



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

Apprendre C++ avec Qt : Leçon 14

Fonctions virtuelles et classes abstraites

1 - Redéfinition d'un membre hérité	2
Redéfinition et surcharge.....	2
Accès par défaut aux membres propres	2
Utilisation de :: pour accéder aux membres hérités.....	3
Redéfinition d'une fonction surchargée dans la classe de base.....	3
2 - Un polymorphisme indésirable.....	4
Redéfinition de variables membre	4
Redéfinition de fonctions membre.....	4
3 - Fonctions virtuelles.....	5
Pourquoi les fonctions membre ne sont-elles pas toutes virtuelles ?	6
Valeurs par défaut des paramètres d'une fonction virtuelle.....	6
Ce que les fonctions virtuelles rendent possible.....	7
4 - Redéfinition de fonctions dont le type est lié à la classe dont elles sont membre	8
5 - Héritage indirect et fonctions virtuelles	9
Transmission de la virtualité	9
Absence de redéfinition dans certaines classes dérivées.....	10
6 - Classes abstraites	10
Fonctions virtuelles pures	10
Version "par défaut" d'une fonction virtuelle pure.....	11
7 - Bon, c'est gentil tout ça, mais ça fait déjà 11 pages. Qu'est-ce que je dois vraiment en retenir ?	12

Nous avons vu ([Leçon 12](#)) qu'il est possible de traiter indifféremment un ensemble d'objets de types divers, à condition que ces types soient tous dérivés d'une même classe autorisant le traitement en question. L'intérêt de cette technique est accru par la possibilité d'adapter à chacune des classes dérivées la façon dont est effectué le traitement demandé. La demande de traitement pourra donc être appliquée aveuglément à une collection hétérogène, même si le détail des opérations doit être adapté au type de chacun des objets concernés.

1 - Redéfinition d'un membre hérité

Lorsqu'une classe dérive d'une autre, il est possible qu'elle redéfinisse certains des membres qui sont rendus disponibles par l'héritage. Une instance de la classe dérivée comporte alors deux membres homonymes : un membre hérité de la classe de base, et un membre propre.

L'usage impose, assez malencontreusement, le terme "redéfinition" pour désigner ce qui n'est en fait que la définition d'un nouveau membre.

Redéfinition et surcharge

Comme la surcharge ([Leçon 10](#)), la redéfinition donne naissance à des fonctions portant le même nom. Ces mécanismes sont cependant bien distincts puisque, dans le cas de la surcharge, c'est la différence des signatures qui permet de résoudre l'ambiguïté créée par l'homonymie, alors que, dans le cas de la redéfinition, les signatures sont toujours identiques.

Si une classe dérivée définit une fonction homonyme d'une fonction héritée, mais possédant une signature différente, il ne s'agit pas d'une redéfinition mais d'une surcharge. Les mécanismes décrits dans la suite de la présente Leçon ne s'appliquent évidemment pas dans ce cas.

L'ambiguïté qui pourrait être créée par l'homonymie entre un membre hérité et un membre propre est évitée par l'adoption d'une règle :

Accès par défaut aux membres propres

L'existence d'un membre propre a pour effet de masquer la présence du membre hérité portant le même nom (un peu comme la définition d'une variable locale masque une éventuelle variable homonyme précédemment accessible).

Si la classe dérivée redéfinit certains des membres dont elle a hérité, c'est certainement parce que, pour une raison quelconque, ceux-ci ne lui conviennent pas parfaitement. Il est donc logique que les manipulations effectuées sur une instance de cette classe opèrent préférentiellement sur les membres qui lui sont propres.

Pour illustrer cette règle, supposons que nous disposons des définitions suivantes :

```
1 class CBase
2 {
3 public:
4     QString texte;
5     QString f() {return "CBase::f()";}
6     QString g(){return "CBase::g()";}
7 };
8
9 class CDerivee : public CBase
10 {
11 public:
12     QString texte; //REDEFINITION d'une variable membre
13     QString f() {return "CDerivee::f()";} //REDEFINITION d'une fonction membre
14     QString h();
15 };
```

Une instance de la classe dérivée comporte donc deux membres nommés `val` : une **variable héritée** de la classe `CBase`, et une **variable propre**. La classe `CDerivee` possède également deux membres nommés `f()` : une **fonction héritée** de la classe `CBase`, et une **fonction propre**.

