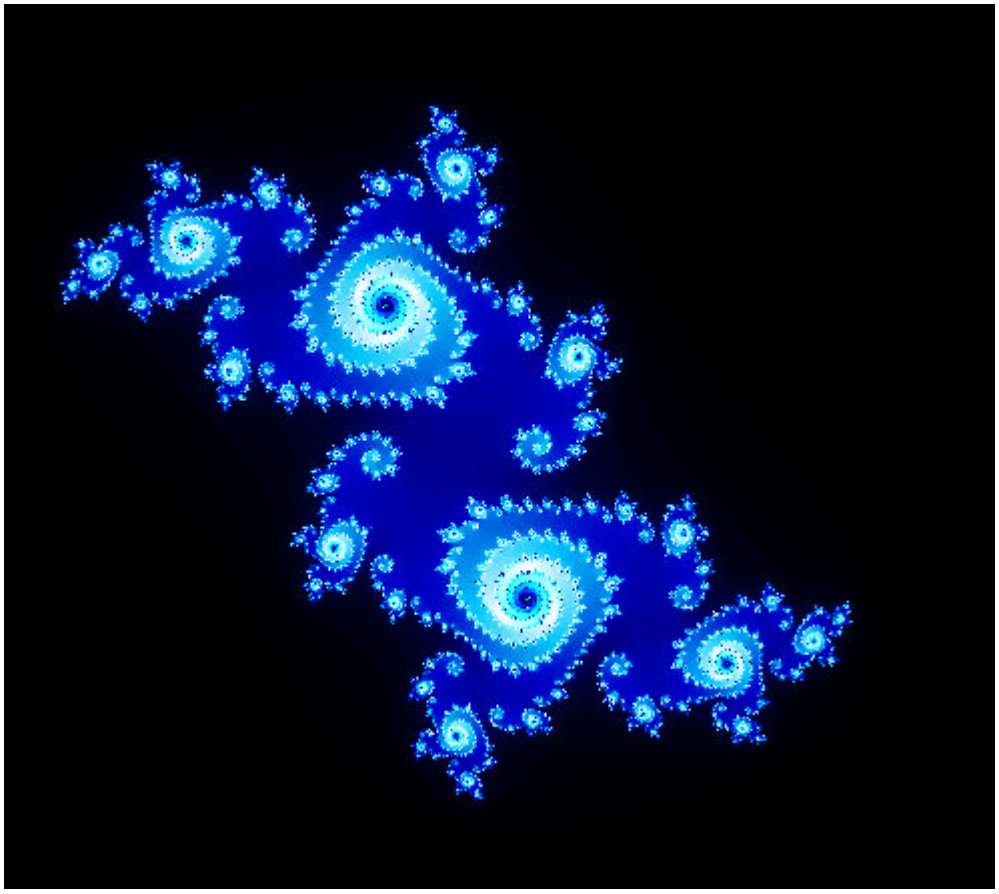
CHAPITRE 15

Héritage multiple



15.1 *Utilisation*

L'héritage multiple est un sujet de controverses multiples. Beaucoup affirment qu'il s'agit là d'une possibilité à ne pas utiliser. La librairie standard d'entrées-sorties de C++, iostream.h, utilise l'héritage multiple pour dériver iostream de istream et de ostream, pour exprimer par là la fait que `iostream` est à la fois un flot d'entrée (`istream`) et un flot de sortie (`ostream`).

L'exemple suivant illustre de manière simple un cas d'héritage multiple. Cet exemple est artificiel et n'illustre que la manière d'utiliser l'outil.

```
#include <iostream.h>

class     A
     {
          int  anInt;
     public :
          A(int a) anInt(a) {}
          void aFunc() { cout<<"aFunc Called"<<endl; }
     };

class     B
     {
          int  anInt;
     public :
          B(int b) anInt(b) {}
          void bFunc() { cout<<"bFunc Called"<<endl: }
     };
class     D : public A, public B
     {
          int  anInt;
     public :
          D(int d, int b, int a) : A(a), B(b), anInt(d) {}
          void dFunc() { cout<<"cFunc called"<<endl;
     };

void      main()

     {
     D    d(10, 20, 30);
     d.bFunc();
     d.aFunc();
     d.dFunc();
     }
```

Dans l'exemple çi-dessus, la classe D hérite publiquement de A et de B. On exprime par là, comme dans le cas de l'héritage simple, que D **est** un (***isa***) **A et** un **B.** Il est possible, dans le cas de l'héritage multiple, de spécifier, pour chaque branche d'héritage et de manière indépendante, le type d'héritage souhaité. Par défaut, le type d'héritage est **privé**. Ainsi, la déclaration suivante exprimerait le fait que D dérive publiquement de A, mais de manière privée de B :

```
class     D: public A, private B {...}
```

