

Héritage

1. Dérivation de classe (héritage)

Maintenant que nous connaissons bien la structure et le fonctionnement d'une classe, nous allons rendre nos programmes plus génériques. Il est fréquent de décrire un problème général avec des algorithmes appropriés, puis de procéder à de petites modifications lorsqu'un cas similaire vient à être traité.

La philosophie orientée objet consiste à limiter au maximum les macros, les inclusions, les modules. Cette façon d'aborder les choses présente de nombreux risques lorsque la complexité des problèmes vient à croître. La programmation orientée objet s'exprime plutôt sur un axe générique/spécifique bien plus adapté aux petites variations des données d'un problème. Des méthodes de modélisation s'appuyant sur UML peuvent vous guider pour construire des réseaux de classes adaptés aux circonstances d'un projet. Mais il faut également s'appuyer sur des langages supportant cette approche, et C++ en fait partie.

a. Exemple de dérivation de classe

Imaginons une classe `Compte`, composée des éléments suivants :

```
class Compte
{
protected:
    int numero; // numéro du compte
    double solde; // solde du compte

    static int num; // variable utilisée pour calculer le prochain numéro
    static int prochain_numero();

public:
    char* titulaire; // titulaire du compte

    Compte(char* titulaire);
```

```

~Compte(void);
void crediter(double montant);
bool debiter(double montant);
void relever();
};

```

Nous pouvons maintenant imaginer une classe `CompteRemunere`, spécialisant le fonctionnement de la classe `Compte`. Il est aisé de concevoir qu'un compte rémunéré admet globalement les mêmes opérations qu'un compte classique, son comportement étant légèrement modifié pour ce qui est de l'opération de crédit, puisqu'un intérêt est versé par l'organisme bancaire. En conséquence, il est fastidieux de vouloir réécrire totalement le programme qui fonctionne pour la classe `Compte`. Nous allons plutôt dériver cette dernière classe pour obtenir la classe `CompteRemunere`.

La classe `CompteRemunere`, quant à elle, reprend l'ensemble des caractéristiques de la classe `Compte` : elle hérite de ses champs et de ses méthodes. On dit pour simplifier que `CompteRemunere` hérite de `Compte`.

```

class CompteRemunere : public Compte
{
protected:
    double taux;

public:
    CompteRemunere(char*titulaire,double taux);
    ~CompteRemunere(void);

    void crediter(double montant);
};

```

Vous aurez remarqué que seules les différences (modifications et ajouts) sont inscrites dans la déclaration de la classe qui hérite. Tous les membres sont transmis par l'héritage : `public Compte`.

Voici maintenant l'implémentation de la classe `Compte` :

```
#include "compte.h"
#include <string.h>
#include <stdio.h>

Compte::Compte(char*titulaire)
{
    this->titulaire=new char[strlen(titulaire)+1];
    strcpy(this->titulaire,titulaire);
    solde=0;
    numero=prochain_numero();
}

Compte::~~Compte(void)
{
    delete titulaire;
}

void Compte::crediter(double montant)
{
    solde+=montant;
}

bool Compte::debiter(double montant)
{
    if(montant<=solde)
    {
        solde-=montant;
        return true;
    }
    return false;
}

void Compte::relever()
{
    printf("%s (%d) : %.2g euros\n",titulaire,numero,solde);
}

int Compte::num=1000;
```

```
int Compte::prochain_numero()
{
    return num++;
}
```

Mis à part l'utilisation d'un champ et d'une méthode statiques, il s'agit d'un exercice connu. Passons à la classe `CompteRemunere` :

```
#include "..\compteremunere.h"

CompteRemunere::CompteRemunere(char*titulaire,
    double taux) : Compte(titulaire)

{
    this->taux=taux;
}

CompteRemunere::~~CompteRemunere(void)
{
}

void CompteRemunere::crediter(double montant)
{
    Compte::crediter(montant*(1+taux/100));
}
```

L'appel du constructeur de la classe `Compte` depuis le constructeur de la classe `CompteRemunere` sera traité dans un prochain paragraphe. Concentrons-nous plutôt sur l'écriture de la méthode `crediter`.

Cette méthode existe déjà dans la **classe de base**, `Compte`. Mais il faut remarquer que le crédit sur un compte rémunéré ressemble beaucoup au crédit sur un compte classique, sauf que l'organisme bancaire verse un intérêt. Nous devrions donc appeler la méthode de base, `crediter`, située dans la classe `Compte`.