Les opérateurs de sélection permettent par défaut l'accès aux membres propres :

```
1 CDerivee unD;
2 CDerivee *ptrD = &unD;
3 ptrD->texte = "coucou"; //la variable propre à CDerivee reçoit "coucou"
4 QString t = unD.f(); //t recoit "CDerivee::f()"
```

De même, dans le corps d'une **fonction propre** à la classe dérivée, le **nom d'un membre redéfini** désigne le membre propre de l'instance au titre de laquelle la fonction est exécutée :

```
QString CDerivee::h()
{
    texte = "CDerivee::h() appelle ";
    return texte + f(); //renvoie "CDerivee::h() appelle CDerivee::f()"
}
```

Utilisation de :: pour accéder aux membres hérités

L'accès aux membres redéfinis reste possible à l'aide de l'opérateur de **résolution de portée** :

```
5 unD.CBase::texte = "xx"; //la variable héritée de CBase reçoit la valeur "xx"
6 t = ptrD->CBase::f(); //t recoit "CBase::f()"
```

La redéfinition n'empêche donc nullement une fonction membre propre d'agir sur les membres hérités, et h() pourrait être définie ainsi :

```
QString CDerivee::h()
{
    CBase::texte = CBase::f(); //la variable héritée reçoit "CBase::f()"
}
```

Le fait qu'il reste possible d'utiliser les membres hérités montre bien qu'ils n'ont aucunement été "redéfinis", mais sont simplement masqués par leurs homonymes.

Redéfinition d'une fonction surchargée dans la classe de base

Si la classe de base comporte plusieurs fonctions homonymes, la redéfinition de l'une d'entre-elles les masque toutes. Ainsi, après

```
class CB
{
public:
    void k() {} //une première fonction k()
    void k(int) {} //une seconde fonction k()
};

class CD : public CB
{
public:
    void k() {} //redéfinition d'une des fonctions k()
};
```

il n'est **pas possible** d'accéder implicitement à la fonction héritée non redéfinie :

```
CD uneInstance;
uneInstance.k(4); //ERREUR : la fonction héritée est masquée
uneInstance.CB::k(4); //OK : appel explicite de la fonction héritée
```

Une **syntaxe particulière** permet toutefois de redonner à la classe dérivée son accès implicite aux fonctions non redéfinies et, avec

```
class CD : public CB
{
public:
    using CB::k;
    void k(); //redéfinition d'une des fonctions k()
};
```

il serait **possible** d'écrire

```
CD uneInstance;
uneInstance.k(4); //OK : les versions non redéfinies de k() sont visibles
```

2 - Un polymorphisme indésirable

Pour examiner les conséquences de la présence de **membres redéfinis**, reprenons l'exemple de la gestion d'une flotte de véhicules de location :

```

1  class CVehicle
2  {
3  public:
4      CVehicle() : prix(0), distance(0) {}
5      void depreciee() {prix *= 0.9;}    //perte de 10% de la valeur
6      void accident() {depreciee();}    //puisque le véhicule est abîmé !
7      double prix;
8      int distance;
9  };
10 class CVoiture : public CVehicle      //les CVoiture SONT des CVehicle
11 {
12 public:
13     CVoiture(int d = 0) : distance(d) {}
14     void deprecie() {prix *= 0.85;}    //perte de 15% de la valeur
15     int distance;
16 };

```

Redéfinition de variables membre

Nous savons ([Leçon 13](#)) qu'un pointeur sur une classe de base (CVehicle, par exemple) peut légitimement contenir l'adresse d'une instance d'une classe dérivée (CVoiture, par exemple), puisqu'un tel objet est un cas particulier d'instance de la classe de base. Toutefois, un pointeur sur la **classe de base** ne permet d'accéder qu'aux membres hérités, et son usage conduit donc à violer la règle d'accès par défaut aux membres propres. Lorsqu'une variable membre est redéfinie, l'utilisation d'un pointeur sur la classe de base contenant l'adresse d'une instance de la classe dérivée donne donc lieu à un effet paradoxal :

```

1  CVoiture titine (45312);           //elle a déjà roulé...
2  CVoiture * ptrD = &titine;
3  CVehicle * ptrB = ptrD; //nos deux pointeurs contiennent la même adresse,
4  int dUn = ptrB->distance;         //mais dUn vaut 0 (la distance héritée) alors
5  int dDeux = ptrD->distance;        //que dDeux vaut 45312 (la distance propre)

```

Le constructeur de CVoiture n'initialise pas sa partie héritée, qui est donc construite avec le constructeur par défaut de CVehicle, ce qui donne à la distance héritée une valeur nulle.

Le même objet (titine) présente donc des caractéristiques différentes selon le type du pointeur qui sert à l'observer (elle a l'air bien plus fatiguée si elle est vue comme une CVoiture que si elle est vue comme un simple CVehicle).

D'après le "[Trésor de la Langue Française](#)", ce qui est polymorphe "offre des apparences, des formes diverses". Une instance d'une classe dérivée mérite cette épithète dans la mesure où elle ne présente pas les mêmes caractéristiques selon le type de pointeur utilisé pour l'observer (elle a donc des apparences diverses). Une classe de base, pour sa part, peut avoir des instances de types différents, donc des formes diverses. Elle peut donc, elle aussi, être dite polymorphe.

