**. Dans ce contexte, on appelle *classe de base* la classe **Forme2D** alors que la classe **Rectangle** est nommée *classe dérivée*. Une classe peut à la fois être classe de base et classe dérivée, telle que **Forme2D** qui est dérivée de la classe plus générale **Forme**, tout en étant une généralisation de la classe dérivée **Rectangle**.

Un rectangle est un type spécifique de forme à deux dimensions; par contre, il est inexact d'affirmer qu'une forme à deux dimensions est un rectangle; la forme à deux dimensions pourrait parfaitement être une ellipse ou une ligne. L'exemple ci-contre illustre plusieurs exemples d'héritage simple.

## Déclaration de dérivation

7

- La classe de base est spécifiée dans la déclaration de la classe dérivée
 

```
class Forme {
}; ...

class Forme2D : public Forme {
}; ...
```

  - Le type d'héritage peut être **public**, **private** ou **protected**
    - Le spécificateur a un impact sur l'accès aux membres hérités de la classe de base
    - Le spécificateur **protected** est exclusivement lié à l'héritage

Marco Lavoie

14728 ORD - Langage C++

Examinons la syntaxe qui permet d'exprimer l'héritage. Pour spécifier que la classe **Forme2D** est dérivée de la classe **Forme**, nous devons définir la classe **Forme2D** de la façon ci-contre.

C'est ce qu'on appelle l'héritage **public**, le type d'héritage le plus couramment utilisé. Nous discuterons également de l'héritage **privé** (**private**) et de l'héritage **protégé** (**protected**). En utilisant l'héritage **public**, les membres **public** et **protected** de la classe de base sont hérités respectivement comme membres **public** et **protected** dans la classe dérivée. Rappelez-vous que les membres **private** d'une classe de base ne sont pas accessibles à partir des classes dérivées de celle-ci. Notez également que les fonctions **friend** ne sont pas héritées.

## Membres protégés (**protected**)

8

- Dans une classe de base
  - Les membres **public** sont accessibles à tous, incluant les classes dérivées
  - Les membres **private** sont exclusifs à la classe de base
  - Les membres **protected** sont exclusifs à la classe de base et ses classes dérivées

Marco Lavoie

14728 ORD - Langage C++

On accède aux membres **public** d'une classe de base par toutes les fonctions du programme, incluant celles des membres de toutes les autres classes. Les membres **private** d'une classe de base ne sont cependant accessibles que par les fonctions membres et amies (**friend**) de la classe de base en question.

L'accès **protected** constitue un niveau de protection intermédiaire entre les accès **public** et **private**. Les membres **protected** d'une classe de base ne sont accessibles que par les membres et amis (**friend**) de la classe de base et par les membres et amis des classes dérivées de cette classe de base.

## Membres protégés (suite)

9

- Exemple

```
class A {
public:
    int aPub;
protected:
    int aProt;
private:
    int aPriv;
};

class B : public A {
public:
    int bPub;
protected:
    int bProt;
private:
    int bPriv;
};

int main {
}
```

### Accessibilité

	Fonctions membres de la classe A	Fonctions membres de la classe B	main() et autres blocs de code
aPub	✓	✓	✓
aProt	✓	✓	✗
aPriv	✓	✗	✗
bPub	✗	✓	✓
bProt	✗	✓	✗
bPriv	✗	✓	✗

- Le type d'héritage ainsi que les fonctions amies (**friend**) circonviennent les spécificateurs d'accès

Marco Lavoie

14728 ORD - Langage C++

Les membres d'une classe dérivée peuvent accéder aux attributs et fonctions membres **public** et **protected** de sa classe de base ainsi que de toutes les *classes ancêtres* de celle-ci, c'est-à-dire de tous les membres des classes constituant la hiérarchie de classe partant de la classe n'ayant aucune classe de base mais dont la succession de classes dérivées mène à la classe visée.

Notez que les membres **protected** fissent l'encapsulation : des modifications apportées aux membres protégés d'une classe de base peuvent nécessiter la modification de toutes les classes dérivées de cette classe de base.

## Transtypage

10

- Une instance de classe dérivée peut être transtypée en instance de classe de base

```
class ExA {
public:
    void func() { cout << "A"; }
    int a;
};

class ExB : public ExA {
public:
    void func() { cout << "B"; }
    int b;
};

void main() {
    ExA exa; // instance de classe ExA
    ExB exb; // instance de classe ExB

    exa.func(); // Affiche A
    exb.func(); // Affiche B

    // L'instruction suivante affiche A
    static_cast< ExA >( exb ).func();
}
```

– L'inverse n'est cependant pas vrai

- L'instance de classe de base n'a pas les attributs supplémentaires potentiels de la classe dérivée

*static\_cast< ExB >( exa ).b = 0;*  
*exb dispose des attributs a et b, mais exa a seulement l'attribut a*

Marco Lavoie

14728 ORD - Langage C++

Un objet d'une classe dérivée d'un type public peut être traité comme un objet de sa classe de base correspondante. Cette pratique permet certaines manipulations intéressantes. Par exemple, en dépit du fait que les objets d'une variété de classes dérivées d'une classe de base particulière peuvent différer sensiblement les uns des autres, nous pouvons tout de même créer un tableau de pointeurs à ces objets, aussi longtemps que nous les traitons comme s'ils étaient des objets de leur classe de base commune.

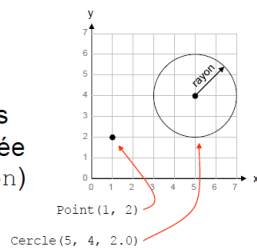
L'inverse ne peut cependant pas s'appliquer : un objet d'une classe de base ne constitue pas automatiquement un objet d'une classe dérivée. Le *transtypage explicite* peut cependant convertir un pointeur de classe de base en un pointeur de classe dérivée. L'anglais désigne cette opération de *downcasting*, où l'on rétrograde un type de classe de base en un autre de classe dérivée. Le *downcasting* est applicable en C++ en autant que l'objet pointé sous effectivement du type correspondant à la classe dérivée.

## Exemple d'héritage

11

- Classe **Cercle** dérivée de la classe **Point**

- Point** : coordonnées d'un point en deux dimensions (x, y)
- Cercle** : coordonnées d'un point accompagnée d'un rayon (x, y, rayon)



Marco Lavoie

14728 ORD - Langage C++

L'exemple ci-contre illustre la définition des classes **Point** et **Cercle**. Un objet de type **Point** est constitué de deux attributs membres donnant les coordonnées du point dans l'espace cartésien : **x** et **y**. Similairement, un objet de type **Cercle** est aussi caractérisé par ses coordonnées cartésiennes (attributs **x** et **y**), ainsi que par un **rayon**.

Les caractéristiques communes aux deux types d'objet (c.à.d. leurs coordonnées) suggèrent une relation de classe de base (**Point**) à classe dérivée (**Cercle**), cette dernière « ajoutant » un attribut supplémentaire à la classe **Point**.

