

## Chapitre *III*

---

# Héritage, virtualité et polymorphisme

---

### Sommaire

---

|     |   |            |
|-----|---|------------|
| 1   | Principes . . . . .                                 | <b>102</b> |
| 2   | Héritage . . . . .                                  | <b>103</b> |
| 2.1 | Mécanisme de dérivation . . . . .                   | 103        |
| 2.2 | Constructeur d'une classe dérivée . . . . .         | 104        |
| 2.3 | Destructeur . . . . .                               | 105        |
| 2.4 | Constructeur par copie ou par déplacement . . . . . | 106        |
| 2.5 | Droit d'accès . . . . .                             | 106        |
| 3   | Héritage multiple . . . . .                         | <b>109</b> |
| 3.1 | Droit d'accès . . . . .                             | 110        |
| 3.2 | Occultation . . . . .                               | 110        |
| 3.3 | Surcharge . . . . .                                 | 112        |
| 4   | Virtualité . . . . .                                | <b>114</b> |
| 4.1 | Héritage virtuel . . . . .                          | 114        |
| 4.2 | Upcasting . . . . .                                 | 116        |
| 4.3 | Méthode virtuelle . . . . .                         | 118        |
| 4.4 | Destructeur virtuel . . . . .                       | 124        |
| 4.5 | Contrôle de surcharge . . . . .                     | 125        |
| 4.6 | Downcasting . . . . .                               | 126        |
| 4.7 | RTTI . . . . .                                      | 127        |
| 5   | Polymorphisme . . . . .                             | <b>130</b> |

---

### Introduction

Nous avons pour l'instant vu deux types de relation entre deux classes :

- une classe peut posséder un membre (=champ) qui est une instance de l'autre classe.
- une classe peut être amie d'une autre classe.

Nous allons maintenant voir d'autres types de lien qui permettent de construire des relations plus complexes entre les classes.

## 1 Principes

Lorsque des types différents possèdent des caractéristiques communes, certains traitements peuvent être appliqués sur ces caractéristiques des instances de ces types sans tenir compte de ces types.

### Exemple :

si on considère une société de location de véhicules qui veut gérer son parc. Les voitures et camions ont des caractéristiques communes :

marque, type, valeur estimée du véhicule, ...

mais sont représentés dans des classes différentes en raison de leurs caractéristiques propres :

- pour les voitures (cCar) : type de carburant, équipements, ...
- pour les camions (cTruck) : volume de chargement, type de permis, ...

Avec les outils déjà vus, le problème est le suivant :

Comment définir un type véhicule qui pourrait être une voiture ou un camion afin par exemple de manipuler une liste de véhicules génériques ?

Pour que cela soit possible, il faut que les types de véhicule spécialisés puissent être "vus" comme de simples véhicules,

La solution apportée à ce problème par le C++ est la suivante :

Afin de manipuler de façon indifférenciée des caractéristiques communes d'objets de types différents, il faut que ces classes aient une origine commune.

C'est-à-dire :

- les caractéristiques communes constituent une **classe de base**,
- les classes sont ensuite construites à partir d'une spécialisation de la classe de base.

On dit alors que la spécialisation est construite **par dérivation** publique de la classe de base (on dit aussi hériter).

Une dérivation peut également être vu, en première approche, comme une augmentation de la classe de base par les membres et les méthodes de la classe spécialisée.

La classe de base n'est donc pas une sous-partie (= un membre) de la classe dérivée, mais intégrée à la classe dérivée.

Dans la classe dérivée qui hérite (publiquement) d'une classe de base, l'accès aux champs et aux méthodes de la classe de base est identique aux champs et aux méthodes issus de la spécialisation.

**Exemple :** En terme de classe,

- la classe de base cVehicle définit les caractéristiques communes.
- la classe dérivé cCar est une spécialisation de la classe cVehicle pour les objets de type voiture.
- la classe dérivé cTruck est une spécialisation de la classe cVehicle pour les objets de type camion.

- les classes dérivées `cCar` et `cTruck` contiennent chacune des caractéristiques qui leurs sont propres, en plus des caractéristiques de la classe de base.

En terme d'instance,

- "Fusion" de la classe de base et de la spécialisation : de l'extérieur, l'ensemble des caractéristiques de `cCar` est vu comme si elles ne provenaient que d'une seule classe.
- pour une instance de `cCar`, chacune de ses caractéristiques (de base ou particulières) a une valeur particulière.
- une instance de classe spécialisée `cCar` peut également être observée comme une instance de `cVehicle`.

Tout se passe comme si seule la partie `cVehicle` de l'instance du `cCar` était visible.

Inversement,

- l'instance d'une classe spécialisée (dérivée publiquement) est aussi une instance de la classe de base.
- un pointeur sur une instance de la classe de base pourra être un pointeur sur une instance de l'une des classes dérivées.

**Exemple :**

- tous les objets `cCar` sont également des objets `cVehicle`.
- toute instance de `cCar` est également une instance de `cVehicle`.  
celle-ci consiste en la valeur des champs de la classe de base (ceux de `Vehicle`) dans la classe `cCar`.
- mêmes remarques en remplaçant `cCar` par `cTruck`.
- un tableau de pointeurs sur des instances `Vehicles` peut pointer sur des adresses d'instances de `cCar` ou `cTruck`.

## 2 Héritage

### 2.1 Mécanisme de dérivation

La définition d'une dérivation s'effectue comme :

**Exemple:**

```
// classe de base
class A { ... };
// classe dérivée
class B : public A { ... };
```

**Conséquence de cette déclaration :** la classe B ainsi définie contient :

- tous les membres de la classe A (intégrés à la classe B)
- toutes les méthodes A à l'exception :
  - ◊ des constructeurs (ils ne permettent pas de construire la partie B de l'objet).
  - ◊ du destructeur (il ne permet de détruire la partie B de l'objet).
  - ◊ de l'opérateur d'affectation `operator=` (il ne permet de copier que la partie A de l'objet).
- tous les membres et méthodes définis pour B.

Noter que les relations d'amitiés de A ne sont pas transmises.

**Remarque :** il est également possible de faire de l'héritage sur les structures (Exemple : `struct A`

; struct B: A ;). Résultat identique aux classes sans contrôle d'accès (= tout est public).

En conséquence, si B hérite de A, et que les méthodes héritées ne sont pas surchargées (*i.e.* redéfinies dans B) :

- pour une méthode héritée, la méthode de A lancée sur un objet de type B est exécutée en ne considérant que la partie A de l'objet B,
- pour un opérateur hérité (rappel tous sauf `operator=`), l'opérateur lancé sur un objet de type B est exécutée sur la partie A de l'objet B.

Le type renvoyé est inchangé (si A, reste de type A).

- à noter que les conversions sont également héritées (ce sont des opérateurs). Donc, s'il était possible de convertir A en T, il devient aussi possible de convertir B en T.
- **Remarque :** il est possible de supprimer une méthode héritée en utilisant la syntaxe `=delete`.

**Exemple :** suppression d'une méthode héritée

```
class A { ... int fun(); ... };
class B : public A { ... int fun() = delete; ... };
```

Considérer un héritage protégé ou privé dans la conception des classes avant d'envisager cette solution.

## 2.2 Constructeur d'une classe dérivée

Un constructeur d'une classe dérivée doit logiquement être défini de manière à construire sa partie héritée (caractéristiques de base) et sa partie propre (spécialisation).

On rappelle les règles de fonctionnement des constructeurs sur une classe :

- si aucun constructeur n'est défini, le compilateur en définit un par défaut qui lance le constructeur par défaut de chacun des membres.
- les membres sont construits dans l'ordre déclaration des membres dans la classe (et non l'ordre donné dans la liste d'initialisation).
- le constructeur d'un type interne (BIT=Build-In Type, exemple : int, float, bool) ne fait rien = pas d'initialisation.
- dès qu'un constructeur est défini, le constructeur par défaut défini par le compilateur n'est plus automatiquement intégré à la classe.

A ces règles, viennent s'ajouter celles spécifiques à l'héritage :

- les classes héritées sont initialisées dans l'ordre exact où elles sont héritées, et **avant** toute initialisation des membres spécialisés.
- par défaut (=sauf spécifications contraires), les membres hérités sont initialisés avec le constructeur par défaut des classes héritées.
- tout constructeur de la classe dérivée peut faire spécifiquement appel à un constructeur particulier d'une classe héritée dans sa liste d'initialisation.