Inutile de dire que, dans ces conditions, le développement d'un programme cohérent devient une entreprise très hasardeuse. Nous pouvons en conclure que

Il ne faut jamais redéfinir les variables membre.

Redéfinition de fonctions membre

Lorsqu'une fonction redéfinie est invoquée au titre d'une instance désignée par un pointeur, un polymorphisme analogue à celui décrit à propos des variables membre peut être observé. Selon le type du pointeur désignant l'instance, ce n'est en effet pas le même code qui est exécuté :

```

6  ptrB->deprecie(); //appel de la fonction héritée : le prix perd 10%
7  ptrD->deprecie(); //appel de la fonction propre : le prix perd 15%

```

Essayer d'écrire un programme dans ces conditions serait un peu comme travailler avec un collaborateur dont les compétences varieraient selon s'il est interpellé par son nom ou par son prénom. Ce n'est peut-être pas totalement impossible, mais le risque de malentendu est élevé.

Même si aucune instance de la classe dérivée n'est explicitement désignée par un pointeur, il suffit qu'une fonction héritée appelle une fonction redéfinie pour que le problème surgisse. Dans notre exemple, la fonction `accident()` n'est pas redéfinie dans la classe dérivée. Cette fonction exécutera donc les mêmes opérations, que l'instance au titre de laquelle elle est exécutée soit de type `CVehicule` ou de type `CVoiture` :

```
8 titine.accident(); //son prix ne perd pas 15%, mais seulement 10% !
```

En clair, les comportements spécifiques d'une classe dérivée sont inaccessibles à ses fonctions non spécifiques...

Le langage C++ prévoit toutefois un mécanisme qui permet d'éliminer les paradoxes liés à la redéfinition des fonctions membre.

3 - Fonctions virtuelles

Lorsqu'une fonction **déclarée virtuelle** dans la classe de base est invoquée au titre d'une instance désignée par un pointeur (ou d'une référence) sur la classe de base, c'est le type de l'objet désigné par le pointeur (ou la référence) et non le type de ce pointeur (ou de cette référence) qui détermine quelle fonction est exécutée. S'il s'agit d'une instance d'une classe dérivée, c'est la fonction propre à cette classe qui sera exécutée. La "virtualité" permet donc de retrouver la cohérence nécessaire à la manipulation des objets concernés. Par conséquent,

Il ne faut redéfinir que des fonctions déclarées virtuelles dans la classe de base.

Ainsi, après

```
1 class CVehicule
2 {
3 public:
4     CVehicule() : prix(0), distance(0) {}
5     virtual void depreciee() {prix *= 0.9;}           //perte de 10% de la valeur
6     void accident() {depreciee();}                   //puisque le véhicule est abîmé !
7     double prix;
8     int distance;
9 };
10 class CVoiture : public CVehicule                    //les CVoiture SONT des CVehicule
11 {
12 public:
13     CVoiture(int d = 0) : distance(d) {}
14     void deprecie() {prix *= 0.85;}                   //perte de 15% de la valeur
15 };

```

on aura bien

```
1 CVoiture choupette;
2 CVehicule *unVehicule = &choupette;
3 unVehicule->deprecie(); //choupette perd 15% de sa valeur

```

De plus, les fonctions de la classe de base appellent la bonne version d'une fonction virtuelle lorsqu'elles sont elles-même exécutées au titre d'une instance de la classe dérivée :

```
4 choupette->accident(); //elle perd 15% de sa valeur

```

Lorsqu'une fonction membre en appelle une autre, elle utilise implicitement le pointeur `this`. Dans le cas de la fonction `accident()`, ce pointeur est de type `CVehicule *`, mais comme la fonction est ici exécutée au titre d'une **instance de CVoiture**, il contient l'adresse de **cet objet**. L'invocation d'une fonction virtuelle (`deprecie()`, en l'occurrence) se traduira donc bien par l'exécution du code propre à `CVoiture` et, donc, par une perte de 15% de la valeur.

Pourquoi les fonctions membre ne sont-elles pas toutes virtuelles ?

Les fonctions virtuelles exigent du compilateur un véritable tour de force. Le premier des exemples proposés ci-dessus est assez simple, parce que l'examen du code révèle à l'évidence que l'objet désigné par `unVehicule` est une instance de la classe `CVoiture`. Le second exemple est bien plus épineux, car `accident()` doit appeler une fonction `deprecie()` différente selon si elle est elle-même exécutée au titre d'une instance de `CVehicule` ou d'une instance de `CVoiture`. Comme il est possible qu'un programme invoque la fonction `accident()` tantôt au titre d'une `CVoiture`, tantôt au titre d'un `CVehicule`, il est clair que le choix ne peut être effectué lors de la compilation. Il faut donc que le compilateur génère du code permettant de différer cette décision, qui devra être prise "à la volée", lors de chaque exécution de la fonction.