Cette formulation est équivalente à :

```
class     D: public A, B { ... }
```

15.2 *Ambiguïtés d'identificateurs*

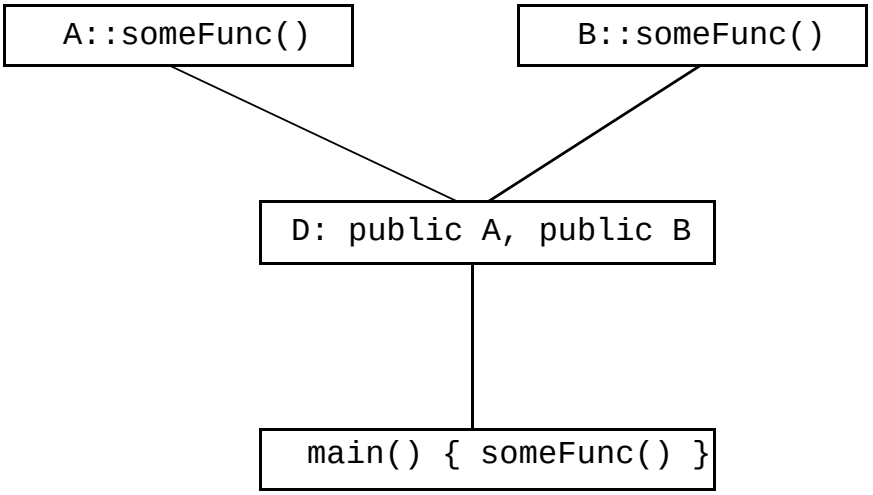
L'exemple ci-dessus illustre un cas idéal. main() peut utiliser les diverses méthodes sans introduire le moindre risque d'ambiguités. Ceci est possible parceque les divers identificateurs sont parfaitement univoques. Il n'en va pas de même dans l'exemple çi-dessous:

```
class     A
     {
          int  anInt;
     public :
          A(int a) anInt(a) {}
          void someFunc() { cout<<"someFunc(A) Called"<<endl; }
     };

class     B
     {
          int  anInt;
     public :
          B(int b) anInt(b) {}
          void someFunc() { cout<<"someFunc(B) Called"<<endl: }
     };
class     D : public A, public B
     {
          int  anInt;
     public :
          D(int d, int b, int a) : A(a), B(b), anInt(d) {}
          void dFunc() { cout<<"dFunc called"<<endl;
     };

void      main()

     {
     D    d(10, 20, 30);
     d.bFunc();
     d.A::someFunc();
     d.B::someFunc();
          // someFunc ne permet pas de dire laquelle des
          // deux fonctions doit être appelée. Il faut donc
          // spécifier le chemin d'accès complètement.
     }
```



Lorsqu'il y a conflit d'identificateurs, que ce soit pour des membres données ou code, il est nécessaire de spécifier intégralement le chemin d'accès pour résoudre le conflit. Les diverses classes de base n'ayant pas forcément la même provenance, il est parfois indispensable de recourir à ce mécanisme; néanmoins, dans la mesure du possible, on évitera, lors de la conception de classes, d'utiliser les mêmes identificateurs dans des classes de base susceptibles d'être utilisées ensemble. Le mécanisme des fonctions virtuelles reste présent, de la même manière que dans le cas de l'héritage simple.

15.3 *Base virtuelle et non virtuelle*

Le nombre de cas où l'héritage multiple est nécessaire est restreint, et dans ces cas, l'utilisation de l'héritage multiple paraît aussitôt évident. L'utilisation de l'héritage multiple est certainement à déconseiller dans un premier temps, parceque l'implémentation en C++ peut receler des complexités cachées. Considérons le cas suivant: A est une classe de base commune aux classes dérivées B et C. D est une classe qui dérive à la fois (par héritage multiple) de B et de C.