Hélas, les règles de visibilité et de portée font qu'elle n'est pas accessible autrement qu'en spécifiant le nom de la classe à laquelle on fait référence :

```

void CompteRemunere::crediter(double montant)
{
    // portée la plus proche => récurrence infinie
    crediter(montant*(1+taux/100));

    // écriture correcte
    Compte::crediter(montant*(1+taux/100));
}

```

Les programmeurs Java et C# ne connaissant pas l'héritage multiple, on a substitué le nom de la classe de base par un mot-clé unique, respectivement `super` et `base`.

Pour étudier le fonctionnement du programme complet, nous fournissons le module principal, `main.cpp` :

```

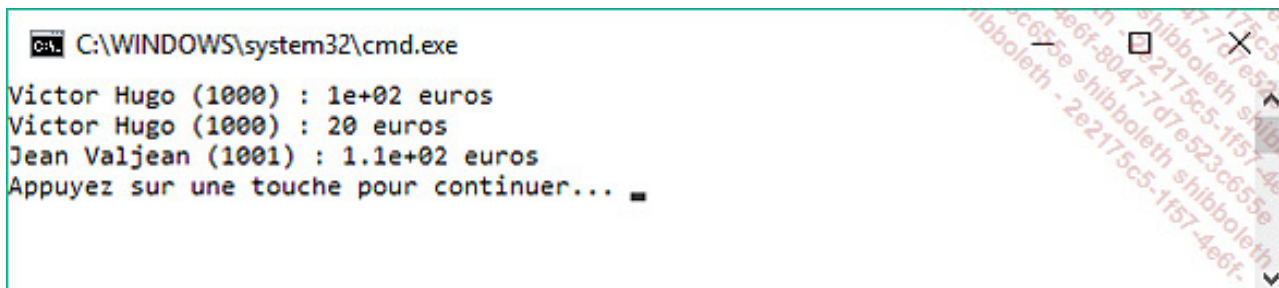
// #include "compte.h" // inutile car déjà présent dans compteremunere.h
#include "compteremunere.h"

int main(int argc, char* argv[])
{
    Compte c1("Victor Hugo");
    c1.crediter(100);
    c1.relever();
    c1.debiter(80);
    c1.relever();

    CompteRemunere r1("Jean Valjean",5);
    r1.crediter(100);
    r1.relever();
    return 0;
}

```

L'exécution donne les résultats suivants :



```

C:\WINDOWS\system32\cmd.exe
Victor Hugo (1000) : 1e+02 euros
Victor Hugo (1000) : 20 euros
Jean Valjean (1001) : 1.1e+02 euros
Appuyez sur une touche pour continuer...

```

Le crédit de 100 euros sur le compte de Jean Valjean a bien été rémunéré à hauteur de 5 %.

b. Héritage public, protégé et privé

Il existe une spécificité du langage C++ : il est possible de restreindre l'accès aux membres de la classe dérivée en contrôlant la publicité de l'héritage.

```

class Derive : public Base
{
...
};

```

Le mot-clé **public** peut être remplacé par **private** ou **protected**. Le tableau suivant énumère les cas d'accessibilité des champs d'une classe de base depuis une classe dérivée, en fonction de leur niveau de protection initial et du type de dérivation.

Visibilité dans la classe de base	Type de dérivation	Visibilité dans la classe dérivée
public	public	public
protected	public	protected
private	public	inaccessible
public	private	private
protected	private	private
private	private	private
public	protected	private
protected	protected	protected
private	protected	protected

Nous vous recommandons toutefois de ne pas abuser de ce type de construction, car les langages Java et C# ne disposent que de l'héritage public.

c. Appel des constructeurs

Nous savons qu'une classe dérivant d'une autre classe hérite de l'ensemble de ses champs, fussent-ils privés. C'est tout à fait normal, car une classe est toujours décrite comme un tout. Elle a besoin de l'ensemble de ses champs pour adosser son algorithmie, aussi une classe n'est jamais décrite avec l'arrière-pensée de la dériver par la suite. Ce

motif de conception est réservé aux classes abstraites, c'est-à-dire contenant au moins une méthode virtuelle pure et peu d'algorithmie.

De ce fait, il faut trouver un moyen d'initialiser l'ensemble des champs, y compris ceux provenant de la classe de base, privés ou non. Ce travail d'initialisation a déjà été réalisé à l'aide d'un ou de plusieurs constructeurs.