Si l'on prend en compte le fait que les classes dérivées concernées peuvent fort bien ne pas encore avoir été définies au moment de la compilation du code appelant la fonction virtuelle, on se rend compte que l'exécution d'une fonction virtuelle n'est pas un exercice anodin. Le mécanisme qui rend cette exécution possible est connu sous le nom de "*dynamic binding*", et il a un coût significatif en termes de vitesse d'exécution. C'est ce coût qui explique que toutes les fonctions membre n'adoptent pas automatiquement ce mode de fonctionnement : les concepteurs du langage n'ont pas voulu faire payer à chaque appel de fonction un luxe qui n'est appréciable que dans certains cas.

Il existe cependant une fonction dont la "non-virtualité" pose souvent des problèmes : il s'agit du destructeur fourni invisiblement par le langage. Comme nous l'avons vu dans la [Leçon 12](#), la destruction d'une instance est souvent ordonnée en désignant celle-ci à l'aide d'un pointeur. Si ce pointeur est un pointeur sur une classe de base contenant l'adresse d'une instance d'une classe dérivée et que le destructeur de la classe de base n'est pas virtuel, c'est lui qui sera exécuté et seule la partie de l'instance définie par héritage sera détruite correctement.

Le destructeur d'une classe de base doit toujours être virtuel.

C'est pour cette raison que certains environnements de développement (Visual C++, par exemple) ajoutent systématiquement la définition d'un destructeur lorsqu'il génèrent le squelette d'une nouvelle classe : le corps de cette fonction reste le plus souvent vide, mais il est nécessaire de la définir explicitement pour pouvoir la rendre virtuelle...

Remarquez que, bien que les destructeurs des classes dérivées ne soient pas réellement des redéfinitions de celui de la classe de base (les destructeurs ne sont pas héritables et ces fonctions ont chacune un nom différent...), la virtualité du destructeur de la classe de base a les mêmes conséquences que si c'était le cas : les destructeurs des classes dérivées deviennent automatiquement virtuels et le polymorphisme de la destruction est géré correctement.

Valeurs par défaut des paramètres d'une fonction virtuelle

Lorsqu'une fonction virtuelle dispose de valeurs par défaut pour ses paramètres, le *dynamic binding* ne garantit pas l'utilisation des valeurs par défaut adéquates si celles-ci sont modifiées lors de la redéfinition de la fonction.

Pourquoi ? Par souci d'efficacité. Rendre sûre la redéfinition des valeurs par défaut entraînerait une complexité supplémentaire qui imposerait un ralentissement de tous les appels de fonctions virtuelles. On a, là encore, préféré sacrifier sur l'autel de l'intérêt général les quelques cas où la gestion correcte de la redéfinition des valeurs par défaut aurait présenté un réel avantage.

Ainsi, après

```
1 class CB
2 {
3 public:
4     virtual int uneFonction(int p = 2) {return p;} //fonction membre virtuelle
5 };
6 class CD : public CB
7 {
8 public:
9     int uneFonction(int p = 3) {return p;} //REDEFINITION
10 };
```

on aura une nouvelle manifestation du "polymorphisme indésirable" :

```

1  CD unCD;
2  CB * ptrCB = & unCD;           //les deux pointeurs contiennent
3  CD * ptrCD = & unCD;           //la même adresse, et pourtant
4  int c = ptrCB->uneFonction();   //c vaut 2
5  int d = ptrCD->uneFonction();   //alors que d vaut 3 !

```

C'est bien la fonction propre à la classe CD qui est invoquée à la ligne 4 (puisque `uneFonction()` est virtuelle). Toutefois, lorsqu'elle est appelée via un pointeur sur CB, cette fonction n'utilise pas sa propre valeur par défaut, mais celle définie pour la fonction héritée.

Quand on redéfinit une fonction virtuelle, il ne faut jamais doter ses paramètres de valeurs par défaut différentes de celles adoptées dans la classe de base.

Ce que les fonctions virtuelles rendent possible

Si la redéfinition n'est pratiquée que sur des fonctions virtuelles, le langage gère correctement le polymorphisme, c'est à dire qu'il le masque totalement aux programmeurs.

