



Centre Informatique pour les **L**ettres
et les **S**ciences **H**umaines

Apprendre C++ avec Qt : Leçon 13

Classes dérivées

| | |
|---|---|
| 1 - Notion de spécialisation | 2 |
| 2 - Le mécanisme de dérivation | 3 |
| Création d'une classe dérivée..... | 3 |
| Ce que l'héritage ne transmet pas..... | 4 |
| Construction et destruction d'une instance d'une classe dérivée..... | 4 |
| Initialisation des membres hérités | 4 |
| Règles d'accès aux membres hérités publiquement..... | 5 |
| Le cas des membres statiques | 6 |
| Ce que la dérivation rend possible | 6 |
| Ce que la dérivation ne suffit pas à rendre possible | 7 |
| 3 - Hiérarchies de classes | 7 |
| 4 - Dérivation multiple | 7 |
| 5 - Dérivation protégée et dérivation privée..... | 8 |
| 6 - Bon, c'est gentil tout ça, mais ça fait déjà 8 pages. Qu'est-ce que je dois vraiment en retenir ? | 9 |

Jusqu'à présent, les rapports liant les différentes classes que nous avons étudiées se limitent à deux cas de figure : une classe peut en accepter une autre comme amie, et elle peut comporter un membre qui est une instance d'une autre classe. Le processus de dérivation, qui est l'objet de cette Leçon et de la prochaine, crée entre les classes un lien d'une toute autre nature.

1 - Notion de spécialisation

Il arrive que des types différents possèdent des caractéristiques communes qui justifient que certains traitements soient appliqués à des objets en ignorant duquel de ces types ces objets sont des instanciations.

Imaginons, par exemple, que nous cherchions à gérer une entreprise de location de véhicules. Cette entreprise possède une flotte qui comporte toutes sortes de véhicules et, notamment, des camions et des voitures. Ces deux types de véhicules sont décrits par des classes différentes, parce que leurs caractéristiques pertinentes pour notre programme sont très différentes : les voitures sont décrites en termes de catégorie tarifaire, type de carburant utilisé, équipements disponibles, alors que les camions sont décrits en termes de catégorie de permis de conduire exigé, volume de chargement, poids du chargement autorisé, etc... Lors de certaines opérations, notre programme est cependant amené à parcourir l'ensemble des véhicules sans s'intéresser à la distinction entre voitures et camions. Un exemple de traitement de ce genre pourrait être le calcul de la valeur totale de la flotte, par addition des valeurs estimées de chacun des véhicules.

Bien entendu, pour qu'un tel traitement soit concevable, il ne doit faire appel qu'à des caractéristiques qui sont communes à tous les types concernés.

On ne pourra pas, par exemple, faire le calcul de la moyenne des poids de chargement autorisés par les véhicules de la flotte si cette donnée n'est disponible que pour les camions. La valeur estimée est, par contre, sans doute disponible pour tous les types de véhicules.

Pour pouvoir manipuler de façon indifférenciée des objets de types différents, il faut trouver à ces types une désignation commune. L'approche adoptée par C++ est de considérer que les caractéristiques communes constituent une classe de base, qui peut être utilisée pour définir des classes publiquement dérivées, qui sont autant de spécialisations de la classe de base.

Dans notre exemple, nous serions donc conduits à concevoir une classe de base (que nous pourrions nommer `CVehicule`) et qui définirait les aspects communs à tous les types de véhicules (la valeur estimée, notamment). Des classes telles que `CVoiture` et `CCamion` peuvent être dérivées de `CVehicule`, ce qui permet de leur conférer les caractéristiques qui leur sont propres, tout en garantissant qu'elles présentent exactement les caractéristiques qui doivent leur être communes.

La propriété essentielle du processus de dérivation est que

Les instances d'une classe dérivée publiquement sont aussi des instances de la classe de base.

En d'autres termes, nos `CVoiture` (tout comme nos `CCamion`) seront des `CVehicule`...