**Exemples :**

- **cas 1 :** pas de constructeurs

```
class A { int a; };
class B : public A { int b; };
main() { B obj; ... }
```

Déroulement de la construction de obj = exécution du constructeur de B (=défaut).

1. appel du constructeur de A (=défaut)  
appel du constructeur de int (pour a) =défaut=pas d'initialisation.
2. appel du constructeur de int (pour b) =défaut=pas d'initialisation.

- **cas 2 :** constructeur par défaut défini pour A.

```
class A { ... public: A(): a(0) {}; ... };
```

Déroulement identique sauf que le constructeur de A existe, celui-ci initialise l'entier a avec 0, et donc l'objet A.

- **cas 3 :** constructeur par entier défini pour A.

```
class A { ... public: A(int v): a(v) {}; };
```

**Échec de compilation :** plus de constructeur par défaut A() car A(int) défini. Donc le constructeur par défaut B() ne peut plus s'exécuter.

- **cas 4 :** constructeur par entier défini pour A et appelé depuis B()

```
class A { ... public: A(int v): a(v) {}; };
class B : public A { ... public: B(): A(0), b(1) { }; };
```

Ok : B() lance spécifiquement le constructeur pour A, puis initialise sa partie.

- **cas 5 :**

```
class A { ... public: A(int v): a(v) {}; };
class B : public A { ... public: B(int u): A(u), b(0) { }; };
main() { B obj; ... }
```

**Échec de compilation :** plus de constructeur par défaut B() car B(int) défini. Donc échec de la déclaration B obj;.

- **cas 6 :**

```
class A { ... public: A(int v): a(v) {}; };
class B : public A { ... public: B(int u): A(u), b(0) { }; };
main() { B obj(3); ... }
```

Ok : le constructeur spécifique B(int) lance spécifiquement le constructeur A(int), puis initialise sa partie.

- **cas 7 :**

```
class A { ... public: A(int v): a(v) {}; };
class B : public A { ... public: B(int u): b(u), A(b) { }; };
main() { B obj(3); ... }
```

Compile et s'exécute mais **erreur** : l'ordre donné dans la liste d'initialisation n'est pas celui exécuté :

1. initialise d'abord les membres hérités : donc initialise A avec la valeur b (**erreur** car b non encore initialisé),
2. puis initialise b avec u (**ok**).

## 2.3 Destructeur

Le destructeur par défaut d'une classe dérivée `class B : public A` consiste en :

- l'appel des destructeur de chaque membre spécialisé de la classe B,
- l'appel au destructeur de la classe de base A, qui par défaut appelle les destructeur de chaque membre de base de la classe.

Plus précisément, l'ensemble des destructeurs est lancé dans l'ordre inverse exact des constructeurs (i.e. le premier membre construit est le dernier détruit).

Évidemment, la définition d'un destructeur soit pour la classe de base, soit pour la classe dérivée remplace le destructeur par défaut.

## 2.4 Constructeur par copie ou par déplacement

On rappelle que le constructeur par copie est l'une des méthodes créées par défaut pour une classe si aucun constructeur n'est défini.

Comment est créé le constructeur par copie d'une classe dérivée ?

- appel du constructeur par copie de la classe de base,
- appel du constructeur par copie sur chaque membre de classe dérivée.

En conséquence, si vous écrivez votre propre constructeur par copie, il est nécessaire d'y intégrer la copie des parties de classe héritée.

**Exemple :** constructeur par copie d'une classe dérivée

```
class A { protected: int a; };  
class B : protected A {  
    protected: int b;  
    public: B(const B& x) : A(x), b(x.b) {};  
};
```

où l'appel `A(x)` correspond à l'appel du constructeur par copie de `A` sur la partie `A` de l'objet `b`.

Le sens de `A(x)` sera précisé dans la section sur l'upcasting.

Même idée pour le constructeur par déplacement : il doit intégrer le déplacement des parties de la classe héritée, et utilise en général dans ce but le constructeur par déplacement de la classe mère.

## 2.5 Droit d'accès

Dans une classe standard, nous avons défini les droits d'accès aux membres de la classe :

- `private` : signifie que ce membre est privé à la classe, à savoir que son accès est limité aux seules méthodes de la classe, et n'est partagé avec aucune autre classe.
- `public` : signifie que ce membre est public, à savoir que l'on peut y accéder depuis l'extérieur de la classe.

Quels sont les droits d'accès dans une classe dérivée sur les membres/méthodes de la classe de base ?

Jusqu'à ce point, la présentation laisse entendre que les membres hérités ont les mêmes droits dans la classe dérivée que dans la classe de base : ceci est faux.

Les droits d'accès dans une classe dérivée dépendent :

- des droits d'accès aux membres dans la classe de base,
- du mode d'héritage utilisée par la classe dérivée pour hériter de la classe de base

Un membre avec un droits d'accès :

**private** n'est accessible que par les méthodes de la classe, et reste privé qu'importe les modes d'héritage (partage avec aucune autre classe).

**public** est accessible depuis n'importe quelle méthode ou fonction externe de la classe, modifiable par héritage.

**protected** est similaire à private dans la classe, modifiable par héritage.

Se souvenir que le mot-clé friend permet également d'ouvrir les accès privés aux fonctions et classes amies.

Conceptuellement :

- un membre privé a pour vocation à rester privé à la classe qui le déclare, et donc inaccessible à ses classes dérivées,
- un membre public a pour vocation à être accessible à tous, mais ces droits d'accès peuvent être restreint lors de l'héritage,
- un membre protégé a pour vocation à rester privé à la classe qui le déclare et à ses classes dérivées.

Pour les modes d'héritage :

- lors d'un héritage **private** :
  - ◊ un membre privé est inaccessible dans la classe dérivée,
  - ◊ un membre protégé ou public devient privé dans la classe dérivée.
 Les membres de la classe héritée deviennent donc privés.
- lors d'un héritage **protected** :
  - ◊ un membre privé est inaccessible dans la classe dérivée,
  - ◊ un membre protégé ou public devient protégé dans la classe dérivée.
 Les membres de la classe héritée deviennent donc protégée.
- lors d'un héritage **public** :
  - ◊ un membre privé est inaccessible dans la classe dérivée,
  - ◊ un membre protégé reste protégé dans la classe dérivée.
  - ◊ un membre public reste public dans la classe dérivée.

Les droits d'accès aux membres protégés ou public dans la classe héritée sont hérités de ceux de la classe de base.

**Exemple :**

```
class A {
    private:    int a;
    protected: int b;
    public:     int c;
};
```

```
class B1 : private A {
    // A::a inaccessible
    // A::b private
    // A::c private
};
```

```
class B2 : protected A {
    // A::a inaccessible
    // A::b protected
    // A::c protected
};
```

```
class B3 : public A {
    // A::a inaccessible
    // A::b protected
    // A::c public
};
```

**Comment choisir le mode d'héritage :**

- **private** : la classe héritée est une classe privée (non héritable) de la classe dérivée.
- **protected** : la classe héritée est une classe privée (héritable) de la classe dérivée.
- **public** : les droits d'accès aux membres privés (héritables) ou publics sont conservés.

**a) Droits par défaut**

- **accès à un membre :**

```
class A {  
    int a;  
};
```

A::a est privé.

- **héritage d'une classe :**

```
class B : A {};
```

L'héritage de A est privé.

Il est fortement déconseillé d'utiliser ces modes par défaut car ils rendent le code moins lisible. Donc, toujours spécifier explicitement les droits d'accès, à savoir les codes ci-dessus devrait être écrits comme :

```
class A {  
    private: int a;  
};  
class B : private A {};
```

**b) Modification manuelle des droits hérités**

Il est possible de modifier les droits d'un membre hérité dans la mesure où cette modification ne contrevient pas aux droits d'accès du membre dans la classe de base (=un accès déclaré privé ne peut être rendu protégé ou public).

Cela s'effectue simplement en déclarant le membre non qualifié (i.e. sans son type) avec sa résolution de portée dans la section de la classe dérivée souhaitée.

