



2^e année – Spécialité Informatique

Année 2024 – 2025

Programmation orientée objet en C++

Notes de cours

Sébastien Fourey

Chapitre 1. Du C au C++

Version du 3 septembre 2024



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

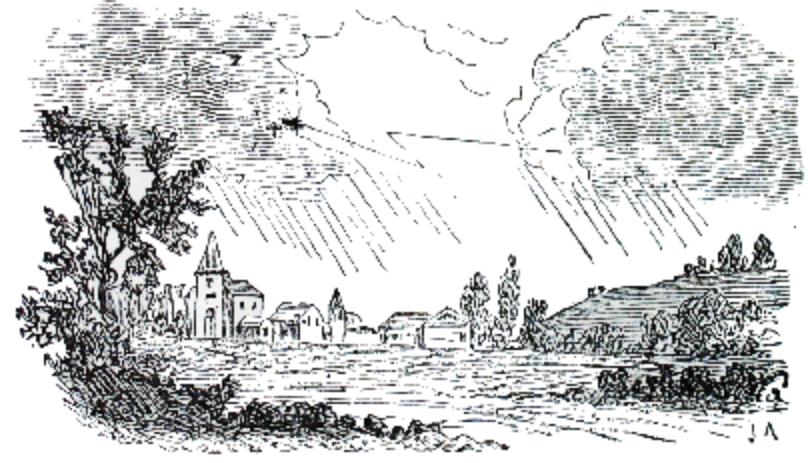


Fig. 73. — Le plus souvent la décharge a lieu entre deux nuages.

Chapitre 1

Du C au C++

Le langage C++ est littéralement un « C augmenté ». Mais les ajouts ne concernent pas uniquement le concept d'objet qui sera traité au chapitre 2. Dans le présent chapitre, on s'intéresse aux particularités du langage qui le distinguent du C mais qui ne sont pas pour autant liées à la *Programmation Orientée Objet* (POO).

1.1 Parmi les nouveautés

1.1.1 Fichiers d'en-têtes

La bibliothèque standard C++ englobe celle du C. Les fichiers d'en-tête du C sont donc utilisables (`<stdio.h>`, `<string.h>`, etc.). Des en-têtes supplémentaires fournissent les déclarations et définitions propres à la bibliothèque C++. Avant la norme de 1998 ils portaient des noms comme `<iostream.h>` ou `<fstream.h>`. Toutefois, suite à l'introduction des espaces de noms (§ 1.2.2) avec la norme [3], des équivalents aux fichiers d'en-têtes de la bibliothèque standard ont été ajoutés pour assurer la compatibilité ascendante. Ainsi :

```
<iostream.h> devient <iostream>,  
<fstream.h> devient <fstream>, etc.
```

L'inclusion de la version `.h` assure un comportement identique à une en-tête sans espace de noms, alors que la version sans `.h` est compatible avec la norme et place les identificateurs dans l'espace de noms standard `std`.

De même, tous les fichiers d'en-têtes de la bibliothèque standard C ont un équivalent, sans `.h` et préfixé par la lettre '`c`', qui place lui aussi les identificateurs dans l'espace de noms `std`.

```
<limits.h> devient <climits>,  
<stdlib.h> devient <cstdlib>,  
<string.h> devient <cstring>,  
<stdio.h> devient <cstdio>, etc.
```

Par exemple, l'en-tête `<cstdio>` déclare la fonction `std::printf` au lieu de `printf` comme le fait `<stdio.h>`.

Remarque 1.1 *Le fichier d'en-tête `<limits>` de la librairie C++ remplit entre autres le rôle*

joué par le fichier `<limits.h>` de la bibliothèque C mais avec une syntaxe propre au C++. Ceci est détaillé au chapitre 8. Il est important de noter que son nom n'est pas préfixé par un '`c`', et que l'en-tête `<climits>` existe aussi.

1.1.2 Commentaires

Comme en C99¹, la séquence de caractères « `//` » permet de mettre tout ce qui la suit, et jusqu'à la fin de ligne, en commentaire (cf. listing 1.1).

```
1 {  
2     float x = 1.414f; // Approximation of the square root of 2.  
3     x = sqrt( 2.0f /* 2 as a float */ );  
4 }
```

Listing 1.1 – Commentaires C++.