## Exemple d'héritage : classe Point

12

```
class Point {
    friend ostream operator<< ( ostream &, const Point & );
public:
    Point( int = 0, int = 0 ); // constructeur par défaut
    void setPoint( int, int ); // change les coordonnées
    int getX() const { return x; } // retourne la coordonnée x
    int getY() const { return y; } // retourne la coordonnée y
protected:
    int x, y; // accessible par les classes dérivées // coordonnées x et y du Point
};

// Constructeur de la classe Point
Point::Point( int a, int b ) { setPoint( a, b ); }

// Ajuste les coordonnées x et y de Point
void Point::setPoint( int a, int b ) {
    x = a;
    y = b;
}

// Sortie de Point (avec l'opérateur d'insertion de flux surchargé)
ostream operator<< ( ostream &sortie, const Point &p ) {
    sortie << "[ " << p.x << ", " << p.y << " ] ";
    return sortie; // permet les appels en cascade
}
```

Marco Lavoie

14728 ORD - Langage C++

Examinons d'abord la définition de la classe **Point**. L'interface **public** de **Point** inclut les fonctions membres accesseurs en ligne **getX** et **getY**, ainsi que la fonction mutateur **setPoint**. Les attributs membres **x** et **y** de **Point** sont d'accès **protected**. Cette situation empêche les clients des objets **Point** d'accéder directement aux données mais permet aux classes dérivées de **Point** de rejoindre directement les attributs membres hérités. Si ces attributs étaient **private**, nous devrions utiliser les accesseurs et mutateur de **Point** afin d'accéder aux données, même pour les classes dérivées.

Notez que la fonction d'opérateur surchargé d'insertion de flux de **Point** peut référencer directement les attributs **x** et **y**, cette fonction étant amie (**friend**) de la classe **Point**.



## Exemple d'héritage : classe Cercle

```
#include "point.h"

class Cercle : public Point { // Cercle hérite de Point
friend ostream &operator<<( ostream &, const Cercle & );
public:
// Constructeur par défaut
Cercle( double = 0.0, int = 0, int = 0 );

void setRayon( double ); // ajuste le rayon
double getRayon() const; // renvoie le rayon
double aire() const; // calcule l'aire
protected:
double rayon;
};

// Le constructeur pour Cercle initialise les coordonnées et le rayon
Cercle::Cercle( double r, int a, int b ) {
setPoint( a, b );
setRayon( r );
}

// Ajuste le rayon de Cercle
void Cercle::setRayon( double r )
{ rayon = ( r >= 0 ? r : 0 ); }
```

Marco Lavoie

14728 ORD - Langage C++

La classe `Cercle` hérite de la classe `Point` par héritage public. On retrouve cette spécification d'héritage dans la première ligne de la déclaration de la classe `Cercle` :

```
class Cercle : public Point { // ...
```

Le deux-points (`:`) dans l'en-tête de la définition de classe indique l'héritage. Le mot-clé `public` indique, quant à lui, le type d'héritage. Ainsi, les membres `public` et `protected` de la classe `Point` sont hérités respectivement comme membres `public` et `protected` dans la classe `Cercle`. Ceci signifie que l'interface `public` de `Cercle` inclut à la fois les membres `public` de `Point` (`getX`, `getY` et `setPoint`) de même que les membres `public` de `Cercle` nommés `aire`, `setRayon` et `getRayon`.

Le constructeur de `Cercle` invoque le mutateur de `Point` pour initialiser les coordonnées du cercle (c.à.d. ses attributs hérités `x` et `y`), puis son mutateur pour ajuster son `rayon`.



## Exemple d'héritage : classe Cercle

```
// Lit le rayon de Cercle
double Cercle::getRayon() const { return rayon; }

// Calcule l'aire de Cercle
double Cercle::aire() const
{ return 3.14159 * rayon * rayon; }

// Sortie d'un Cercle sous la forme:
// Centre = [x, y]; Rayon = ###
ostream &operator<<( ostream &sortie, const Cercle &c ) {
sortie << "Centre = " << static_cast<Point>(c) << "\n";
sortie << "Rayon = "
<< setiosflags( ios::fixed | ios::showpoint )
<< setprecision( 2 ) << c.rayon;

return sortie; // permet les appels en cascade
}
```

- Le transtypage est requis afin d'invoquer la fonction de surcharge `ostream &operator<<( ostream &, const Point & )`;

Marco Lavoie

14728 ORD - Langage C++

La fonction de l'opérateur<< surchargé de `Cercle` peut produire la sortie de la partie `Point` de `Cercle` en forçant le type de la référence `c` de `Cercle` pour le convertir en un `Point`. Cette situation produit un appel à l'opérateur<< de `Point` et produit la sortie des coordonnées `x` et `y` du cercle en utilisant la forme du `Point` appropriée.

Cette stratégie de transtypage d'un objet de classe dérivée (i.e. `c` de classe `Cercle`) en classe de base (i.e. `c` interprété comme un `Point`) afin d'exploiter les fonctions membres héritée de la classe de base dans la classe dérivée est exploitée couramment en programmation orientée objet. En effet, elle favorise la *réutilisation du code* (la classe `Cercle` n'a pas à déterminer comment afficher ses coordonnées `x` et `y`).



## Exemple d'héritage : programme principal

```
#include "point.h"
#include "Cercle.h"

int main() {
Point p( 30, 50 );
Cercle c( 2.7, 120, 89 );

cout << "Point p: " << p << "\nCercle c: " << c << "\n";

// Traite Cercle comme un Point (ne voit que la partie de la classe de base)
cout << "\nCercle c (via transtypage): " << static_cast<Point>(c) << "\n";

return 0;
}
```

```
C:\>cercle.exe
Point p: [30, 50]
Cercle c: Centre = [120, 89]; Rayon = 2.70

Cercle c (via transtypage): [120, 89]
C:\>
```

Marco Lavoie

14728 ORD - Langage C++

Le code client (fonction `main`) crée un objet de classe `Point` (variable `p`) et un objet de classe `Cercle` (objet `c`), puis affiche les deux objets via leur opérateur de surcharge respectif `operator<<`.

Le code démontre ensuite l'effet du transtypage de l'objet `c` en objet de classe `Point` afin d'afficher uniquement ses coordonnées. Le transtypage « transforme » temporairement `c` en objet de type `Point`, seulement le temps que l'invocation de `operator<<` (de la classe `Point`) soit exécutée.





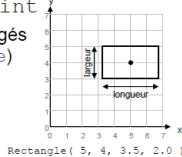
## Exercice 9.1

16

- Récupérez le code source distribué par l'instructeur (Exercice\_9\_1\_dist.zip)

- Dérivez la classe **Rectangle** de **Point**

- Cette-ci doit disposer des attributs protégés **longueur** et **largeur** (de type **double**) et offrir des fonctionnalités similaires à celles de la classe **Cercle**
- Modifiez le programme principal afin qu'il teste la classe **Rectangle**



- N'oubliez pas les conventions d'écriture
- Soumettez votre projet selon les indications de l'instructeur

Marco Lavoie

14728 ORD - Langage C++

Vous devez ajouter deux nouveaux fichiers au projet Visual Studio : **Rectangle.h** et **Rectangle.cpp**, qui doivent contenir respectivement la définition de la classe **Rectangle** dans le premier, et les définitions de fonctions membres de **Rectangle** dans le deuxième.