D'autre part, c'est bien la classe héritant qui est instanciée, et son constructeur appelé pour initialiser ses propres champs. Il paraît dès lors légitime de chercher à appeler au plus tôt un constructeur dans la classe de base, ou les constructeurs des classes de base en cas d'héritage multiple.

Le concepteur de C++ a voulu souligner que cet appel devait figurer avant l'exécution de la première instruction du constructeur de la classe héritant, aussi a-t-il prévu une syntaxe particulière :

```
CompteRemunere::CompteRemunere(
    char*titulaire, double taux) : Compte(titulaire)
{
    this->taux=taux;
}
```

Le constructeur de la classe `Compte` est appelé avant l'affectation du champ `taux`. En cas d'héritage multiple, on sépare les constructeurs par des virgules :

```
ConstructeurHerite(params) : Constructeur1(params),
Constructeur2(params)
{
}
```

Vous aurez noté la possibilité de transmettre au constructeur de la classe supérieure des paramètres reçus dans la classe héritant, ce qui est tout à fait légitime.

Si votre classe de base n'a pas de constructeur, ou si elle en possède un par défaut (sans argument), le compilateur peut réaliser cet appel automatiquement. Toutefois, l'absence de constructeur ou l'absence de choix explicite d'un constructeur constituent une faiblesse dans la programmation, et probablement dans la conception.

2. Polymorphisme

Le polymorphisme caractérise les éléments informatiques existant sous plusieurs formes. Bien qu'il ne s'agisse pas à proprement parler d'une caractéristique nécessaire aux langages orientés objet, nombre d'entre eux la partagent.

a. Méthodes polymorphes

Pour définir une méthode polymorphe, il suffit de proposer différents prototypes avec autant de signatures (listes d'arguments). La fonction doit toujours porter le même nom, autrement elle ne serait pas polymorphe. Le compilateur se base sur les arguments transmis lors de l'invocation d'une méthode, en nombre et en types, pour déterminer la version qu'il doit appeler.

```
class Chaîne
{
...
void concat(char c);
void concat(Chaîne s);
void concat(int nombre);
}:
```

Faites attention lorsqu'une méthode polymorphe est invoquée à l'aide d'une valeur littérale comme paramètre, il y a parfois des ambiguïtés délicates à résoudre. Le transtypage (changement de type par coercition) apparaît alors comme l'une des solutions les plus claires qui soit.

Dans notre exemple de classe `Chaîne`, nous allons invoquer la méthode `concat()` avec la valeur littérale 65. Si 65 est le code ASCII de la lettre B, c'est aussi un entier. Quelle version choisit le compilateur ?

```
Chaîne s="abc";
s.concat(65);
```

Suivant la version appelée, nous devrions obtenir soit la chaîne `"abcA"` soit la chaîne

```
"abc65" .
```

Nous pouvons résoudre l'équivoque à l'aide d'un transtypage adapté aux circonstances :

```
s.concat( (char) 65 );
```

Nous sommes certains cette fois que c'est la version recevant un char qui est appelée. Il vaut mieux être le plus explicite possible car le compilateur C++ est généralement très permissif et très implicite dans ses décisions !

b. Conversions d'objets

S'il est une méthode qui est fréquemment polymorphe, c'est certainement le constructeur. Celui-ci permet d'initialiser un objet à partir d'autres objets :

```
Chaine x('A');  
Chaine t(x);
```

Lorsque nous aurons étudié la surcharge des opérateurs, nous verrons comment tirer parti de cette multitude de constructeurs offerts à certaines classes.

3. Méthodes virtuelles et méthodes virtuelles pures

Le langage C++ est un langage compilé, et de ce fait, fortement typé. Cela signifie que le compilateur suit au plus près les types prévus pour chaque variable.

Livrons-nous à l'expérience suivante : considérons deux classes, `Base` et `Derive`, la seconde dérivant naturellement de la première. La classe `Base` ne contient pour l'instant qu'une méthode, `methode1()`, surchargée (réécrite) dans la seconde classe :

```
class Base  
{  
public:
```

```

void methode1()
{
    printf("Base::methode1\n");
}
};

class Derive : public Base
{
public:
    void methode1()
    {
        printf("Derive::methode1\n");
    }
};

```

Nous proposons maintenant une fonction `main()` instanciant chacune de ces classes dans le but d'invoquer la méthode `methode1()`.