**Exemple :**

```
class A { protected: int a; };  
class B : protected A { public: int b; };  
class C : private B {  
    protected: B::A::a;  
    public: B::b;  
};
```

par cette écriture, B::A::a devient protégés et B::b public alors que le mode d'héritage (=privé) aurait dû rendre ces deux membres privés.



### 3 Héritage multiple

L'héritage multiple consiste à faire dériver une classe de plusieurs classes de base à la fois lors d'un seul héritage.

**Exemple : héritage multiple**

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

**Remarques :**

- Interdiction d'hériter plusieurs fois de la même classe (raison : impossibilité par la syntaxe de différencier les deux héritages)

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

mais ceci peut être réalisé en utilisant une classe intermédiaire :

```
class A { protected: int a; ... };
class B1 : public A { ... };
class B2 : public A { ... };
class C : public B1, public B2 { ... };
```

B1::A et B2::A sont deux sous-parties différentes de l'objet C.

Exemple d'accès aux a hérités dans C : avec B1::A::a et B2::A::a (voir ci-après).

- Interdiction pour une classe d'hériter d'elle-même (qu'importe le sens que vous croyez donner à cela) : crée une définition circulaire.

**Conséquences :**

- **construction d'un objet issu d'une classe avec héritage multiple** : construction des sous-parties de l'objet dans l'ordre dans lequel les classes sont héritées.

Dans l'exemple précédent : construit la partie A, la partie B puis les membres de C.

Si des constructeurs spécifiques des classes héritées doivent être appelés, les placer dans la liste d'initialisation :

```
class A { ... public: A(int); };
class B { ... public: B(int,int); };
class C : public A, public B {
    public: C(int u, int v) : A(u*v), B(u,v) { ... };
    ... };
```

- **destruction d'un objet issu d'une classe avec héritage multiple** : dans le sens inverse de la construction.
- **ambiguïtés** : s'il y a des ambiguïtés d'accès à un membre ou une méthode, elle peut toujours être résolue avec l'opérateur de résolution de portée (voir l'exemple de l'héritage multiple de la même classe).

#### a) Organisation mémoire

- **Rappel** : les champs d'une classe sont stockés dans l'ordre dans lequel ils sont déclarés.
- les instances des classes héritées sont en général stockées dans l'ordre dans lesquelles elles sont déclarées (attention : non requis par la norme).

- par défaut, l'inclusion multiple d'une même classe dans une hiérarchie de classes stocke dans une instance de la hiérarchie autant de copies de la classe que d'héritages de cette classe.

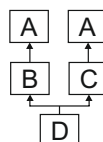
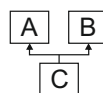
### Exemples :

#### Définition C++

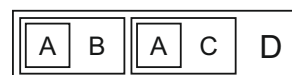
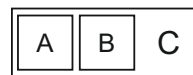
```
class A { ... };
class B { ... };
class C : public A , public B { ... };
```

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

#### Hiérarchie



#### Stockage



## 3.1 Droit d'accès

### Droit et héritage multiple :

si une classe hérite de plusieurs autres classes, le mode d'héritage doit être précisé pour chacune des classes.

### Exemple :

```
class A { ... };
class B { ... };
class C1 : A, B { ... };
class C2 : public A, B { ... };
class C3 : A, public B { ... };
class C4 : public A, public B { ... };
```

est équivalent à :

```
class C1 : private A, private B { ... };
class C2 : public A, private B { ... };
class C3 : private A, public B { ... };
class C4 : public A, public B { ... };
```

Les problèmes techniques apportés par l'héritage multiple seront abordés dans une section spécifique.

## 3.2 Occultation

Lorsqu'un objet membre portant le même nom est présent dans plusieurs classes de la chaîne d'héritage. Alors, les règles sont les suivantes :

- le nom fait référence à celui déclaré dans la classe courante, où sinon à la classe la plus récemment héritée.
- il est possible d'accéder au membre souhaité en utilisant l'opérateur de résolution de portée `::` en indiquant dans quelle classe on souhaite utiliser le membre.
- si un même nom apparaît plusieurs fois au même niveau de la chaîne d'héritage, alors il est susceptible de générer un conflit de nom.

**Exemple 1 :** même nom à des niveaux différents de la chaîne d'héritage

```
class A : { protected: int a; ... };
class B : public A { protected: int a; ... };
class C : public B { protected: int a; ... };
```

| classe            | C         | B    | A               |
|-------------------|-----------|------|-----------------|
| a de cette classe | a ou C::a | B::a | A::a ou B::A::a |

Le a de la classe courante C occulte celui des classes B et A.

Le a de la classe B occulte celui de la classe A.

**Exemple 2 :** même nom à des niveaux différents de la chaîne d'héritage

```
class A : { protected int a; ... };
class B : public A { protected: int a; ... };
class C : public B { ... };
```

| classe            | C   | B         | A               |
|-------------------|-----|-----------|-----------------|
| a de cette classe | n/a | a ou B::a | A::a ou B::A::a |

Le a visible dans la classe courante C est celui issu de la classe B.

Le a de la classe B occulte celui de la classe A.

**Exemple 3 :** même nom à des niveaux identiques de la chaîne d'héritage

```
class A1 : { protected: int a; ... };
class A2 : { protected: int a; ... };
class B : public A1, public A2 { ... };
```

| classe            | B   | A1    | A2    |
|-------------------|-----|-------|-------|
| a de cette classe | n/a | A1::a | A2::a |

Le a n'est pas utilisable dans la classe courante car il y a ambiguïté sur celui à utiliser.

**Exemple 4 :** même nom à des niveaux différents de la chaîne d'héritage

```
class A1 { protected: int a; ... };
class B1 : public A1 { protected: int b; ... };
class A2 { protected: int b; ... };
class B2 : public A2 { protected: int a; ... };
class C : public B1, public B2 { ... };
```

| classe | C   | B1                 | A1    | B2    | A2                 |
|--------|-----|--------------------|-------|-------|--------------------|
| a      | n/a | B1::a<br>B1::A1::a | A1::a | B2::a | n/a                |
| b      | n/a | B1::b              | n/a   | B2::b | A2::b<br>B2::A2::b |

Les membres a ou b ne sont pas utilisables directement dans la classe courante car il y a ambiguïté sur celui à utiliser.

Autrement dit, la résolution des noms s'effectue à chaque niveau de la chaîne d'héritage (*i.e.* la profondeur n'a pas d'importance).

**Note :** ces problèmes seront approfondis lorsque nous aborderons l'héritage multiple.

### 3.3 Surcharge

Les règles d'héritage pour les fonctions membres sont sensiblement identiques.

La recherche d'une fonction membre s'effectue par résolution de nom seulement (*i.e.* le type des arguments et le type de retour ne sont pas pris en compte lors de cette recherche),

et ceci, même si l'utilisation du type des arguments permettrait de déduire une fonction unique sur toute la chaîne d'héritage.

#### Conséquence :

la surcharge d'une fonction membre à un niveau de la chaîne d'héritage "domine" toutes les fonctions membres de même nom définies antérieurement dans la chaîne.

#### Exemple 1 : même fonction à des niveaux différents de la chaîne d'héritage

```
class A { public: int fun(int c) { return 2*c; } };
class B : public A {
    public: void fun(void) { ... } };
class C : public B {
    public: char fun(char c) { return c+1; } };
main() {
    C c;
    c.fun(2);
}
```

`char C::fun(char)` surcharge toutes les méthodes héritées de même nom indépendamment de leurs paramètres.

Donc,

- l'appel à `c.fun(2)` fait appel à `char C::fun(char)` (2 est converti en `char`).
- un appel à `c.fun()` provoque une erreur car la seule méthode `fun` définie dans la classe C est `char C::fun(char)` qui surcharge `void C::B::fun(void)`.

#### Exemple 2 : ambiguïté résultante de méthodes issue de deux branches

```
class A1 { public: int fun(int b) { return b+1; } };
class A2 { public: float fun(float c) { return c/2.f; } };
class B : public A1, public A2 { ... };
main() {
    C c;
    c.fun(2);
}
```

Les deux méthodes `fun` étant issues de deux branches différentes de la chaîne d'héritage, il n'y a donc pas moyen de déterminer quelle méthode doit être appelée.

Ce code ne compile pas et signale l'ambiguïté de l'appel à `fun`.

Autrement dit, le type des paramètres n'est jamais utilisé pour lever une ambiguïté issue de la chaîne d'héritage.