Implantez dans **Rectangle** des fonctionnalités semblables à celles offertes par la classe **Cercle**, soient

- un constructeur paramétré;
- des accesseur et mutateurs aux attributs membres **longueur** et **largeur**;
- une fonction **aire** retournant l'aire du rectangle.

Augmentez le code client dans **main**, qui teste déjà les classes **Point** et **Cercle**, afin qu'il teste la classe **Rectangle**.



## Pointeurs et hiérarchie de classes

17

- Un pointeur peut contenir l'adresse d'une instance de classe dérivée

```
int main() {
    Point *pointPtr = 0, p( 30, 50 );
    Cercle *cerclePtr = 0, c( 2.7, 120, 89 );

    // Traite Cercle comme un Point (ne voit que la partie de la classe de base)
    pointPtr = &c; // affecte l'adresse de Cercle à pointPtr
    cout << "UnCercle c (via *pointPtr): " << *pointPtr << '\n';

    return 0;
}
```

```
C:\>cercle2.exe
Cercle c (via *pointPtr): [120, 89]
C:\>
```

- Mais l'inverse n'est pas vrai (encore dû au fait que la classe de base n'a pas tous les attributs de la classe dérivée)

```
cerclePtr = &p; // ERREUR: p n'a pas de rayon
```

Marco Lavoie

14728 ORD - Langage C++

Le programme ci-contre crée **pointPtr**, un pointeur vers un objet **Point**, mais y affecte l'adresse d'un objet **Cercle**. Ainsi, l'adresse d'un objet de la classe dérivée (**&c**) est placée dans un pointeur de la classe de base (**pointPtr**), ce qui produit la sortie de l'objet **c** de type **Cercle** en employant l'opérateur **<<** de **Point** via le pointeur déréférencé **\*pointPtr**. Notez que seule la portion **Point** de l'objet **c** est affichée.

En utilisant l'héritage **public**, l'affectation d'un pointeur de classe dérivée en un pointeur de classe de base demeure valable, puisqu'un objet de classe dérivée est *implicitement* un objet de classe de base. Le pointeur de la classe de base ne voit que la partie de la classe de base de l'objet de classe dérivée.

Lors de telles affectations, le compilateur effectue une conversion implicite du pointeur de classe dérivée en pointeur de classe de base.



## Pointeurs et instances dynamiques

18

- Corrolairement, un pointeur peut manipuler des instances de classes dérivées

```
int main() {
    Point *pointPtr1 = new Point( 30, 50 );
    Point *pointPtr2 = new Cercle( 2.7, 120, 89 );

    // Traite Cercle comme un Point (ne voit que la partie de la classe de base)
    cout << "UnCercle c (via *pointPtr2): " << *pointPtr2 << '\n';

    delete pointPtr1;
    delete pointPtr2;

    return 0;
}
```

```
C:\>cercle3.exe
Cercle c (via *pointPtr2): [120, 89]
C:\>
```

- Notez encore une fois que le compilateur suppose que l'instance pointée par **pointPtr2** est de type **Point**

- Aucun rayon n'est affiché

Marco Lavoie

14728 ORD - Langage C++

En programmation orientée objets, un pointeur de classe de base peut se voir affecter l'adresse d'un objet de toute classe dérivée directement ou indirectement de cette classe de base. Dans une telle circonstance, le compilateur « suppose » que l'objet pointé par ce pointeur en est un de la classe de base.

Cette supposition du compilateur n'est pas erronée puisqu'un objet de classe dérivée *hérite* de tous les membres (attributs et fonctions), même **private**, de la classe de base. Conséquemment, cet objet dispose de toutes les fonctionnalités de la classe de base et peut ainsi être considéré comme tel.

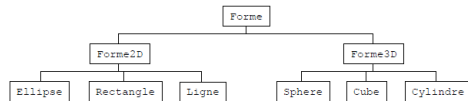
Le fait qu'un membre soit **private** dans la classe de base ne prive pas ces membres des objets de classes dérivées. En fait ces objets disposent des membres **private** hérités de leur classe de base; ils ne peuvent cependant pas y référer directement.



## Pointeurs et instances (suite)

19

- En fait, un pointeur peut contenir l'adresse de toute instance de classe de la hiérarchie dont elle est la classe de base



- Un pointeur de type `*Forme` peut pointer à toute instance de classe de la hiérarchie
- Un pointeur de type `*Forme3D` peut pointer à une instance de `Forme3D`, `Sphere`, `Rectangle` ou `Cylindre`

Marco Lavoie

14728 ORD - Langage C++

Les logiciels de grande envergure sont souvent constitué d'une hiérarchie de classes, et même parfois de plusieurs hiérarchies distinctes. Dans un tel contexte, la classe à la racine d'une telle hiérarchie de classes est en fait la classe de base de toutes les autres classes de la hiérarchie, soit directement (pour les classe directement sous celle racine) ou indirectement (c.à.d. les classes sous ces dernières). Donc un pointeur de la classe racine peut conséquemment contenir l'adresse de tout objet instancié d'une classe de la hiérarchie.

Le fait qu'un pointeur de classe de base puisse pointer à tout objet de classe dérivée (directement ou indirectement) est à la base du *polymorphisme*, que nous introduirons au prochain chapitre.



## Pointeurs et instances (suite)

20

- Cet aptitude d'une instance à être affectée à un pointeur de classe de base s'étend aussi aux alias

```

int main() {
    Cercle c( 2.7, 120, 89 );
    Point &p = c;

    // Traite Cercle comme un Point (ne voit que la partie de la classe de base)
    cout << "Cercle c (via p): " << p << '\n';

    return 0;
}
  
```

- Cette caractéristique du langage constitue un des fondements du *polymorphisme* (chapitre 10)

Marco Lavoie

14728 ORD - Langage C++

Tel un pointeur, une référence de classe de base peut aussi être alias d'un objet de classe dérivée. Dans l'exemple ci-contre, la référence `&p` de type `Point` est alias de l'objet `c` de type `Cercle`. Comme pour le pointeur `pointPtr` de l'exemple précédent, l'affichage de l'objet `c` via la référence `p` invoque l'opérateur `<<` de la classe `Point`, puisque `p` est de type `Point`, et ce même si l'objet référencé est en réalité une instance de la classe `Cercle`.

Comme pour l'exemple précédent, le code client ci-contre affiche seulement les coordonnées du cercle `c`, mais non son rayon (car `Point::operator<<` affiche seulement les attributs `x` et `y` de l'objet).