Les instances de la classe de base qui sont des instances de différentes classes dérivées ne sont pas identiques, puisqu'elles possèdent des membres propres différents. Un programmeur qui manipule une collection hétérogène d'instances de la classe de base n'a pas à se soucier de ce polymorphisme, car le *dynamic binding* lui garantit l'exécution du code le plus adapté. Lorsqu'on dit que C++ "permet le polymorphisme", on fait allusion à cette gestion correcte du polymorphisme des classes de base, et non au fait que la redéfinition irraisonnée de membres hérités peut créer des situations où le même objet donne, selon comment on le regarde, l'impression d'être dans des états différents...

Imaginons, par exemple, deux classes ainsi définies :

```

1  class CBase
2  {
3  public:
4      int a;
5      virtual void initialisation() {a = 4;}
6  };
7
8  class CDerivee : public CBase
9  {
10 public:
11     int b;
12     void initialisation() {b = 5; CBase::initialisation();}

```

La classe CBase ne possède qu'une variable membre, dont on suppose que la valeur 4 correspond à une représentation d'un quelconque "état initial" d'un objet. On suppose également que le programme exige que cet état initial puisse être instauré par une fonction autre qu'un constructeur, en l'occurrence la fonction `initialisation()`.

CDerivee est une spécialisation de CBase qui possède un membre supplémentaire qui doit, dans l'état "initial", adopter 5 pour valeur. La fonction `initialisation()` propre à CDerivee procède donc à l'affectation de cette valeur au membre propre, puis appelle la fonction `initialisation()` héritée de CBase, de façon à ce que le membre hérité reçoive, lui aussi, la valeur désirée.

Un appel à la fonction héritée est préférable à un accès direct à la variable héritée, car, en cas d'évolution de la classe de base, il sera plus simple de n'avoir à modifier que la fonction d'initialisation qui y est définie que d'avoir à mettre à jour les fonctions d'initialisation de toutes les classes dérivées.

La virtualité de la fonction `initialisation()` permet de remplacer tous les objets d'une collection hétérogène dans leur état "initial" sans s'inquiéter des différences de type qui exigent que des traitements différents soient provoqués par des appels apparemment identiques :

```

1  CBase objetUn;
2  CDerivee objetDeux;
3  QList <CBase *> listeDePtr ;
4  listeDePtr.append(&objetUn);
5  listeDePtr.append(&objetDeux);   //une collection hétérogène

```

```

6  QValueList <CBase *>::Iterator it;
7  for(it = listeDePtr.begin() ; it != listeDePtr.end() ; ++it)
8      (*it)->initialisation();           //une initialisation "polymorphe"

```

4 - Redéfinition de fonctions dont le type est lié à la classe dont elles sont membre

Lorsqu'une fonction virtuelle est redéfinie, elle doit conserver la même signature (faute de quoi il s'agit d'une surcharge) et le même type (sous peine de déclencher une erreur de compilation du genre "*overriding function differs only by return type*").

Pourquoi une telle exigence ? La redéfinition des fonctions virtuelles est destinée à permettre leur invocation indifférenciée, comme dans l'exemple de la réinitialisation des éléments d'une collection hétérogène proposé ci-dessus. Dans ce contexte, on voit mal comment le code appelant pourrait exploiter les valeurs éventuellement renvoyées par les fonctions redéfinies si ces valeurs étaient de types complètement différents.

Il existe pourtant des types que le code appelant accepte de ne pas différencier : les pointeurs (ou références) sur la classe de base et les pointeurs (ou références) sur des classes dérivées. Le langage C++ admet donc ce que les anglophones appellent des "*covariant return types*" :

Si une fonction membre d'une classe de base renvoie un **pointeur** (resp. une référence) **sur une instance de cette classe**, elle peut être redéfinie dans une classe dérivée par une fonction renvoyant un **pointeur** (resp. une référence) **sur une instance de cette classe dérivée**.

Ceci n'est qu'une relativisation de la règle : les types renvoyés n'ont pas à être *littéralement* les mêmes, il suffit qu'ils soient *formés de la même façon* à partir du nom de la classe concernée.

```

1  class CBase
2  {
3  public:
4      virtual double interdit() {return 2.5;}
5      virtual CBase * autorise() {return this;}
6  };
7
8  class CDerivee : public CBase
9  {
10 public:
11     //int interdit() {return 3;} //provoquerait une erreur de compilation !
12     CDerivee * autorise() {return this;} //ok pour un compilateur ISO

```

Microsoft Visual C++ 6.0 ne connaît pas les *covariant return types*. Un compilateur plus récent (Visual C++ 2005, par exemple) est donc nécessaire pour exploiter cet aspect du langage.