La justification de ce comportement est simple :

- la surcharge définie dans la dernière spécialisation prime sur toutes les autres. La spécialisation suppose donc que toute surcharge rend l'ensemble des autres méthodes définies antérieurement obsolètes.
- il est possible de "ramener" les méthodes de la classe de base dans la classe dérivée en changeant son mode d'accès. On rappelle que dans ce cas, seul le nom non qualifié doit être donné.

**Exemple 3 : transfert d'exécution**

```
class A { public: void fun(void) { ... } };
class B : public A { public: int fun(int c) { ... } };
class C : public B {
public:
    B::A::fun;
    char fun(char c) { return c + 1; }
};
```

La déclaration en `public` de `B::A::fun` permet de ramener l'ensemble des méthodes de nom `fun` de `B::A` dans la portée de la classe dérivée.

- si une méthode particulière définie dans la chaîne d'héritage veut être conservée alors qu'au moins une surcharge existe dans la classe dérivée (et donc que cette surcharge efface la méthode initiale),
  - ◇ **méthode 1** : on déclare le nom non qualifié avec son chemin complet (ramène toutes les méthodes de ce nom),
  - ◇ **méthode 2** : une nouvelle méthode doit être définie dans la classe dérivée pour transférer l'exécution à la méthode qui doit être conservée.

**Exemple 4 : transfert d'exécution**

```
class A { public: int fun(int c) { return 2*c; } };
class B : public A { public: void fun(void) { ... } };
class C : public B {
public:
    char fun(char c) { return c+1; }
    // méthode 1:
    B::A::fun;
    // méthode 2:
    inline int fun(int c) { return B::A::fun(c); }
};
```

## 4 Virtualité

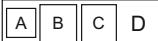
### 4.1 Héritage virtuel

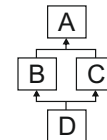
#### a) Introduction

Comment faire si l'on souhaite qu'une classe héritée plusieurs fois soit commune à la hiérarchie de l'héritage ?

Par exemple, on voudrait construire la hiérarchie ci-contre :

Si A est commune à B et C, le stockage de A devrait être unique.

On rêve de ceci : 



#### b) Principe

- ajouter le mot-clé **virtual** lors de l'héritage de la classe à rendre virtuelle,
- **attention** : n'utiliser ce mot-clé **uniquement** lors de l'héritage de la classe à virtualiser sur la classe qui en dérive immédiatement (*i.e.* et non lors des héritages ultérieurs sur une classe qui contient une classe virtuelle).

**Exemple :**

```

// A virtuelle , B dérivation de A => mot-clé virtual
// A virtuelle , C dérivation de A => mot-clé virtual
// B et C non virtuels dans D => pas de mot-clé virtual
class A { ... };
class B : virtual public A { ... };
class C : virtual public A { ... };
class D : public B, public C { ... };
  
```

#### c) Stockage

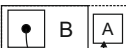
**Problème** : si on fait place un pointeur sur la partie B ou C de D, alors l'utilisation de ce pointeur devra permettre de "trouver" le A commun.

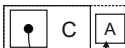
**Conséquences** :

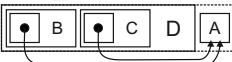
- la partie A commune n'est ni stockée dans B, ni dans C, mais à la fin de l'instance de l'objet contenant l'instance de la classe virtuelle A.
- à l'intérieur de la classe qui dérive immédiatement de la classe virtuelle A, est placé un pointeur qui pointe vers A à la place du stockage de A.

**Exemple 1 :**

Avec les définitions données ci-dessous, le résultat est le suivant pour les instances issues des différentes classes :

instance de B : 

instance de C : 

instance de D : 

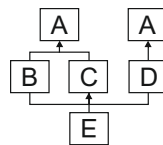
**Exemple 2 :**

```

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

```

La hiérarchie associée est :



Dans toute instance de cette classe, il y a donc :

- une instance de A virtuelle partagée entre les classes B et C,
- une instance de A propre à la classe D (non virtualisée).

Le stockage de la classe E est

**Construction d'une instance avec classes virtuelles :**

- si l'héritage d'une classe A est virtuel alors :
  - ◊ si une instance de la classe virtuelle n'existe pas, la créer derrière le stockage de la classe de l'objet en cours de construction.
  - ◊ créer un pointeur à l'endroit où A doit être stocké, et faire pointer le pointeur vers l'instance de la classe virtuelle.
- sinon créer une instance de A à la position courante de l'objet en construction.

**Attention :**

un héritage virtuel ajoute donc systématiquement un pointeur à la place de l'instance de la classe, et le fait pointer vers l'instance virtuelle placée à la fin de l'objet.

**Conséquences :**

- La classe grossit donc d'autant de pointeurs qu'il y a d'héritages virtuels (par rapport au stockage optimal de la classe).  
Pour les classes de petite taille, l'impact de stockage peut être important.
- L'accès à une classe virtuelle nécessite un déréférencement supplémentaire.

**d) Initialisation des classes virtuelles**

- si une classe est déclarée comme virtuelle, alors son constructeur doit être déclaré dans la classe finale.
- donc, dans une hiérarchie de classes héritant d'une classe virtuelle, le constructeur de la classe virtuelle doit être déclaré à tous les niveaux de la hiérarchie pour lequel des objets sont instanciés.

**Exemple :**

```

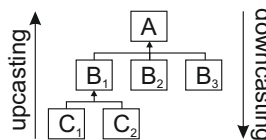
class A { protected: int a;
public: A(int x) : a(x) {}; };
class B : virtual public A { protected: int b;
public: B(int x) : A(1), b(x) {}; };
class C : virtual public A { protected: int c;
public: C(int x) : A(2), c(x) {}; };
class D : public A { protected: int d;
public: D(int x) : A(3), d(x) {}; };
class E : public B, public C { protected: int e;
public: E(int x) : A(4), B(5), C(6), e(x) {}; };
class F : public D, public E { protected: int f;
public: F(int x) : E::A(7), D(8), E(9), f(x) {}; };

```

Dessiner la classe, et donner la valeur de la valeur de a pour un objet de type B, C, D, E, et F.

**4.2 Upcasting**

L'upcasting est un mécanisme permettant de convertir sans jamais aucune perte de donnée une classe dérivée en l'une des classes dont elle dérive (car les données de l'objet contenu dans la classe de base est hérité, donc intégré à la classe dérivée).



La transformation se fait d'une classe dérivée vers une classe de base, **sans nécessité de conversion explicite**, d'où le terme d'"up"-casting.

**Exemple : upcasting**

```

class A {
    protected: int a;
    ...
    friend int fun(A&);
};

class B : public A { protected b; ... };
int fun(A &x) { ... };
main() {
    B    b;
    fun(b);
}

```

fun(b) prend en argument un objet de type B et l'upcaste en objet de type A (revient à ne transmettre à la fonction qu'une sous-partie de B).



L'upcasting est également possible pour les pointeurs et les références (toujours sans nécessité de conversion explicite).

Si l'upcasting permet d'appeler les méthodes héritées avec tout sous-objet de l'objet dérivé, il pose un problème dans le cas ci-dessous :

**Exemple :** upcasting d'un pointeur ou d'une référence

```
class A { protected: int a;
public: int fun() { return a+1; }; };
class B : public A { protected: int b;
public: int fun() { return a+b; }; };
class C : public B { protected: int c;
public: int fun() { return a*b*c; }; };
int main() {
    C c;
    B *bp = &c, &b = c;
    A *ap = &c, &a = c;
    c.fun();    // exécute int C::fun()
    b.fun();    // exécute int B::fun()
    a.fun();    // exécute int A::fun()
}
```

En effet, la fonction exécutée est celle du type courant de l'objet, et non celle associée à l'objet sous-jacent. Ce problème est résolu par la virtualité.

**Sens d'un upcasting :** comment un objet d'un type dérivé peut-il être transformé en l'un des types de base utilisés ?

**Exemple :** upcasting d'un pointeur ou d'une référence

```
class A1 { protected: int a1; };
class A2 { protected: int a2; };
class B : public A1, public A2 { protected: int b; };
```

Stockage associé à B :

|        |
|--------|
| A1::a1 |
| A2::a2 |
| B::b   |

```
int main() {
    B    b, *bp = &b;
    A1   *a1p = bp;
    A2   *a2p = bp;
}
```

- bp pointe (évidemment) au début du stockage.
- a1p pointe sur A1::a1, donc a1p = bp.
- a2p pointe sur A2::a2, donc a2p = bp + sizeof(A1::a1).

