



2^e année – Spécialité Informatique

Année 2024 – 2025

Programmation orientée objet en C++

Notes de cours

Sébastien Fourey

Chapitre 2. Programmation orientée objet en langage C++

Version du 3 septembre 2024



Ce travail est publié sous licence Creative Commons
Paternité – Pas d'utilisation commerciale – Pas de modification



Chapitre 2

Fig. 27. — L'eau de chaux se trouble.
C'est que Pierre, en y soufflant de l'acide carbonique, a formé du carbonate de chaux.

Programmation orientée objet en langage C++

2.1 Notion d'objet

Une classe est un type qui regroupe au sein d'une même *entité* :

- des données (à l'instar des structures du C) ;
- des traitements, sous la forme de fonctions particulières appelées *méthodes* ou *fonctions membres*.

Les données et méthodes sont les *membres* de la classe. Un *objet* est une variable d'une certaine classe. On dit aussi *instance de classe*.

2.1.1 Encapsulation

Les méthodes constituent l'ensemble des services offerts par la classe. Ces services utilisent et éventuellement agissent sur les données de la classe sans que l'utilisateur ne connaisse nécessairement la façon exacte dont les données sont manipulées. Idéalement, l'utilisateur de la classe ne peut accéder *directement* aux données membres.

Exemple

La figure 2.1 montre deux versions possibles d'une classe représentant des nombres complexes. Dans ces deux classes, les données ne sont pas manipulées de la même manière mais la liste des méthodes est la même. Du point de vue de l'utilisateur, ces deux classes présentent des *interfaces* similaires, même si leur fonctionnement interne est différent.

Illustration

La notion d'encapsulation peut être illustrée par le dessin de la figure 2.2 qui montre que l'utilisateur ne peut modifier l'état interne de l'objet qu'en passant par l'appel d'une des

Complex	ComplexBis
<ul style="list-style-type: none"> - re: double - im: double 	<ul style="list-style-type: none"> - modulus: double - argument: double
<ul style="list-style-type: none"> + argument(): double + modulus(): double + getRe(): double + getIm(): double + normalize() 	<ul style="list-style-type: none"> + argument(): double + modulus(): double + getRe(): double + getIm(): double + normalize()

FIGURE 2.1 – Deux versions de la classe des nombres complexes.

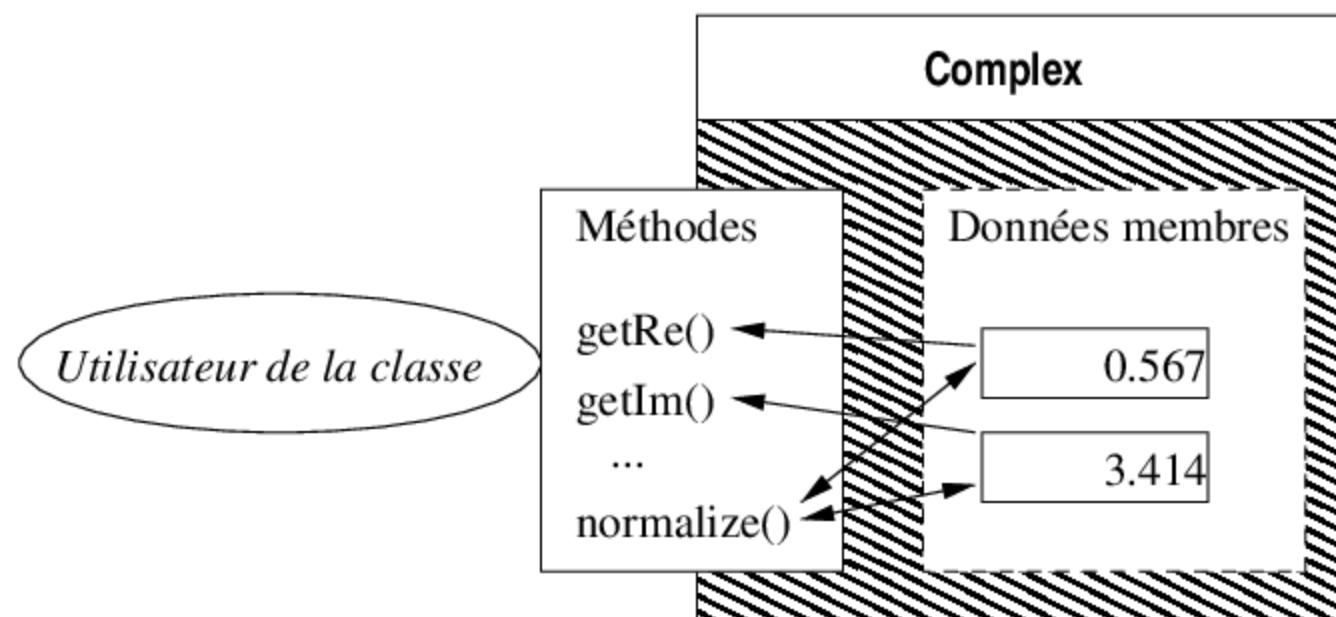


FIGURE 2.2 – Encapsulation

méthodes (Laquelle ?). C'est un moyen de garantir la cohérence de cet état, si on suppose que le concepteur de la classe a bien programmé toutes les méthodes.

2.1.2 Éléments de syntaxe

Déclaration

Le listing 2.1 donne un exemple de déclaration d'un type **Complex** rudimentaire.

```

1 // File: Complex.h
2 class Complex {
3 private:
4     double re, im;
5
6 public:
7     double getRe() const; // Real part
8     double getIm() const; // Imaginary part
9     void normalize();
10    double modulus() const;
11    double argument() const;
12 };

```

Listing 2.1 – Déclaration d'une classe pour les nombres complexes.

Contrôle d'accès

Les mots-clés **public**, **protected** et **private** précisent les autorisations d'accès aux membres d'une classe (et de ses dérivées). Leurs significations sont les suivantes :

- **public**: membres accessibles depuis n'importe où, et entre autre depuis une fonction classique avec la même syntaxe que pour les champs d'une structure ('.' ou '->').
- **private**: membres accessibles depuis une fonction membre, depuis une *fonction amie* de la classe, ou depuis une fonction membre d'une *classe amie*. [...]
- **protected**: restrictions d'accès similaires à **private**: mais les membres sont accessibles en plus depuis les méthodes et fonctions amies des classes dérivées. Attention : contrairement au langage Java pour lequel le modificateur **protected**: apporte une visibilité de *package*, en C++ il n'est pas question de visibilité d'espace de nom.

Concernant cette syntaxe propre aux classes et structures, on notera que

- chaque mot-clé affecte *les* déclarations qui le suivent ;
 - sans précision, tout est privé dans une classe alors que tout est public dans une structure.
- Le premier **private**: du listing 2.1 est donc optionnel. (Question d'examen récurrente mais révolue.)

Définition des méthodes

```

1 // File: Complex.h
2 class Complex {
3 private:
4     double re, im;
5
6 public:
7     double getRe() const {
8         return re;
9     }
10    inline double getIm() const;
11    double modulus() const;
12 };
13
14 double Complex::getIm() const {
15     return im;
16 }
```

Listing 2.2 – Déclaration de la classe **Complex** et définition de ses méthodes « inline ».

```

1 // File: Complex.cpp
2 #include <cmath>
3 #include "Complex.h"
4
5 double Complex::modulus() const {
6     return std::sqrt(re * re + im * im);
7 }
```

Listing 2.3 – Définition des méthodes non-inline de la classe **Complex**.

Une méthode peut être définie dans le corps de la déclaration de classe, ou bien en dehors mais dans le fichier d'en-tête si elle a été déclarée `inline`; enfin et plus classiquement dans un fichier source séparé. On se souviendra que :

- Une définition de méthode dans une déclaration de classe revient à déclarer la méthode `inline`. (Cas de `getRe()` dans le listing 2.2.)
- Une méthode déclarée `inline` doit être définie dans le fichier d'en-tête (`.h`).

Instanciation

Instancier une classe signifie déclarer ou allouer un objet de la classe en question. La syntaxe est similaire à n'importe quel type de base pour la définition de variables statiques ou pour une allocation dynamique (sur la pile ou sur le tas) :

```
IDClasse id_objet;
new IDClasse;
new[] IDClasse;
```

A noter (programmeurs Java), qu'il n'y a pas de parenthèses dans la syntaxe d'allocation dynamique qui utilise le constructeur par défaut (c.-à-d. sans arguments). On pourra écrire par exemple :

```
1 Complex z;
2 Complex * pz = new Complex;
```

Listing 2.4 – Allocations explicites d'objets.

Appel des méthodes

L'appel d'une méthode s'effectue à l'aide de l'opérateur *point* (`.`) ou *flèche* (`->`) selon le modèle suivant :

```
objet.méthode( arguments ... );
pointeur->méthode( arguments ... );
```

Par exemple,

```
1 double x;
2 Complex z;
3 Complex * pz = new Complex;
4 x = z.getRe(); // Correct, but what is the value of x?
5 x = pz->getRe(); // Again.
```

Listing 2.5 – Appels de méthodes.

Remarque 2.1

- Chaque instance d'une classe possède ses propres données.
- Les méthodes sont d'une certaine façon « partagées » par l'ensemble des objets d'une même classe.

- Une donnée membre peut être déclarée statique (mot-clé `static`), comme montré en exemple dans le listing 2.6. Elle est alors partagée par toutes les instances de la classe.
- Une fonction membre peut aussi être déclarée statique. Elle pourra alors être appelée sans faire référence à un objet de la classe mais en utilisant à la place le nom de la classe et l'opérateur « `::` ». Bien entendu, une telle méthode n'a accès qu'aux données membres statiques de la classe et le mot-clé `this` (§ 2.4) n'y est pas non plus défini. La syntaxe de l'appel d'un méthode statique est la suivante :

IdClasse::méthode(arguments ...);

Une donnée membre statique doit être définie une fois et une seule dans un programme au moment de l'édition de liens. Cette définition doit être *globale*, c.-à-d. en dehors de tout bloc (Listing 2.6). L'initialisation d'une donnée membre statique se fait lors de sa définition (ligne 8 du listing 2.6). Dans un cas réel avec compilation séparée, la variable statique est déclarée dans la classe dans le fichier `.h`; elle est définie (et éventuellement initialisée) dans le fichier `.cpp` de cette même classe. De cette façon, la définition est propre à une seule unité de compilation.

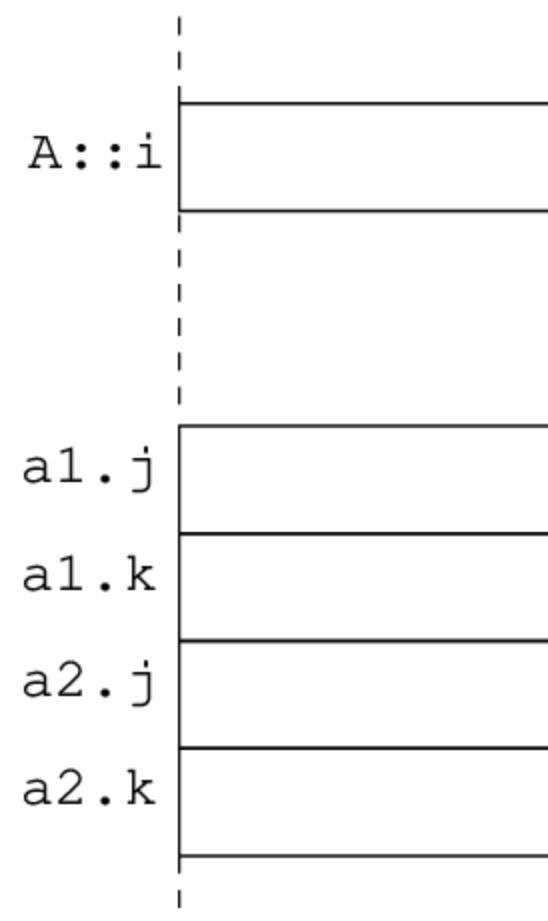
Les données membres statiques *constantes* de types *integrals* (char, bool, int) et flottants (float, double) font exception à cette règle : elles peuvent être initialisées, donc définies, dans le corps de la définition de classe.

Exercice 2.1 Écrivez un programme utilisant la compilation séparée, dans lequel un fichier définira de manière convenable une classe munie d'une donnée membre statique. Prenez garde aux définitions multiples !

```

1 | class A {
2 |     static int i; // Decl.
3 |     int j, k;
4 |
5 | public:
6 |     // ...
7 | };
8 | int A::i = 8; // Def. & init.
9 | A a1, a2;
```

Listing 2.6 – Déclaration, définition et initialisation d'une donnée membre statique.



Contrôle de l'accès aux méthodes

Il suit les mêmes règles et la même syntaxe que pour les données membres. Le listing 2.7 en donne un exemple.

2.1.3 Constructeurs

Jusqu'ici, nous n'avons pas vu comment les objets peuvent être initialisés, comme il est possible de le faire au moment de la déclaration de n'importe quelle variable. On aimerait effectivement

```
1 class Engine {
2     void changeOil();
3 public:
4     void start();
5 };
6
7 class Tank {
8     float level;
9 public:
10    void fill();
11 };
12
13 class Car {
14     bool engineIsRunning;
15 public:
16     void start();
17     Engine & getEngine() {
18         return engine;
19     }
20     Tank tank;
21 private:
22     Engine engine;
23 };
24
25 int main(int , char *[])
26 {
27     Car car;
28     car.start(); // Correct
29     car.tank.level = 1.0; // Error
30     car.tank.fill(); // Correct
31     car.engine.start(); // Error
32     car.getEngine().start(); // Allowed
33     car.getEngine().changeOil(); // Error
34 }
```

Listing 2.7 – Exemple d'utilisation du contrôle d'accès aux méthodes.

pouvoir écrire des déclarations semblables à celles du listing 2.8. Si dans ce cas on peut aisément donner un sens à l'initialisation d'un complexe par un réel à l'aide de l'opérateur `=`, une syntaxe doit aussi être possible pour des initialisations plus élaborées¹.

```
1 | Complex z = 1.0;
2 | Complex y = 0;
3 | Complex x = z;
```

Listing 2.8 – Déclarations de complexes avec initialisations.

Présentation

Les constructeurs sont des méthodes particulières qui portent le même nom que la classe. L'un d'eux est appelé lors de l'instanciation (c.-à-d. déclaration ou allocation d'un objet). Le rôle du constructeur qui sera utilisé au moment de l'instanciation est :

- D'initialiser les données membres ;
- De déclencher des actions voulues à chaque création d'un objet de la classe. (Par exemple, il pourra effectuer les éventuelles allocations dynamiques sur le tas.)

Il ont quelques caractéristiques qui les distinguent des méthodes ordinaires :

- Un constructeur n'a pas de valeur de retour (pas même `void`) ;
- Il est *généralement* public (contre-exemple ?) ;
- Dans le corps d'un constructeur, la résolution des appels de méthodes virtuelles est *ad hoc*. (Des questions ?)

Par contre, comme toute méthode :

- Un constructeur peut être surchargé ;
- Il peut avoir des valeurs d'arguments par défaut.

On appelle *constructeur par défaut* le seul constructeur qui puisse être utilisé sans arguments. C'est ce constructeur qui est appelé quand aucune précision n'est donnée lors de l'instanciation. C'est aussi le seul qui puisse être appelé lors de l'allocation d'un tableau d'objets avec l'opérateur `new[] . . .`

De plus, tout constructeur acceptant un seul paramètre autorise à la fois la syntaxe d'initialisation avec l'opérateur `=`

```
|| Complex z = 0.0;
```

et la syntaxe *fonctionnelle*

```
|| Complex z( 0.0 );
```

Les deux lignes précédentes sont équivalentes, à moins que le constructeur correspondant n'ait été déclaré « `explicit` » (cf. plus loin). La syntaxe fonctionnelle permet d'utiliser un constructeur particulier lorsque plusieurs sont définis, en suivant les règles habituelles de résolution de la surcharge.

Pour finir, un exemple de classe munie de trois constructeurs est donné dans le listing 2.9.



Si (au moins) un constructeur avec arguments est défini, le compilateur ne fournit pas de constructeur par défaut. Il faut alors le définir si on veut pouvoir l'utiliser !

1. Plus « complexes » eût été maladroit.

```
1 class Complex {
2
3 public:
4     Complex() {
5         re = im = 0.0;
6     }
7
8     Complex(double r) {
9         re = r;
10    im = 0.0;
11 }
12
13    Complex(double re, double im) {
14        Complex::re = re;
15        Complex::im = im;
16    }
17
18 private:
19     double re, im;
20 };
21
22 int main() {
23     Complex z;           // Complex()
24     Complex o(3.14, 2.5); // Complex(double, double)
25     Complex r(3);        // Complex(double)
26     Complex * pz;
27
28     r = o;               // ?
29     pz = new Complex(.4); // Complex(double)
30     return 0;
31 }
```

Listing 2.9 – Exemple de classe munie de plusieurs constructeurs.

Conversion implicite

Un constructeur d'une classe C ayant un unique argument de type T (type de base ou bien défini par l'utilisateur) définit une conversion implicite du type T vers le type C .