1.1.3 Structures

La déclaration de variables de types composés (structures) ne nécessite plus la répétition du mot-clé `struct`. Le nom de la structure suffit.

```
1 struct SComplex {  
2     double re, im;  
3 };  
4 struct SComplex z;  
5 typedef struct SComplex Complex;  
6 Complex r = { 0.0, 1.0 };
```

Listing 1.2 – Structures en C.

```
1 struct Complex {  
2     double re, im;  
3 };  
4 Complex r;
```

Listing 1.3 – Structures en C++.

1.1.4 Types de base

Mis à part les types *référence* (§ 1.5) et booléen (`bool`), ce sont les mêmes que ceux du langage C². De plus, la norme 2011 du langage (C++11) introduit le type `long long` (`int`) que plusieurs compilateurs proposaient déjà. Le tableau 1.1 donne les désignations et domaines de ces types de base.

Les tailles des différents types ne sont pas précisées par la norme et sont dépendantes des implémentations (i.e., architecture ou compilateur). Par contre, la norme précise certaines relations d'ordre entre ces tailles. Dans ce qui suit, \preceq signifie « occupe au plus autant d'espace que » et \doteq signifie « possède la même taille que ».

$\text{sizeof(char)} == 1$ (mais ≥ 8 bits !) $\text{char} \preceq \text{short} \preceq \text{int} \preceq \text{long} \preceq \text{long long}$ $1 \preceq \text{bool} \preceq \text{long}$ $2 \text{ octets} \preceq \text{short}$ $4 \text{ octets} \preceq \text{long}$ $8 \text{ octets} \preceq \text{long long}$ $\text{char} \preceq \text{wchar_t} \preceq \text{long}$ $\text{float} \preceq \text{double} \preceq \text{long double}$

Pour tout type entier T , on a $T \doteq \text{signed } T \doteq \text{unsigned } T$.

1. Pour être précis, en C90 amendement AMD1.

2. La norme C99 introduit les booléens `true` et `false` ainsi que le type `bool` sous forme de macros via l'en-tête `<stdbool.h>`. En C++, ce sont des mots-clés.

Désignation	Mot-clé	Exemples littéraux
booléen	<code>bool</code>	<code>true, false</code>
caractère	<code>char</code>	<code>'a', '\n', '0', 48, 117, 234 ou -122</code>
caractère large	<code>wchar_t</code>	<code>'ab', 2345</code>
entier	<code>short (int), int, long (int) et long long (int)</code>	<code>1, -256, -1234</code>
virgule flottante	<code>float, double et long double</code>	<code>-1.06e-8, 1.4, 255.0f (double par défaut)</code>
énuméré	<code>enum</code>	
indéfini / inexistant	<code>void</code>	<code>void foo(); void *p;</code>
pointeur	<code>T *</code>	
tableau	<code>T []</code>	<code>{true, false, true}</code>
référence	<code>T &</code>	

TABLE 1.1 – Types de base du langage.



La taille des types *integral* (c.-à-d. entiers, dont `char`, et booléens) dépend de l'implémentation. Généralement, c'est 4 pour un `int` sur une architecture 32 ou 64 bits, mais ce n'est pas du tout une obligation.

Pour connaître les intervalles de valeurs possibles des types numériques, il est possible d'utiliser les constantes définies dans les fichiers d'en-têtes de la bibliothèque C. En effet :

```
limits.h définit INT_MAX, UINT_MAX, etc.  
float.h définit FLT_MAX, DBL_MAX, DBL_EPSILON, etc.
```

En C++, on pourra utiliser un code ressemblant à celui du listing 1.4 sur lequel nous reviendrons dans la section 8.1.

```
1 || int iMax = std::numeric_limits<int>::max();
```

Listing 1.4 – La plus grande valeur possible pour un entier (`int`) en C++.

La syntaxe précédente est sans doute obscure pour qui ne connaît pas encore :

- les espaces de noms ;
- les notions de classe, de modèle de classe et de spécialisations ;
- la notion de méthode statique.