## Dangers du transtypage

21

- Attention au transtypage via pointeurs
  - Le compilateur ne peut interdire un transtypage invalide mais forcé

```

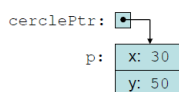
int main() {
    Point *pointPtr = 0, p( 30, 50 );
    Cercle *cerclePtr = 0, c( 2.7, 120, 89 );

    cerclePtr = &p;
    cerclePtr = static_cast< Cercle * >( &p ); // Compile OK

    cout << "Naire du cercle: " << cerclePtr->aire() << '\n'; // ERREUR à l'exécution

    return 0;
}
  
```

- Lors de l'exécution, le programme va planter car l'objet pointé par `cerclePtr` est une instance de `Point` (pas d'attribut `rayon`)



Marco Lavoie

14728 ORD - Langage C++

On ne peut affecter directement un pointeur de classe dérivée à un objet de classe de base. Il s'agit d'une affectation fondamentalement dangereuse, les pointeurs de classes dérivées s'attendant à pointer vers des objets de classes dérivées. Dans ce cas, le compilateur n'effectue pas de conversion implicite. Le programmeur peut cependant forcer l'affectation à l'aide d'un transtypage explicite. Une telle affectation est considérée dangereuse car le compilateur « suppose » que l'objet pointé dispose de tous les attributs et fonctions membres de la classe dérivée, ce qui n'est pas le cas. Si le code source suivant l'affectation dangereuse invoque un membre étant déclaré dans la classe dérivée, le compilateur acceptera cette invocation, même si l'objet pointé ne dispose pas de ce membre.

L'utilisation d'un transtypage explicite pour forcer une telle affectation informe le compilateur que le programmeur est au courant des dangers inhérents à ce type de conversion de pointeurs. Le programmeur porte alors la responsabilité de l'emploi adéquat du pointeur.



## Substitution des fonctions membres héritées

22

- Une classe dérivée peut substituer à une fonction membre héritée d'une classe de base sa propre fonction membre
  - En autant qu'elles aient la même signature

```
class ExA {
public:
    void func( int i )
    { cout << "A1"; }
    void func( float f )
    { cout << "A2"; }
};

class ExB : public ExA {
public:
    void func( float g )
    { cout << "B1"; }
};

int main {
    ExB exb; // instance de ExB
    exb.func( 1 ); // Affiche A1
    exb.func( 1.1 ); // Affiche B1
    return 0;
}
```

Marco Lavoie

14728 ORD - Langage C++

Une classe dérivée peut substituer une fonction membre d'une classe de base en fournissant une nouvelle version de cette fonction avec la même signature. Si la signature est différente, il s'agira alors d'une surcharge de fonction membre et non d'une substitution de fonction membre. Lorsqu'on invoque cette fonction dans la classe dérivée, la version de la classe dérivée est choisie automatiquement. On peut cependant utiliser l'opérateur de résolution de portée (::) pour invoquer explicitement la version de la classe de base à partir d'une fonction de la classe dérivée.



## Types d'héritage

23

- Une dérivation peut être de type **public**, **protected** ou **private**

```
class ExB : public ExA {
    ...
};
```

  - Généralement, les dérivation sont **public**
- Le type d'héritage (ou type de dérivation) restreint l'accessibilité des membres de la classe de base dans la classe dérivée
  - Une classe dérivée ne peut cependant pas élargir l'accessibilité des membres hérités (p. ex. rendre **public** un membre hérité **protected**)

Marco Lavoie

14728 ORD - Langage C++

Lorsqu'on dérive une classe à partir d'une classe de base, l'héritage de cette dernière peut être **public**, **protected** ou **private**. L'emploi de l'héritage **protected** ou **private** demeure rare et ne devrait intervenir qu'avec de nombreuses précautions. Normalement, nous préférons dans le cours 14728ORD l'héritage **public**. Nous présentons cependant brièvement les deux autres formes d'héritage dans ce chapitre.

Les types d'héritage **protected** et **private** ont comme objectif de restreindre l'accessibilité du code client aux membres hérités. Lors d'un héritage **public**, l'accessibilité des membres hérités (attributs et fonctions) de la classe de base demeure la même dans la classe dérivée (p.ex. les membres hérités d'accès **public** dans la classe de base demeurent **public** dans la classe dérivée). Il n'en est cependant pas de même pour les héritages **protected** et **private**, qui changent l'accessibilité aux membres hérités dans la classe dérivée.



## Types d'héritage (suite)

24

- Propagation d'accessibilité des membres à la classe dérivée

	Type d'héritage		
	Héritage <b>public</b>	Héritage <b>protected</b>	Héritage <b>private</b>
Membre dans la classe de base			
Membre <b>public</b>	Membre <b>public</b> dans la classe dérivée	Membre <b>protected</b> dans la classe dérivée	Membre <b>private</b> dans la classe dérivée
Membre <b>protected</b>	Membre <b>protected</b> dans la classe dérivée	Membre <b>protected</b> dans la classe dérivée	Membre <b>private</b> dans la classe dérivée
Membre <b>private</b>	Membre <b>inaccessible</b> dans la classe dérivée	Membre <b>inaccessible</b> dans la classe dérivée	Membre <b>inaccessible</b> dans la classe dérivée

Marco Lavoie

14728 ORD - Langage C++

Lorsqu'une classe est dérivée d'une classe de base **public**, les membres **public** de la classe de base deviennent membres **public** de la classe dérivée, tandis que les membres **protected** de la classe de base deviennent membres **protected** de la classe dérivée. Les membres **private** d'une classe de base ne sont jamais accessibles directement à partir d'une classe dérivée mais demeurent visibles par le biais d'appels vers les fonctions membres **public** et **protected** hérités de la classe de base.

Dérivés à partir d'une classe de base **protected**, les membres **public** et **protected** de la classe de base deviennent membres **protected** de la classe dérivée. Lorsqu'ils sont dérivés d'une classe **private**, les membres **public** et **protected** de la classe de base deviennent membres **private** de la classe dérivée; en d'autres termes les fonctions deviennent des fonctions utilitaires.





## Types d'héritage (suite)

25

- Objectif de changer l'accessibilité aux membres hérités
  - Restreindre l'accès aux membres hérités
    - Le programmeur de la classe dérivée peut décider de restreindre l'accès à ces membres
  - Il ne peut cependant pas élargir l'accessibilité aux membres hérités
    - Seul le programmeur de la classe de base peut élargir cette accessibilité (en modifiant la classe de base)

Marco Lavoie

14728 ORD - Langage C++

Le principal objectif des types d'héritage **protected** et **private** est de restreindre l'accès du code client aux membres hérités. Ces types d'héritage sont généralement exploités lorsqu'une classe dérivée doit offrir une interface distincte de celui de sa classe de base : dans un tel contexte, l'héritage **protected** ou **private** permet à la classe dérivée de bloquer l'accès du code client à l'interface héritée de la classe de base. La classe dérivée peut alors implanter des fonctions alternatives servant d'interface, tout en ayant accès aux membres hérités, telles des fonctions utilitaires.