Cette conversion s'applique donc lors d'une affectation ou bien un appel de fonction. [...]

Constructeur explicite

Soient les déclarations suivantes faisant appel au constructeur de la classe `String` ayant comme argument un entier (la taille de la chaîne) :

```
1 String s(10);      // Constructor with int parameter
2 String s = 10;     // <=> String s(10);
3 String s = 'z';    // <=> String s('z'); <=> String s(122);
```

La syntaxe utilisée à la ligne 3 pose un problème de lisibilité. De même que la conversion implicite n'est pas souhaitable dans l'exemple suivant (ligne 6).

```
1 void foo(String str) {
2     // ...
3 }
4
5 int main() {
6     foo(50); // Correct!
7 }
```

La solution dans ce cas consiste à utiliser le mot-clé `explicit` comme illustré par le listing 2.10.

```
1 class String {
2     // ...
3 public:
4     String() {
5         // ...
6     };
7     explicit String(int size) {
8         // ...
9     }
10    // ...
11};
12
13 void foo(String) {
14     // ...
15}
16
17 int main() {
18     String s1(10);           // OK
19     s1 = String(10);        // OK
20     s1 = static_cast<String>(10); // OK
21     String s2 = 10; // Error
22     foo(20); // Error
```

23 }

Listing 2.10 – Constructeur explicite.

Comme le montre cet exemple, un constructeur explicite ne définit donc plus de conversion implicite (ligne 22) mais autorise la conversion explicite (lignes 19 & 20).

Nouvelle forme d'instanciation

La possibilité d'initialiser explicitement ou par défaut un objet lors de sa création permet aussi de construire avec une syntaxe concise des objets temporaires et anonymes. La syntaxe utilisée pour cela est présentée dans le listing 2.11.

```
1 double x = 0.0;
2 x = modulus(Complex(0, 1)); // Value of x?
```

Listing 2.11 – Création d'un objet temporaire et anonyme.

La durée de vie de ce type d'objet est, sauf exception concernant les références constantes, celle de *l'instruction* dans laquelle il est créé.

Exercice 2.2 Une fonction peut-elle retourner une référence à un objet temporaire ?

Remarque 2.2 Les initialisations faites dans le 3^e constructeur des complexes (cf. listing 2.9, ligne 13) montrent l'utilisation de l'opérateur de résolution de portée pour lever le masquage des données membres *re* et *im* par les paramètres de mêmes noms du constructeur.

Cependant, une autre syntaxe est préférable pour initialiser dans le constructeur les données membres ; du moins lorsque l'initialisation se fait à l'aide d'une expression simple (p. ex. une constante ou toute expression dépendant de variables globales). Ainsi, on utilise dans le listing 2.12 une liste d'initialiseurs de données membres. Cette syntaxe est d'autant plus recommandée si les données membres à initialiser sont elles-mêmes des instances de classes. En effet, elle permet dans ce cas de court-circuiter l'appel (sinon automatique) des constructeurs par défaut de ces données membres, ceci en appelant explicitement des constructeurs avec paramètre(s).

```
1 class Complex {
2     public:
3         Complex() : re(0.0), im(0.0) {
4             }
5         Complex(double r) : re(r), im(0.0) {
6             }
7         Complex(double re, double im) : re(re), im(im) {
8             }
9
10    private:
11        double re, im;
12    };
```

Listing 2.12 – Utilisation d'une liste d'initialisation de données membres.

```

1 // File: BankAccount.h
2 class BankAccount {
3 public:
4     BankAccount(int number);
5     BankAccount(int number, float balance);
6
7 private:
8     int number;
9     float balance;
10 };
11
12 // File: BankAccount.cpp
13 BankAccount::BankAccount(int number)
14     : BankAccount(number, 0.0f) {
15 }

```

Listing 2.13 – Délégation de constructeur.

C++11 : Délégation de constructeur

Avant la norme de 2011, il n'était pas possible d'appeler un constructeur d'une classe depuis un autre constructeur de cette même classe. C'est contraignant car cela oblige parfois à définir une méthode pour factoriser le code commun à plusieurs constructeurs. En C++11, le code du listing 2.13 est correct. Cette syntaxe similaire à celle des listes d'initialiseurs de données membres (utilisant les « `:` ») est la seule possible.

On remarque que cette syntaxe offre une alternative aux arguments par défaut, puisqu'il est possible de définir un constructeur qui se contente d'en appeler un autre qui possède plus de paramètres. Il y a toutefois une différence : la valeur effective des paramètres fait partie de la *définition* du constructeur qui délègue ; alors que dans le cas des arguments par défaut c'est la consultation du fichier d'en-tête à chaque utilisation dans une unité de compilation qui détermine la valeur de l'argument utilisé. Ainsi, la modification de la valeur par défaut donnée dans un fichier d'en-tête doit être suivie de la recompilation de toutes les unités de compilation qui utilisent le constructeur concerné. Avec la délégation de constructeur, seule la classe (ou structure) dont le constructeur est modifié doit être recompilée !

Restrictions Cette syntaxe n'autorise pas d'autres initialisations ni l'appel de constructeurs supplémentaires (p. ex. celui d'une classe de base). Ainsi, le code du listing 2.14 est erroné car l'initialisation de l'attribut `balance` à la ligne 14 n'est pas possible. En effet, l'objet est considéré *construit* dès lors qu'un de ses constructeurs s'est exécuté (autrement dit, le premier de ceux qui sont appelés). De ce fait, l'initialisation « anticipée » d'attributs, ou d'une classe de base (via une liste d'initialiseurs de membres), perd son sens quand l'objet est considéré construit.

C++11 : Initialisations des données membres

Il est possible en C++11 de spécifier une valeur par défaut pour une donnée membre de classe directement dans la déclaration de la classe. Un exemple est donné dans le listing 2.15. En fait,

```

1 // File: BankAccount.h
2 class BankAccount {
3 public:
4     BankAccount(int number);
5     BankAccount(int number, float balance);
6
7 private:
8     int number;
9     float balance;
10 }
11
12 // File: BankAccount.cpp
13 BankAccount::BankAccount(int number, float balance)
14     : BankAccount(number), balance(balance) // Error!
15 {
16 }
```

Listing 2.14 – Délégation de constructeur erronée.

la syntaxe d’initialisation uniforme (section 7.1) peut aussi être utilisée ici.

L’initialisation par défaut qui est spécifiée ainsi peut être court-circuitée dans n’importe quel constructeur via la liste d’initialiseurs de membres (présentée dans le listing 2.12), et seulement grâce à celle-ci. Bien entendu, il est possible dans le bloc d’un constructeur de modifier la donnée membre, mais elle aura déjà été initialisée.

Exercice 2.3 *À l’aide de deux classes et d’un peu d’affichage, écrivez un code qui vous permette de vérifier que l’initialisation par défaut fonctionne bien, mais aussi qu’elle peut être court-circuitée via la liste d’initialiseurs de membres.*

C++11 : new et initialisation

En C++11, il est possible de combiner la syntaxe généralisée d’initialisation avec l’opérateur `new` lors de l’allocation d’un tableau d’éléments. Ceci permet d’appeler un constructeur spécifique pour les premiers éléments du tableau alloué, voire pour la totalité. Ceci est illustré par le code du listing 2.16.

2.1.4 Destructeur

Motivation Si un constructeur alloue de la mémoire sur le tas, comme c’est le cas dans le listing 2.17, qui se charge de la libérer quand l’objet est détruit (en fin de bloc pour un objet alloué sur la pile, ou en cas de désallocation explicite s’il a été alloué sur le tas)? Attention, il est bien question ici de la mémoire allouée sur le tas par l’objet et non pas de la mémoire occupée par l’ensemble des données membres de l’objet. Il faut bien faire la différence entre une donnée membre de type pointeur et la zone mémoire référencée par ce pointeur. Par défaut, le pointeur sera désalloué en même temps que l’objet qui le contient, pas la zone référencée.

```

1 #include <iostream>
2 #include <string>
3 class Student {
4 public:
5     Student() {
6         std::cout << "A student has been created" << std::endl;
7     }
8
9 private:
10    int number = 0;
11    int birthYear{1900};
12    std::string lastname{"Doe"};
13    std::string firstname{"John"};
14 };

```

Listing 2.15 – Initialisation par défaut de données membres.

```

1 class Complex {
2 public:
3     Complex() {
4         // ...
5     }
6     Complex(double re) {
7         // ...
8     }
9     Complex(double re, double im) {
10        // ...
11    }
12 };
13
14 int main() {
15     int * array = new int[5]{0, 1, 2}; // array = {0, 1, 2, ?, ?}
16     Complex * tz;
17     tz = new Complex[10]{
18         1.0,           // Complex(double)
19         {},            // Complex()
20         {1.0, 1.0},   // Complex(double, double)
21         2.0           // Complex(double)
22     };
23 }

```

Listing 2.16 – Utilisation de new avec initialisations (C++11).

```
1 #include <cstring> // For strlen()
2 #include <iostream>
3
4 class String {
5     char * array;
6
7 public:
8     String() {
9         array = nullptr;
10    }
11    String(const char * s) {
12        if (s) {
13            array = new char[strlen(s) + 1];
14            strcpy(array, s);
15        } else {
16            array = nullptr;
17        }
18    }
19    // ...
20 };
21
22 void foo() {
23     String a_string("C++ rocks");
24     std::cout << a_string << std::endl; // Possible?
25 } // What happens at the end of the block?
```

Listing 2.17 – Exemple de code erroné.

De plus, si on peut souhaiter réaliser des opérations automatiquement lors de chaque création d'un objet, pourquoi n'en serait-il pas autrement lors de sa destruction ? (Exemple d'application : compteur d'instances à l'aide d'une variable statique.)

Présentation Le *destructeur* est une méthode dont le nom est celui de la classe précédé du caractère ~, qui est appelée juste avant la libération de la mémoire occupée par l'objet.

Son rôle est :

- de déclencher des actions voulues à chaque destruction d'un objet de la classe ;
- d'effectuer d'éventuelles désallocations, par exemple de données liées qui auraient été allouées à la construction ou au cours de la vie de l'objet.

Il se distingue des autres méthodes car :

- il n'a pas de valeur de retour ;
- il n'accepte aucun argument (raison suffisante pour qu'il soit unique car un destructeur ne peut par nature pas être déclaré constant) ;
- il est le plus souvent public ;
- il ne peut pas être surchargé.

Le destructeur nécessaire dans la classe **String** est défini dans le listing 2.18.

```

1 class String {
2     char * array;
3
4 public:
5     String() {
6         array = nullptr;
7     }
8     String(const char *) {
9         // ...
10    }
11    ~String(); // Declaration
12 };
13
14 String::~String() { // Definition
15     delete[] array;
16 }
```

Listing 2.18 – Exemple de classe munie d'un destructeur.

Quand le destructeur est-il appelé ?

D'une manière générale, on peut dire que le destructeur d'un objet est appelé automatiquement lorsque cet objet est désalloué, qu'il ait été alloué sur la pile (allocation automatique) ou sur le tas (allocation dynamique). Dans le premier cas, le destructeur est donc appelé en fin de bloc et, dans le second cas, lors de l'utilisation de l'opérateur **delete** (voir listing 2.19).

```

1 int main()
2 {
3     String * firstnames = new String[3];
4     String * lastname = new String;
```

```

1 class String {
2     char * array;
3
4 public:
5     String() {
6         array = nullptr;
7     }
8     String(const char *) {
9         // ...
10    }
11    ~String() {
12        delete [] array;
13    }
14 };
15
16 int main() {
17     String school("ENSICAEN");
18     String s;
19     s = school;
20     return 0; // Segmentation fault! Why?
21 }
```

Listing 2.20 – Exemple de code compilé sans erreur mais néanmoins erroné.

```

5     String address;
6     /* ... */
7     delete [] firstnames; // Call 3 destructors
8     delete lastname; // Call 1 destructor
9 } // <- End of block , call address' destructor
```

Listing 2.19 – Trois appels automatiques du destructeur.



Que se passe-t-il dans le cas de la désallocation d'un tableau d'objets ?

2.1.5 Opérateur d'affectation (=)

Par défaut, l'opérateur d'affectation « = » fonctionne pour les objets comme pour les structures : il recopie les données membre à membre, avec ce même opérateur². C'est parfois suffisant (complexes), mais le listing 2.20 montre qu'il en est souvent autrement. C'est notamment le cas lorsque des membres sont des pointeurs vers des données allouées dynamiquement sur le tas.

Pour résoudre ce problème, il est possible de surcharger l'opérateur d'affectation. Il s'agit là d'une particularité importante et essentielle du C++ qui sera présentée de façon exhaustive dans la section 2.2 (pour les opérateurs en général).

2. Ceci n'est pas anodin car ces données membres peuvent être elles mêmes des instances de classes.

Méthode operator=

Cette méthode peut être déclarée de la manière suivante :

 $T \& T::operator=(const T\&);$

```

1  class String {
2      char * array;
3      // ...
4  public:
5      // ...
6      String & operator=(const String &);
7  };
8
9  String & String::operator=(const String & other) {
10     if (this == &other) {
11         return *this;
12     }
13     if (array) { // Test is optional
14         delete [] array;
15     }
16     if (other.array) {
17         array = new char[strlen(other.array) + 1];
18         strcpy(array, other.array);
19     } else {
20         array = nullptr;
21     }
22     return *this; // Why?
23 }
```

Listing 2.21 – Classe munie d'un opérateur d'affectation.

Un exemple de classe munie de cet opérateur est donné dans le listing 2.21. Cette surcharge de l'opérateur assure par exemple que l'instruction de la ligne 12 du listing 2.20 ne provoquera pas une erreur *à retardement* du fait d'un partage de données non maîtrisé.

Toutefois, l'opérateur d'affectation intervient, comme son nom l'indique, seulement lors d'une instruction d'affectation. On peut alors se demander comment s'effectuent :

- la recopie des valeurs des arguments de fonctions de types objets ;
- l'initialisation lors de la déclaration d'une instance selon la syntaxe ci-dessous.

```
|| String a_string = another_string;
```

En effet, si aucun contrôle sur ces opérations n'est donné au programmeur, le problème du partage de données persiste.

2.1.6 Constructeur par recopie

Ce constructeur particulier répond au besoin de contrôle, sur les opérations d'initialisations et de recopies d'objets, évoqué dans la section précédente.

```

1 class String {
2     // ...
3 public:
4     String() {
5         array = 0;
6     }
7     String(const String & other) {
8         if (other.array) {
9             array = new char[strlen(other.array) + 1];
10            strcpy(array, other.array);
11        } else {
12            array = nullptr;
13        }
14    }
15};

```

Listing 2.22 – Classe `String` munie d'un constructeur par recopie.

De plus, si on considère les déclarations ci-après :

```

1 Complex z(0,1);
2 Complex r = z;

```

Alors, le fonctionnement suivant serait idiot et inefficace pour la déclaration de la ligne 2 :

- Appel du constructeur par défaut de l'objet `r`, puis
- Exécution d'une instruction `r.operator=(z)`.

Constructeur par recopie

Comme tout constructeur il n'a pas de valeur de retour ; et son prototype est de la forme ci-dessous si `T` est l'identifiant d'une classe :

`T::T(const T &);`

Il est mis en jeu :

- Lors d'une initialisation à la déclaration d'une instance : `T objet2 = objet1;`
- Lors du passage d'arguments par valeur ;
- Lors d'un retour de fonction [...].

Exemple

Le listing 2.22 montre l'utilisation classique de ce constructeur, dans la classe `String` introduite précédemment, pour éviter le partage de données.



Le passage de l'argument par référence dans le constructeur par recopie est obligatoire. Pourquoi ?

Par défaut, un constructeur par recopie est automatiquement fourni par le compilateur. Il recopie les données membres, non statiques, qu'il s'agisse de types de base comme d'instances

de classes. Dans le cas de données membres qui sont des instances de classes, c'est le constructeur par recopie de ces classes qui est appelé.

2.1.7 Règle des trois

On a vu dans les sections précédentes que, pour désallouer les données référencées par un objet, il fallait équiper la classe correspondante d'un destructeur (section 2.1.4). Afin d'éviter une partage de données lors des recopies ou affectations, il est aussi nécessaire de définir un constructeur par recopie (section 2.1.6) et de surcharger l'opérateur d'affectation (section 2.1.5). En effet, le comportement de ceux qui sont fournis par défaut par le compilateur ne suffit pas, puisque ces derniers se contentent de recopier les données membres (copie membre à membre). Ceci aboutit à des erreurs comme celle mise en évidence dans le listing 2.20, où plusieurs instances « pointent » sur une même donnée, qu'elles vont chacune essayer de désallouer.

Les trois méthodes dont il est question ici (destructeur, constructeur par recopie et opérateur d'affectation) forment finalement un triplé indivisible. La règle des trois, attribuée à un certain Marshall Cline, stipule simplement que si le programmeur a besoin de définir l'une de ces méthodes, alors il devrait définir les trois.

Attention : cette règle ne s'applique pas, par exemple, à une classe qui ne ferait aucune allocation dynamique. En effet, il paraît difficile dans ce cas d'aboutir à une situation de partage de donnée (que cette règle vise à éviter).