Le cas du pointeur nul

En langage C, la constante pré-processeur `NULL` est parfois définie de la manière suivante :

```
1 ||#define NULL ((void*)0)
```

C'est tout à fait convenable en C puisque les pointeurs sur `void` sont convertis implicitement, dans ce langage, en tout type de pointeur. Mais ce n'est pas le cas en C++. Ainsi, le code C++ du listing 1.5 provoque une erreur de compilation car le pointeur `NULL` (de type `void*`) ne peut pas être converti implicitement en un `const char*`.

```

1 // Possible definition in C
2 // #define NULL ((void *)0)
3
4 void foo(const char *) {
5     // ...
6 }
7
8 int main() {
9     foo(NULL);
10}

```

Listing 1.5 – Code erroné du fait de la définition de NULL.

```

1 // Possible definition of NULL
2 #define NULL 0
3
4 void foo(int i) {
5 }
6 void foo(const char *) {
7 }
8
9 int main() {
10    foo(0);           // Call foo(int)
11    foo(NULL);        // Same thing!
12    foo((const char *)0); // Finally!
13}

```

Listing 1.6 – Utilisation malheureuse du pointeur NULL.

Le C++ apporte une solution : la valeur constante 0 est une valeur constante valide pour tout type de pointeur. Il faut donc préférer son utilisation à celle de la macro NULL.

C++11 : le mot-clé nullptr

Le double sens de la constante littérale 0 (à la fois entier et pointeur nul) pose un problème lors de l'appel d'une fonction surchargée. En effet, pour peu que la macro NULL soit définie comme suit :

```
1 #define NULL (0)
```

le code du listing 1.6 n'aboutira pas au résultat escompté (lignes 10 et 11).

Pour remédier à cet inconvénient, le C++11 introduit le mot-clé `nullptr` qui est une constante de type `nullptr_t`, pour laquelle une conversion implicite en n'importe quel type de pointeur est possible, ainsi qu'en la valeur booléenne `false`. Comme ce sont là les seules conversions possibles de cette constante, elle permet de lever toute ambiguïté. Ainsi, dans le listing 1.6, l'utilisation de `nullptr` (ligne 12) en lieu et place de NULL provoquera l'appel de la seconde fonction `foo()`.

```

1 int main() {
2     int i, j;
3     cin >> i >> j;
4     unsigned long n;
5     cin >> n;
6     i = (int) n;
7 }
```

Listing 1.7 – Déclaration de variables au sein d'un bloc.

1.1.5 Position des déclarations de variables

En C++ comme en C99, les déclarations de variables ne se font pas nécessairement en début de bloc avant toute instruction, mais elle peuvent se faire *entre* deux instructions (cf. listing 1.7). Dans tout les cas, la portée d'une variable s'étend de la fin de sa déclaration jusqu'à la fin du bloc.

1.1.6 Fonctions sans arguments

Il existe une différence, qui peut paraître anodine, entre les fonctions sans arguments déclarées en C par rapport au C++. En effet, le code C ci dessous :

```
float sum();
```

signifie « il existe une fonction de nom `sum` retournant un `float` et qui accepte des arguments, sans que le nombre et le type des arguments ne soient précisés dans cette déclaration. Le prototype d'une fonction sans arguments s'écrit, toujours en C :

```
float random(void); // This is C (and not C++)
```

En C++, par contre, le prototype ci-dessous est, sans aucune ambiguïté, celui d'une fonction qui n'accepte *aucun* argument.

```
float random(); // Two distinct meanings between C and C++
```

Finalement, on pourra bannir en C++ l'utilisation de `void` comme synonyme de l'absence d'arguments [10].

1.1.7 Valeurs d'arguments par défaut

Les paramètres de fonction peuvent avoir des valeurs par défaut. Le type est vérifié lors de la déclaration, qui doit être *unique*, et la valeur par défaut est évaluée au moment de l'appel de la fonction. La valeur par défaut peut être une variable globale ou une expression faisant intervenir des variables globales. Attention : la valeur par défaut doit être précisée dans la déclaration de la fonction mais pas dans sa définition (cf. listing 1.8).