A savoir, un upcasting renvoie :

- pour un objet (exemple : A2 a2=b;), l'objet a2 est une copie de la sous-partie de b contenant l'objet a2.
- pour une référence sur un objet (exemple : A2 &a2r=b;), la référence a2r est une référence vers la sous-partie de b contenant l'objet a2.
- pour un pointeur sur un objet (exemple : A2 \*a2p=&b;), la référence a2p est un pointeur vers la sous-partie de b contenant l'objet a2.

### 4.3 Méthode virtuelle

#### a) Introduction

Considérons :

- Une classe B dérivée à partir d'une classe de base A.
- La classe A possède une méthode Fun, surchargée dans la classe B avec une implémentation propre et correcte de Fun sur B.
- Un pointeur pB vers objet de type B peut être upcasté comme un pointeur pA sur un objet A.

**Problème** : si on applique la méthode Fun sur pA (*i.e.* pA->Fun(. . .)), alors :

- la méthode exécutée est A : : Fun (cette méthode s'applique uniquement à la sous-partie A de l'objet B).
- or, le type sous-jacent de l'objet pointé par pA est B, et en conséquence la méthode qui aurait dû être exécutée est B : : Fun.

Le problème est double :

- Comme l'implémentation de A : : Fun est utilisée à la place de B : : Fun, le résultat est incorrect,
- En même temps, le pointeur étant de type A, comment le programme ou le compilateur peuvent-ils savoir que l'objet sous-jacent est de type B ?

D'où l'introduction des méthodes virtuelles.

#### b) Principe

Ce problème n'est pas seulement un problème technique mais conceptuel.

Par exemple, on prend une classe cInstrument qui propose une méthode Play,

- les spécialisations cPiano, cTrompette, ... dérivent de cette classe,
- chaque spécialisation a une implémentation particulière de Play permettant de jouer le morceau souhaité avec l'instrument associé à la classe.

Si on construit un orchestre comme étant un tableau de pointeurs sur des cInstruments, alors on souhaite que l'appel de Play sur chaque instrument exécute la version spécialisée de Play.

Ceci n'est possible que si la méthode Play est déclarée comme étant **virtuelle**.

Il y a ce titre deux approches :

- soit cInstrument fournit une implémentation effective de Play (implémentation par défaut), et dans ce cas, toute classe dérivée qui ne fournirait pas une surcharge spécialisée de Play est en mesure d'utiliser l'implémentation par défaut de cInstrument.

**Déclaration** : virtual [définition-fonction];

- soit cInstrument n'a pas vocation à fournir une implémentation par défaut, et dans ce cas :
  - ◊ La classe cInstrument ne peut pas être instanciée (mais un pointeur ou une référence de cInstrument peut faire référence à une spécialisation),
  - ◊ Toute classe qui dérive de cInstrument **doit** fournir une implémentation de Play sauf si elle n'est pas destinée à être instanciée elle non plus.

La méthode Play est alors dite **virtuelle pure**.

**Déclaration** : virtual [déclaration-fonction] = 0;

**Exemple : méthode virtuelle**

```

class A { public: virtual int fun() { ... } };
class B1 : public A { public: virtual int fun() { ... } };
class B2 : public A { public: virtual int fun() { ... } };
class B3 : public A { ... };
int main() {
    A *tA[4] = { new B1(), new A(), new B2(), new B3() };
    tA[0]->fun();    // exécute int B1::fun()
    tA[1]->fun();    // exécute int A::fun()
    tA[2]->fun();    // exécute int B2::fun()
    tA[3]->fun();    // exécute int A::fun()
    ... }

```

**Exemple : méthode virtuelle pure**

```

class A { public: virtual int fun() = 0 };
class B1 : public A { public: virtual int fun() { ... } };
class B2 : public A { public: virtual int fun() { ... } };
class B3 : public A { ... }; // fun non surchargée
int main() {
    A *tA[2] = { new B1(), new B2() };
    tA[0]->fun();    // exécute int B1::fun()
    tA[1]->fun();    // exécute int B2::fun()
    ... }

```

pas de code défini pour `A::fun()`.

impossible de définir un objet de type `A` ou `B3` (dans l'exemple 2).

**Remarque :**

Le type de retour d'une méthode virtuelle ne peut pas être changé (le compilateur doit garantir que l'utilisation de la méthode de la classe de base sur un objet polymorphe) sauf si ce type de retour peut être upcasté dans le type de retour de la classe de base.

En revanche, toute surcharge occulte les méthodes virtuelles de même nom et qui n'ont pas le même prototype.

**Définitions :**

- **classe abstraite** : classe qui contient au moins une méthode virtuelle pure. Elle est dite abstraite car elle ne peut pas être instanciée (et donc, tout objet de ce type ne peut être qu'abstrait).
- **classe abstraite pure** : classe qui ne contient que des méthodes virtuelles pures. Ce type de classe est qualifié souvent d'interface car elle n'expose que des fonctionnalités qui doivent être implémentés dans les classes qui représentent des objets concrets.

**c) Lien d'appel de fonction (function call binding)**

C'est le procédé qui lie l'appel de la fonction au code de la fonction appelée.

Il existe deux types de lien d'appel :

- **lien anticipé (early binding)** : le lien est effectué avant l'exécution du programme (à savoir, par le compilateur ou l'éditeur de lien).

**Exemples :**

- ◇ fonctions classiques.
- ◇ méthodes non virtuelles (type de l'objet connu, donc méthode à appeler connue).
- ◇ méthodes virtuelles sur un objet de type non pointeur (car le type de l'objet est certain).
- **lien tardif** (late, dynamic ou **runtime binding**) : le lien est effectué lors de l'exécution du programme.

**Exemples :**

- ◇ méthodes virtuelles (type de l'objet déterminé à l'exécution, donc méthode à appeler inconnue à la compilation) sur un objet de type pointeur.
- ◇ pointeurs de fonction.
- ◇ méthodes virtuelles sur un objet de type pointeur (car le type du pointeur n'est peut-être pas le type de l'objet pointé).

La définition de méthodes virtuelles requiert donc de déterminer à l'exécution quel est le type de l'objet qui appelle une méthode.

Rappelons comment un objet classique est géré :

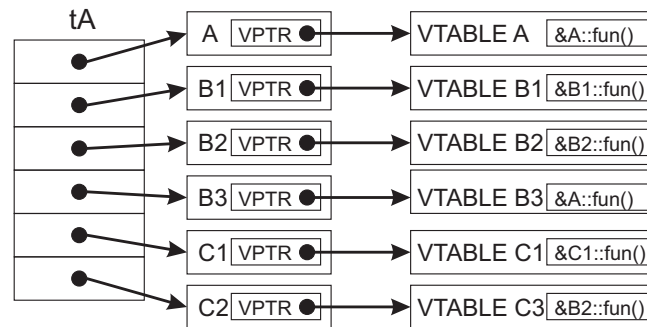
- une instance d'un objet ne stocke que les **données** de la classe  
*i.e.* la présence ou le nombre de méthodes dans la classe n'a aucune incidence sur la taille de la classe,
- l'affectation de la méthode à appeler est effectué par lien anticipé (=appel direct).

Lorsqu'un objet contient une méthode virtuelle :

- une table (la virtual table ou VTABLE) contenant l'ensemble des **méthodes virtuelles** de la classe est créée et disponible en mémoire pour la classe,
- un vpointer (ou VPTR) supplémentaire pointant sur la VTABLE est ajouté à la zone de stockage des données,
- lorsqu'un appel est effectué, VPTR permet l'accès à la VTABLE (qui contient l'ensemble des **méthodes virtuelles** pour un objet de ce type), et permet d'identifier la méthode à appeler (par décalage dans la VTABLE), et d'effectuer le lien tardif.