Un exemple concret d'une telle application est pour une classe dérivée d'implanter des accesseurs et mutateurs pour des attributs membres **public** hérités de la classe de base. Pour forcer le code client à invoquer ces accesseurs et mutateurs pour manipuler les attributs d'un objet de la classe dérivée, il suffit de dériver cette dernière de façon **protected** ou **private**, le code client n'ayant alors plus accès directement aux attributs membres de l'objet qui étaient **public** dans la classe de base.



## Constructeurs et destructeurs

26

- Invocation des constructeurs

```
class Point {
public:
    Point( int a = 0, int b = 0 ) {
        x = a;
        y = b;
    }
protected:
    int x, y;
};

class Cercle: public Point {
public:
    Cercle( double r = 0.0, int u = 0, int v = 0 ) {
        rayon = r;
        x = u;
        y = v;
    }
protected:
    double rayon;
};
```

– Si la classe dérivée n'initialise pas les attributs membres hérités, ceux-ci seront automatiquement initialisés via le constructeur par défaut de la classe de base

Le constructeur initialise son attribut membre (*rayon*) ainsi que ceux hérités (*x* et *y*)

Marco Lavoie

14728 ORD - Langage C++

Puisque la tâche principale d'un constructeur est d'initialiser les attributs membres de l'objet, un constructeur de classe dérivée doit s'assurer que les attributs membres hérités de sa classe de base soient aussi initialisés. Cette tâche est réalisable en autant que la classe dérivée ait accès à ces attributs membres hérités, qui doivent conséquemment être d'accès **protected** ou **public**.

Mais qu'en est-il des attributs membres hérités d'accès **private** dans la classe de base? Le constructeur de la classe dérivée ne peut alors pas initialiser directement ces attributs membres puisque, même si la classe dérivée dispose de ces attributs (puisque'elle en hérite), elle ne peut pas y accéder directement.



## Invocation des constructeurs

27

- Et attributs membres privés de la classe de base ?

```
class Point {
public:
    Point( int a = 0, int b = 0 ) {
        x = a;
        y = b;
    }
private:
    int x, y;
};

class Cercle: public Point {
public:
    Cercle( double r = 0.0, int u = 0, int v = 0 ) {
        rayon = r;
        x = u;
        y = v;
    }
protected:
    double rayon;
};
```

– Alors comment initialiser les attributs hérités *x* et *y* aux valeurs fournies via les paramètres *u* et *v* ?

ERREUR : Cercle n'a pas accès à *x* ni *y* (même si elle dispose de ces attributs)

Marco Lavoie

14728 ORD - Langage C++

Évidemment, il serait possible d'ajouter des mutateurs **public** à la classe de base afin de permettre à la classe dérivée d'initialiser indirectement (c.à.d. via leur mutateur) les attributs membres privés de la classe de base. Il existe cependant une stratégie plus simple et n'exigeant aucune modification) à la classe de base (telle que l'ajout de mutateurs : *invoker le constructeur de la classe de base à même le constructeur de la classe dérivée*.

Mais comme nous l'avons vu précédemment, il est impossible d'invoquer explicitement un constructeur. Alors comment faire?

Les classes dérivées n'héritent pas des constructeurs et des opérateurs d'affectation de la classe de base. Toutefois, les constructeurs et les opérateurs d'affectation des classes dérivées peuvent invoquer les constructeurs et les opérateurs d'affectation de leur classe de base.



## Invocation des constructeurs (suite)

28

- En invoquant un constructeur de la classe de base via un initialiseur

```
class Point {
public:
    Point( int a = 0, int b = 0 ) {
        x = a;
        y = b;
    }
private:
    int x, y;
};

class Cercle: public Point {
public:
    Cercle( double r = 0.0, int u = 0, int v = 0 )
        : Point( u, v ) {
        rayon = r;
    }
protected:
    double rayon;
};
```

– Le constructeur invoqué de la classe de base est exécuté avant celui de la classe dérivée

- Dans l'exemple, les attributs `x` et `y` sont initialisés avant `rayon`

Marco Lavoie

14728 ORD - Langage C++

Puisqu'une classe dérivée hérite des membres de sa classe de base, on doit appeler son constructeur pour initialiser les membres de classe de base de l'objet de classe dérivée lorsqu'un objet d'une classe dérivée est instancié. Un *initialiseur de classe de base*, utilisant la syntaxe d'initialiseur de membre que nous avons vue précédemment, peut être fourni dans le constructeur de la classe dérivée pour invoquer explicitement le constructeur de la classe de base; sinon, le constructeur de la classe dérivée invoquera implicitement le constructeur par défaut de la classe de base.

Un constructeur de classe dérivée invoque toujours en premier le constructeur de sa classe de base afin d'initialiser les membres hérités de celle-ci. Si l'on omet d'implanter un constructeur de classe dérivée, le constructeur par défaut de la classe dérivée invoque le constructeur par défaut de la classe de base.



## Invocation des destructeurs

29

- Puisqu'une classe ne peut avoir qu'un seul destructeur, il est inutile d'invoquer explicitement le destructeur de la classe de base dans le destructeur de la classe héritée
  - Le compilateur l'invoque automatiquement, en ordre inverse d'invocation des constructeurs (i.e. de la classe dérivée vers la classe de base)
  - Concrètement, la syntaxe ne permet pas à un destructeur d'invoquer un destructeur de la classe de base

Marco Lavoie

14728 ORD - Langage C++

Si une classe dispose d'un destructeur, celui-ci est implicitement invoqué lors de la destruction d'un objet de cette classe. Il en est de même si l'objet détruit en est un instancié d'une classe dérivée. Si cette dernière dispose d'un destructeur, il sera invoqué à la destruction de l'objet, ainsi que le destructeur de la classe dérivée si celle-ci en possède un.

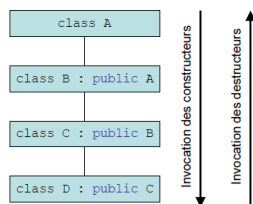
Un destructeur de classe dérivée n'a pas à explicitement invoquer le destructeur de la classe de base; cette invocation sera faite implicitement par le compilateur. En fait, même si la classe dérivée ne dispose pas d'un destructeur mais que la classe de base en possède un, ce dernier sera tout de même invoqué lors de la destruction d'un objet de la classe dérivée.



## Constructeurs et destructeurs

30

- En résumé, lors de l'instanciation -----> D objet;
- Les constructeurs sont explicitement invoqués de la classe de base vers la classe dérivée
- Les destructeurs sont implicitement invoqués de la classe dérivée vers la classe de base



Marco Lavoie

14728 ORD - Langage C++

Comme mentionné précédemment, un constructeur de base dérivée invoque (explicitement ou implicitement) en premier le constructeur de la classe de base avant de s'exécuter. Dans l'exemple ci-contre, lors de l'instanciation d'un objet de classe dérivée `D`, le constructeur de cette classe invoquera celui de la classe `C` avant de s'exécuter, qui invoquera celui de la classe `B` avant de s'exécuter, qui invoquera celui de la classe `A` avant de s'exécuter. Ainsi, l'ordre d'exécution des constructeurs sera celui de `A`, suivi de celui de `B`, suivi de celui de `C` et finalement celui de `D`.