2.1.8 C++11 : *rvalue references* et constructeur par déplacement

Comme cela a été vu dans la section 2.1.6, le constructeur par recopie se charge de copier les données d'un objet dans trois situations bien identifiées. Pourtant, il arrive que cette recopie soit inutile, et donc inefficace, par exemple lorsque l'objet à copier est un objet qui sera détruit juste après avoir été copié. C'est le cas d'un objet temporaire, ou bien encore d'une variable locale renvoyée par une fonction. Dans ces deux cas, il est en effet plus efficace de *déplacer* les données de l'objet amené à disparaître en les attribuant, via une simple affectation de pointeur, à l'objet qui reçoit la copie.

On remarque ainsi que dans le code du listing 2.23, le constructeur par recopie est appelé à la ligne 8 pour copier le contenu de la chaîne `result` dans l'objet temporaire utilisé pour stocker le résultat de l'appel. Ce dernier objet étant lui-même utilisé comme argument de l'opérateur d'affectation. Soit deux copies intégrales de la chaîne créée par la fonction, qui peuvent toutes les deux être évitées.

De même, l'opérateur `+` appliqué à deux objets de type `String` (listing 2.23, ligne 19), produit un objet temporaire anonyme qui est recopié dans une variable locale à la fonction `foo` (son paramètre « `s` »). Là encore, c'est une recopie inutile car l'objet temporaire disparaît dès que la copie s'est terminée. Les données pourraient là encore être simplement déplacées.

À tout ce qui vient d'être évoqué, le C++11 apporte une solution appelée « sémantique de déplacement » (*move semantics*) à l'aide des *rvalue references*. En résumé, ce type de référence permet d'indiquer au compilateur qu'une fonction (constructeur, opérateur d'affectation ou autre) peut être utilisée quand elle a pour argument une *rvalue* (voir section 1.1.9).

```

1 #include "String.h"
2
3 String nospace(const String & str) {
4     String result(str);
5     //
6     // ... replace spaces with '*'
7     //
8     return result;
9 }
10
11 void foo(String s) {
12     // ...
13 }
14
15 int main() {
16     String s("A very, very long string... ");
17     String r;
18     r = nospace(s);
19     foo(String("OK") + String("KO"));
20 }
```

Listing 2.23 – Retour de fonction avec recopies multiples.

Constructeur par déplacement

Il suffit, pour éviter la première copie intégrale de la variable locale `resultat` évoquée plus haut, de définir un *constructeur de déplacement* comme dans le listing 2.24.

Il est important de noter que la variable de type *rvalue* qui est recopiée est modifiée de sorte qu'elle ne possède plus de pointeur (*array*) sur les données. Ainsi, la destruction imminente de cette variable sera rapide et n'aura pas d'effet sur les données qu'elle possédait avant que ces dernières soient transférées à un nouveau propriétaire.

Dans le prototype de ce constructeur, l'opérateur `&&` indique une référence à une *rvalue* (cf. section 1.1.9). Autrement dit, ce constructeur sera utilisé pour assurer la recopie de tout objet dont l'adresse ne peut être manipulée que par le compilateur (et pas par le programmeur). Notez aussi que, contrairement au constructeur par recopie, la référence n'est pas constante. Il s'agit en effet d'un déplacement et donc généralement d'une modification de l'objet dont on va déplacer les données.

En complément de ce constructeur, la surcharge de l'opérateur d'affectation à partir d'une *rvalue reference* permet d'éviter elle aussi son lot de recopies inutiles.

Affectation par déplacement

En l'absence d'optimisation par le compilateur, on peut détailler l'effet de la ligne 18 du listing 2.23 comme suit³ :

3. Avec g++ il faut utiliser l'option `-fno-elide-constructors` pour observer ce comportement.

```

1 class String {
2     char * array;
3     unsigned int length;
4
5 public:
6     String();
7     String(const char *);
8     String(const String &);
9     String(String && other) noexcept {
10         length = other.length;
11         array = other.array;
12         other.length = 0;
13         other.array = nullptr;
14     }
15     // ...
16 };

```

Listing 2.24 – Constructeur par déplacement.

- Appel de `nospace` pour laquelle `str` est une référence à la variable locale `s` de la fonction `main`.
- Réservation, au moment de l'appel de fonction, d'un espace sur la pile pour un objet anonyme qui contiendra le résultat de la fonction `nospace` (appelons le `tmp_ret`).
- Création de la variable locale `result` par recopie de `str`.
- (`return result;`) Copie, par appel du constructeur par recopie, de la variable `result` dans `tmp_ret`.
- Destruction de la variable locale `result`.
- Appel de l'opérateur d'affectation de l'objet `r` à partir de l'objet temporaire `tmp_ret`.

On observe ici que la variable temporaire `tmp_ret` peut être déplacée dans `r` au lieu d'être recopiée. Il s'agit bien d'une *rvalue* qui disparaît de toute façon après l'affectation. Ici, l'opérateur d'affectation par déplacement s'applique, s'il est défini.

Le modèle de fonction `std::move`

Les *rvalue references* ont une particularité : une fois nommées (comme c'est le cas de la variable `other` du constructeur par déplacement de la classe `String` dans le listing 2.24), elles se comportent comme des *lvalues*. En effet, si `other` fait bien référence par exemple à un objet temporaire, il a une durée de vie qui est au moins égale au temps d'exécution de la méthode ! Dans ce contexte, il ne faut donc pas considérer `other` *implicitement* comme temporaire dans les opérations réalisées sur cette variable. Par exemple, la recopie (resp. l'affectation) d'une donnée membre de `other` ne doit pas être faite automatiquement en appelant le constructeur par déplacement (resp. l'opérateur d'affectation par déplacement) du type en question. Une telle opération serait par exemple dangereuse si elle était répétée plusieurs fois. Pour cela, le C++11 introduit le modèle de fonction `std::move`⁴ qui permet d'expliciter la conversion entre une *lvalue* et une *rvalue reference* afin de forcer l'appel du constructeur par déplacement, ou de l'opérateur d'affectation par déplacement, aux endroits précis où le programmeur souhaite

4. Entête `<utility>`

```

1 class String {
2     char * array;
3     unsigned int length;
4
5 public:
6     String();
7     String(const char *);
8     String(const String &);
9     String(String &&);
10    String & operator=(const String &);
11    String & operator=(String && other) {
12        delete [] array;
13        array = other.array;
14        length = other.length;
15        other.array = nullptr;
16        other.length = 0;
17        return *this;
18    }
19    // ...
20};

```

Listing 2.25 – Affectation par déplacement.

qu'ils interviennent.

Ainsi, dans le constructeur par déplacement de la classe `Set` du listing 2.26, l'identifiant `other` est considéré par le compilateur comme une *lvalue*. À la ligne 12, c'est donc le constructeur par recopie (classique) de la classe `std::string` qui est appelé, et non pas le constructeur par déplacement. Afin de forcer le compilateur à optimiser la recopie en utilisant ce dernier, il faut convertir explicitement `other.name` en une *rvalue reference* à l'aide du modèle de fonction `std::move`, comme dans le listing 2.27 (ligne 12).

Finalement, une bonne nouvelle est que l'on ne peut pas provoquer « accidentellement » une construction ou une affectation par déplacement en transformant une *lvalue* en une *rvalue reference*. Cela ne peut se faire que de manière explicite, et c'est heureux. Dans le cas contraire, on risquerait de provoquer des déplacements sans le vouloir.



S'il est possible de convertir un *lvalue* en une *rvalue reference* (via un `static_cast` ou bien à l'aide du modèle de fonction `std::move`), il est la plupart du temps préférable de laisser au compilateur le soin de décider quand appeler un constructeur ou une affectation par déplacement (via le principe de la surcharge). L'exception à cette règle a été expliquée dans les deux précédents paragraphes.

2.1.9 C++11 : Contrôle des méthodes générées par défaut

Le modificateur `default` Il est possible en C++11 de demander explicitement la création par le compilateur d'un constructeur ou d'une méthode autrement créée par défaut (constructeur par recopie, opérateur d'affectation, etc.). Ceci passe par l'utilisation du modificateur `default` comme dans le listing 2.28.

```
1 class Set {
2 public:
3     Set(const std::string & name)
4         : name(name),
5          size(0),
6          values(nullptr) {
7     }
8     Set(const Set & other) {
9         // ...
10    }
11    Set(Set && other) noexcept
12        : name(other.name), // <---- INEFFICIENT
13          size(other.size),
14          values(other.values) {
15     other.values = nullptr;
16     other.size = 0;
17   }
18   void insert(double);
19
20 private:
21     std::string name;
22     int size;
23     double * values;
24 };
```

Listing 2.26 – Utilisation incomplète de la recopie par déplacement.

```

1  class Set {
2  public:
3      Set(const std::string & name)
4          : name(name),
5              size(0),
6              values(nullptr) {
7      }
8      Set(const Set & other) {
9          // ...
10     }
11     Set(Set && other) noexcept
12         : name(std::move(other.name)),
13             size(other.size),
14             values(other.values) {
15         other.values = nullptr;
16         other.size = 0;
17     }
18     void insert(double);
19
20 private:
21     std::string name;
22     int size;
23     double * values;
24 };

```

Listing 2.27 – Utilisation du modèle de fonction `std::move`.

```

1  class A {
2  public:
3      A(int) {
4      }
5  };
6
7  class Empty {
8  public:
9      Empty() = default;
10     Empty(char * a_string) {
11         std::cout << a_string;
12     }
13 };
14
15 int main() {
16     A a;           // ERROR: A::A() does not exist.
17     Empty e; // OK since Empty::Empty() is well-defined.
18 }

```

Listing 2.28 – Spécification d'un constructeur par défaut.

```

1 class String {
2 public:
3     String(const char *) {
4         // ...
5     }
6     String(const String &) = delete;
7     String & operator=(const String &) = delete;
8     ~String() {
9         // ...
10    }
11
12 private:
13     char * data;
14 };
15
16 void foo(String) {
17 }
18
19 void bar(const String &) {
20 }
21
22 int main() {
23     String s("Ensi");
24     foo(s); // ERROR: Copying is not possible
25     bar(s); // OK: does not require a copy
26 }
```

Listing 2.29 – Classe interdisant sa recopie.

Le modificateur `delete` Il est aussi possible d'empêcher l'utilisation de *n'importe quelle méthode* à l'aide du modificateur `delete` (listing 2.29).

Spécification explicite et génération par défaut de méthodes Même s'il y a des exceptions, les cinq méthodes ci-dessous forment généralement un ensemble cohérent :

1. constructeur par recopie ;
2. constructeur par déplacement ;
3. opérateur d'affectation par recopie ;
4. opérateur d'affectation par déplacement ;
5. destructeur.

Si l'une de ces méthodes est spécifiée (c.-à-d. déclarée, définie, `=default` ou `=delete`), alors aucune des deux méthodes de déplacement n'est générée par le compilateur, et il est fortement recommandé de définir aussi les méthodes de recopie ([12], *control of defaults*).

2.1.10 Méthodes constantes

Un méthode peut être déclarée *constante*, ce qui signifie intuitivement qu'elle ne modifie pas l'état interne de l'objet. Plus précisément, si une méthode est déclarée constante, toutes les données membres⁵ de la classe ont le statut de constante dans le corps de la méthode ; et seules des méthodes constantes de la classe peuvent être appelées par celle-ci. En contrepartie, une telle méthode peut être à son tour appelée à partir d'un objet constant. En effet, l'appel d'une méthode non-constante d'un objet déclaré constant provoque, logiquement, une erreur de compilation. C'est aussi le cas pour un objet manipulé via une référence constante ou un pointeur vers un objet constant.



Une erreur classique consiste à prendre soin d'effectuer des passages d'arguments par références pour éviter des recopies inutiles d'objets ; ces références étant déclarées constantes pour permettre l'utilisation des fonctions en question sur des objets temporaires, pour enfin s'apercevoir que les références constantes provoquent des erreurs liées à l'appel de méthodes non-constantes.

Afin de tenir compte de la mise en garde précédente, une bonne habitude de programmation consiste à utiliser le mot-clé `const` comme moyen d'amélioration de la lisibilité, la prévention des erreurs venant ensuite d'elle même. En effet, dès lors qu'une méthode n'a aucun effet sur l'état de l'objet, il est souhaitable de la déclarer `const`, ce qui dispense de tout commentaire sur le fait que la méthode ne modifie pas l'objet à partir duquel elle est appelée. Les méthodes classiques d'accès aux données membres rentrent bien entendu dans cette catégorie.

La syntaxe pour la déclaration d'une méthode constante est la suivante :

```
TypeRetour nomMéthode( Type1 , ... ) const;
```

Notez que le mot-clé `const` fait partie intégrante du prototype de la méthode, et doit donc être répété dans la définition, qui sera de la forme :

```
TypeRetour NomClasse::nomMéthode( Type1 arg1 , ... ) const {
    ...
}
```

Finalement, le listing 2.30 illustre tout ce qui vient d'être dit.

2.1.11 Fonctions et classes amies

Le langage C++ possède une particularité qui n'est pas *standard* dans la famille des langages de POO : la relation d'amitié entre classes et fonctions (non membres), mais aussi l'amitié entre classes.

Définition

- Une fonction déclarée *amie* (mot-clé `friend`) dans la définition d'une classe peut accéder aux membres privés ou protégés des instances de cette classe.

5. Exception faite des données membres déclarées `static` (non constantes), qui ont un statut particulier de ce point de vue.

```
1 #include <cmath>
2 #include <iostream>
3 #include <string>
4 using namespace std;
5
6 class Matrix {
7 public:
8     Matrix(unsigned int rows = 0, unsigned int columns = 0){
9         // ...
10    };
11    double determinant() const;
12
13 protected:
14    double ** array;
15    void allocate(unsigned int rows, unsigned int cols);
16    // ...
17 };
18
19 Matrix product(const Matrix & a, const Matrix & b) {
20     Matrix c;
21     // ...
22     return c;
23 }
24
25 Matrix inverse(const Matrix & m) {
26     Matrix r;
27     // Here, calling m.determinant() would result in a
28     // compilation error if the method had not been
29     // declared "const".
30     if (fabs(m.determinant()) < 1.0e-5) { // ?
31         throw std::string("inverse(): Singular matrix");
32     }
33     // ...
34     return r;
35 }
36
37 istream & operator>>(istream & in, Matrix & m);
38
39 ostream & operator<<(ostream & out, Matrix & m);
40
41 int main() {
42     Matrix a(50, 100), b(100, 50), c(50, 50);
43     cin >> a >> b; // ?
44     c = inverse(product(a, b));
45     cout << c;
46 }
```

Listing 2.30 – Exemple pour lequel l'usage de méthodes constantes s'impose.

```

1 class Circle {
2     double xCenter, yCenter, radius;
3
4 public:
5     Circle() : xCenter(0), yCenter(0), radius(1.0) {
6     }
7     Circle(double x, double y, double r);
8
9     friend ostream & operator<<(ostream & out, Circle c);
10    friend class Geometer;
11};
12
13 class Geometer { // The friend of circles.
14     // ...
15 public:
16     Geometer();
17     void move(Circle & circle) {
18         circle.xCenter += 10.0;
19         circle.yCenter += 10.0;
20     }
21 };
22
23 ostream & operator<<(ostream & out, Circle c) {
24     out << "Circle(" << c.xCenter << ',' << c.yCenter;
25     out << ',' << c.radius << ')';
26     return out; // Why ?
27 }
```

Listing 2.31 – Classe ayant deux amies : une fonction et une autre classe.

- Les méthodes d'une classe *B* déclarée *amie* d'une classe *A* peuvent accéder aux membres privés ou protégés des objets de *A*.

Remarque 2.3 *En C++, l'amitié ne s'hérite pas.*

Syntaxe par l'exemple

Le listing 2.31 fournit un exemple de définition d'une classe qui possède une classe amie ainsi qu'une fonction amie.

Discussion Comment s'effectue le choix entre méthode et fonction amie pour un traitement opérant sur un objet ?

2.2 Surcharge d'opérateurs

Comme on l'a vu, l'utilisateur peut surcharger l'opérateur d'affectation. En fait, cette particularité est partagée par la majorité des opérateurs du langage.

2.2.1 Principe

Les opérateurs peuvent être surchargés par des fonctions dont l'un des arguments doit être une classe ou une structure définie par le programmeur. Il peuvent aussi l'être sous la forme de méthodes d'une classe. Dans les deux cas l'arité des opérateurs est conservée et les règles suivantes s'appliquent :

- Un opérateur binaire peut être implémenté par une fonction membre à un argument, ou par une fonction (éventuellement amie) à deux arguments.
- Un opérateur unaire peut être implémenté par une fonction membre sans argument, ou par une fonction (éventuellement amie) à un argument.

Si T est une classe, x et y sont deux objets de la classe T , et OP est un opérateur, l'expression « $x OP y$ » peut être interprétée selon les cas comme « $x.operatorOP(y)$ », ou bien comme « $operatorOP(x, y)$ ».

Opérateurs pouvant être surchargés

Tous les opérateurs ne peuvent pas être surchargés. L'encadré ci-dessous énumère ceux qui peuvent l'être.