**Exemple : VTABLE**

```
class A { public: virtual int fun() };
class B1 : public A { public: virtual int fun() { ... } };
class B2 : public A { public: virtual int fun() { ... } };
class B3 : public A { ... }; // fun non surchargée
class C1 : public B1 { public: virtual int fun() { ... } };
class C2 : public B2 { ... }; // fun non surchargée
main() {
    A *tA[6] = { new A, new B1, new B2, new B3, new C1, new C2 };
    ... }
```



### Conséquence pour un objet contenant une méthode virtuelle :

- la structure de donnée est augmentée avec VPTR  
 $\text{sizeof(B avec virtual)} = \text{sizeof(B sans virtual)} + \text{sizeof(VPTR)}$
- l'appel de toute méthode s'effectue à travers un pointeur de fonction (**late binding**) plutôt que directement (**early binding**).

L'utilisation d'une méthode virtuelle est donc complètement transparente pour l'utilisateur.

### Conséquences indirectes :

- toute classe qui dérive d'une classe contenant une méthode virtuelle contient également une VTABLE.
- une méthode virtuelle ne peut pas être **inline** (car elle doit avoir une adresse dans la VTABLE).
- une classe abstraite ne peut pas être passée à une fonction par valeur
- ne jamais utiliser `memset` pour initialiser à 0 ce type de classe sinon VPTR sera lui aussi mis à zéro (idem pour toutes opérations bas-niveaux sur la classe pouvant affecter le VPTR),
- éviter l'appel à des méthodes virtuelles dans des sections intensives du code (en raison des appels indirects),
- si une classe est sans données mais contient au moins une méthode virtuelle, alors elle est de taille `sizeof(VPTR)`.

Notons qu'une classe sans donnée ni méthode virtuelle a une taille 1 (un membre factice est ajouté afin que sa taille ne soit pas nulle ; sinon il serait impossible de faire des tableaux avec de tels objets).

- l'attribut `__declspec(novtable)` appliquée à une classe d'interface pure (= une classe ne contenant que des méthodes virtuelles pures) indique qu'elle ne sera jamais instanciée toutes seules (toujours héritée mais jamais instanciée), et que le constructeur et le destructeur n'initialiseront jamais la VTABLE. En conséquence, le compilateur peut supprimer cette VTABLE (rappel : une instance peut contenir plusieurs VPTR, voir plus loin)

### d) Définitions de méthodes pure virtuelles

Lorsqu'une méthode est définie comme virtuelle pure, il est néanmoins possible de lui donner une définition afin de partager une implémentation par défaut avec certaines classes dérivées.

**Exemple : définition d'une méthode virtuelle pure**

```

class A {
    public: virtual int fun() = 0;
    // public: virtual int fun() = 0 { ... }; [ILLEGAL]
};
// définition de la méthode virtuelle pure
int A::fun() { ... };
// classe dérivée
class B : public A {
    // utilisation de la définition de la classe de base
    public: int fun() { return A::fun(); };
};

```





**Note : destructeur virtuel pur**

Lorsque l'on souhaite créer une classe mère à toutes les autres classes, le destructeur doit être virtuel, et la classe ne doit être spécifique en rien, d'où l'idée d'un destructeur virtuel pur.

Le code du destructeur doit être donné (pour résolution de l'appel du destructeur sur un objet générique) : même technique que ci-dessus.

Notons tout d'abord que le le VPTR d'une classe abstraite est un pointeur invisible stocké comme premier champ de la classe.

**e) Règles de construction de la VTABLE**

|   |   |                                 |
|---|---|---------------------------------|
| class B : public A  |  |                                 |
| class B (?VPTR)<br>: public A (+VPTR)                         |  | VPTR → VTABLE de B              |
| class B (+VPTR)<br>: public A                                 |  | VPTR → VTABLE de B              |
| class B (?VPTR)<br>: public A1 (+VPTR)<br>, public A2 (+VPTR) |  | VPTR <sub>1</sub> → VTABLE de B |

**Conséquence :**

- lors d'un upcasting d'un type B vers une référence ou un pointeur de type A, le VPTR pointe vers la VTABLE de B (*i.e.* associée au type original de l'objet),
- lors d'un upcasting d'un type B vers un objet de type A (constructeur par copie), le VPTR pointe vers la VTABLE de A (initialisé par le constructeur).

**f) VTABLE, constructeurs et destructeurs**

- lorsqu'un objet est construit de la classe de base jusqu'à la classe la plus dérivée, alors avant chaque lancement du constructeur pour la classe en cours de construction, VPTR pointe vers la VTABLE de la classe.
- inversement, lors de l'exécution du destructeur (du plus dérivé jusqu'à la classe de base), alors avant chaque lancement du destructeur pour la classe en cours de destruction, VPTR pointe vers la VTABLE de la classe.

**Conséquences :**

- VPTR est toujours redirigé vers la VTABLE associée au type courant de l'objet avant l'appel d'un constructeur ou d'un destructeur.  
Permet de faire en sorte que les appels aux méthodes virtuelles soient toujours fonctionnelles dans les constructeurs et les destructeurs, mais que la méthode appelée soit localisée aux données de l'objet courant (i.e. dans le cas d'un destructeur, un appel à une méthode virtuelle aurait pu utiliser sinon une version de la méthode utilisant une partie de l'objet déjà détruite).
- Le compilateur ajoute donc l'initialisation du VPTR et l'allocation / desallocation de this en code caché aux constructeurs / destructeurs.

**g) Découpage d'objet (object slicing)**

Lorsqu'un objet issu d'une classe dérivée subit un upcasting (passage par valeur ou construction), alors il est découpé (VTABLE comprise) pour ne conserver que la partie de base.

**Exemple : object slicing**

```
class A {
protected: int a;
public: A(int x) : a(x) {};
        virtual void view() { cout << a << endl; }; };
class B : public A {
protected: int b;
public: B(int x, int y) : A(x), b(y) {};
        void view() { cout << a << ", " << b << endl; };
        virtual int add() { return a+b; }; };
int main() {
    B    b(2,3);      b.view();    // => 2,3
    A    a = b;       a.view();    // => 2
    A    &ar = b;     ar.view();   // => 2,3
    ar = B(4,5);     ar.view();   // => 4,3
}
```

Dans l'exemple précédent :

- a est construit en utilisant le constructeur par copie et un upcasting de b. En conséquence, a est initialisée avec la sous-partie A de b, et son VPTR pointe vers la VTABLE de A.
- ar est une référence à la sous-partie A de b, mais le VPTR de ar continue à pointer vers la VTABLE de B.  
Ceci est bien le comportement attendu : l'objet sous-jacent est de type B, la méthode exécutée est bien B::view().
- l'affectation d'un objet de type B à ar provoque un découpage de l'objet B : seule la partie A de ar est affectée avec la partie A de l'objet B, bien que l'objet sous-jacent de ar soit de type B.  
Rendre operator= virtuel ne change rien au problème sauf à surcharger A& operator=(const A&) dans B, et y effectuer un dynamic cast vers B.

## 4.4 Destructeur virtuel

Si une hiérarchie de classe a pour but de manipuler des objets polymorphiques qui alloue de la mémoire en interne, alors il est possible de se retrouver dans le cas suivant.

**Exemple :**

```
class A { protected: int *x;
public: A(int n) : x(new int[n]) {};
      ~A() { delete [] x; } };
class B : public A { protected: int *y;
public: B(int n, int m) : A(n), y(new int[m]) {};
      ~B() { delete [] y; } };
int main() {
    A *tA[2] = { new A(), new B() };
    delete tA[0]; // type A, ~A() appelé, ok
    delete tA[1]; // type B, ~A() appelé, memory leak
}
```

**Solution :** tous les destructeurs d'une hiérarchie de classes polymorphiques doivent être déclarés virtuels afin d'être sûr que le destructeur adéquat sera appelé lors de la destruction de l'objet sous l'une de ses formes polymorphiques quelconques.

**Notes :** la notion de constructeur virtuel n'a aucun sens (réfléchir !).

Par défaut, une méthode n'est capable de déterminer (=dispatch) le type que d'un seul objet (= l'objet courant).