Les destructeurs sont par contre invoqués dans l'ordre inverse de celui des constructeurs; un destructeur de classe dérivée est donc invoqué avant son destructeur de classe de base. Dans l'exemple ci-contre, les destructeurs seront donc exécutés dans l'ordre suivant lors de la destruction d'un objet instancié de la classe `D`: le destructeur de `D` sera exécuté, suivi de celui de `C`, suivi de celui de `B` et finalement celui de la classe `A` sera exécuté en dernier.



## Exercice 9.2

31

- Poursuivez l'exercice 9.1
  - Déclarez **private** les **attributs** membres des trois classes (**Point**, **Cercle** et **Rectangle**)
  - Modifiez le code source en conséquence afin qu'il soit fonctionnel
- N'oubliez pas les conventions d'écriture
- Soumettez votre projet selon les indications de l'instructeur

Marco Lavoie

14728 ORD - Langage C++

Dans cet exercice vous devez modifier les constructeurs des classes dérivées **Point** et **Cercle** afin qu'ils invoquent (via un initialiseur) le constructeur de leur classe de base **Point**.



## Études de cas

32

### • Classe Point

```
class Point {
    friend ostream &operator<<( ostream &, const Point & );
public:
    Point( int a = 0, int b = 0 ) { setPoint( a, b ); } // Constructeur
    void setPoint( int, int ); // change les coordonnées
    int getX() const { return x; } // retourne la coordonnée x
    int getY() const { return y; } // retourne la coordonnée y
protected: // accessible aux classes dérivées
    int x, y; // coordonnées du point
};

// Change les coordonnées x et y
void Point::setPoint( int a, int b ) {
    x = a;
    y = b;
}

// Produit la sortie du Point
ostream &operator<<( ostream &sortie, const Point &p ) {
    sortie << '[' << p.x << ", " << p.y << ']'<< '\n';

    return sortie; // permet la mise en cascade
}
```

Marco Lavoie

14728 ORD - Langage C++

Examinons maintenant l'exemple principal de ce chapitre, soit la hiérarchie point, cercle et cylindre. Nous développons premièrement la classe **Point** et présentons ensuite la classe **Cercle** dérivée de la classe **Point**. Finalement nous présentons une classe dérivée de la classe **Cercle**, soit la classe **Cylindre**.

Le code ci-contre résume la classe **Point** que nous avons étudiée aux pages précédentes de ce chapitre. Cette classe de base dispose de deux attributs membres **protected** (les coordonnées **x** et **y** du point) ainsi que leurs accesseurs et mutateur **public**. Un constructeur par défaut inline initialise toute instance de **Point** aux coordonnées (0, 0), à moins que d'autres coordonnées soient fournies aux constructeur.

Puisque les attributs membres de **Point** sont d'accès **protected**, les classes dérivées **Cercle** et **Cylindre** peuvent directement accéder à ces derniers.



## Études de cas (suite)

33

### • Classe Cercle

```
class Cercle : public Point {
    friend ostream &operator<<( ostream &, const Cercle & );
public:
    // Constructeur par défaut
    Cercle( double r = 0.0, int x = 0, int y = 0 );
    void setRayon( double ); // change le rayon
    double getRayon() const; // retourne le rayon
    double aire() const; // calcule l'aire
protected: // accessible aux classes dérivées
    double rayon; // rayon du Cercle
};

// Constructeur de Cercle appelle le constructeur de Point
// avec un initialiseur membre et initialise rayon
Cercle::Cercle( double r, int a, int b )
: Point( a, b ) // appelle le constructeur de la classe de base
{ setRayon( r ); }

// Change le rayon
void Cercle::setRayon( double r )
{ rayon = ( r >= 0 ? r : 0 ); }
```

Marco Lavoie

14728 ORD - Langage C++

Le code ci-contre présente la définition de la classe **Cercle** qui hérite de la classe **Point** par un héritage **public**. Cela signifie que l'interface **public** de **Cercle** comprend les fonctions membres **public** de **Point** (**setPoint**, **getX** et **getY**), de même que les fonctions membres **public** de **Cercle** (**getRayon**, **setRayon** et **aire**).

Le constructeur par défaut de la classe dérivée **Cercle** invoque celui de la classe de base **Point** via un initialiseur. Cet initialiseur permet de passer en arguments au constructeur de **Point** les coordonnées reçues en paramètres par le constructeur de **Cercle** (paramètres **a** et **b**). Ce dernier peut ainsi initialiser ses attributs membres hérités **x** et **y** sans les manipuler directement. Notez cependant que le constructeur de **Cercle** aurait pu explicitement initialiser ses attributs membres hérités **x** et **y** puisque ceux-ci sont d'accès **protected** dans la classe de base **Point**.



## Études de cas (suite)

34

```
// Retourne le rayon
double Cercle::getRayon() const
{ return rayon; }

// Calcule l'aire du Cercle
double Cercle::aire() const
{ return 3.14159 * rayon * rayon; }

// Produit la sortie d'un cercle selon la forme:
// Centre = [x, y]; Rayon = ###
ostream & operator<<( ostream &sortie, const Cercle &c ) {
    sortie << "Centre = " << static_cast< Point >( c )
    << " "; Rayon = "
    << setiosflags( ios::fixed | ios::showpoint )
    << setprecision( 2 ) << c.rayon;

    return sortie; // permet les appels en cascade
}
```

Marco Lavoie

14728 ORD - Langage C++

Notez que la fonction d'`operator<<` surchargée de `Cercle`, amie (`friend`) de la classe `Cercle`, peut produire la sortie de la partie `Point` du `Cercle` en forçant le type de la référence de `c`, de type `Cercle`, pour le convertir en un `Point`. Ce procédé provoque un appel vers `operator<<` de `Point`, produisant la sortie des coordonnées `x` et `y` en utilisant le formatage `Point` approprié.



## Études de cas (suite)

35

### • Classe Cylindre

```
class Cylindre : public Cercle {
    friend ostream operator<<( ostream &, const Cylindre & );
public:
    // Constructeur par défaut.
    Cylindre( double h = 0.0, double r = 0.0,
              int x = 0, int y = 0 );

    void setHauteur( double ); // change la hauteur
    double getHauteur() const; // retourne la hauteur
    double aire() const; // calcule et renvoie l'aire
    double volume() const; // calcule et renvoie le volume

protected:
    double hauteur; // hauteur du Cylindre
};

// Le constructeur de Cylindre appelle le constructeur de Cercle
Cylindre::Cylindre( double h, double r, int x, int y )
: Cercle( r, x, y ) // appelle le constructeur de la classe de base
{ setHauteur( h ); }
```

Marco Lavoie

14728 ORD - Langage C++

Le code ci-contre implante la classe `Cylindre`, dérivée de la classe de base `Cercle` puisqu'un cylindre dispose non seulement d'une hauteur, mais aussi d'un rayon (hérité de `Cercle`) et de coordonnées (héritées de `Point`). Puisque la classe `Cylindre` hérite de la classe `Cercle` avec un héritage `public`, l'interface `public` de `Cylindre` comprend les membres `public` de `Cercle` (`getRayon` et `setRayon`) ainsi que les membres `public` de `Point` (`setPoint`, `getX` et `getY`). Notez que la classe `Cylindre` surcharge la fonction membre `aire` héritée de `Cercle` car l'aire d'un cylindre (c.à.d. la surface de celui-ci) n'est pas calculée de la même façon que celle d'un cercle.