La relaxation de l'exigence d'identité des types permet notamment de rendre virtuelles les fonctions prenant en charge les opérateurs ++ et -- préfixés, qui renvoient (cf. [Leçon 11](#)) une **référence à l'objet sur lequel ils ont opéré** :

```

1  class CBase
2  {
3  public:
4      CBase(): m_a(0) {};
5      virtual CBase & operator ++() {++m_a; return *this;}
6      const CBase operator ++(int) {CBase temp(*this); ++*this; return temp;}
7      int m_a;
8  };
9
10 class CDerivee : public CBase
11 {
12 public:
13     CDerivee() : m_b(0) {}
14     CDerivee & operator ++() {CBase::operator ++(); ++m_b; return *this;}
15     int m_b;

```

Remarquez que les fonctions prenant en charge l'usage postfixé de ces opérateurs ne sauraient, pour leur part, être rendues virtuelles. Ces fonctions doivent en effet renvoyer la **valeur** qu'avait l'objet avant l'opération, et non un pointeur ou une référence.

Il suffit d'imaginer un contexte d'utilisation de ces fonctions pour se rendre compte que leur virtualité n'aurait, de toute façon, aucun intérêt. En effet, disposer d'un pointeur (ou d'une référence) désignant un objet d'un type imparfaitement connu peut avoir un intérêt, car cela permet d'invoquer toutes les fonctions de l'interface de la classe de base. Disposer d'une valeur d'un type imparfaitement connu ne sert par contre à rien, puisqu'on ne peut strictement rien faire de cette valeur, pas même l'affecter à un objet (de quel type d'objet pourrait-il s'agir ?).

Dans bien des cas, cependant, les opérateurs ++ et -- sont utilisés uniquement pour leur effet, sans que la valeur de l'expression intervienne en quoi que ce soit dans le déroulement du programme. Les versions pré et postfixée sont alors perçues comme essentiellement équivalentes et, si la première est virtuelle, cette équivalence doit être maintenue :

```
1 CDerivee maVariable;      //maVariable.a et maVariable.b valent 0
2 CBase & alias = maVariable;
3 ++alias;                  //maVariable.a et maVariable.b passent à 1
4 alias++;                  //maVariable.a et maVariable.b passent à 2
```

La fonction CBase::operator++(int) n'étant pas virtuelle, c'est elle qui est appelée lorsque la ligne 4 est exécutée, en dépit du fait que alias désigne un objet de type CDerivee. Toutefois, comme cette fonction n'incrémente pas elle-même m_a mais appelle CBase::operator++(), celle-ci est exécutée au titre de l'instance désignée par this. Bien que ce pointeur soit de type CBase *, il contient l'adresse d'une instance de CDerivee (maVariable, en l'occurrence). Comme CBase::operator++() est virtuelle, c'est en fait CDerivee::operator++() qui est exécutée, ce qui se traduit par l'incrémement de m_a (via un **appel** à CBase::operator++() au titre d'un objet de type CBase) et par celle de m_b. La valeur de l'expression alias++ reste cependant de type CBase, et n'est donc pas celle de maVariable avant incrémementation...

Il est souvent préférable d'implémenter les fonctions prenant en charge l'usage postfixé de -- et de ++ en appelant les fonctions prenant en charge l'usage préfixé de ces opérateurs.

5 - Héritage indirect et fonctions virtuelles

Les mécanismes de redéfinition et de *dynamic binding* ne se limitent évidemment pas à l'héritage direct. En cas d'héritages en cascade, les questions de transmission de la virtualité et d'absence de redéfinition de certaines fonctions doivent être envisagées.

Transmission de la virtualité

Lorsqu'une classe dérivée redéfinit une fonction déclarée virtuelle dans la classe de base, cette fonction reste virtuelle, même si cela n'est pas répété lors de la redéfinition. Ainsi, après

```
1 class CMere
2 {
3 public:
4     int a;
5     virtual void initialisation() {a = 4;}
6 };
7 class CFille : public CMere
8 {
9 public:
10     int b;
11     void initialisation() { CMere::initialisation();b = 5;} //virtuelle !
12 };
```

les deux variables membre de uneFille seront correctement traitées par

```
CFille uneFille;
CMere * ptr = &alice;
ptr->initialisation(); //appel de la fonction propre à CPetiteFille
```

Toutefois, cette propagation automatique de la virtualité ne favorise pas la lisibilité du code source, puisque le seul examen de la classe CFille ne permet pas d'appréhender une des

caractéristiques essentielles de cette hiérarchie (la virtualité de la fonction). Il est donc préférable de **répéter** le mot `virtual` à tous les niveaux de la hiérarchie, même si cela ne change rien au fonctionnement du programme.