**Exemple :** dispatch multiple

```
// ajouter les forward declarations ici.
class AlgObj { public:
    virtual AlgObj operator*(AlgObj&)=0;
    virtual AlgObj multiply(Matrix*)=0;
    virtual AlgObj multiply(Vector*)=0;
    virtual ~AlgObj() {}; };
class Matrix : public AlgObj { public:
    AlgObj operator*(AlgObj &x) { return x.multiply(*this); };
    AlgObj multiply(Matrix&) { return /* Matrix*Matrix */; };
    AlgObj multiply(Vector&) { return /* Vector*Matrix */; }; };
class Vector : public AlgObj { public:
    AlgObj operator*(AlgObj &x) { return x.multiply(*this); };
    AlgObj multiply(Matrix&) { return /* Matrix*Scalar */; };
    AlgObj multiply(Vector&) { return /* Vector*Scalar */; }; };
```

Le dispatch lors d'un appel `Left*Right` (deux `AlgObj`) a lieu en deux étapes :

- l'appel à `operator*` identifie par l'appel à la méthode virtuelle le type de l'objet `Left` (le type de `this` est le type réel de l'objet).
- la fonction `operator*` exécute ensuite l'appel à la fonction virtuelle `Right.multiply`, qui permet à son tour d'identifier le type de `Right`.



## 4.5 Contrôle de surcharge

### a) Override

Le mot-clé `override` permet d'indiquer qu'une méthode dans une classe fille est la redéfinition d'une méthode de la classe mère.

**Exemple :**

```
class Base {
public: virtual void mf1() const;
       virtual void mf2(int x);
       virtual void mf3() &;
       void mf4() const;
};

class Derived : public Base {
public:
    // ok, redéfinition d'une méthode de la classe mère
    virtual void mf1() const override;
    // ok, mais mf2(int) est issu de la classe mère
    virtual void mf2(unsigned int x);
    // erreur, ce n'est pas la redéfinition
    virtual void mf3() && override;
    // ok, redéfinition non signalée comme telle
    void mf4() const;
};
```

En conséquence, la bonne méthode devrait être la suivante :

toute méthode surchargée dans une classe  
fille doit avoir le mot-clé `override`.

En effet,

- cela assure une vérification que la signature de la méthode est bien présente dans la classe mère,  
en cas de modification de la signature de la fonction, produit une erreur permettant de s'apercevoir qu'il s'agit d'une surcharge et non d'une redéfinition.
- cela permet, à la lecture de la classe fille, de savoir quelles sont les méthodes redéfinies par la classe fille,

### b) Restriction de redéfinition

Il est aussi possible de limiter les redéfinitions :

- d'une méthode **virtuelle** en plaçant le mot-clé `final` sur les méthodes qui ne doivent pas être redéfinies,
- d'une classe en plaçant le mot-clé `final` derrière le nom de classe sur les classes afin d'indiquer qu'on ne peut hériter de cette classe.

**Exemple :**

```

struct A {
    // A::foo est final
    virtual void foo() final;
    // Erreur: une fonction non-virtuelle ne peut être final
    void bar() final;
};

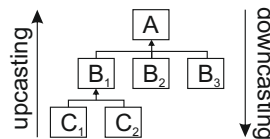
struct B final : A { // struct B est final
    // Erreur: foo surchargée car final dans A
    void foo();
};

// Erreur: B est final
struct C : B {};

```

**4.6 Downcasting**

Le downcasting permet de convertir un objet polymorphique d'une classe de base vers une classe dérivée.



Ce mécanisme est potentiellement dangereux :

- un downcasting n'a de sens que si l'objet transformé contient de manière sous-jacente le type spécialisé vers lequel il est converti.
- sinon, la conversion doit mener à un échec.

**Exemple :** downcasting dans la hiérarchie ci-dessus

- type sous-jacent  $C_2$ , type courant A, downcasting légal :  $B_1$  et  $C_2$ .
- type sous-jacent  $B_2$ , type courant A, downcasting illégal :  $B_1$ ,  $B_3$ ,  $C_1$ ,  $C_2$ .

Un downcasting s'effectue avec la fonction template `dynamic_cast`.

- n'a d'intérêt que sur les objets réellement polymorphiques, car `dynamic_cast` utilise le VPTR pour identifier le type courant et donc les types vers lesquels la conversion est possible.
- si la conversion échoue, la fonction renvoie un pointeur null de ce type.

**Exemple :** utilisation `dynamic_cast`

```

class A { ... };
class B : public A { ... };
class C : public B { ... };
int main() {
    A *a = new B();
    B *b = dynamic_cast<B*>(a); // succès: b valide
    C *c = dynamic_cast<C*>(a); // échec: c = nullptr
}

```

**Notes :**

- la conversion s'effectue à l'exécution, en conséquence ce type de conversion a un surcout (overhead).
- attention, la hiérarchie de classe peut avoir pour conséquence que l'adresse obtenue par le `dynamic_cast` soit différente de l'adresse du pointeur à convertir.
- si la conversion est sûre (et le pointeur converti a la même adresse), utiliser plutôt `static_cast`.
- `dynamic_cast` utilise le RTTI (RunTime Type Identification).

## 4.7 RTTI

### a) Définition

**RTTI** est l'acronyme de RunTime Type Identification.

Dans un code C++, c'est la possibilité, à partir d'un pointeur, de déterminer le type sous-jacent d'un **objet polymorphe** pointé.

**Rappel :** un objet polymorphe possède nécessairement au moins une méthode virtuelle (= son destructeur).

Cette possibilité est offerte à travers deux fonctions :

- `dynamic_cast` qui permet de vérifier si une conversion de pointeur est valide lors d'un downcasting (*i.e.* renvoie un pointeur si valide et `nullptr` sinon).  
permet d'obtenir un type compatible avec le type réel de l'objet, et non le type réel de l'objet (sauf si le downcasting est vers la classe la plus dérivée).
- `typeid(T)` qui renvoie une référence vers un objet constant contenant des informations sur le type et permettant d'effectuer des opérations sur ce type T.  
à savoir obtenir un descripteur caractéristique de la classe ou de comparer des types.

`typeid` :

- l'expression renvoyée par `typeid` est une lvalue qui fait référence à un stockage statique de type `const std::type_info`
- cet opérateur fonctionne sur tous les types, y compris les types non polymorphiques, pour lequel il ne donne que le type immédiat.  
peut avoir un intérêt dans le cadre d'un template.

`const std::type_info` expose essentiellement 3 méthodes :

- les opérateurs de comparaison `==` et `!=` qui permettent de comparer deux `type_info`
- `size_t hash_code()` qui retourne un code unique associé au type (permet une comparaison plus rapide des types).
- `const char* name()` qui retourne un chaîne de caractères contenant le nom du type (ce nom peut être explicite ou décoré).

**Notes :**

- inclure l'entête `<typeinfo>`.
- **attention :** ne jamais comparer l'adresse des `type_info` pour vérifier si deux types sont égaux (pas de garantie que toutes les instances d'un type feront référence à la même structure `type_info`).

**Exemple :**

```
#include <typeinfo>
class A { ... };
const std::type_info& ti1 = typeid(A);
const std::type_info& ti2 = typeid(A);
// affichage: noms garantis identiques
cout << "Type=" << ti1.name() << " " << ti2.name() << endl;
// comparaison direct du type (pas de garantie)
assert(&ti1 == &ti2); // possiblement faux
// comparaison à travers le hash-code
assert(ti1.hash_code() == ti2.hash_code()); // ok
```

### Remarques :

- les qualificateurs `const` et `volatile` sont ignorés par `typeid` (i.e. `typeid(T)==typeid(const T)`),
- l'appel de `typeid` dans le constructeur (resp. destructeur) fait référence à la classe en cours de construction (reps. destruction).  
voir plus sur l'implémentation de `typeid` + se souvenir que le VPTR est constamment réinitialisé