Le constructeur de `Cylindre` invoque explicitement celui de la classe de base `Cercle`, qui à son tour invoque explicitement celui de la classe de base `Point`. Un objet de classe `Cylindre` voit donc l'ensemble de ses attributs membres initialisés.



## Études de cas (suite)

36

```
// Ajuste la hauteur du Cylindre
void Cylindre::setHauteur( double h )
{ hauteur = ( h >= 0 ? h : 0 ); }

// Lit la hauteur du Cylindre
double Cylindre::getHauteur() const
{ return hauteur; }

// Calcule l'aire du Cylindre (i.e. l'aire de la surface)
double Cylindre::aire() const {
    return 2 * Cercle::aire() +
           2 * 3.14159 * rayon * hauteur;
}

// Calcule le volume du Cylindre
double Cylindre::volume() const
{ return Cercle::aire() * hauteur; }

// Produit la sortie des dimensions du Cylindre
ostream operator<<( ostream &sortie, const Cylindre &c ) {
    sortie << static_cast< Cercle >( c ) << " "; Hauteur = " << c.hauteur;

    return sortie; // permet les appels en cascade
}
```

Marco Lavoie

14728 ORD - Langage C++

Portez une attention particulière aux fonctions membres `aire` et `volume` de la classe `Cylindre`. Étudions premièrement la fonction `aire`. L'aire d'un cylindre correspond à sa surface, c'est-à-dire la somme de l'aire de ses deux extrémités additionnée à la surface de sa partie latérale, soit un rectangle aux dimensions correspondant à la hauteur du cylindre et au périmètre de ses extrémités (c.à.d.  $2 * \pi * rayon$ ). Pour calculer l'aire des extrémités, la fonction `aire` de `Cylindre` invoque explicitement la fonction membre `aire` de sa classe de base `Cercle` en utilisant l'opérateur de portée :

```
Cercle::aire()
```

Notez que l'opérateur de portée est requis dans cette invocation, sinon la fonction `aire` de `Cylindre` s'invoquerait elle-même à l'infini.

La fonction membre `volume` invoque aussi explicitement la fonction membre `aire` héritée de `Cercle` pour obtenir l'aire d'une extrémité du cylindre.



## Études de cas (suite)

37

### • Programme principal

```
#include "point2.h"
#include "cercle2.h"
#include "cylindre2.h"

int main() {
    // Crée un objet Cylindre
    Cylindre cyl( 5.7, 2.5, 12, 23 );

    // Utilise des fonctions get pour afficher le Cylindre
    cout << "La coordonnée X et Y sont " << cyl.getX() << ", " << cyl.getY()
    << "\nLe rayon vaut " << cyl.getRayon()
    << ", la hauteur vaut " << cyl.getHauteur() << "\n\n";

    // Utilise des fonctions set pour changer les attributs de Cylindre
    cyl.setHauteur( 10 );
    cyl.setRayon( 4.25 );
    cyl.setPoint( 2, 2 );
    cout << "Le nouvel emplacement, le rayon et la hauteur de cyl sont:\n"
    << cyl << '\n';

    cout << "L'aire de cyl vaut: " << cyl.aire() << '\n';
}
```

Marco Lavoie

14728 ORD - Langage C++

Le code client des classes, soit le programme principal, instancie un objet de la classe **Cylindre** et utilise par la suite des fonctions accesseurs pour obtenir les informations relatives à l'objet **cylindre**. Une fois de plus, **main** n'est ni fonction membre ni **friend** de la classe **Cylindre**. Par la suite, le programme utilise les fonctions mutateurs **setHauteur**, **setRayon** et **setPoint** pour initialiser à zéro la hauteur, le rayon et les coordonnées **x** et **y** du cylindre.



## Études de cas (suite)

38

### • Programme principal (suite)

```
// Affiche le Cylindre comme un Point
Point pRef = cyl; // pRef "croit" qu'il est un Point
cout << "\nLe Cylindre affiché comme un Point est: " << pRef << "\n\n";

// Affiche le Cylindre comme un Cercle
Cercle cRef = cyl; // cRef "croit" qu'il est un Cercle
cout << "Le Cylindre affiché comme un Cercle est:\n" << cRef
<< ", Aire: " << cRef.aire() << endl;

return 0;
}
```

```
C:\>cylindre.exe
La coordonnée X et Y sont: 12, 23
Le rayon vaut 2.5, la hauteur vaut 5.7

Le nouvel emplacement, le rayon et la hauteur de cyl sont:
Centre = [2, 2]; Rayon = 4.25; Hauteur = 10.00
L'aire de cyl vaut: 380.53

Le Cylindre affiché comme un Point est: [2, 2]

Le Cylindre affiché comme un Cercle est:
Centre = [2, 2]; Rayon = 4.25; Aire = 56.74
C:\>
```

Marco Lavoie

14728 ORD - Langage C++

Finalement, le programme initialise la variable de référence **pRef**, de type « référence à un objet **Point** » (**Point &**), à l'objet **Cylindre** appelé **cyl**. Le programme affiche ensuite **pRef** qui, en dépit de son initialisation avec un objet **Cylindre**, « croit » qu'il est un objet **Point** et l'objet **Cylindre** s'affiche en réalité comme un objet **Point**.

Par la suite, le programme initialise la variable de référence **cRef**, de type référence à un objet **Cercle** (**Cercle &**), à l'objet **cyl**. Le programme affiche finalement **cRef** qui, en dépit de son initialisation avec un objet **Cylindre**, « croit » qu'il est un objet **Cercle** et l'objet **Cylindre** s'affiche en pratique comme un objet **Cercle**: la sortie de l'**aire** du **Cercle** est donc affichée.

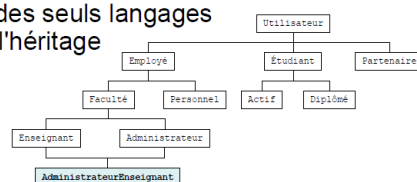
Cet exemple démontre la justesse de l'héritage public de même que la façon de définir et de référencer des membres **protected** dans une classe dérivée.