Cette propriété est essentielle car, grâce à elle, il est possible de stocker, dans un pointeur destiné à recevoir l'adresse d'une instance de la classe de base, l'adresse d'une instance de l'une des classes dérivées. Un tel pointeur permet donc de traiter un ensemble d'objets de types différents, en ne s'intéressant qu'aux aspects communs à tous ces types.

Dans notre exemple, le même pointeur sur `CVehicule` pourra donc contenir à certains moments l'adresse d'un `CCamion`, à d'autres l'adresse d'une `CVoiture`, ou celle d'une instance de n'importe laquelle des classes dérivées (publiquement) de `CVehicule`. Ce pointeur peut donc être utilisé pour parcourir l'intégralité de la flotte et faire, par exemple, la somme des valeurs estimées des véhicules.

Il s'agit là de la véritable raison d'être du processus de dérivation de classes, au point que, dans le langage courant,

Lorsqu'on dit simplement d'une classe qu'elle est dérivée d'une autre, on veut dire qu'elle en dérive publiquement.

Il existe également deux autres formes de dérivation (protégée et privée), mais leur usage est beaucoup plus exceptionnel, et elles ne feront ici l'objet que d'une présentation succincte.

2 - Le mécanisme de dérivation

Le principe fondamental de la dérivation est que la classe dérivée comporte implicitement tous les membres définis dans la classe de base. Cette transmission est généralement connue sous le nom d'héritage, et l'on dit donc couramment que la classe dérivée hérite des membres définis dans la classe de base.

Le terme "héritage" est particulièrement heureux, mais son succès mondain conduit malheureusement souvent à des contresens quant à la nature profonde du processus de dérivation, qui n'est pas un simple moyen de transfuser du code d'une classe vers une autre pour éviter d'avoir à le réécrire.

Pour simplifier la discussion, admettons que nous disposons d'une classe ainsi définie :

```

1 class CVehicle
2 {
3 public:
4     CVehicle(double valeurInitiale = 0) : argus(valeurInitiale) {}
5     double valeurEstimee() const {return argus;}
6     void valeurEstimee(double nouvelle) {argus = nouvelle;}
7 protected:
8     double argus;
9 };

```

Création d'une classe dérivée

La création d'une classe dérivée exige que le nom de la **classe créée** soit suivi de deux points, de la spécification du **type de dérivation** souhaitée et du nom de la **classe de base** devant être utilisée :

```

1 class CVoiture : public CVehicle //les CVoiture SONT des CVehicle
2 {
3 };

```

Bien que la définition de cette classe soit ostensiblement vide, toute instance de CVoiture possède une variable membre nommée argus et deux fonctions membre nommées valeurEstimee(). La classe CVoiture permet donc déjà d'écrire des choses comme

```

1 CVoiture titine; //création d'une instance
2 titine.valeurEstimee(1000); //modification de sa valeur
3 double prix = titine.valeurEstimee(); //utilisation de sa valeur

```

Bien entendu, si la classe dérivée ne comporte aucune caractéristique supplémentaire par rapport à la classe de base, elle ne présente aucun intérêt. Les caractéristiques supplémentaires en question sont ajoutées en déclarant des membres propres à la classe CVoiture, qui viennent simplement s'ajouter aux membres hérités de CVehicle :

```

1 enum E_CATEGORIE {COMPACTE, MOYENNE, FAMILIALE, LUXE, CAT_INCONNUE};
2 enum E_CARBURANT {ESSENCE, GAZOLE, ELECTRICITE, CARB_INCONNU};
3 enum E_BOITE_VITESSE {AUTO, MANUELLE, BOITE_INCONNUE};
4 class CVoiture : public CVehicle //les CVoiture SONT des CVehicle
5 {
6 public:
7     double prixLocation();
8 protected:
9     E_CATEGORIE categorie;
10    E_CARBURANT carburant;
11    E_BOITE_VITESSE bv;
12 };

```

La façon dont la fonction CVoiture::prixLocation() détermine la valeur qu'elle renvoie est sans grand intérêt dans le cas présent. Imaginons simplement que cette fonction effectue un calcul où interviennent les caractéristiques de l'instance au titre de laquelle elle est appelée.