<code>+</code>	<code>-</code>	<code>*</code>	<code>/</code>	<code>%</code>	<code>^</code>	<code>&</code>	<code> </code>	<code>~</code>	<code>!</code>
<code>=</code>	<code><</code>	<code>></code>	<code>+=</code>	<code>-=</code>	<code>*=</code>	<code>/=</code>	<code>%=</code>	<code>^=</code>	<code>&=</code>
<code> =</code>	<code><<</code>	<code>>></code>	<code>>>=</code>	<code><<=</code>	<code>==</code>	<code>!=</code>	<code><=</code>	<code>>=</code>	
<code>&&</code>	<code> </code>	<code>++</code>	<code>--</code>	<code>-></code>	<code>*</code>	<code>,</code>	<code>-></code>	<code>[]</code>	<code>()</code>
<code>new</code>		<code>new []</code>		<code>delete</code>		<code>delete []</code>			

Remarque 2.4 Les opérateurs `=`, `[]`, `()` et `->` ne peuvent être que des méthodes.

Opérateurs ne pouvant pas être surchargés

- `::` Résolution de portée.
- `.` Sélection de membre.
- `.*` Sélection de membre via un pointeur de fonction.
- `_?_:_` Opérateur ternaire.

Exemple

Le listing 2.32 donne un exemple de classe pour laquelle deux des opérateurs du langage sont surchargés : le '-' unaire et le '-' binaire en tant que méthodes ; le '+' binaire sous forme de fonction non membre.

```
1 class Complex {
2     double re , im ;
3
4 public :
5     Complex() : re(0) , im(0) {
6     }
7     Complex(double , double );
8     friend Complex operator+(Complex , Complex );
9     Complex operator-(Complex) {
10        // ...
11    }
12    Complex operator-();
13    Complex operator*(Complex );
14 };
15
16 Complex operator+(Complex a , Complex b) { // V1
17     return Complex(a.re + b.re , a.im + b.im);
18 }
19
20 Complex Complex::operator-() { // V2
21     Complex z ;
22     z .re = -re ;
23     z .im = -im ;
24     return z ;
25 }
26
27 Complex Complex::operator*(Complex z) { // V3
28     return Complex(re * z.getRe() - im * z.getIm() ,
29                     re * z.getIm() + im * z.getRe());
30 }
```

Listing 2.32 – Exemple de surcharge d’opérateurs du langage.

2.2.2 Opérateurs -- et ++

La différenciation entre pré- et post-incrémantation se fait à l'aide d'un argument fictif inutilisé :

- $T \& T::operator++()$; pré-incrémantation.
- $T T::operator++(int)$; post-incrémantation.

L'argument `int`, auquel il est inutile de donner un nom, ne sert qu'à différencier les déclarations.

On rappelle que l'opérateur de pré-incrémantation retourne généralement (dans l'optique de lui conserver sa sémantique habituelle) une référence à l'objet incrémenté lui-même. Ainsi, dans le code du listing 2.33 à la ligne 34, l'opérateur `*=` est appliqué sur le résultat de l'opérateur `++`. De son côté, l'opérateur de post-incrémantation retourne une *copie* de l'objet dans l'état où il était avant d'être incrémenté (listing 2.33 ligne 10, utilisé à la ligne 36).

Exercice 2.4 Définir les opérateurs de pré- et post-incrémantation pour la classe `String` des listings précédents. Incrémenter une chaîne signifie ici la transformer en celle qui lui succède dans l'ordre lexicographique, sans s'inquiéter de l'appartenance du résultat à un quelconque dictionnaire. Comment gérez vous le retour de l'opérateur de post-incrémantation ?

2.2.3 Opérateur []

C'est un opérateur binaire qui doit être obligatoirement une méthode. Son utilisation typique concerne la syntaxe permettant de faire référence à une position donnée dans les classes *conteneurs*. Il doit alors généralement renvoyer une référence (ou un pointeur) pour permettre la modification de l'élément désigné (Ex. : `vecteur[10] = 4.5;`), mais aussi d'une manière plus générale pour permettre d'agir sur l'élément référencé et pas sur une copie qui serait retournée. Il faut remarquer que le type de l'indice n'est pas limité aux types entiers. Le listing 2.34 donne un exemple de classe pour laquelle l'opérateur crochet a un sens, raison suffisante pour qu'il soit défini ici.

```

1 class String {
2     char * array;
3
4 public:
5     String();
6     String(const char *);
7     char operator[](int i) { // Version A
8         return array[i];
9     }
10    char & operator[](int i) { // Version B
11        return array[i];
12    }
13 };
14
15 void foo() {
16     String s("ENSICAEN");
17     std::cout << s[2] << std::endl;
18     s[3] = '-';
19 }
```

```

1 #include <iostream>
2 #include <string>
3 using namespace std;
4 class StarString {
5 public:
6     StarString & operator++() { // Prefix
7         _text += '*';
8         return *this;
9     }
10    StarString operator++(int) { // Postfix
11        StarString savedValue = (*this);
12        _text += '*';
13        return savedValue;
14    }
15    StarString & operator*=(unsigned int n) {
16        string pattern = _text;
17        _text.clear();
18        while (n--) {
19            _text += pattern;
20        }
21        return *this;
22    }
23    const string & text() const {
24        return _text;
25    }
26
27 private:
28     string _text;
29 };
30
31 int main(int, char *[]) {
32     StarString stars;
33     ++stars;                                // stars._text == "*"
34     (++stars) *= 2;                          // stars._text == "****"
35     cout << stars.text() << endl;          // ****
36     cout << (stars++).text() << endl;       // **** (idem !)
37     cout << stars.text() << endl;           // *****
38     return 0;
39 }
```

Listing 2.33 – Une structue de chaines

Listing 2.34 – Classe surchargeant l'opérateur crochets.

Remarque 2.5 La double surcharge de l'opérateur `[]` comme dans le listing 2.34 n'est pas permise. Les deux méthodes ne diffèrent en effet que par les types retournés.

2.2.4 L'opérateur `->`

C'est un opérateur dont le comportement diffère des autres, notamment parce que son type de retour est partiellement imposé. Il est notamment indispensable à l'implémentation de classes utilitaires comme les *pointeurs intelligents* (ou *smart pointers*) qui sont présentés dans la section 7.6.

- C'est un opérateur postfixe *unaire* qui permet le contrôle des déréférencements.
- Il doit retourner un pointeur ou un objet pour lequel l'opérateur `->` s'applique.
- Sa définition est de la forme :

$$T^* \ idClasse::operator->() \ \{ \ /* \ [\dots] \ */ \ };$$

- La syntaxe de l'appel est :

$$\quad \quad \quad expression->membre$$

- Qui est évalué de la façon suivante :

$$(expression.operator->())->membre$$

2.2.5 L'opérateur `()`, les objets fonctions

Cet opérateur permet d'utiliser un objet comme une fonction. Ajouté au polymorphisme, il apporte notamment une alternative élégante aux pointeurs de fonctions hérités du langage C. En effet, une fonction classique pourra par exemple accepter des arguments de types *classes d'objets fonctions*.

Il généralise la syntaxe de l'appel de fonction :

$$\quad \quad \quad fonction(arguments \dots)$$

au cas où **fonction** est en fait une instance de classe.

Exemple L'objet additionneur.

Exercice 2.5 (Question d'examen) Écrivez le code de la déclaration d'un objet fonction précédé de la définition de la classe correspondante. Vous choisirez librement le type de la valeur de retour et celui des paramètres.

2.2.6 Opérateur de conversion

Il est possible d'effectuer des conversions explicites ou implicites grâce aux constructeurs qui acceptent un seul argument. Ainsi, la ligne 4 du listing 2.36 est correcte dès lors que le constructeur de `Complex` à partir d'un `double` est défini, et ce *sans que l'opérateur d'affectation ne le soit*.

Toutefois, on note que :

```

1 | class Additioner {
2 |     int value;
3 |
4 | public:
5 |     Additioner(int v) : value(v) {
6 |     }
7 |     void operator()(int & x) {
8 |         x += value;
9 |     }
10| };
11|
12| void foo() {
13|     int array[5] = {1, 2, 3, 4, 5};
14|     Additioner add(10);
15|     for (int i = 0; i < 5; i++) {
16|         add(array[i]);
17|     }
18| }
```

Listing 2.35 – Un objet fonction *additionneur*.

```

1 | double x = 1.414;
2 | double y = 0.0;
3 | Complex z;
4 | z = x; // Calls Complex::Complex(double);
```

Listing 2.36 – Exemple de conversion automatique souhaitable.

- La conversion vers un type de base est dans ce cas impossible.
- De même, la conversion vers une classe déjà définie et non « modifiable » par le programmeur n'est pas permise de cette façon puisqu'il faut ajouter un constructeur à la classe cible.

Opérateur de conversion

La définition d'un opérateur de conversion permet de gérer dans une classe *sa propre* conversion en un autre type. (Et pas l'inverse.)

C'est une fonction membre dont le prototype est de la forme :

`Classe::operator TYPE();`

Il permet alors la conversion de *Classe* en *TYPE* qui peut être implicite ou explicite (par exemple pour lever les ambiguïtés lors d'un appel de fonction). En toute logique, aucun type de retour n'est à préciser.

Cas pratique

Le code du listing 2.37 illustre l'utilisation d'une conversion définie par l'utilisateur pour les classes de flux d'entrée/sortie de la bibliothèque standard (§ 4.1).

```

1 istream & istream ::operator>>(char &);
2 istream ::operator bool();
3
4 void pass_through()
5 {
6     char c;
7     while (cin >> c) {
8         cout << c;
9     }
10 }
```

Listing 2.37 – Idiome de lecture d'un flux de fichier en entrée.

C++11 : Opérateur de conversion explicite

Les opérateurs de conversion tels que définis précédemment sont valables pour des conversions implicites ou explicites. Il peut être utile de limiter un opérateur donné aux seules conversions explicites. Par exemple, on peut souhaiter qu'un objet soit converti en un booléen uniquement si on l'utilise comme condition (`if`, `while`, etc.) ou bien avec un opérateur logique. Il peut en effet être regrettable que la conversion implicite en un type `bool` rende de fait possible l'utilisation de l'objet en question dans une opération arithmétique⁶. Pour cela, le C++11 introduit la notion d'opérateur de conversion « `explicit` » (listing 2.38).

6. Même si cela peut aussi être souhaitable.

```
1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 class Rational {
6 public:
7     Rational(int, unsigned int) {
8         // ...
9     }
10    operator string() {
11        // ...
12    }
13    explicit operator bool() {
14        // ...
15    }
16
17 private:
18     int numerator;
19     unsigned int denominator;
20 };
21
22 void display(string) {
23 }
24 void test(bool) {
25 }
26
27 int main() {
28     Rational r(3, 2);
29     display(r);           // OK
30     test(r);             // Error: no implicit conversion
31     test(static_cast<bool>(r)); // OK
32     int a = r + r;        // Error: no implicit conversion
33     if (r) {               // OK (condition)
34     } // OK (logical operator)
35     if (!r) {              // OK (logical operator)
36     } // OK (logical operator)
37 }
```

Listing 2.38 – Opérateur de conversion explicite.

2.3 Héritage

2.3.1 Notions de classe dérivée et d'héritage

Motivation

Une modélisation correcte peut amener à définir plusieurs classes ayant des données ou fonctionnalités communes. On peut alors avoir un souci de *généralisation* pour *factoriser* ce qui est commun. Il arrive aussi d'avoir à définir une classe qui est très proche d'une autre, mais avec certaines particularités supplémentaires. On parlera alors de *spécialisation*.

Généralisation et spécialisation

A la vue du diagramme de classes de la figure 2.3, on peut faire les remarques suivantes :

- *Tout* véhicule peut démarrer, acquérir une certaine vitesse, etc.
- Un avion *est un* véhicule motorisé qui peut prendre de l'altitude.
- Un avion, *comme* une fusée, peut avoir une altitude.
- Un avion peut avoir une hélice, un réacteur.
- Un avion et une voiture ne se démarrent pas de la même façon.

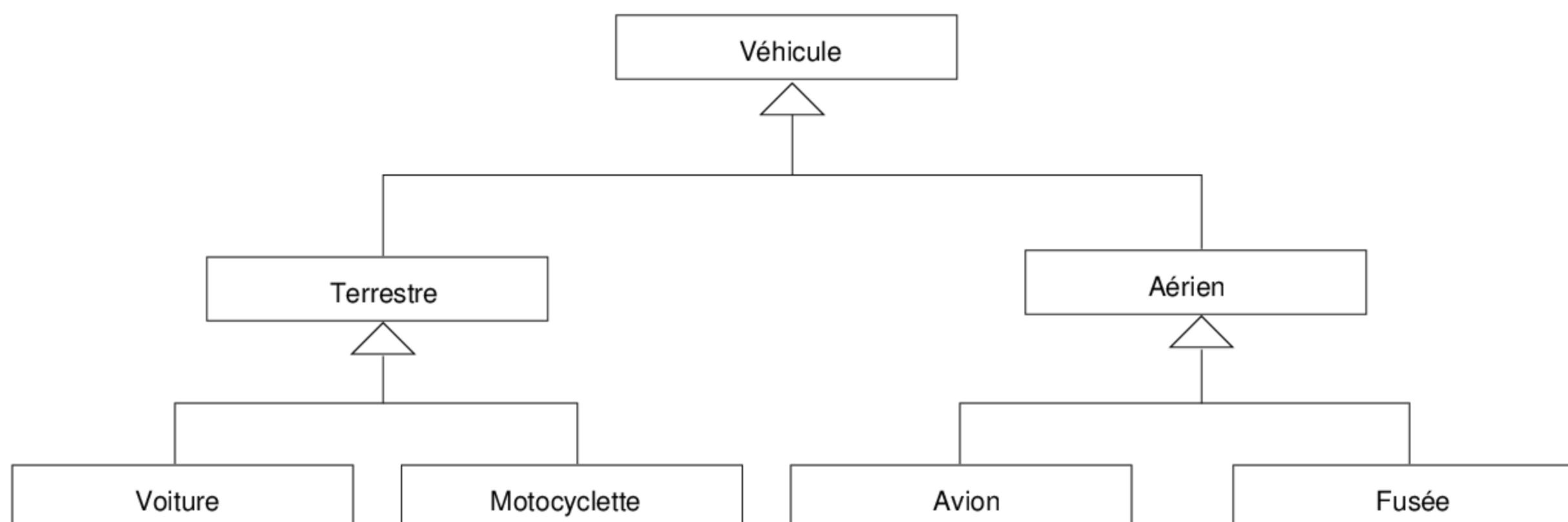


FIGURE 2.3 – Arbre d'héritage de types de véhicules.

Il est raisonnable de souhaiter programmer la classe Voiture sans avoir à récrire le code qu'elle a en commun avec une classe Camion, autre véhicule terrestre !

Solution en POO/C++

Une classe B peut *hériter* d'une classe A ; elle possède alors toutes les caractéristiques de la classe A (c.-à-d., données et méthodes). La classe A est appelée *classe de base*. La classe B est appelée *classe dérivée*.