Absence de redéfinition dans certaines classes dérivées

Lorsque, dans une hiérarchie de classes, certaines classes dérivées ne redéfinissent pas une fonction virtuelle, l'appel de la fonction en question au titre de l'une de leurs instances provoque l'exécution de la fonction héritée. Rien ne s'oppose à ce qu'une classe dérivée d'une telle classe reprenne l'initiative et redéfinisse sa propre version de la fonction :

```
1 class CMere
2 {
3 public:
4     int a;
5     virtual void initialisation() {a = 4;}
6 };
7
8 class CFille : public CMere
9 { //cette classe ne redéfinit pas la fonction initialisation()
10 };
11
12 class CPetiteFille : public CFille
13 {
14 public:
15     int b;
16     virtual void initialisation() { CFille::initialisation();b = 5;}
17 };
```

Dans cet exemple, la classe `CFille` ne dispose que d'une seule fonction `initialisation()`, celle qu'elle hérite de la classe `CMere` et c'est donc cette fonction qui est exécutée à la suite de l'appel de la ligne 14. La "chaîne de la virtualité" n'est aucunement brisée par ce "chaînon manquant" :

```
1 CPetiteFille alice;
2 CMere * ptr = &alice;
3 ptr->initialisation(); //appel de la fonction propre à CPetiteFille
```

6 - Classes abstraites

L'utilisation du mécanisme d'héritage conduit souvent à définir des classes qui ne servent qu'à fournir un moyen de manipuler de façon indifférenciée des instances de diverses classes dérivées. De telles classes ne sont pas destinées à être elles-même directementinstanciées, mais seulement à servir de classes de base à d'autres classes (qui, elles, serontinstanciées).

Lorsqu'une classe de base est dans ce cas, il peut arriver que certaines fonctions doivent y être définies (parce qu'elles assurent des traitements qui doivent pouvoir être appliqués à des collections hétérogènes d'instances de classes dérivées) sans qu'il soit possible de leur donner un corps (parce que les traitements impliquent d'accéder à des membres propres aux classes dérivées). Le langage C++ satisfait ces exigences contradictoires en autorisant la création de...

Fonctions virtuelles pures

La syntaxe utilisée pour créer une fonction virtuelle pure est très particulière : elle exige que la parenthèse clôturant la liste des paramètres soit suivie du signe `=`, du nombre 0 et d'un point virgule en lieu et place du bloc de code définissant le corps d'une fonction ordinaire.

La présence d'une fonction virtuelle pure dans une classe a deux conséquences :

- la classe en question ne peut plus êtreinstanciée directement ;
- toute classe dérivant de cette classe doit normalement redéfinir cette fonction.

Lorsqu'une classe comporte une fonction virtuelle pure, elle peut être qualifiée d'abstraite, puisqu'elle ne peut donner naissance à aucune instance. Les classes abstraites sont, par nature, destinées à servir de classes de base. Certains auteurs les désignent donc parfois en utilisant l'acronyme ABC (pour **Abstract Base Class**).

L'exemple traditionnel de mise en œuvre d'une classe abstraite concerne des formes géométriques. C'est un bon exemple, parce qu'il est facile d'imaginer une situation où un programme aurait besoin de parcourir un ensemble de pièces de formes diverses pour, par exemple, faire la somme de leurs surfaces. Le parcours de cet ensemble peut être effectué facilement à l'aide d'un pointeur sur une classe `CForme`, à la seule condition que toutes les classes décrivant des objets susceptibles d'apparaître dans la collection soient dérivées de `CForme`. Pour que le calcul de surface puisse être fait par l'intermédiaire d'un pointeur sur `CForme`, il faut que cette classe dispose d'une fonction `surface()`. Toutefois, le calcul de la surface d'une `CForme` n'a guère de sens : on sait, par exemple, calculer la surface d'un `CRectangle` ou d'un `CDisque` en fonction de leurs dimensions, mais la notion de forme est trop abstraite pour se prêter à une véritable définition du corps de la fonction `CForme::surface()`. Nous pouvons donc définir ainsi notre classe de base :

```
1 class CForme //une ABC
2 {
3 public:
4     virtual double surface() = 0; //une fonction virtuelle pure
5 };
```

Bien que la classe `CForme` comporte comme membre unique une fonction dépourvue de corps, son existence est loin d'être inutile, car elle permet la gestion correcte du polymorphisme :

```
1 class CDisque : public CForme
2 {
3 public:
4     CDisque(double r) : rayon(r) {}
5     virtual double surface() { return 3.14 * rayon * rayon; }
6 protected:
7     double rayon;
8 };
9
10 class CRectangle : public CForme
11 {
12 public:
13     CRectangle(double l, double L) : largeur(l), longueur(L) {}
14     virtual double surface() { return largeur * longueur; }
15 protected:
16     double largeur;
17     double longueur;
18 };
```

Le seul rapport concret entre les classes `CDisque` et `CRectangle` est le fait que toutes deux disposent d'une fonction `surface()`. Ces fonctions `surface()` ont la même signature, mais c'est leur seul point commun, puisque les calculs qu'elles effectuent ne font intervenir que des données propres à leurs classes respectives. Un rapport syntaxique subtil est toutefois établi entre ces fonctions : elles sont toutes deux des redéfinitions d'une fonction virtuelle pure déclarée dans la classe dont `CDisque` et `CRectangle` dérivent. La raison d'être de la classe `CBase` est d'établir ce rapport syntaxique qui va, par la suite, nous permettre d'appeler ces deux fonctions très différentes exactement comme s'il s'agissait d'une seule et même fonction.