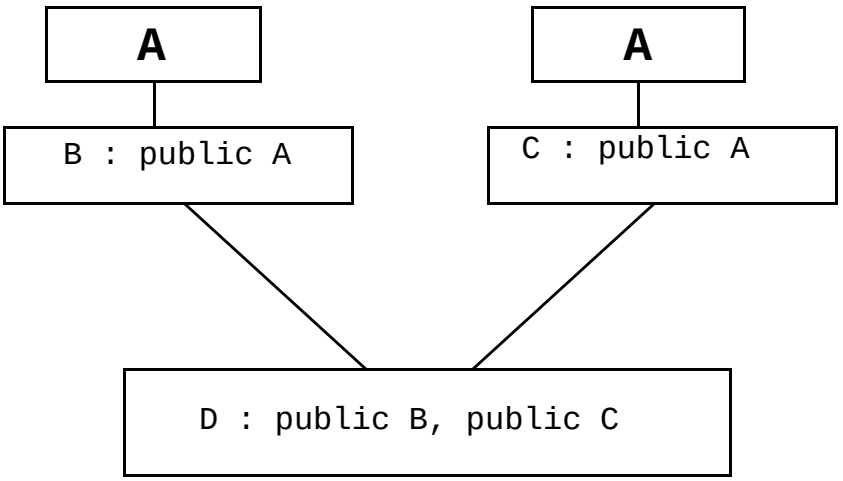
```
class    A
     {
     int  anInt;
     public :
          A(int ent) : anInt(ent) {}
          void helloFunc()
               { cout<<"Hello from A"<<endl; }
     };
class    B : public A
     {
     public :
          B(int i2) : A(i2) {}
     };
class    C : public A
     {
     public :
          C(int i3) : A(i3) {}
     };
class    D : public B, public C
     {
     public :
          D(int i4, int i5) : B(i4), C(i5) {}
     }; // Héritage multiple de B et de C

 void    main()

     {
     D    dc(1, 2);
     dc.helloFunc();
CC:"file.C":ambiguous A::helloFunc() and A::helloFunc() (no virtual base)
     }
```

Ce schéma cache une grave potentialité d'erreurs : B étant une sous-classe de A, il contient donc une instance de A. Il en va de même pour C. D, qui contient B et C, contient donc deux fois A! D'où le message de protestation du compilateur, qui ne peut savoir à quel A on s'adresse.

De fait, si je veux faire un test sur l'adresse de A dans le cadre d'une instance de D, ce test peut se révéler faux alors même que les objets sont identiques, parceque D contenant deux fois A, A possède plus d'une adresse. Graphiquement, cette situation correspond à la figure ci-dessous :



Ainsi, si je désire appeler la fonction A::helloFunc(), le compilateur ne pourra pas résoudre l'appel, parcequ'il sera dans l'incapacité de déterminer si c'est l'instance contenue dans B ou celle contenue dans C que je veux appeler. Je suis dans l'obligation de spécifier le chemin d'accès complet, par exemple B::helloFunc(), ou C::helloFunc().

```
void     main()

     {
     D    dc(1, 2);
     dc.C::helloFunc();
     }
```

Ce que le programmeur veut exprimer par le schéma çi-dessus, c'est que tant B que C ont besoin d'une instance de A, et que cette instance est propre tant à B qu'à C. Ainsi, on pourrait imaginer l'exemple de B et C implémentant tous deux des fichiers d'un genre particulier, et que la classe de base A serait alors `fstream` ou l'un de ses dérivés. Pour éviter toute ambiguité dans ce cas, il serait préférable de faire dériver B et C de A de manière privée, ce qui implique que A devient invisible depuis `main()`. Ceci peut nécessiter évidemment une adaptation des classes B et C, pour implémenter toutes les fonctionnalités désirées:

```
class     A    { ... };
class     B: private A { ... };
class     C: private A { ... };
class     D: public B, public C { ... };
```

Il s'agit ici de dérivation normale; A est une classe de base **non virtuelle**.

Comment faire si l'on désire en fait que B et C dérivent du même A? Il faut alors transformer la dérivation de A en dérivation **virtuelle**. Cette dérivation est illustrée çi-après:

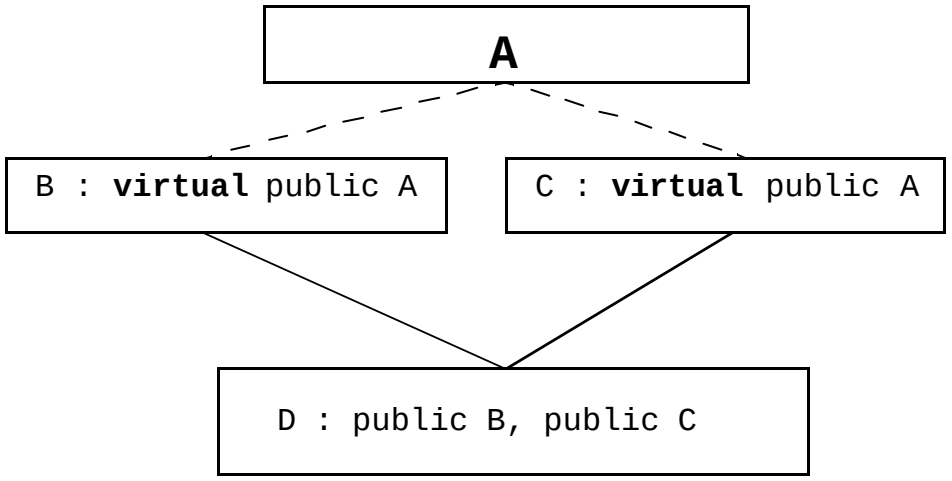
```
class     A
     {
```

```
     int  anInt;
     public :
          A(int ent) : anInt(ent) {}
          void helloFunc()
               { cout<<"Hello from A"<<endl; }
     };
class     B : virtual public A
     {
     public :
          B(int i2) : A(i2) {}
     };
class     C : virtual public A
     {
     public :
          C(int i3) : A(i3) {}
     };
class     D : public B, public C
     {
     public :
          D(int i4, int i5, int i6) : B(i4), C(i5), A(i6) {}
     }; // Héritage multiple de B et de C

void main()

     {
     D    dc(1, 2, 3);
     dc.helloFunc();
     }
```