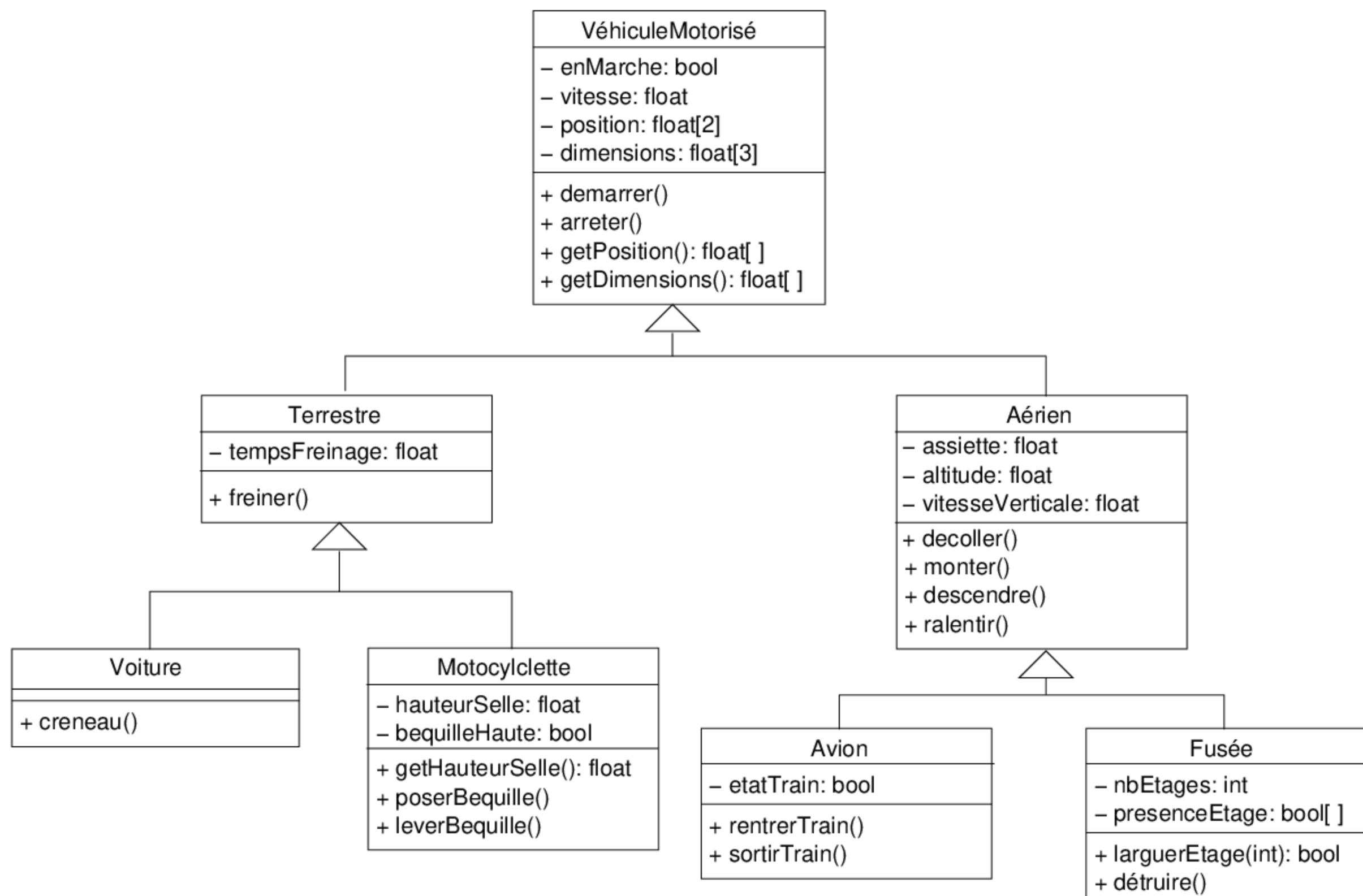
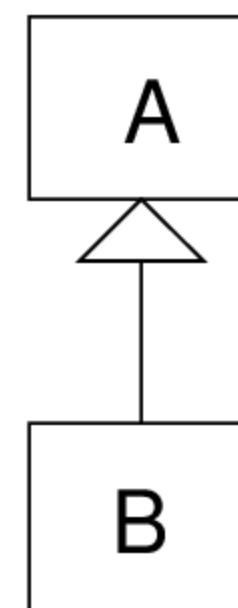


FIGURE 2.4 – Détail des classes modélisant les véhicules, avec héritage.

2.3.2 Syntaxe

```

class A { ... } ;
class B : public A { ... };
class B : private A { ... };
class B : protected A { ... };
class B : A { ... };
  
```



Dans la suite, on utilisera sans précision A et B comme noms de classes, la seconde (B) étant une classe dérivée de la première (A), selon le schéma précédent.

2.3.3 Intérêt

La figure 2.4 montre un diagramme de classes pour la hiérarchie des véhicules, plus détaillé que celui de la figure 2.3. Il est à comparer avec les deux diagrammes de classes que l'on pourrait être amené à définir en se passant de l'héritage (cf. figure 2.5). Dans ce dernier cas, on a le choix entre une classe unique mais exagérément complexe et un grand nombre de classes présentant de nombreuses redondances.

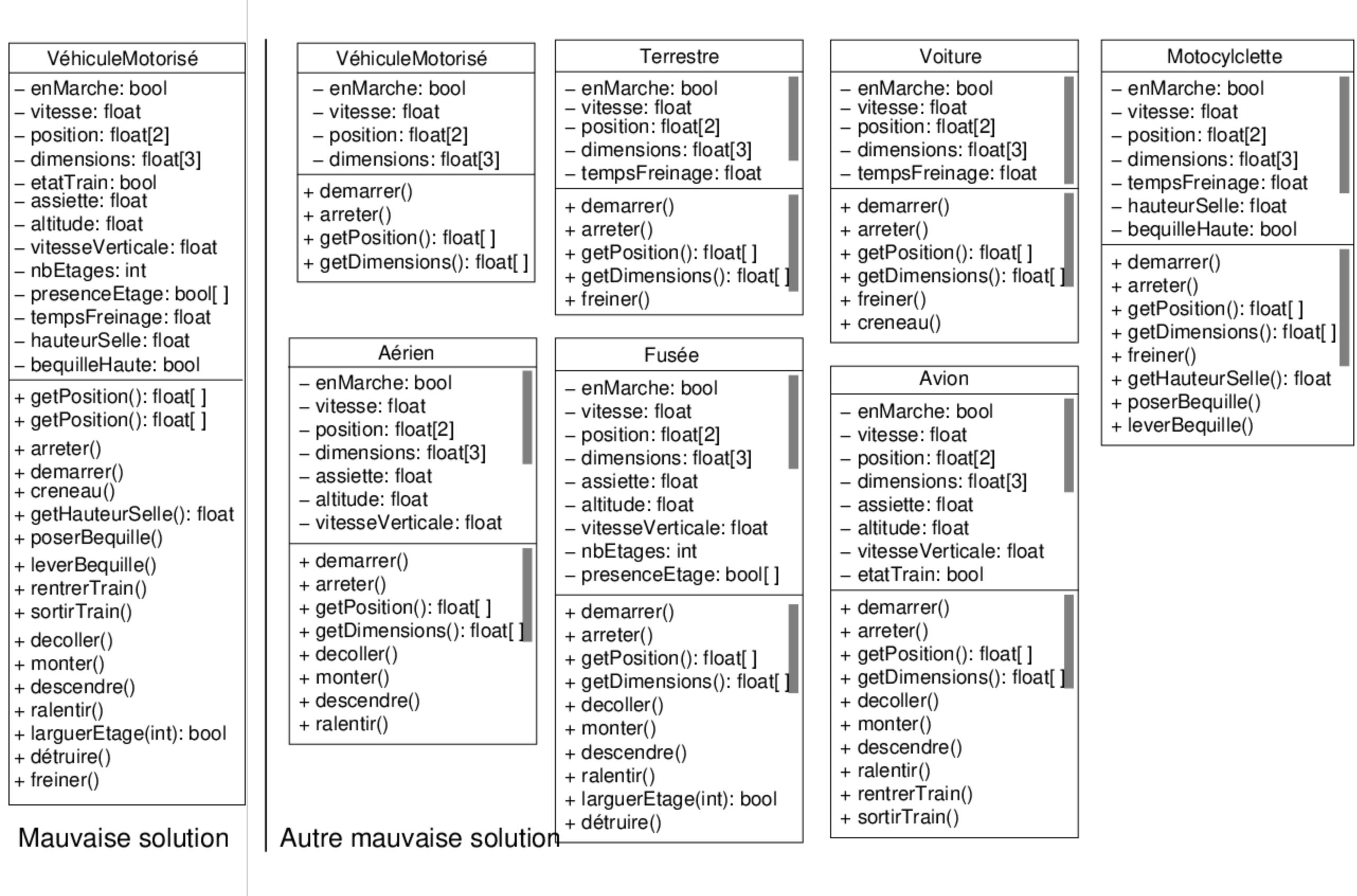


FIGURE 2.5 – Classes modélisant les véhicules, sans héritage.

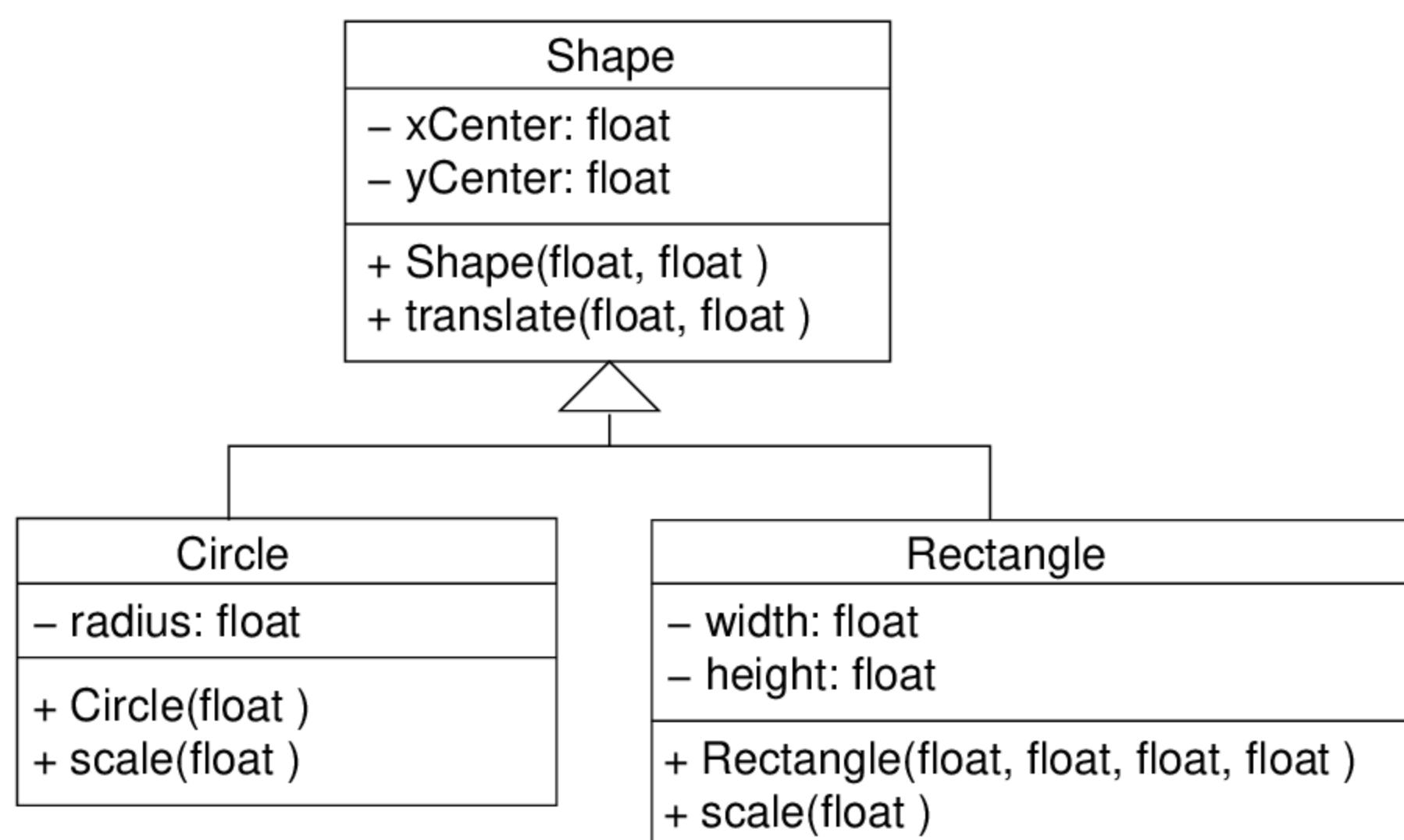


FIGURE 2.6 – Arbre d'héritage de formes géométriques (Diagramme de classes).

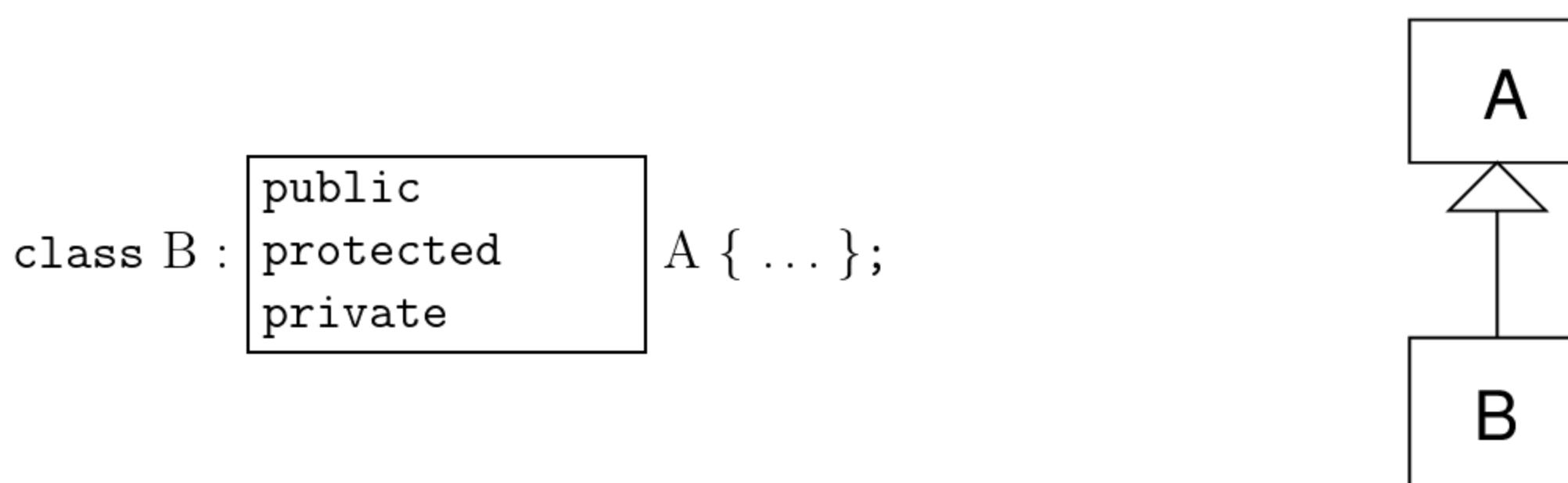
2.3.4 Exemple simple

La figure 2.6 (page 67) présente un diagramme de classes qui montre la relation d'héritage entre des formes géométriques. A la lecture de ce diagramme, on peut énoncer qu'un rectangle « est une » forme. On peut notamment déplacer un rectangle, comme c'est le cas pour toute forme. Des déclarations qui correspondent à ce diagramme sont données dans le listing 2.39.

2.3.5 Dérivation et contrôle d'accès

Les 3 modes de dérivation présentés en 2.3.2 influencent la visibilité des membres de la classe de base depuis les méthodes de la classe dérivée (et aussi depuis les fonctions extérieures à cette classe).

Dans le cas suivant :



la visibilité des membres de A depuis les fonctions membres, les fonctions amies, les autres fonctions et les classes dérivées de B est donnée par le tableau 2.1.

Membre de A	Visibilité / A	Visibilité relative à B		
		Dérivation public	Dérivation protected	Dérivation private
public:	Membres Amies Dérivées Autres	Membres Amies Dérivées Autres	Membres Amies Dérivées	Membres Amies
protected:	Membres Amies Dérivées	Membres Amies Dérivées	Membres Amies Dérivées	Membres Amies
private:	Membres Amies	∅	∅	∅

TABLE 2.1 – Héritage et contrôle d'accès.

Le listing 2.40 illustre les différents types d'héritage (`public`, `private` et `protected`) et leur impact sur la visibilité des membres hérités (en fonction de celle qu'ils ont dans la classe de base A).



Les dérivation privées et protégées vont à l'encontre du polymorphisme.[...]

```
1 class Shape {
2     float xCenter, yCenter;
3
4 public:
5     Shape() {
6         xCenter = 0;
7         yCenter = 0;
8     }
9     Shape(float x, float y) : xCenter(x), yCenter(y) {
10    }
11    void translate(float x, float y) {
12        xCenter = x;
13        yCenter = y;
14    }
15};
16
17 class Circle : public Shape {
18     float radius;
19
20 public:
21     Circle() {
22         radius = 0;
23     }
24     Circle(float x, float y, float r) : Shape(x, y) {
25         radius = r; // OK
26     }
27     void scale(float s) {
28         radius *= s;
29     }
30 };
31
32 class Rectangle : public Shape {
33     float width, height;
34
35 public:
36     Rectangle();
37     Rectangle(float x, float y, float l, float h) {
38         Shape(x, y); // NO WAY!
39         width = l;
40         height = h;
41     }
42     void scale(float s) {
43         width *= s;
44         height *= s;
45     }
46 };
```

Listing 2.39 – Définition des trois formes géométriques du diagramme de la figure 2.6.

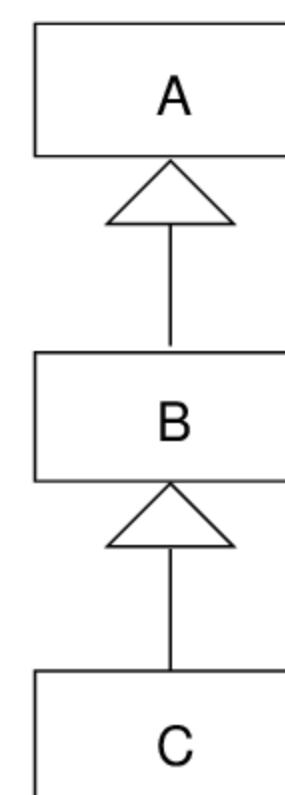
```

1  class A {
2  private:
3      int a;
4
5  protected:
6      int b;
7
8  public:
9      int c;
10 };
11
12 class B : public A { // Public inheritance
13 public:
14     void f() {
15         a = 5; // ERROR
16         b = 5; // OK
17         c = 5; // OK
18     }
19     friend void g(B x);
20 };
21
22 void g(B x) {
23     x.a = 5; // ERROR
24     x.b = 5; // OK
25     x.c = 5; // OK
26 }
27
28 class C : private A { // Private inheritance
29 public:
30     void f() {
31         a = 5; // ERROR
32         b = 5; // OK
33         c = 5; // OK
34     }
35     friend void h(C x);
36 };
37
38 void h(C x) {
39     // ...
40 }
```

Listing 2.40 – La dérivation publique comparée à la dérivation privée.