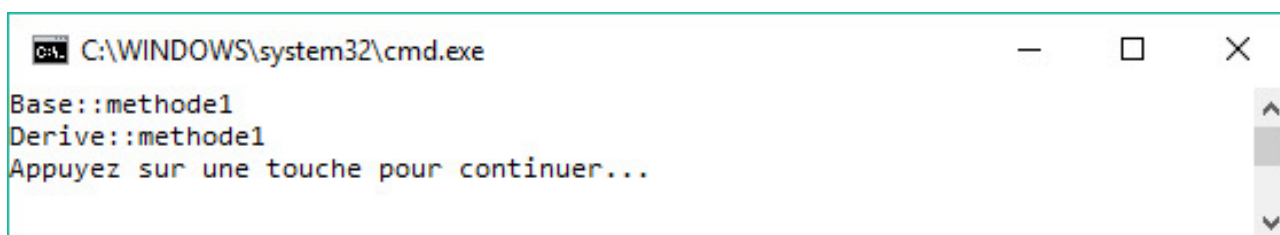
```

int main(int argc, char* argv[])
{
    Base*b;
    Derive*d;
    b = new Base();
    b->methode1();

    d = new Derive();
    d->methode1();
    return 0;
}

```

L'exécution du programme fournit des résultats en concordance avec les instructions contenues dans la fonction `main()` :



The screenshot shows a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe". The output of the program is displayed as follows:

```

Base::methode1
Derive::methode1
Appuyez sur une touche pour continuer...

```

Le compilateur a suivi le typage des pointeurs `b` et `d` pour appliquer la bonne méthode.

Maintenant considérons la classe `Base` comme un sous-ensemble de la classe `Derive`. Il est légal d'écrire `b=d` puisque toutes les opérations applicables à `b` sont disponibles dans `d`, l'opération est sans risque pour l'exécution. Quelle va être alors le choix du compilateur lorsque nous écrirons `b->methode1()` ?

```
printf("-----\n");
b = d;
b->methode1();
```

L'exécution du programme nous fournit le résultat suivant, le compilateur ayant finalement opté pour la version proposée par `Base`.

```
C:\WINDOWS\system32\cmd.exe
Base::methode1
Derive::methode1
-----
Base::methode1
Appuyez sur une touche pour continuer...
```

Ce résultat est tout à fait logique, en tenant compte du fait que l'édition des liens a lieu avant l'exécution. Aucun mécanisme n'est appliqué pour retarder l'édition des liens et attendre le dernier moment pour déterminer quel est le type désigné par le pointeur `b`.

Ce mécanisme existe dans C++, il s'agit des méthodes virtuelles.

Complétons à présent nos classes par `methode2()`, dont le prototype est marqué par le mot-clé **virtual** :

```
class Base
{
public:
    void methode1()
    {
        printf("Base::methode1\n");
```

```

    }

    virtual void methode2()
    {
        printf("Base::methode2\n");
    }
};

class Derive : public Base
{
public:
    void methode1()
    {
        printf("Derive::methode1\n");
    }

    void methode2()
    {
        printf("Derive::methode2\n");
    }
};

```

Vous aurez noté qu'il n'est pas nécessaire de marquer la version héritière, l'attribut `virtual` est automatiquement transmis lors de la dérivation.

Un nouveau `main()` fournit des résultats différents :

```

printf("-----\n");
b = d;
b->methode2();

```

La seconde méthode étant virtuelle, le compilateur retarde l'éditeur de liens. C'est à l'exécution que l'on détermine quel est le type désigné par le pointeur `b`. L'exécution est plus lente mais le compilateur appelle la "bonne" version.

En fait, les méthodes virtuelles représentent la grande majorité des besoins des programmeurs, aussi le langage Java a-t-il pris comme parti de supprimer les méthodes non virtuelles. De ce fait, le mot-clé `virtual` a disparu de sa syntaxe. Le langage C# est

resté plus fidèle au C++ en prévoyant les deux systèmes mais en renforçant la syntaxe de surcharge, afin que le programmeur exprime son intention lorsqu'il dérive une classe et surcharge une méthode.

Les méthodes virtuelles sont particulièrement utiles pour programmer les interfaces graphiques et les collections. Signalons qu'elles fonctionnent également avec des références :

```
Base& br = *d;
br.methode1();
br.methode2();
```

Il arrive fréquemment qu'une méthode ne puisse être spécifiée complètement dans son algorithmie, soit parce que l'algorithmie est laissée aux soins du programmeur, soit parce que la classe exprime un concept ouvert, à compléter à la demande.