La situation peut être schématisée de la manière suivante :



Que veut-on exprimer par là ? Simplement que B et C dérivent d'un seul et même A. B et C implémentent tous deux des détails de comportement de A que D veut voir réunis dans une même classe. La dérivation est dite virtuelle, parceque ni B ni C ne contiennent A. Qui dès lors, va devoir initialiser (**instancier**) A? Dans l'exemple de code çi-dessus, on peut voir que tant B, C et D initialisent A. Syntactiquement, le code est corect, mais il y a deux initiali-

sations, dans le cas qui nous préoccupe, qui sont inutiles (mais pas inutiles dans tous les cas !): il s'agit des initialisations faites par les constructeurs de B et de C. Ces initialisations (C::`A(i3) B::A(i2)`) n'ont pas d'effet dans le cas qui nous préoccupe, car A est une base **virtuelle**, qui doit être **initialisée par D**. Ceci correspond à la logique: si A devait être initialisé par B ou C, il serait impossible de savoir (sinon par des règles obscures et peu transparentes) qui de B ou de C aurait raison en cas de conflit!

En revanche, si l'on avait besoin, dans une autre portion de code, d'instancier **B** ou **C**, l'initialisation par B ou par C devrait prendre effet, ce qui justifie la possibilité offerte d'initialiser A depuis B ou C, même si dans l'exemple considéré, cette initialisation n'a pas d'effet.

Ceci pose néanmoins des problèmes difficilement maîtrisables, si l'on prévoit la possibilité d'instanciation de B ou C : comment B ou C peuvent-ils savoir si leur méthode d'initialisation est appliquée, ou si, par suite de dérivation multiple, elle n'a pas d'effet? Il est fort possible que des constructeurs différents soient appelés depuis B, C ou D: à la limite, ils pourraient même ne pas avoir les mêmes effets, ce qui conduirait B ou C à attendre un comportement de A qui ne serait pas respecté, du fait de l'initialisation par D !

Cette situation peut être évitée de plusieurs façons, si l'on respecte certaines règles lors de l'écriture de classes:

- Les constructeurs devraient tous avoir le même effet final.
- Une classe héritant du comportement d'une autre ne devrait jamais avoir à faire d'hypothèses sur le comportement de sa classe de base.
- N'utiliser l'héritage multiple que s'il s'impose de lui-même, et dans ce cas, essayer d'éviter la situation décrite dans l'exemple çi-dessus, avec un arbre (ou un fragment d'arbre) d'héritage en forme de losange (*diamond shaped inheritance subtree*).

15.4 *Utilisation de l'héritage multiple*

Comme mentionné au début de ce chapitre, l'héritage multiple peut être un outil très puissant pour définir des comportements et réutiliser du code; souvent, il s'avère également une source potentielle d'ambiguïtés et de difficultés de compréhension de la part de lecteurs.

Il est hélas souvent malaisé de réutiliser du code en provenance de schémas d'héritage multiple. Chaque fois que l'on se heurte à des difficultés, il s'avère que l'héritage multiple (spécialement **public**) a été utilisé à la légère, et ne reflète en réalité pas le fait qu'une dérivation implique une signification logique (**est un**, **isa** pour l'héritage publique). En pratique, l'héritage multiple s'avère assez simple à utiliser, pour autant qu'il n'y ait **qu'une branche publique** visible par l'utilisateur.

Aussi, avant de recourir à un outil aussi puissant que l'héritage (simple, mais surtout multiple), il est absolument indispensable de se demander si l'outil est approprié, et si ce qu'il représente correspond bien à la réalité que l'on désire exprimer dans le code. Cela peut paraître un truisme, mais il est toujours plus facile de faire simplement les choses simples, et de garder les outils complexes pour des problèmes à leur mesure.