2.3.6 Construction, destruction et héritage

Une classe B dérivée de A « hérite » des constructeurs de A. Si aucune précision n'est donnée, le constructeur par défaut de la classe A est appelé automatiquement lors de l'instanciation d'un objet de la classe B. Mais le constructeur de A ne fait rien pour B. Il faut donc généralement définir un constructeur spécifique à la classe dérivée, qui peut logiquement s'appuyer sur les opérations déjà réalisées par celui de la classe A. Ainsi, dans la situation décrite par le schéma ci-contre :



- A l'instanciation d'un objet de la classe C, les constructeurs sont logiquement appelés dans l'ordre : classe de base puis classe dérivée. C'est à dire `A::A()` puis `B::B()` et enfin `C::C()`. (Inversement pour les destructeurs.)
- Les éventuels membres d'une classe A qui sont eux mêmes des instances de classes sont construits *avant* l'appel du constructeur de A.
- Ces appels automatiques peuvent être *court-circuités* grâce à la syntaxe particulière déjà rencontrée pour l'initialisation des données membres (cf. listing 2.41).

Rappel Si un constructeur avec argument(s) est défini, le constructeur par défaut n'est pas généré par le compilateur.

2.3.7 C++11 : Héritage explicite des constructeurs

Comme indiqué dans la section précédente, le constructeur d'une classe mère peut être appelé par le constructeur de sa classe dérivée. Si aucun appel explicite n'est fait dans le constructeur de la classe fille, c'est le constructeur par défaut de la classe mère qui est appelé. C'est en cela que le constructeur est « hérité ». Toutefois, l'instruction même de création d'une instance de la classe B n'a pas accès au constructeur de la classe mère de B. Le code du listing 2.42 est en effet erroné car la classe B ne possède pas, comme A, de constructeur ayant un paramètre.

En C++11, il est possible d'expliciter l'héritage de *tous* les constructeurs d'une classe de base, grâce au mot-clé `using`. Le listing 2.43 montre comment le code précédent peut être rendu correct. Il faut bien noter que cet héritage explicite est en « tout ou rien » : on ne peut hériter d'un sous-ensemble des constructeurs de la classe de base.

2.3.8 Héritage et conversions

L'héritage permet, avec les méthodes virtuelles, le polymorphisme d'exécution qui fera l'objet de la section 2.6. Avant d'aborder cette notion essentielle de la POO, il convient de présenter les règles de conversions automatiques qu'autorise l'héritage.

```
1 class Point {
2     float x, y;
3
4 public:
5     Point( float a = 0.0, float b = 0.0 ) {
6         x = a;
7         y = b;
8     }
9 };
10
11 class Shape {
12     Point center;
13
14 public:
15     Shape() {
16     }
17     Shape( float x, float y ) : center(x, y) {
18     }
19 };
20
21 class Circle : public Shape {
22     float radius;
23
24 public:
25     Circle( float x, float y, float r ) : Shape(x, y) {
26         radius = r; // Another possible syntax
27     }
28 };
```

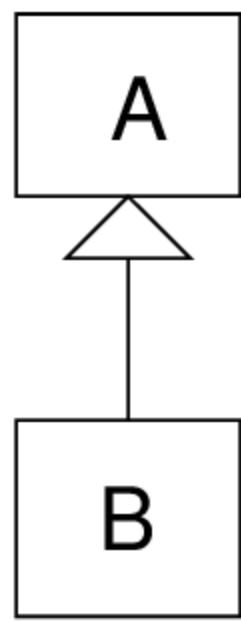
Listing 2.41 – Constructeurs et héritage.

```
1 class A {  
2     public:  
3         A(int value) : _value(value) {  
4             }  
5     private:  
6         int _value;  
7     };  
8  
9  
10    class B : public A {  
11        public:  
12            B() : A(0) {  
13                }  
14        };  
15  
16        int main() {  
17            B b(10); // Error!  
18        }
```

Listing 2.42 – Code erroné : un constructeur de la classe A ne peut pas être utilisé lors de l’instanciation de sa classe dérivée B.

```
1 class A {  
2     public:  
3         A() : _value(0) {  
4             }  
5         A(int value) : _value(value) {  
6             }  
7     private:  
8         int _value;  
9     };  
10  
11    class B : public A {  
12        float _x{9.81};  
13  
14        public:  
15            using A::A;  
16        };  
17  
18        int main() {  
19            B b(10);  
20        }
```

Listing 2.43 – Héritage explicite des constructeurs d’une classe mère.



```

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

Les règles de conversion suivent l’adage « Qui peut le plus, peut le moins ». Dans le cas ci-contre, il y a donc conversion *implicite* d’un objet de type B en un objet de type A. Plus précisément :

- un pointeur sur un objet de B peut être converti en un pointeur sur un objet de A ;
- une référence à un objet de B peut être convertie en une référence à un objet de A ;
- enfin, tout objet de type B peut être converti en un objet de type A.

Attention La réciproque du 3^e point est bien sûr fausse, sauf si une conversion est définie par l’utilisateur. [...]

Exemple

Quelques exemples de conversions (im)possibles sont donnés dans le listing 2.44.

2.4 Implémentation de l’appel des méthodes et pointeur this

2.4.1 Implémentation des méthodes

Il n’y a que peu de différence entre une classe C++ et une structure C classique. Une méthode est implémentée sous la forme d’une fonction ayant pour premier argument « caché » l’adresse de l’objet *dès lors* lequel elle a été appelée. Ainsi,

objet.méthode(arg1, arg2, ..., argN);

est implémenté par :

méthode(&objet, arg1, arg2, ..., argN);

Ce premier argument ajouté est disponible pour le programmeur : sa valeur est donnée par le mot-clé **this**, utilisable dans toute méthode non statique [...]. On rappelle que dans une méthode, les noms des variables et fonctions sont liés en priorité aux membres de la classe concernée.

Exemple

Des exemples d’appels de méthodes sont donnés dans le listing 2.45.

Exercice 2.6 Compilez et interprétez l’affichage produit par le programme du listing 2.45, notamment en ce qui concerne la taille des objets.

```
1  class Shape {
2      Point center;
3  public:
4      void display() {
5          // ...
6      }
7  };
8
9  class Circle : public Shape {
10     float radius;
11 public:
12     void display();
13 };
14
15 int main() {
16     Shape f, *p_shape;
17     Circle c, *pc;
18
19     pc = &c;           // Obviously correct!
20     pc->display(); // Circle :: display()
21     pc = &f;           // Error
22     p_shape = &c;     // OK (implicit cast)
23     pc = p_shape;   // Error
24
25     // Correct because explicit
26     // but VERY dangerous!
27     // (what if p_shape was the address of a Rectangle?)
28     pc = (Circle*)p_shape;
29
30     p_shape->display(); // Shape :: display() !
31 }
```

Listing 2.44 – Conversions entre références et pointeurs sur des formes.

```

1 #include <iostream>
2 using namespace std ;
3
4 class A {
5 public:
6     void showYou() {
7         cout << this << endl ;
8     }
9 };
10
11 int main() {
12     A a, b;
13     A * pa = &a;
14     cout << &a << endl ;
15     a.showYou();
16     cout << pa << endl ;
17     pa->showYou();
18     cout << &b << endl ;
19     b.showYou();
20     return 0;
21 }
```

Listing 2.45 – Exemples d'appels de méthodes.

2.4.2 Héritage et conversions

Terminons cette section sur l'implémentation par une remarque sur la façon dont les conversions entre types dérivés sont réalisées.

Implémentation des conversions

- Lors de la conversion implicite d'un objet de type B en un objet parent de type A, seules les données communes à A et B (c.-à-d., celles de A) sont recopiées, comme illustré dans le haut de la figure 2.7(b). Les données supplémentaires de la classe B sont donc simplement ignorées, et perdues. Dans la partie inférieure de cette même figure, on voit que la conversion inverse laisserait indéterminées les valeurs des données membres d'un `Circle` qui sont absentes de la classe `Shape`.
- Dans le cas de la conversion entre pointeurs (`B*` vers `A*`), aucune modification n'intervient au niveau des données. Les opérations valides pour le type A sont bien sûr valides pour un objet de type B effectivement pointé ; les méthodes de A se contentent de manipuler les données propres à la classe A que possède aussi la classe B.

Les deux remarques précédentes montrent que l'héritage est en fait une « simple » imbrication de structures dans laquelle les données de chaque classe sont stockées de façon contiguë en suivant la relation de descendance.

```

1 Shape shape;
2 Circle circle(6.2, 2, 3.0);
3 Shape * p_shape = &circle;
4 shape = circle;
5 shape.translate(1, 1);      // Act on a shape
6 p_shape->translate(10, 0); // Act on a circle

```

Listing 2.46 – Conversions entre classes.

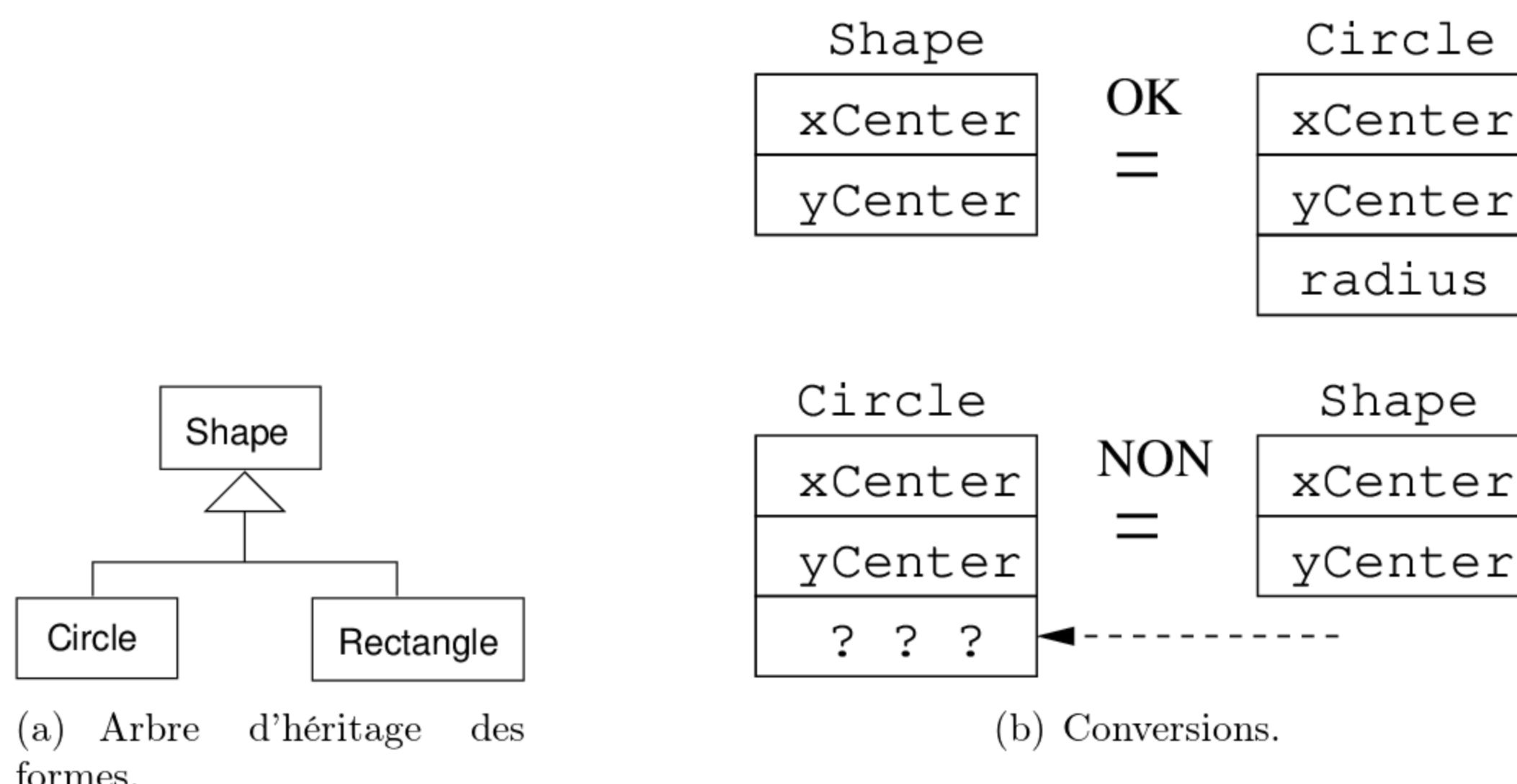


FIGURE 2.7 – Héritage et conversion.

Exemple

Des exemples de conversions sont donnés dans le listing 2.46.

2.5 Opérateurs portant sur les types

En C++ , de nouveaux opérateurs qui permettent de manipuler les types font leur apparition. Le premier, `typeid`, concerne les vérifications. Trois autres permettent un contrôle fin des conversions explicites, en autorisant par exemple un contrôle dynamique de la conversion, c.-à-d. pendant l'exécution d'un programme. Il s'agit des trois opérateurs dit de coercition : `static_cast`, `reinterpret_cast` et `dynamic_cast`.

2.5.1 L'opérateur `typeid`

Le mot-clé `typeid` retourne une référence à un objet constant (de type `std::type_info`) de la bibliothèque standard (en-tête `<typeinfo>`). Il peut être utilisé *si besoin* pour déterminer à l'exécution le type d'un objet pointé ou référencé, puisque les opérateurs `==` et `!=` ainsi qu'une relation d'ordre sont définis pour ce type (cf. listing 2.47). Son utilisation classique concerne les classes polymorphes (c.-à-d., qui possèdent des méthodes virtuelles) mais elle est possible aussi avec les autres types.

```

1 class type_info {
2 public:
3     virtual ~type_info();
4     bool operator==(const type_info &) const;
5     bool operator!=(const type_info &) const;
6     bool before(const type_info &) const;
7     const char * name() const;
8
9 private: // ???
10    type_info(const type_info &);
11    type_info & operator=(const type_info &);
12 };

```

Listing 2.47 – Définition partielle de la classe standard `type_info`.

A l'instar du mot-clé `sizeof`, cet opérateur s'applique à une expression ou bien à un nom de type. Dans le cas d'une expression, celle-ci n'est pas évaluée.

Exercice 2.7 *Quel principe d'implémentation vu précédemment peut être utilisé avantageusement pour définir le type `type_info` dans le cas des classes polymorphes ?*

2.5.2 L'opérateur `static_cast`

Il permet la conversion explicite entre types d'une même famille, le contrôle de validité étant réalisé à la compilation. Schématiquement, il autorise les conversions :

- $T1^* \rightarrow T2^*$, $T1& \rightarrow T2&$ (si les types pointés ou référencés sont parents) ;
- `enum` \rightarrow `bool`, `char`, `int` ;
- flottant \rightarrow intégral.

La syntaxe de l'opérateur est la suivante :

```
static_cast<type_résultat>( expression )
```

Exemple

```

1 double x = 12.5;
2 Shape * shape = new Circle(10, 10, 1.5);
3 Circle * circle = 0;
4 int i;
5
6 i = static_cast<int>(x);
7 circle = static_cast<Circle *>(shape);

```

Listing 2.48 – Utilisation de la coercition statique.