Les méthodes virtuelles pures sont des méthodes qui n'ont pas de code. Il faudra dériver la classe à laquelle elles appartiennent pour implémenter ces méthodes :

```
class Vehicule
{
public:
    virtual void deplacer() =0 ;
};

class Voiture : public Vehicule
{
public:
    void deplacer()
    {
        printf("vroum");
    }
};
```

La méthode virtuelle pure est signalée par la conjonction de la déclaration `virtual` et par l'affectation d'un pointeur nul (`=0`). De ce fait, la classe `Vehicule`, abstraite, n'est pas directement instanciable. C'est `Voiture` qui est chargée d'implémenter tout ou partie des méthodes virtuelles pures reçues en héritage. S'il reste au moins une méthode

virtuelle pure n'ayant pas reçu d'implémentation, la classe `Voiture` reste abstraite. Ce n'est certes pas le cas de notre exemple.

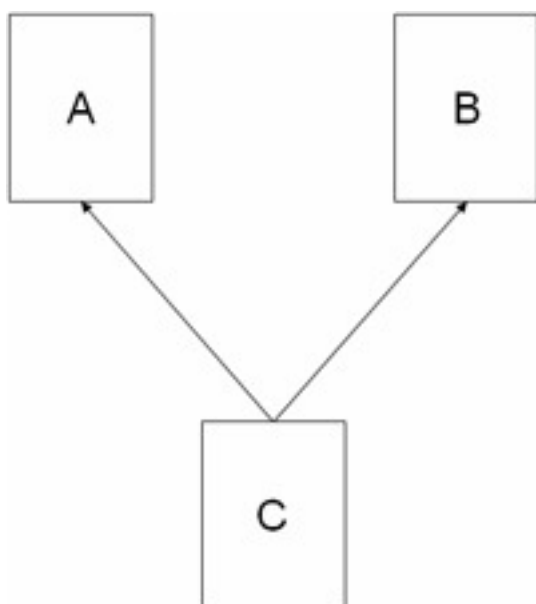
```
Vehicule*veh = new Vehicule; // interdit car classe abstraite
Voiture *voit1 = new Voiture; // ok
voit1->deplacer;
veh = new Voiture; // ok
```

Le langage Java ne connaissant pas les pointeurs, il a remplacé l'écriture `=0` ou `=NULL` par le mot-clé `abstract`. Cela signifie que les langages C++ et Java (en C#) peuvent exprimer les mêmes notions objet, ce qui est déterminant lorsque l'implémentation suit une modélisation comme UML.

Les langages successeurs de C++ proposent également la notion d'interface : il s'agit d'une classe abstraite dont toutes les méthodes sont abstraites, autrement dit virtuelles pures dans le vocabulaire C++. L'interface a sans aucun doute été proposée pour simplifier l'écriture des programmes, mais elles ne constituent pas une nouveauté en tant que telle. C++ est donc capable d'exprimer des modélisations faisant intervenir des interfaces.

4. Héritage multiple

Une situation inconfortable s'est présentée aux concepteurs de programmes orientés objet : une classe devait spécialiser plusieurs concepts simultanément. De ce fait, elle devait dériver d'au moins deux classes, et hériter de chacun de ses membres. L'auteur de C++ a pourvu son langage d'une syntaxe respectant cette modélisation que l'on désigne sous l'appellation héritage multiple.



Dans notre exemple, la classe **C** hérite à la fois de la classe **A** et de la classe **B**.

En termes C++, nous aurions traduit cette modélisation de la façon suivante :

```

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

a. Notations particulières

La première notation concerne naturellement l'appel des constructeurs. La classe **C** héritant des classes **A** et **B**, elle doit appeler les deux constructeurs, dans l'ordre de son choix. En fait, l'ordre d'initialisation n'a normalement pas d'importance, une classe étant un tout, elle n'est pas conçue en fonction d'autres classes. Si l'ordre revêt une importance particulière pour que le programme fonctionne, la modélisation est probablement à revoir.