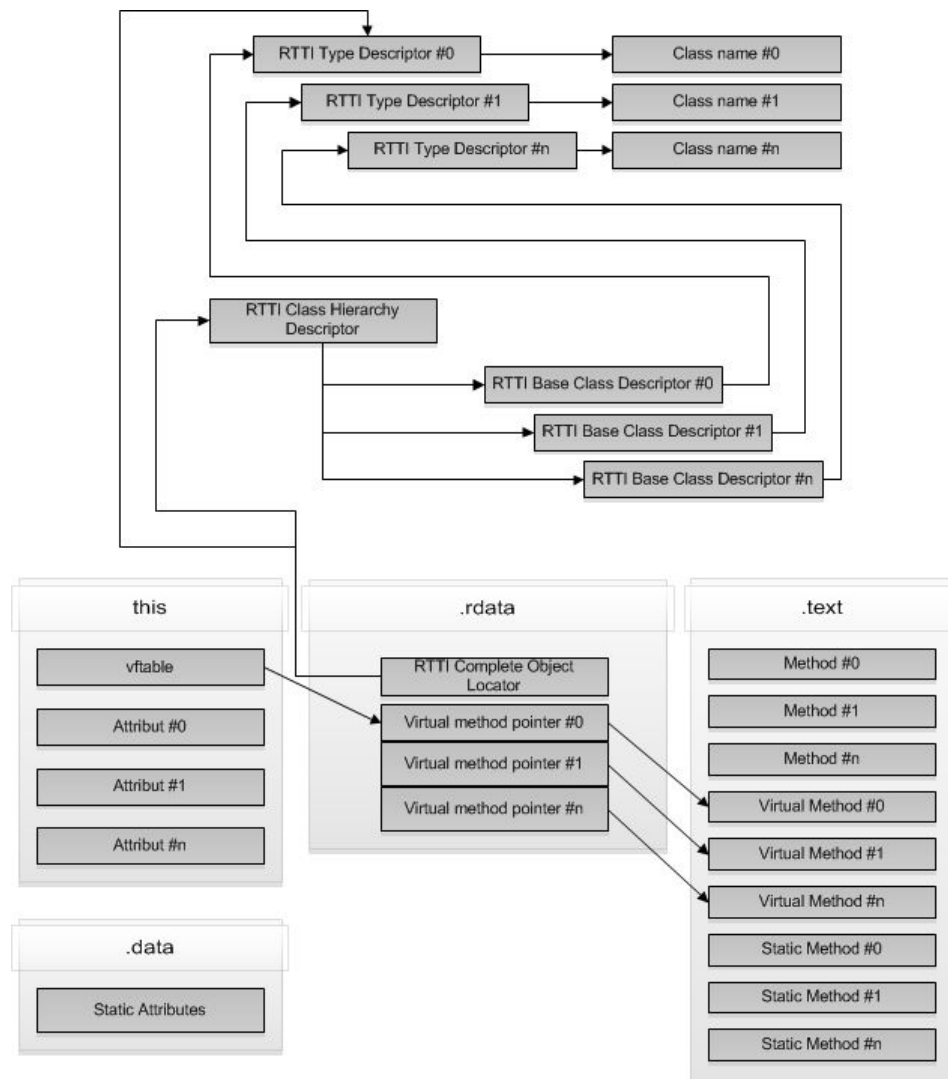
### b) Implémentation

**Implémentation de `typeid`** : sur un objet polymorphe, le compilateur insère dans la VTABLE une structure (RTTI Complete Object Locator) constituée de deux parties :

- un descripteur de la hiérarchie associée à cette classe (= arborescence où il est fait référence à chaque classe à travers un pointeur vers son descripteur de classe)
- un pointeur vers la liste des descripteurs de classes (= `type_info`) contenues dans la hiérarchie, dont le premier élément est la descripteur de la classe courante.

### Fonctionnement du RTTI :

- l'appel à `typeid(A)` sur l'instance d'un objet A polymorphe, utilise son VPRT pour accéder à la VTABLE associée au type réel de l'objet. Puis, grâce au RTTI COL, retourne le descripteur de classe associé.
- **downcasting** : l'appel à `dynamic_cast<A*>(B*)` sur le pointeur d'un objet C polymorphe, utilise son VPRT pour accéder à la VTABLE associée au type réel de l'objet. Puis, grâce au RTTI COL, descend la hiérarchie afin de déterminer si le type A en fait partie.
- **upcasting** : pas besoin de RTTI, simple cast à la compilation.



En terme de performance :

- l'augmentation de la taille de la VTABLE est négligeable au regard du coût associé à l'existence de celle-ci.
- l'appel à typeid est rapide (typiquement quelques indirections),
- un downcasting est lent (typiquement 20 fois plus lent qu'un upcasting), indépendamment du nombre de niveaux à descendre dans la hiérarchie,
- un cross-casting est très lent (2 à 3 fois qu'un downcasting).  
cross-casting = conversion en un type situé dans une autre branche de la hiérarchie : nécessite possiblement plusieurs parcours de la hiérarchie.

#### Remarques :

- l'implémentation du RTTI dépend du compilateur : la structure utilisée peut varier en fonction de celui-ci,
- les performances mesurées ci-dessus (IOS/IEC TR 18015 :2006) ont pu être améliorées.

**Conséquence :** à éviter dans les parties les plus actives du code.

## 5 Polymorphisme

Au total, il y a 4 type de polymorphisme en C++ :

- le **polymorphisme par sous-typage** (polymorphisme à l'exécution)  
polymorphisme en utilisant le RTTI (la classe à utiliser est déterminée à l'exécution en utilisant l'identification de type).
- le **polymorphisme polymorphique** (polymorphisme à la compilation)  
polymorphisme utilisant les modèles de classe (la classe à utiliser est remplacée dans le modèle de classe et compilée).
- le **polymorphisme ad-hoc** (surcharge)  
polymorphisme utilisant les surcharges de fonction (plusieurs fonctions avec le même nom peuvent être définies avec des types de paramètre différents, rendant la fonction polymorphique et celle à appeler devant être déterminée à la compilation à partir du type de ses paramètres).
- le **polymorphisme par coercition** (casting)  
polymorphisme utilisant les constructeurs ou les opérateurs de conversion de classe qui permettent automatiquement de construire automatiquement un objet d'un type à partir d'un autre type (desactivé par le mot clé `explicit`).  
**Attention** : cette approche peut amener à du slicing.

Les trois derniers types sont des facilités d'écriture : elles permettent l'écriture sous une forme polymorphique d'objets qui à la base ne le sont pas (puisque les types à utiliser et les fonctions à appeler sont déterminés à la compilation).

Sur le fond,

- un objet concret possède un type A unique (tout objet est typé),
- l'objet est rendu polymorphe par la présence du VPTR et par le fait que l'objet peut être interprété comme étant de tout type dont il hérite,
- si l'objet est converti en un type B (tel que A hérite de B), par conversion directe ou transmission par copie, alors le VPTR est perdu, et l'objet découpé à sa partie B,
- si on place une référence ou un pointeur de type B sur un objet de type A, le VPTR est conservé.  
**inconvénient de la référence** : elle doit faire référence à un objet concret.

En conséquence, la meilleure façon de stocker un objet polymorphe :

- pour une variable, un pointeur permet de garder l'objet vers lequel il pointe quelque soit son type,
- pour un membre d'une structure, un pointeur permet de pointer vers un objet de tout type compatible.

**Exemple:**

```
struct A { virtual int v(); };
struct B : A { int v(); };

void fun1(A a) { a.v(); }
void fun2(A &a) { a.v(); }
void fun3(A *a) { a->v(); }

void test() {
    B b;
    fun1(b); // appelle A::v() dans fun1
    fun2(b); // appelle B::v() dans fun2
    fun3(&b); // appelle B::v() dans fun3
}

// un pointeur sur un A peut pointer sur un
// objet concret de type A ou B.
struct C { A *c; };
```

## Conclusion

Nous avons vu dans ce chapitre différents mécanismes spécifiques de la programmation orientée objet :

- **L'héritage simple** permet de construire une classe comme spécialisation d'une classe de base connue. La spécialisation permet d'ajouter des champs, des méthodes ou de spécialiser des méthodes existantes.
- **L'héritage multiple** permet de construire une classe comme l'assemblage de plusieurs classes, ce qui permet d'assembler une classe spécialisée à partir de plusieurs autres.
- Le **polymorphisme** consiste à considérer qu'un objet issu d'une classe spécialisée comme pouvant être utilisé à la place d'un objet d'une de ses classes de base, ce qui est logique puisqu'il en possède en partie les caractéristiques.  
Le polymorphisme induit qu'un pointeur sur un objet d'une certaine classe peut en fait pointer sur une spécialisation de cette classe.
- La **virtualité** permet, lorsque l'on possède un pointeur d'un certain type sur un objet, d'exécuter, pour une méthode, celle correspondant au type sous-jacent de l'objet pointé, et non celle du type pointé.

La façon dont les champs sont stockés en pratique dans une classe composite, et la manière dont la virtualité fonctionne sont également décrits. Il est important de comprendre ces mécanismes, car ils permettent de comprendre ce que ces concepts impliquent en pratique.