```

1 Circle c(10, 10, 50), *pc;
2 Rectangle r(10, 10, 30, 10), *pr;
3 Shape * p_shape = &c;
4 pc = dynamic_cast<Circle *>(p_shape); // OK
5 pr = dynamic_cast<Rectangle *>(p_shape); // pr = 0;

```

Listing 2.49 – Exemple d'utilisation de la coercition dynamique.

2.5.3 L'opérateur reinterpret_cast

Il est utilisé pour *forcer* une conversion entre types de familles différentes, comme celle d'un pointeur en un entier, ou entre pointeurs sur des classes sans relation de parenté. Il est à utiliser avec grande précaution.



Code peu portable, aucun contrôle.

2.5.4 L'opérateur dynamic_cast

Cet opérateur permet la conversion contrôlée d'un *pointeur sur* ou une *référence à* un objet de type polymorphe. Son utilisation est restreinte à ces deux cas bien précis.

Sa seconde particularité est que la validité de la conversion est vérifiée *pendant l'exécution* du programme d'après le type de l'objet source pointé ou référencé, mais uniquement dans le cas d'un type polymorphe (?). En effet, dans le cas d'une classe non polymorphe il n'apporte, comparé au `static_cast`, qu'un contrôle supplémentaire sur le type d'héritage qui relie les deux classes impliquées. (Voir les conséquences de l'héritage privé ou protégé sur le polymorphisme...)

Enfin, l'opérateur retourne la valeur spéciale `nullptr` en cas d'échec dans la conversion d'un pointeur, alors qu'il lève une exception standard de type `bad_cast` si la conversion d'une référence est invalide.

Syntaxe

Les deux seules syntaxes possibles sont les suivantes :

- `dynamic_cast< T*>(p)`, où *p* est un pointeur ; et
- `dynamic_cast< T&>(r)`, où *r* est une référence.

Exemple

Le listing 2.49 donne un exemple d'utilisation du `dynamic_cast` avec des pointeurs.

Exercice 2.8 Construisez un exemple comparable à celui du listing 2.49 mais en utilisant des références et en gérant l'erreur à l'aide des exceptions.

2.6 Méthodes virtuelles : le polymorphisme

Le polymorphisme est une notion essentielle de la programmation orientée objet. En C++ on distingue le *polymorphisme d'exécution* dont il est question ici, du *polymorphisme de compilation* qui fera l'objet du chapitre 3.

2.6.1 Motivations

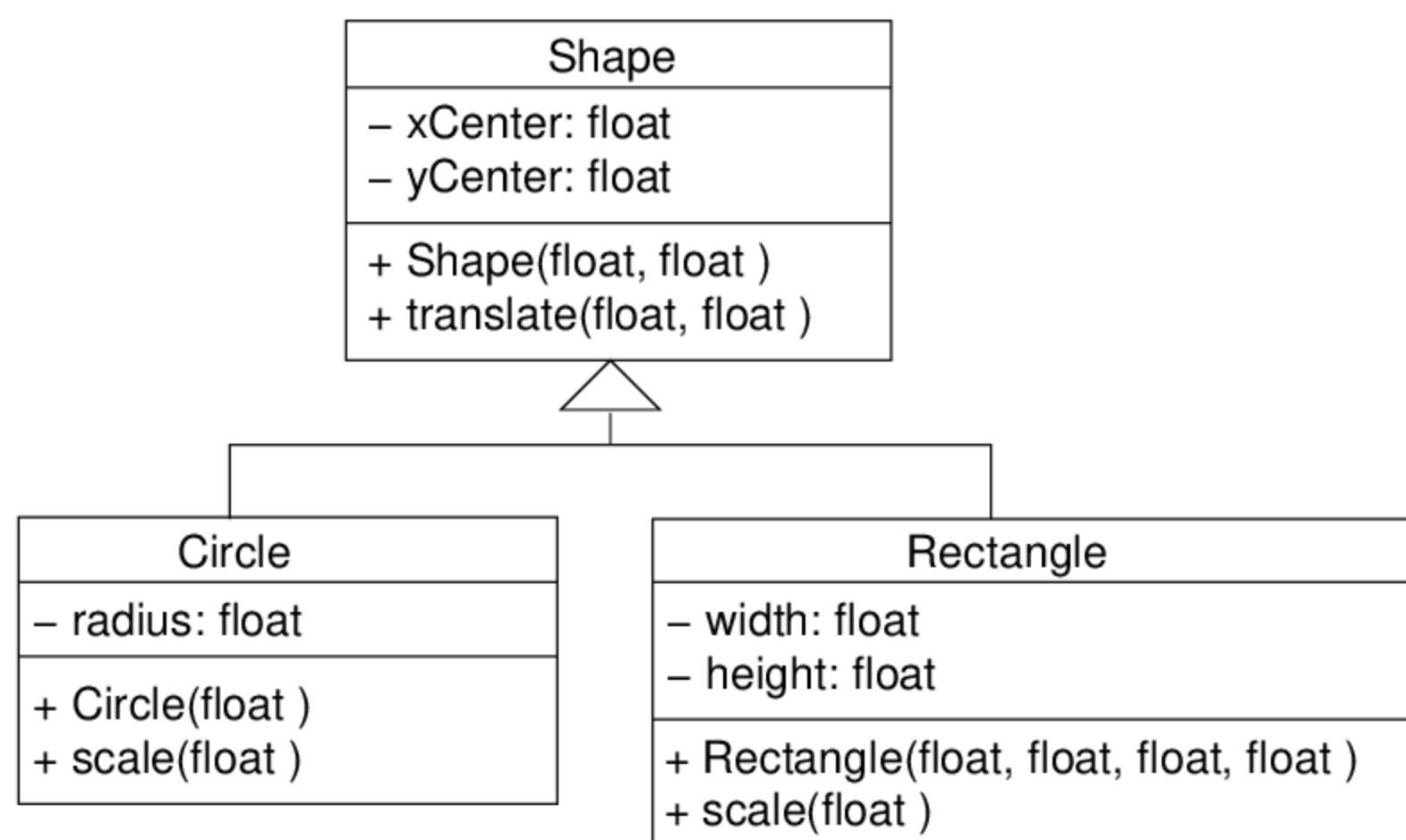


FIGURE 2.8 – Formes géométriques.

Soient les classes **Shape**, **Circle** et **Rectangle** telles que définies précédemment et dont le diagramme de classes est rappelé dans la figure 2.8.

Dans ce cas, `scale()` est une méthode de **Circle**, mais aussi de **Rectangle**. Si on veut manipuler une liste de formes, par exemple sous la forme d'un tableau de pointeurs, et que l'on souhaite appliquer une homothétie à toutes les formes de la liste ; alors on aimeraït écrire le code ci-dessous :

```

1  {
2      Shape * shapes[10];
3      shapes[0] = new Circle(10, 10, 1);
4      shapes[1] = new Rectangle(10, 10, 20, 10);
5      // ...
6      for (int i = 0; i < 10; i++) {
7          shapes[i]->scale(2); // ERROR!
8      }
9  }
  
```

Listing 2.50 – Exemple de polymorphisme souhaitable.

Pour que ce code soit valide, il faudrait au moins que la fonction `scale()` soit une méthode de la classe **Shape**, redéfinie par chacune des classes **Circle** et **Rectangle**. On retrouve bien dans cet exemple les relations de généralisation et spécialisation. Toutefois, définir une méthode `Shape::scale(float)` ne suffirait pas...

En effet, une classe dérivée peut redéfinir des méthodes de sa (ses) classe(s) de base. Mais quelle en est la conséquence lors de l'utilisation des conversions de références ou de pointeurs comme dans l'exemple qui suit ?

```

1 class A {
2 public:
3     void display() {
4         cout << "I am A" << endl;
5     }
6 };
7
8 class B : public A {
9 public:
10    void display() {
11        cout << "I am B" << endl;
12    }
13 };
14
15 void foo() {
16     A a, *pa = &a;
17     B b;
18     a.display();
19     pa->display();
20     pa = &b;
21     pa->display();
22 }
```

Listing 2.51 – Appel de méthodes après conversion.

On cherche alors à résoudre le problème suivant :

- Disposant d'un pointeur (sur) ou d'une référence à une classe de base, on veut appeler une méthode spécifique à l'objet *effectivement* pointé ou référencé.
- Le compilateur traduit un appel de fonction (entre autres) par un saut à une adresse mémoire. Mais il est impossible de prévoir à la compilation quel sera le type d'objet pointé ou référencé.

Exercice 2.9 *Construisez un programme d'exemple dans lequel le compilateur ne peut certainement pas connaître la méthode à appeler pour un objet manipulé via un pointeur.*

Solution : la liaison retardée

Afin de retarder la « décision » concernant la méthode à appeler, le compilateur ajoute dans la structure de l'objet un champ servant d'entrée dans une table : la table des méthodes virtuelles, ou TMV (cf. figure 2.9). Cette table est un tableau de *pointeurs de méthodes*.

La table des méthodes virtuelles est créée dès lors qu'une classe possède une méthode virtuelle, et elle contiendra une entrée pour chacune des méthodes virtuelles définies dans la classe ou héritées de ses classes de base.

Le champ caché qui permet de retrouver la table associée à une classe donnée est lui aussi ajouté dès qu'une nouvelle méthode virtuelle est déclarée, il est donc présent dans toutes les classes dérivées.

Ensuite, tout appel d'une méthode virtuelle subira une *indirection* via la TMV de telle sorte que la fonction appelée dépendra d'un champ particulier de la structure pointée ou référencée.

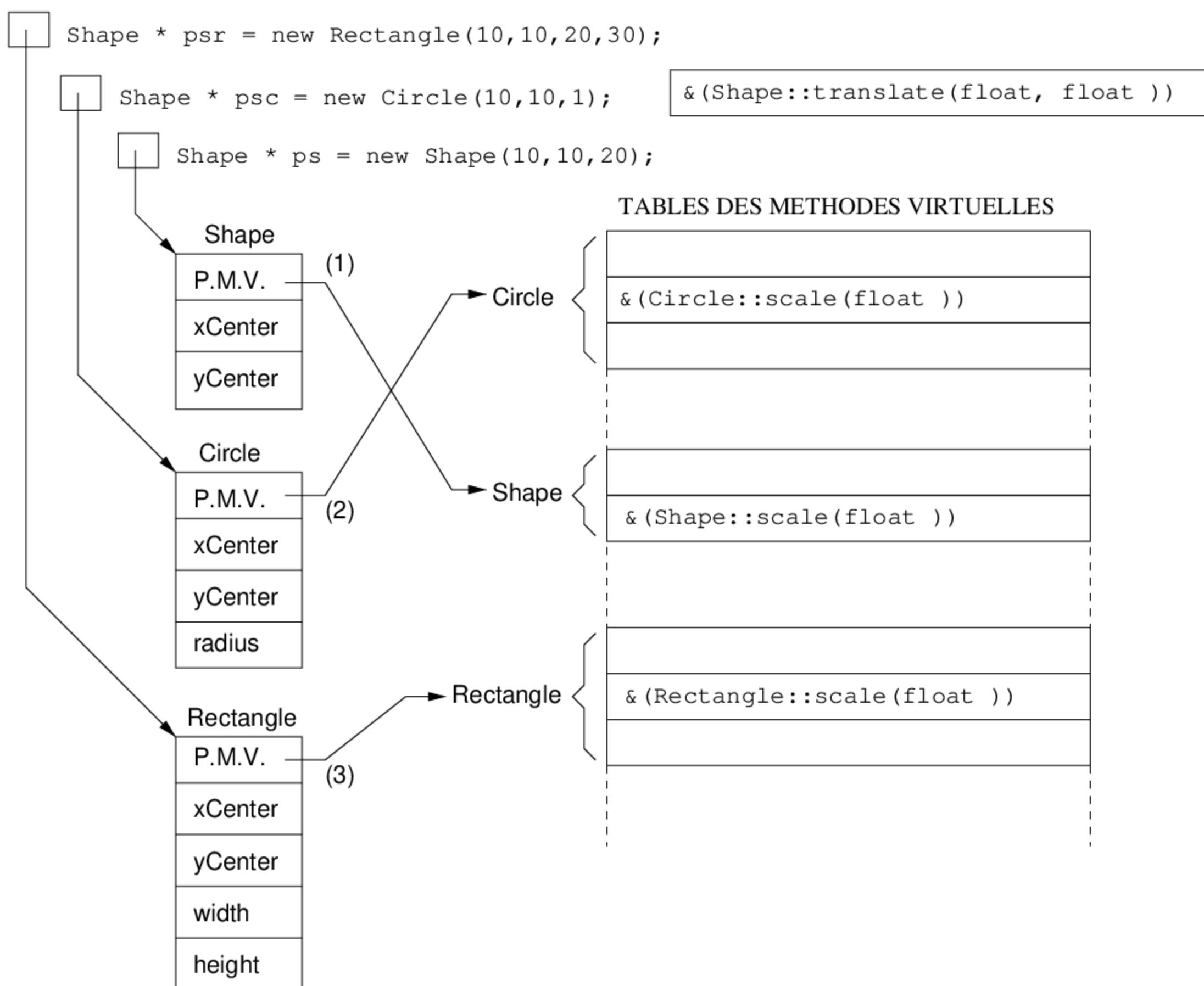


FIGURE 2.9 – Table des méthodes virtuelles.

[...]

D'une certaine façon, si l'approche objet associe des traitements à des données au sein d'une même entité (la classe), on peut dire que le procédé qui vient d'être décrit implémente de manière concrète cette idée puisque chaque instance d'une classe polymorphe « embarque » une référence à la liste des méthodes (virtuelles) qui lui sont associées.

Syntaxe

- Une méthode est déclarée virtuelle dans une classe en y faisant précédé son prototype du mot-clé `virtual`. (Il ne doit pas être répété dans une définition à l'extérieur de la classe. Mais un commentaire est le bienvenu.)
- Lors des redéfinitions dans les classes dérivées, les méthodes correspondantes doivent avoir exactement la même signature, sinon il ne s'agira pas d'une redéfinition mais d'une surcharge ! (Le mot-clé `virtual` y est optionnel. Mais...)



Le polymorphisme d'exécution s'applique uniquement aux pointeurs et aux références.

Exemple

Une exemple de définition de méthode virtuelle est donné dans le listing 2.52.

```
1 class Shape {
2 public:
3     void translate(float x, float y);
4     virtual void scale(float s) { /* Empty ? */ }
5
6 private:
7     float xCenter, yCenter;
8 };
9
10 class Circle : public Shape {
11 public:
12     void scale(float s) {
13         radius *= s;
14     }
15
16 private:
17     float radius;
18 };
19
20 class Rectangle : public Shape {
21 public:
22     void scale(float s) {
23         width *= s;
24         height *= s;
25     }
26
27 private:
28     float width, height;
29 };
```

Listing 2.52 – Exemple de définition d'une méthode virtuelle.

Exercice 2.10 (Question d'examen) Quand peut-on écrire le code ci-dessous ? (Il n'y a aucun piège.)

```
B *pb = new B;
A *pa = pb;
```

2.6.2 Méthodes virtuelles pures et classes abstraites

En Java, l'impossibilité d'utiliser l'héritage multiple (§ 2.7) est en partie compensée par la possibilité de définir des interfaces. Une interface est finalement une liste de services qui sont uniquement *déclarés* comme étant offerts par les classes qui l'*implémentent*. La particularité supplémentaire qui distingue les classes des interfaces en langage Java est qu'une classe peut implémenter plusieurs interfaces alors qu'elle ne peut avoir qu'au plus une classe mère.

D'une manière plus générale, certaines classes ne servent qu'à factoriser un ensemble de fonctionnalités partagées par leurs classes dérivées, sans qu'aucun de ces services n'ait de définition possible dans la classe mère. Il se peut aussi qu'une seule méthode ait cette propriété (cf. `Shape::scale()`). Dans tous les cas, instancier ce type de classe n'a aucun sens puisqu'elle possède au moins une méthode qui n'est pas réellement définie. On parle alors de *classe abstraite*.

Le langage C++ autorise à la fois l'héritage multiple, mais offre aussi la possibilité de laisser une méthode *virtuelle* sans définition dans une classe. On parle alors de *méthode virtuelle pure*.

Syntaxe

Une *méthode virtuelle pure* est déclarée par :

```
virtual TypeRetour nomFonction( Type1 , ... , TypeN ) = 0;
```

Les points importants à retenir sont :

- Un classe possédant (au moins) une méthode virtuelle pure est dite *classe abstraite*.
- Une classe abstraite ne peut pas être instanciée.

Exemple



En cas d'utilisation du polymorphisme, le destructeur de la classe de base devra généralement être déclaré *virtual*.

Exercice 2.11

- a) Écrivez un programme d'exemple définissant des classes, mère et fille, pour lesquelles un destructeur est nécessaire mais ne sera pas déclaré virtuel. Le programme devra provoquer une fuite de mémoire.
- b) Pourquoi un constructeur n'est-il jamais virtuel ?
- c) Que peut-il se passer quand une méthode virtuelle est appelée depuis un constructeur ?

```

1  class Shape {
2      float xCenter, yCenter;
3
4  public:
5      // ...
6      void translate(float dx, float dy);
7      virtual void scale(float factor) = 0;
8  };
9
10 class Circle : public Shape {
11     float radius;
12
13 public:
14     // ...
15     void scale(float factor) {
16         radius *= factor;
17     }
18 };

```

Listing 2.53 – Exemple de définition d'une classe abstraite.

C++11 : override et final

La redéfinition d'une méthode virtuelle dans une classe dérivée peut poser problème si on se trompe, même légèrement, dans la signature de la méthode. En effet, on veut dans ce cas donner un sens nouveau à une méthode héritée alors que, par erreur, on définit une toute nouvelle méthode qui pourrait n'être finalement jamais utilisée. C'est le cas avec la méthode `Bank::deposit` du listing 2.54 qui n'est pas du tout redéfinie dans la classe dérivée `AlternativeBank` puisque c'est une toute autre version qui y est définie (inversion des deux arguments).

En C++11, il est possible d'utiliser l'*identifiant spécial*⁷ `override` afin de spécifier au compilateur

7. Ce n'est en effet pas un mot-clé, mais bien un identifiant de signification spéciale à cet endroit précis.

```

1  class Bank {
2  public:
3      // ...
4      virtual void deposit(int account_number, float amount);
5  };
6
7  class AlternativeBank : public Bank {
8  public:
9      // ...
10     void deposit(float amount, int account_number);
11 };

```

Listing 2.54 – Redéfinition « ratée » d'une méthode virtuelle.