Ce que l'héritage ne transmet pas

Les constructeurs, le destructeur et les fonctions définissant l'opérateur d'affectation ne sont jamais hérités.

La raison de ces exceptions est assez évidente : étant donné qu'une classe dérivée peut comporter des variables membre qui n'existent pas dans la classe de base, un constructeur de la classe de base ne peut pas initialiser complètement une instance d'une classe dérivée. Le même argument vaut pour le destructeur et les opérateurs d'affectation de la classe de base, qui ne sauraient traiter correctement des membres qui sont propres à une classe dérivée et dont ils ignorent nécessairement tout.

Les relations d'amitié ne sont pas non plus transmises par héritage. Si une classe dérivée doit accepter comme amie une classe (ou une fonction), elle doit exprimer elle-même cette acceptation, même si la classe (ou la fonction) en question est déjà amie de la classe de base.

Construction et destruction d'une instance d'une classe dérivée

L'instanciation d'une classe dérivée implique nécessairement l'exécution d'un constructeur de la classe de base (qui initialise les membres hérités), puis celle d'un constructeur de la classe dérivée (qui initialise les membres propres à celle-ci).

Symétriquement, lors de la disparition d'une instance d'une classe dérivée, l'exécution du destructeur de cette classe (qui "fait le ménage" dans les membres propres à cette classe) s'accompagne automatiquement d'un appel au destructeur de la classe de base (qui "fait le ménage" dans les membres hérités).

Si vous définissez explicitement le destructeur d'une classe dérivée, aucun appel au destructeur de la classe de base ne doit y figurer. L'exécution de ce destructeur est rendue indispensable dans ce contexte par la relation d'héritage liant les deux classes, et le langage assure qu'il sera implicitement invoqué au moment judicieux.

Initialisation des membres hérités

Dans notre exemple, la classe `CVehicule` définit un constructeur qui accepte un paramètre permettant d'initialiser la valeur estimée du véhicule. On peut donc écrire

```
CVehicule unVehicule(10000); //on peut créer des CVehicule d'une valeur quelconque
```

La classe `CVoiture`, par contre, ne définit explicitement aucun constructeur. Comme les constructeurs ne sont pas héritables, elle ne dispose que des constructeurs automatiquement générés par le compilateur (c'est à dire, au maximum, un constructeur par défaut et un constructeur par copie). Il n'est donc pas possible d'écrire

```
CVoiture batmobile(9999999); //IMPOSSIBLE : le constructeur requis n'existe pas !
```

Dans la version générée automatiquement du constructeur par défaut d'une classe dérivée, les membres hérités sont initialisés par appel du constructeur par défaut de la classe de base.

Nos `CVoiture` ont donc une valeur estimée initiale nulle.

Pour rendre possible l'initialisation avec une valeur arbitraire des instances des classes dérivées de `CVehicule`, il faut pourvoir ces classes de constructeurs explicitement définis. La **liste d'initialisation** permet alors de choisir quel **constructeur de la classe de base** va être utilisé pour initialiser les membres hérités. Si nous négligeons ses caractéristiques spécifiques, la classe `CCamion` pourrait, par exemple, procéder ainsi :

```
1 class CCamion : public CVehicule
2 {
3 public:
4     CCamion() : CVehicule(100) {}
5 };
```

Définie de cette façon, la classe `CCamion` donne naissance à des instances dont la valeur estimée initiale est de 100. Le constructeur de `CCamion` peut, bien entendu, être lui-même doté d'un **paramètre** lui indiquant quelle valeur transmettre au **constructeur** de `CVehicule` :

```
1 class CCamion : public CVehicle
2 {
3 public:
4     CCamion(double valeurInitiale = 0) : CVehicle(valeurInitiale) {}
5 };
```

Il devient alors possible de créer des CCamion d'une valeur initiale quelconque.

Règles d'accès aux membres hérités publiquement

Pour une fonction membre d'une classe dérivée, la possibilité d'accéder aux membres hérités de la classe de base dépend des privilèges d'accès associés à ceux-ci.