## Héritage multiple

39

### • C++ est un des seuls langages à supporter l'héritage multiple



### • Exploiter l'héritage multiple est très complexe

- C'est pourquoi la plupart des nouveaux langages (C#, Java, VB) ne le supportent pas
- Même en C++, il est fortement conseillé d'éviter l'héritage multiple

Marco Lavoie

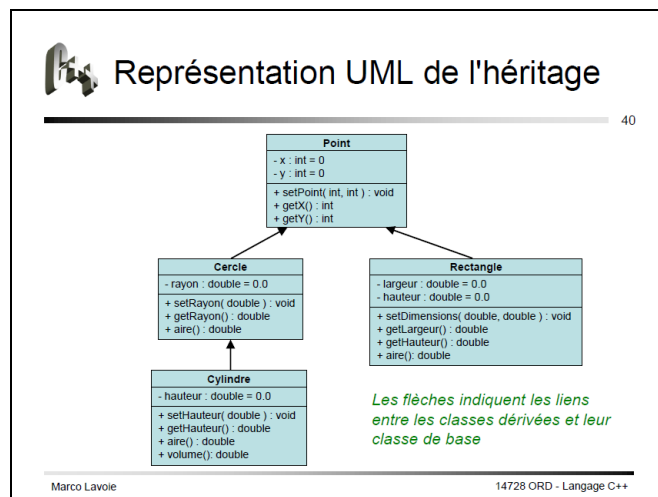
14728 ORD - Langage C++

Jusqu'ici, nous avons discuté d'héritage simple, dans lequel chaque classe est dérivée à partir d'au plus une seule classe de base. Nous pouvons cependant dériver une classe à partir de plus d'une classe de base; ce procédé s'appelle l'*héritage multiple*. L'héritage multiple signifie qu'une classe dérivée hérite des membres de plusieurs classes de base n'ayant par nécessairement de relations entre elles. Cette caractéristique du C++ favorise des formes importantes de réutilisation de logiciels, mais elle engendre habituellement une pléiade de problèmes ambigus.

Dans l'exemple ci-contre, la classe dérivée **AdministrateurEnseignant** regroupe les fonctionnalités de l'**Enseignant** ainsi que celles de l'**Administrateur**.

L'héritage multiple constitue une puissante caractéristique mais peut rendre le système plus complexe. Il est préférable de ne pas l'utiliser lorsque l'héritage simple peut faire le travail, ce qui est généralement le cas.





Le *diagramme de classes* UML est un schéma utilisé en génie logiciel pour présenter les classes et les interfaces des systèmes ainsi que les différentes relations entre celles-ci.

Les classes peuvent être liées entre elles grâce au mécanisme d'héritage qui permet de mettre en évidence des relations de parenté. Une flèche illustre la relation de parenté entre la classe dérivée (origine de la flèche) à sa classe de base (destination de la flèche). D'autres relations sont possibles entre des classes, chacune de ces relations est représentée par un arc spécifique dans le diagramme de classes.

## Erreurs de programmation

- Traiter une instance de classe de base comme instance de classe dérivée constitue une erreur de syntaxe ou de logique (s'il y a transtypage)
  - Affecter une instance de classe de base à un pointeur de classe dérivée
- Définir un attribut de classe dérivée du même nom qu'un attribut hérité de la classe de base
- Tenter d'élargir l'accessibilité d'un membre hérité (p. ex. rendre `public` un membre hérité `protected`)
- Tenter d'accéder à un membre hérité `private`

Marco Lavoie 14728 ORD - Langage C++

Le fait d'utiliser un transtypage explicite pour convertir un pointeur de classe de base en un pointeur de classe dérivée et de faire ensuite référence à des membres de classe dérivée n'existant pas dans cet objet provoque généralement des erreurs de logique à l'exécution.

Une autre erreur généralement fatale consiste à substituer dans une classe dérivée une fonction membre d'une classe de base, puis d'invoquer la fonction de la classe de base dans celle substituée dans la classe dérivée tout en omettant d'utiliser l'opérateur de résolution de portée (`::`). Le fait de ne pas utiliser l'opérateur de résolution de portée pour référencer la fonction membre de la classe de base produit une récursion infinie puisque, en réalité, la fonction membre de la classe dérivée s'invoque elle-même. Cette situation finit par épuiser la mémoire du système, produisant une erreur fatale à l'exécution.

## Bonnes pratiques de programmation

- Déclarer `private` les membres d'une classes et n'utiliser `protected` qu'en dernier recours
  - Alternative à `protected` : fournir des accesseurs et mutateurs dans la classe de base
- Toujours invoquer explicitement (via initialiseur) un constructeur de la classe de base dans les constructeurs de la classe dérivée
  - Pour s'assurer que les attributs hérités sont adéquatement initialisés
- Favoriser l'héritage `public`
  - L'héritage `protected` et `private` sont rarement essentiels
- Éviter l'héritage multiple

Marco Lavoie 14728 ORD - Langage C++

Le fait de privilégier la déclaration `private` les membres d'une classe favorise l'encapsulation et la réutilisation du code. En effet, un attribut membre ou une fonction membre `private` ne peut être invoqué par aucun code client ni par les fonctions membres des classes dérivées, ce qui offre une grande latitude pour modifier ces membres sans occasionner une refonte de ces codes.

L'héritage multiple constitue une puissante caractéristique lorsqu'elle est utilisée convenablement. Il est cependant difficile d'exploiter adéquatement cette caractéristique du langage C++, ce qui en fait généralement un « cadeau empoisonné ». La plupart des autres langages orientés objets, tels que *Java*, *C#* et *VB*, ne supportent pas l'héritage multiple par soucis de facilité de maintenance du code source des applications.



## Devoir #8

43

- Poursuivez l'exercice 9.1
  - Dérivez les classes suivantes
    - **Sphere** dérivée de **Cercle**
    - **Boite** dérivée de **Rectangle**
  - Ces classes doivent pouvoir s'afficher ainsi que retourner leur aire et volume
  - Créez un programme principal testant les deux nouvelles classes
- Respectez l'échéance imposée par l'instructeur
- Soumettez votre projet selon les indications de l'instructeur
  - **Attention** : respectez à la lettre les instructions de l'instructeur sur la façon de soumettre vos travaux, *sinon la note EC pourrait être attribuée à ceux-ci*

Marco Lavoie

14728 ORD - Langage C++

Ce devoir consiste à poursuivre le développement de la hiérarchie de classes étudiée dans ce chapitre (**Point**, **Cercle** et **Cylindre**), en y ajoutant de nouvelles classes de base et classes dérivées.

N'oubliez pas que chaque classe doit être implantée dans deux fichiers séparés : un fichier `.h` (p.ex. `Sphere.h`) implantant la définition de la classe, et un fichier `.cpp` (p.ex. `Sphere.cpp`) implantant les fonctions membres de la classe.



## Pour la semaine prochaine

44

- Vous devez relire le contenu de la présentation du chapitre 9
  - Il y aura un **quiz** sur ce contenu au prochain cours
    - À livres et ordinateurs fermés
  - Profitez-en pour réviser le contenu des chapitres précédents

Marco Lavoie

14728 ORD - Langage C++

En début de classe la semaine prochaine vous aurez à répondre à des questions sur le C++ sans consultation du matériel pédagogique. Vous êtes donc fortement encouragé à relire les notes de cours du chapitre 9.

Profitez-en pour réviser le contenu des chapitres précédents.