```

1 class Bank {
2 public:
3     // ...
4     virtual void deposit(int account_number, float amount);
5 };
6
7 class AlternativeBank : public Bank {
8 public:
9     // ...
10    void deposit(int account_number, float amount) override;
11 };

```

Listing 2.55 – Redéfinition correcte d'une méthode virtuelle.

que la méthode concernée doit exister avec la même signature *et être déclarée virtuelle* dans une classe de base. L'absence d'une telle méthode se soldera alors par une erreur de compilation.

Il est aussi possible de spécifier qu'une méthode *virtuelle* ne pourra pas être redéfinie dans une classe dérivée, et même qu'une classe ne pourra pas être dérivée. Dans les deux cas, on utilise l'identifiant spécial **final** (cf. listing 2.56).

2.7 Héritage multiple

Comme décrit schématiquement sans plus de précision dans la section 2.3.2 et évoqué en 2.6.2, une classe peut hériter des caractéristiques de plusieurs autres (cf. figure 2.10). Ceci peut apporter quelques ambiguïtés, que nous allons lever ici.

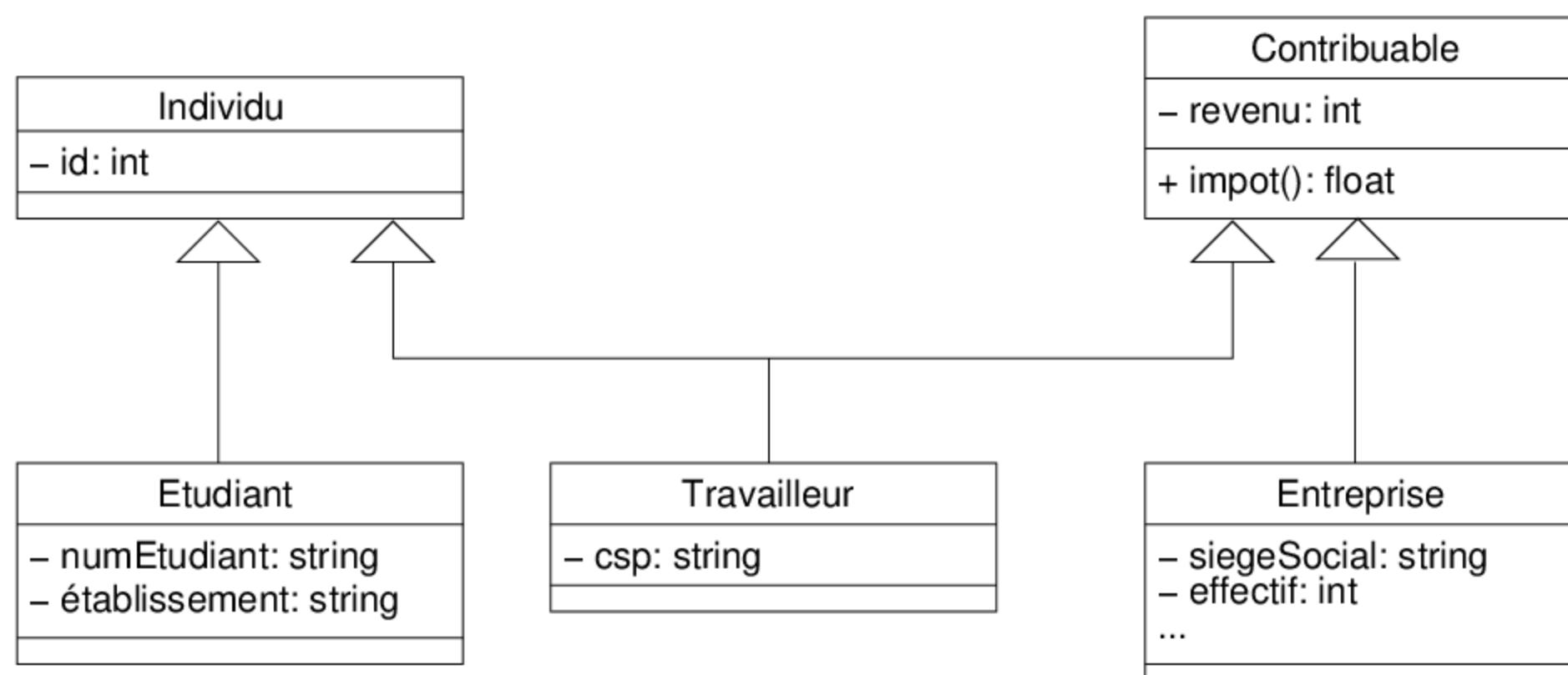


FIGURE 2.10 – Un travailleur est un individu *et* un contribuable.

La syntaxe générale de l'héritage est rappelée ici :

```

1 class Number final {};
2
3 class Complex : public Number { // Error
4                                     // Inheritance is not allowed
5 };
6
7 class Shape {
8 public:
9     virtual void translate(double dx, double dy);
10}
11
12 class Polygon : public Shape {
13 public:
14     void translate(double dx, double dy) final;
15 }
16
17 class Square : public Polygon {
18 public:
19     void translate(double, double); // Error: Redefinition is
20                                     // not allowed
21 };

```

Listing 2.56 – Utilisation de l’identifiant `final`.

```

public           public
class Nom :    private        BaseA,   private        BaseB, ...
private          protected      /*           protected
protected
*/
*   Déclaration des membres
*/
};
```

Le code du listing 2.57 donne une exemple d’utilisation de ce type d’héritage. La situation mise en évidence par cet exemple est la présence de méthodes dont les prototypes sont identiques dans les deux branches d’héritage. Il y a dans ce cas ambiguïté au moment de l’appel de l’une de ces méthodes. Logiquement, c’est une fois encore l’opérateur de résolution de portée `::` qui est utilisé pour lever cette ambiguïté.

Un exemple plus conséquent et concret d’utilisation de l’héritage multiple, qui s’inspire du *design pattern* COMPOSITE est donné par le listing 2.58.

Comme illustré dans le diagramme de la figure 2.11(a), l’héritage multiple d’une même classe n’est pas possible. Si l’intérêt d’un tel héritage semble de toute façon limité, la figure 2.11(b) montre que l’héritage multiple *indirect* d’une même classe peut tout à fait être envisagé et que l’empêcher serait une limite parfois contraignante.

Dans le cas illustré par la figure 2.11(b), les données de la classe A sont dupliquées dans D, et l’opérateur de résolution de portée peut à nouveau être utilisé pour lever les ambiguïtés. Il faut alors préciser le nom de l’une des premières classes mères qui lève cette ambiguïté en remontant vers la racine de l’arbre d’héritage (cf. listing 2.59).

```

1 class A {
2 public:
3     normalize();
4 };
5
6 class B {
7 public:
8     normalize();
9 };
10
11 class C : public A, public B {
12 public:
13     foo() {
14         normalize(); // Error
15     }
16 };
17
18 C c;
19 c.normalize(); // Error
20 c.B::normalize(); // Correct

```

Listing 2.57 – Exemple « syntaxique » d'héritage multiple.

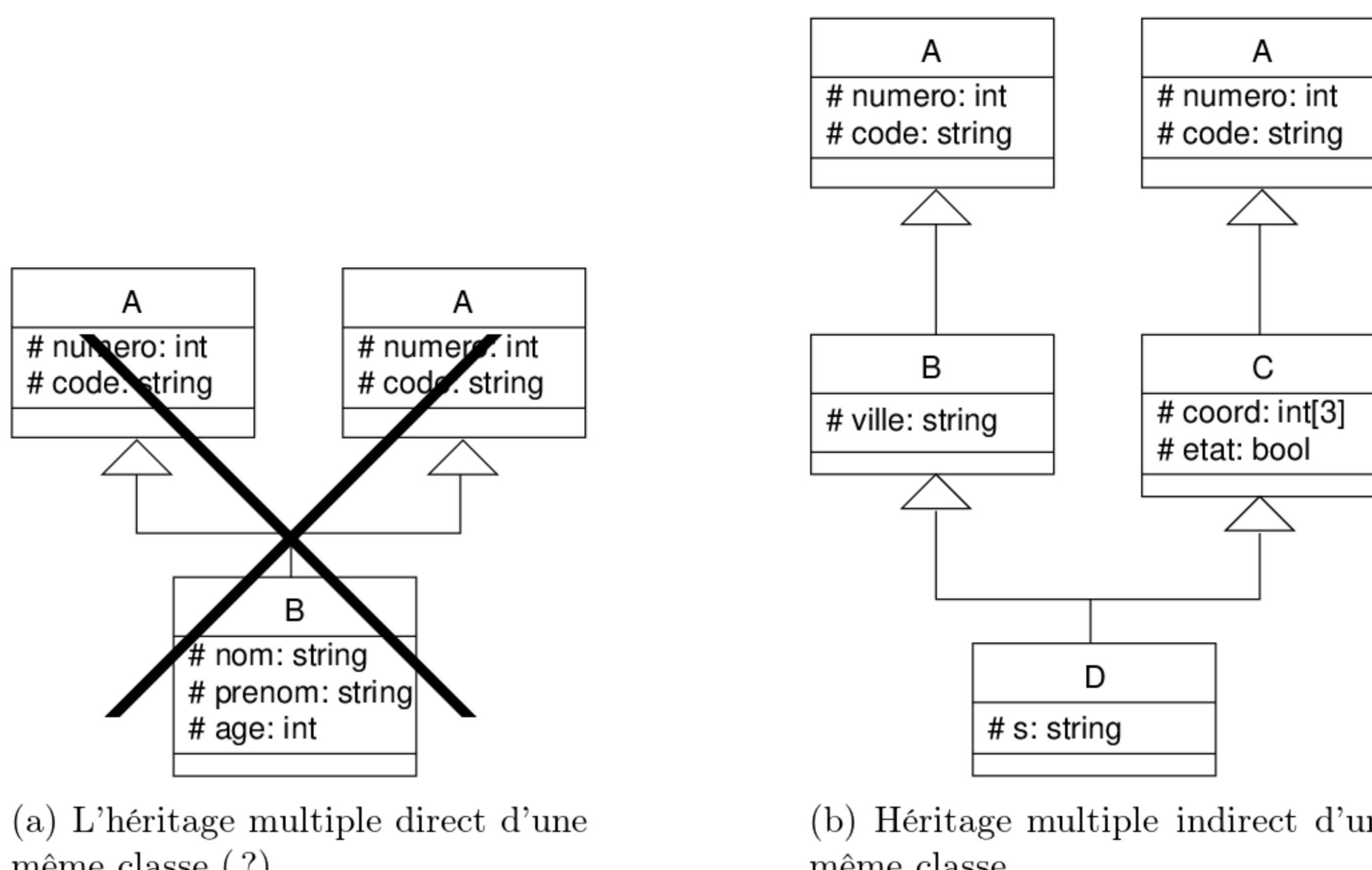


FIGURE 2.11 – Héritage multiple d'une même classe.

```
1 #include <iostream>
2 class Object {
3     virtual ~Object() {};
4 }
5
6 class List {
7 public:
8     bool add(Object *);
9     bool remove(Object *);
10    void apply(unary_function<Object *, void>); // Quid
11 };
12
13 class GeometricShape : public Object {
14 public:
15     ~GeometricShape();
16     virtual string toSVG() = 0;
17 };
18
19 class Circle : public Object {
20 public:
21     Circle(double x, double y, double radius);
22     ~Circle();
23     string toSVG();
24 };
25
26 class Rectangle : public Object {
27 public:
28     Rectangle(double x, double y, double l, double h);
29     ~Rectangle();
30     string toSVG();
31 };
32
33 class Drawing : public Object,
34                 public List,
35                 public GeometricShape {
36 public:
37     Drawing();
38     ~Drawing();
39     string toSVG();
40 };
41
42 int main() {
43     Drawing drawing;
44     drawing.add(new Circle(10.0, 10.0, 1.0));
45     drawing.add(new Rectangle(10.0, 10.0, 100.0, 20.0));
46     std::ofstream("my_drawing.svg") << drawing.toSVG()
47                                     << std::endl;
48 }
```

Listing 2.58 – Exemple « concret » d'utilisation de l'héritage multiple.

```

1 {
2   D d;
3   d.number = 10;    // Error
4   d.C::number = 20 // Correct
5 }
```

Listing 2.59 – Ambiguïté lors d'un héritage multiple indirect d'un même classe.

Exercice 2.12 Écrire les déclarations C++ des classes de la figure 2.11(b).

Il arrive aussi que la duplication des données, dans le cas de l'héritage multiple d'une même classe, ne soit pas souhaitable. C'est par exemple le cas dans le diagramme de la figure 2.12(a). Dans pareille situation, la *dérivation virtuelle* garantit que les données d'une classe B qui sont héritées d'une classe A ne seront pas dupliquées dans une classe dérivée de B qui hérite plusieurs fois et indirectement de A. L'absence de duplication des données peut aussi être schématisée par le diagramme de la figure 2.12(b).

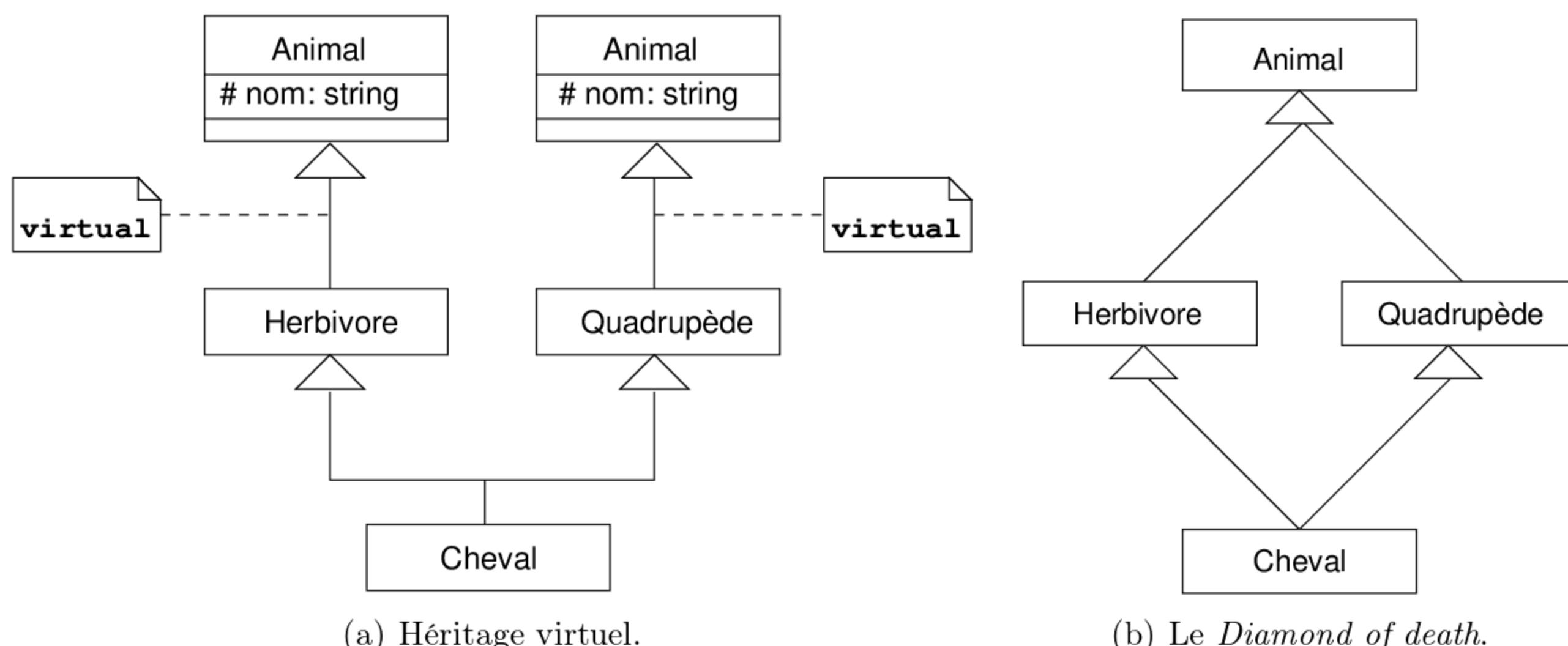


FIGURE 2.12 – Héritage multiple indirect sans duplication.

La syntaxe de la dérivation virtuelle est la suivante :

```

public
class Fille : private Mère, [...] {
protected
};
```

Le listing 2.60, qui correspond aux diagrammes de la figure 2.12, montre qu'il n'y a alors plus d'ambiguïté lorsqu'une donnée de la classe ancêtre est référencée depuis la classe qui en hérite plusieurs fois et de manière indirecte. Dans cet exemple, il n'y a qu'une occurrence de la variable `name` dans la classe `Cheval`.



Ne pas oublier que la surcharge des méthodes est faite avec une portée de classe. Pour bénéficier de la surcharge entre fonctions définies à différents niveaux d'une hiérarchie, on peut avoir recours aux `using-declarations`.

```
1 #include <iostream>
2
3 class Animal {
4     std::string name;
5 };
6
7 class Herbivorous : public virtual Animal {
8     // ...
9 };
10
11 class Quadriped : public virtual Animal {
12     // ...
13 };
14
15 class Horse : public Herbivorous, public Quadriped {
16 public:
17     Horse() {
18         name = "horse"; // No ambiguity!
19     }
20 };
```

Listing 2.60 – Utilisation de l'héritage virtuel.