S'il s'agit d'un membre **public** ou d'un membre **protégé** de la classe de base, aucune restriction n'est appliquée et les fonctions membre de la classe dérivée peuvent accéder librement au membre hérité.

Dans l'état actuel de la classe CVehicle, la fonction CVoiture::prixLocation(), par exemple, peut donc parfaitement utiliser la variable `argus`, bien que cette variable soit protégée.

Remarquez que cette règle ne s'applique que pour l'accès aux membres de l'instance au titre de laquelle la fonction est exécutée. Pour les autres instances, une fonction membre propre à la classe dérivée n'aura accès aux membres hérités que s'ils sont publics.

Si la fonction CVoiture::prixLocation() possède, par exemple, une variable locale de type CVehicle, il lui reste interdit d'accéder au membre `argus` de celle-ci, puisqu'il est protégé.

Le langage C++ comporte un troisième régime d'accès : lorsqu'un membre de la classe de base est **privé**, seules les fonctions membres ou amies de la classe de base y ont accès.

En dehors du contexte des fonctions membre d'une classe dérivée, les membres protégés et privés ont exactement les mêmes propriétés. C'est ce qui explique que nous n'avons pas rencontré cette distinction au cours des Leçons précédentes.

Ceci ne signifie pas que la classe dérivée possède des membres auxquels aucune de ses fonctions n'a accès. Les fonctions membre de la classe dérivée qui sont elles-même héritées de la classe de base ont en effet accès à tous les membres hérités (puisque ces fonctions sont, par définition, également membre de la classe de base). La signification de la "privatisation" est donc claire : la manipulation des membres concernés est du seul ressort des fonctions membres définies au niveau de la classe de base, ce qui implique qu'elle sera la même pour toutes les classes dérivées.

Dans le cas de notre classe CVehicle, il serait donc sans doute préférable d'écrire :

```
1 class CVehicle
2 {
3 public:
4     CVehicle(double valeurInitiale = 0) : argus(valeurInitiale) {}
5     double valeurEstimee() const {return argus;}
6     void valeurEstimee(double nouvelle) {argus = nouvelle;}
7 private:
8     double argus;
9 };
```

Cette modification du statut du membre `argus` a pour effet d'imposer également aux membres des classes dérivées de CVehicle la contrainte subie par le code qui se contente d'instancier la classe : l'accès à l'information n'est plus possible que via une fonction membre.

Imposer cette contrainte présente toujours le même intérêt fondamental (cf. Leçon 9), qui est d'isoler efficacement la signification l'information (qui intéresse le code utilisateur) de la façon dont celle-ci est représentée (qui ne concerne que le code définissant la classe). Dans le cas d'une classe de base, choisir entre des membres protégés et des membres privés, c'est décider si l'on considère les classes dérivées comme du code utilisateur ou comme du code participant à la définition, non pas d'une classe, mais d'un ensemble de classes étroitement liées. Etant donné qu'une classe de base peut comporter à la fois des membres protégés et des membres privés, cette décision peut, selon les besoins, être nuancée de façon très fine.

Le cas des membres statiques

Lorsqu'une classe de base comporte un membre statique (cf. [Annexe 3](#)), ce membre est commun non seulement à toutes les instances de la classe de base, mais aussi à toutes les classes dérivées de cette classe de base et à leurs instances.

Si l'on souhaite, par exemple, doter notre système de gestion de véhicules d'un dénombrement automatique du nombre d'instances, il nous faut modifier la classe de base de façon à

- déclarer (11) une variable statique privée qui servira de compteur d'occurrences ;
- définir (9) une fonction statique permettant au code utilisateur d'obtenir la valeur courante de cette variable ;
- veiller (4 et 5) à ce que tous les constructeurs incrémentent le compteur ;
- veiller à ce que le destructeur décrémente ce pointeur ;
- définir et initialiser à zéro (dans un fichier CVehicle.cpp ?) la variable CVehicle::nb.

```

1 class CVehicle
2 {
3 public:
4     CVehicle(double valeurInitiale = 0) : argus(valeurInitiale) {++nb;}
5     CVehicle(const CVehicle & modele) : argus(modele.argus) {++nb;}
6     ~CVehicle() {--nb;}
7     double valeurEstimee() {return argus;}
8     void valeurEstimee(double nouvelle) {argus = nouvelle;}
9     static int nbVehicules() {return nb;}
10 private:
11     static int nb;
12     double argus;
13 };

```

Aucune modification des classes dérivées n'est nécessaire.