Faire de `CBase` une classe abstraite présente ici deux avantages : cela résout de façon simple la question du corps de la fonction `CForme::surface()`, et cela indique clairement au lecteur humain que la classe `CBase` n'existe que pour permettre l'utilisation du polymorphisme.

Version "par défaut" d'une fonction virtuelle pure

La présence d'une fonction virtuelle pure garantit que la classe ne pourra pas être instanciée directement. Lorsqu'un ensemble de classes est conçu pour être mis à la disposition d'autres programmeurs (sous la forme d'une librairie, par exemple), cette garantie constitue une protection radicale contre une utilisation inadaptée de classes qui, comme la classe `CForme` évoquée ci-dessus, ne sont destinées qu'à permettre le polymorphisme.

Les utilisateurs d'une classe abstraite doivent cependant payer cette sécurité en acceptant de redéfinir les fonctions virtuelles pures dans toutes les classes qu'ils dérivent de telles classes de bases. Il arrive que certaines classes dérivées puissent très bien s'accommoder d'une

"version générique" de la fonction, et avoir à recopier un bloc de code identique dans toutes les classes est non seulement fastidieux mais aussi source de difficultés de maintenance. Il est alors possible de donner à la fonction virtuelle pure un corps dans la classe de base.

Si, par exemple, nous envisageons de créer des classes dérivées de `CForme` pour lesquelles le calcul de la surface n'est pas pertinent, nous pouvons définir ainsi notre classe de base :

```
1 class CForme //une ABC
2 {
3 public:
4     virtual double surface() = 0 {return -1;} //virtuelle pure avec un corps
5 };
```

Dans ces conditions, la classe `CForme` reste abstraite (elle ne peut être instanciée directement), mais les classes qui en sont dérivées n'ont pas à redéfinir la fonction `surface()` si celle-ci ne les concerne pas : elles héritent de la "version par défaut" définie dans la classe de base.

Le fait que la syntaxe du langage permette de procéder ainsi ne doit pas être interprété comme une incitation à le faire. J'ai personnellement tendance à penser que, si une classe dérivée n'est "pas concernée" par une fonction virtuelle de la classe de base, il y a probablement une erreur d'analyse quelque part. Remarquez, en outre, qu'une classe héritant de la version "par défaut" d'une fonction virtuelle pure reste elle-même abstraite...

7 - Bon, c'est gentil tout ça, mais ça fait déjà 11 pages. Qu'est-ce que je dois vraiment en retenir ?

- 1) Une classe dérivée peut comporter un membre propre homonyme à l'un des membres dont elle hérite. On dit alors (un peu à la légère) qu'elle redéfinit ce membre hérité.
- 2) En cas de redéfinition, on accède par défaut au membre propre.
- 3) Il ne faut JAMAIS redéfinir une fonction non virtuelle ou une variable.
- 4) Le destructeur d'une classe qui risque de servir de classe de base doit être virtuel.
Indice : si une classe définit une fonction virtuelle, elle va sans doute servir de classe de base.
- 5) La redéfinition d'une fonction surchargée masque toutes les fonctions héritées homonymes.
- 6) Lorsqu'une fonction est virtuelle, son invocation via un pointeur (ou une référence) sur la classe de base se traduit par l'exécution du code disponible le plus adapté au type de l'objet désigné par le pointeur (ou la référence).
Cas particulier important : lorsqu'une fonction membre en appelle une autre, elle le fait (implicitement) via un pointeur (`this`). Qu'elle soit ou non virtuelle, une fonction membre appelle donc toujours la bonne version des fonctions virtuelles de sa classe.
- 7) La redéfinition d'une fonction héritée ne doit pas spécifier pour ses paramètres des valeurs par défaut différentes de celles utilisées par la classe de base.
- 8) Une fonction virtuelle pure est habituellement dépourvue de corps dans la classe de base.
- 9) Les fonctions virtuelles pures ne servent que d'interface pour exécuter les redéfinitions qu'en font les classes dérivées.
- 10) Si une classe contient une fonction virtuelle pure, elle ne peut plus être instanciée. C'est ce qu'on appelle une classe abstraite.
- 11) Si une classe dérivée d'une classe abstraite n'en redéfinit pas toutes les fonctions virtuelles pures, elle reste elle-même abstraite.