```

C(param) : A(params), B(params)
{
  ...
}
  
```



```
}
```

Évidemment, les paramètres applicables aux trois constructeurs peuvent être différents en valeur, en type et en nombre.

Par la suite, nous devons distinguer les méthodes qui portent le même nom dans les classes `A` et `B`, et qui seraient invoquées par une méthode de `C`.

Imaginons la situation suivante :

```
class A
{
    void methode1() { ... }
};
```

et

```
class B
{
    void methode1() { ... }
};
```

Que se passe-t-il si une méthode de `C` appelle `methode1()`, dont elle a en fait hérité deux fois ? En l'absence d'un marquage particulier, le compilateur soulève une erreur car l'appel est ambigu :

```
class C : public A, public B
{
    void methode2()
    {
        methode1(); // appel ambigu
    }
};
```

Lorsque de tels cas se présentent, le programmeur doit faire précéder le nom de la méthode du nom de la classe à laquelle il fait référence. C'est bien entendu l'opérateur de

résolution de portée qui intervient à ce moment- là :

```
class C : public A, public B
{
    void methode2()
    {
        A::methode1() ; // ok
    }
};
```

À l'extérieur de la classe, le problème demeure, car le compilateur ne sait pas quelle version appeler :

```
class A
{
public:
    int f(int i)
    {
        printf("A::f(int=%d)\n",i);
        return i;
    }
    char f(char i)
    {
        printf("A::f(char=%c)\n",i);
        return i;
    }
};

class B
{
public:
    double f(double d)
    {
        printf("B::f(double=%g)\n",d);
        return d;
    }
};

class AetB : public A, public B
{
```

```

public:
    char f(char a)
    {
        printf("AetB::f(char=%c)\n",a);
        return a;
    }
    bool f(bool b)
    {
        printf("AetB::f(boolean=%d)\n",b);
        return b;
    }
};

int main(int argc, char* argv[])
{
    AetB x;
    x.f(1);    // appel ambigu
    x.f(2.0);  // appel ambigu
    x.f('A');
    x.f(true);
    return 0;
}

```

Il existe deux moyens pour lever l'ambiguïté des appels signalés dans la fonction `main()`. Le premier consiste à faire précéder l'invocation de `f` par `A::` ou `B::`, comme indiqué :

```

x.A::f(1000);
x.B::f(2.0);

```

Le second moyen consiste à utiliser dans le corps de la déclaration des classes `A` et `B` des déclarations d'utilisation : **using**

```

using A::f;
using B::f;

```

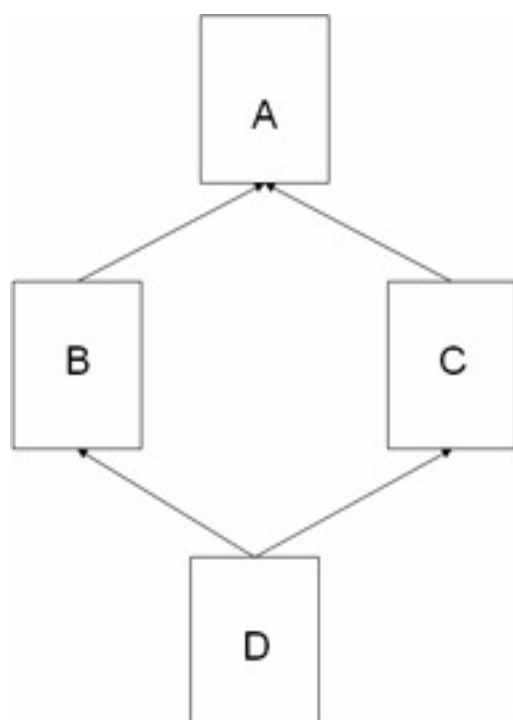
Le second moyen est à privilégier car il augmente la qualité de la modélisation.

b. Conséquences sur la programmation

Cette caractéristique a fortement augmenté la complexité des compilateurs, en même temps que la vie des programmeurs. Imaginons une classe **A** dérivant en deux classes **B** et **C**. Ajoutons à notre diagramme une classe **D** qui hérite simultanément de **B** et de **C**. Elle devrait hériter deux fois des membres de **A**.

Les langages Java et C# ont supprimé l'héritage multiple, en autorisant seulement l'héritage d'une classe et l'implémentation d'une interface. Une interface étant abstraite, elle ne contient ni champ, ni code. Les doutes sont par conséquent levés.

L'auteur de C++ a, lui, proposé le mécanisme des classes virtuelles.



En principe, tous les membres de la classe **A** sont hérités deux fois par la classe **D**. L'utilisation de l'opérateur de résolution de portée `::` contrôle l'accès à ces membres. Lorsque le concepteur (ou le programmeur) ne veut qu'un seul héritage, il peut utiliser le mécanisme de la dérivation virtuelle :

```

class B : public virtual A
{ ... };
class C : public virtual A

```

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

Le compilateur s'assurera alors que chaque membre n'est hérité qu'une fois. Toutefois, le programmeur ne pourra pas différencier l'origine des membres hérités. Ce mécanisme est donc à manier avec prudence.