Lorsque l'une de ces classes est instanciée, les membres hérités sont initialisés par appel d'un constructeur de la classe de base, ce qui assure l'incrémement du compteur. Conformément au principe selon lequel les instances de la classe dérivée sont aussi des instances de la classe de base, la création d'une CVoiture se traduira donc par une augmentation du nombre de CVehicle. De même, la disparition d'une instance d'une classe dérivée s'accompagne automatiquement d'une exécution du destructeur de la classe de base, ce qui garantit l'exactitude du dénombrement.

La fonction nbVehicules() peut être appelée de multiples façons :

```

1 CVehicle::nbVehicules(); //appel au titre de la classe de base
2 CVoiture::nbVehicules(); //appel au titre d'une classe dérivée
3 CVehicle x;
4 x.nbVehicules(); //appel au titre d'une instance de la classe de base
5 CVoiture a;
6 a.nbVehicules(); //appel au titre d'une instance d'une classe dérivée

```

Tous ces appels se résolvant par l'exécution de la même fonction, il semble toutefois préférable de privilégier l'appel au titre de la classe de base, qui exprime clairement aussi bien le fait qu'il s'agit d'une fonction statique que le fait qu'elle est définie dans la classe de base.

Ce que la dérivation rend possible

La prise en compte explicite du fait que les camions comme les voitures (et, éventuellement, d'autres types d'engins) sont des véhicules permet d'écrire du code qui néglige les particularités de ces objets. On peut, par exemple, créer une collection contenant les adresses de tous les CVehicle de la flotte et définir des fonctions traitant cette collection :

```

1 double valeurTotale(const QList < CVehicle * > & flotte)
2 {
3     double total = 0;
4     QList< CVehicle * >::ConstIterator it;
5     for (it = flotte.begin() ; it != flotte.end() ; ++it)
6         total += (*it)->valeurEstimee();
7     return total;
8 }

```

L'originalité profonde d'une telle fonction est qu'elle est parfaitement indifférente à la composition de la flotte. Il est même possible de créer de nouvelles classes dérivées de `CVehicule` sans avoir à reconsidérer le code qui n'accède aux objets que par l'intermédiaire d'un pointeur (ou d'une référence) sur la classe de base.

Bien entendu, cette caractéristique présente un intérêt pratique proportionnel à la complexité des traitements qui peuvent être définis au niveau de la classe de base. Un exemple de quelques lignes, tel que ceux envisageables dans une Leçon, permet d'illustrer le fonctionnement de l'héritage, mais certainement pas d'engranger un bénéfice qui justifierait à lui seul la complexité de ce mécanisme. De ce point de vue, l'exemple proposé dans le TD 12 est sans doute plus convainquant, puisqu'il va s'agir d'hériter d'un ensemble de fonctions que nous ne serions tout bonnement pas en mesure de mettre au point.

Ce que la dérivation ne suffit pas à rendre possible

La manipulation d'instances d'une classe dérivée au moyen d'un pointeur sur la classe de base n'autorise l'accès qu'aux membres hérités. Il n'est en particulier pas possible d'appliquer à une collection hétérogène telle que la `flotte` de notre exemple précédent un traitement qui ne s'effectue pas exactement de la même façon dans toutes les classes dérivées.

Dans l'état actuel de nos connaissances, le traitement d'une collection hétérogène ne peut faire appel qu'à du code défini au niveau de la classe de base. Il faut donc non seulement que l'opération envisagée soit possible pour tous les types dérivés, mais aussi qu'elle puisse être effectuée par une seule et même fonction.

Du point de vue de la gestion de collections hétérogènes, il faut aussi souligner qu'un conteneur ne peut en aucun cas stocker des objets de types différents. Il serait illusoire de créer un conteneur d'instances d'une classe de base et d'essayer ensuite d'y stocker des instances de différentes classes dérivées : la copie d'une valeur de ce genre exige évidemment la prise en compte des variables propres à la classe dérivée concernée, et le "moule" obtenu en annonçant une collection d'instances de la classe de base ne laisse aucune place à ces variables.

Le transfert d'une valeur d'un type dérivé dans une instance de la classe de base laisse de côté les membres propres à la classe dérivée.

C'est pour cette raison que, dans l'exemple précédent, la collection hétérogène est gérée par une liste de pointeurs sur `CVehicule`, et non par une liste de `CVehicule`.

3 - Hiérarchies de classes

La notion de "classe de base" n'implique pas de caractéristique spéciale. D'un point de vue syntaxique, n'importe quelle classe peut donc, a priori, servir de base à la création d'une classe dérivée. Il est en particulier tout à fait possible qu'une classe dérivée serve elle-même de classe de base à une classe dérivée "de seconde génération". On peut ainsi créer des hiérarchies de classes dérivant les unes des autres.

Lorsqu'une classe dérive d'une classe elle-même dérivée, on se trouve dans une situation que l'on peut décrire comme des héritages successifs (ou un héritage indirect).

Aucune règle de fonctionnement particulière ne découle du fait que la classe de base d'une classe est elle-même déjà une classe dérivée. La seule véritable complexité introduite par ce type de situation est celle inhérente à la hiérarchie créée : il est particulièrement important de procéder à une analyse préalable soigneuse et, notamment, de veiller à ce que toutes les relations de dérivation correspondent effectivement à une spécialisation progressive par ajout de caractéristiques supplémentaires.

4 - Dérivation multiple

Une classe peut aussi dériver simultanément de plusieurs classes de base.

On réserve habituellement l'expression "héritage multiple" à cette situation où une même classe a plusieurs "ancêtres" directs, par opposition à celle créée par des héritages successifs (qui, d'un certain point de vue, pourrait également mériter cette appellation).

Si notre entreprise de location diversifie son activité, nous pourrions être amenés à créer la classe `CHabitation`, en vue de décrire les maisons et appartements proposés (surface habitable, équipements disponibles, etc). Il n'est pas impossible d'imaginer que le cas des "mobile homes" pourrait alors être traité par une classe héritant certains de ses membres de `CVehicule` et d'autres de `CHabitation`.

La création d'une classe bénéficiant de plusieurs héritages simultanés n'implique aucun effort syntaxique remarquable, puisqu'il s'agit simplement d'énumérer les classes de base :

```
1 class CMobileHome : public CVehicule, public CHabitation
2 {
3 };
```

En toute honnêteté, si cet exemple suggère bien l'idée générale d'une situation d'héritage multiple, je doute qu'il puisse être réellement utilisé dans un programme. Concevoir les "mobile homes" comme une spécialisation du cas "habitation" risque en effet de nous conduire à retirer tellement de caractéristiques de cette classe qu'elle cessera de présenter le moindre intérêt (un "mobile home" est dépourvu d'adresse postale, par exemple).

La mise au point de hiérarchies de classes utilisant l'héritage multiple peut s'avérer un problème redoutable, aussi bien pour des raisons conceptuelles (cohérence de la modélisation réalisée) que pour des raisons purement techniques (les interactions entre les différents aspects du langage peuvent devenir complexes).

L'utilisation de bibliothèques commerciales reposant sur de telles hiérarchies, en revanche, peut offrir un rapport difficulté/efficacité très attrayant, même pour un programmeur relativement débutant. Une fois les difficultés d'analyse et d'implémentation résolues (par des professionnels de la question), l'héritage multiple permet en effet de "piocher" assez facilement dans un ensemble de caractéristiques disponibles.

La bibliothèque Qt est un exemple de produit permettant cette approche.

5 - Dérivation protégée et dérivation privée

Outre la dérivation publique dont nous avons discuté jusqu'à présent, le langage C++ propose deux autres type d'héritages, qui sont obtenus au prix d'une variation syntaxique très mineure. Les mots `protected:` et `private:` peuvent en effet remplacer `public:` dans la déclaration de "filiation" d'une classe :

```
1 class CDerivee : private CBase //héritage privé
2 {
3
4 };
```

Les droits d'accès aux membres sont alors différents de ceux présentés précédemment :

- dans le cas d'un **héritage protégé**, les membres qui sont déclarés `public:` dans la classe de base deviennent protégés dans la classe dérivée (les membres protégés et privés de la classe de base gardent leur statut dans la classe dérivée) ;
- dans le cas d'un **héritage privé**, tous les membres de la classe de base deviennent des membres privés de la classe dérivée.

Ces règles définissent le cas "normal". Les membres de la classe de base dont le contrôle d'accès est modifié par un héritage protégé ou privé peuvent, par ailleurs, faire l'objet d'une redéfinition dans une classe dérivée, de façon à leur restaurer leur statut initial...

La différence essentielle entre ces types de dérivation et la dérivation publique ne réside cependant pas là, mais dans le fait que

Les instances d'une classe dérivée de façon protégée (ou privée) d'une classe de base NE SONT PAS des instances de la classe de base.

Ces types de dérivation ne correspondent donc pas la notion de spécialisation définie en début de Leçon, et ils ne peuvent donner lieu au traitement indifférencié d'instances de diverses classes dérivées. En dépit de leur similarité de surface avec l'héritage public, les mécanismes

d'héritage protégé et privé sont fonctionnellement plus proche du cas où une classe possède comme membre une instance d'une autre classe.

De fait, dans la plupart des cas, l'alternative à considérer est entre une dérivation (publique) qui rend compte d'une véritable spécialisation (dans les cas où un `ClasseDerivee` est un `ClasseDeBase`) et une relation "possède un membre du type..." (dans les cas où un `ClasseComplexe` comporte un `ClassePlusSimple`).

Ce n'est qu'exceptionnellement qu'il peut s'avérer plus avantageux de recourir à un héritage protégé ou privé.

6 - Bon, c'est gentil tout ça, mais ça fait déjà 8 pages. Qu'est-ce que je dois vraiment en retenir ?

- 1) Lorsque différents types d'objets ont assez de caractéristiques communes pour que des traitements non triviaux puissent être effectués sans savoir exactement à quel type d'objet on a affaire, il peut être judicieux de créer une classe de base et des classes qui en sont (publiquement) dérivées.
- 2) Lorsqu'une classe dérive d'une classe de base, elle hérite des membres qui constituent celle-ci. Les membres explicitement déclarés dans la classe dérivée lui seront propres.
- 3) Constructeurs, destructeurs, opérateurs d'affectation et liens d'amitié ne sont jamais hérités.
- 4) L'initialisation des membres hérités passe par l'exécution d'un constructeur de la classe de base.
- 5) La destruction d'une instance d'une classe dérivée passe par l'exécution d'un destructeur de la classe de base.
- 6) Le destructeur de la classe de base est en général insuffisant pour détruire une instance d'une classe dérivée.
- 7) Une fonction membre propre à une classe dérivée n'a accès aux membres hérités que si ceux-ci ne sont pas déclarés `private` dans la classe de base.
- 8) Une classe peut être dérivée d'une classe qui est elle-même déjà dérivée d'une autre. On parle alors d'une hiérarchie de classes et d'héritage indirect.
- 9) Une classe peut dériver simultanément de plusieurs autres. On parle alors d'héritage multiple.
- 10) La conception de hiérarchies de classes impliquant l'héritage multiple n'est pas une entreprise à conseiller à un débutant.
- 11) Créer une classe par dérivation d'une (ou plusieurs) classe(s) de base prévue(s) à cet effet est, en revanche, à la portée de n'importe qui.