Règle Si un argument possède une valeur par défaut, tous ceux qui le suivent dans la déclaration aussi. [...]

```

1 // Compute a price "All Taxes Included"
2 // (declaration in "taxes.h")
3 float priceATI(float price, float vat = 18.6f);
4
5 // (definition in "taxes.cpp")
6 float priceATI(float price, float vat = 18.6f) { // Error!
7     return price * (1 + vat / 100.0f);
8 }
```

Listing 1.8 – Arguments de fonction par défaut.

1.1.8 Boucle `for` et portée des déclarations

```

1 int a, b;
2 for (int i = 0; i < 10; ++i) {
3     // Body of the loop
4     // ...
5 }
```

Listing 1.9 – Déclaration de variable dans une instruction `for`.

Dans le cas du listing 1.9, la variable `i` n'est visible que dans le corps de la boucle. Dans ce corps, elle masque bien entendu toute variable de même nom définie dans le bloc de niveau supérieur.

1.1.9 Définitions : objets et lvalues

Dans la suite, on conviendra d'appeler :

- **objet**, une zone contiguë de mémoire (\neq instance de classe) ;
- **lvalue**, une expression faisant référence à un objet en mémoire (et dont on peut récupérer l'adresse avec l'opérateur `&`) ;
- **lvalue modifiable**, une expression faisant référence à un objet non constant ;
- **rvalue**, une expression qui n'est pas une *lvalue*, ou un objet temporaire qui n'existe que durant l'évaluation d'une expression.

1.2 Portée et espaces de noms

1.2.1 Opérateur de résolution de portée

Rappel En C++ les déclarations « volantes » sont possibles.

Lorsqu'une variable définie localement à un bloc masque une variable définie en dehors de ce bloc (p. ex. une variable globale), l'opérateur de *résolution de portée* `::` utilisé sans préfixe permet de faire référence à la variable masquée (cf. listing 1.10).

```

1 int i;
2 void someFunction() {
3     int x = i; // global variable 'i'
4     cout << "x=" << x << endl;
5     int i = 0; // Declaration of a local 'i' (masking)
6     i = 10 * i; // Set and use the local 'i'
7     ::i = 100; // Set the global 'i'
8 }
```

Listing 1.10 – Résolution de portée.

1.2.2 Espaces de noms

Toute portion de code de niveau global peut être placée dans un espace de noms. Il peut s’agir d’une simple déclaration comme d’un fichier complet. La syntaxe est donnée par le listing 1.11. Dans cet exemple, l’identificateur `inputMatrix` situé entre les accolades est alors défini dans l’espace de noms `my_lib_name`. Cela revient à dire que d’un point de vue global son nom est en fait `my_lib_name::inputMatrix`.

```

1 namespace my_lib_name {
2     // Declarations of variables
3     // Definitions of types (e.g. typedef)
4     // Definitions of classes, functions, etc.
5     // For example :
6     void inputMatrix() { /* ... */ };
7 }
```

Listing 1.11 – Code placé dans un espace de noms.

Afin d’alléger la syntaxe, il est possible d’importer un symbole depuis un espace de noms vers la portée courante à l’aide du mot-clé `using`. L’importation est alors valable pour le reste de l’unité de compilation ou du bloc courant, comme dans l’exemple du listing 1.12. On parle dans ce cas de *using-declaration*.

```

1 #include <iostream>
2
3 namespace mathematics
4 {
5     const double e = 2.71828182845904523536;
6 }
7
8 using mathematics::e;
9
10 std::cout << e << std::endl;
```

Listing 1.12 – Import d’un symbole dans l’espace de noms global.

Il est aussi possible d’importer *tout* le contenu d’un espace de noms dans l’espace courant à l’aide de la directive `using namespace` selon le modèle du listing 1.13 pour l’utilisation des identifiants `cout` et `endl` de l’espace de noms `std`. On parle alors de *using-directive*.

```

1 #include <iostream>
2 namespace mathematics
3 {
4     const double e = 2.718;
5     const double pi = 3.1416;
6 }
7
8 using namespace std;
9
10 int main() {
11     cout << mathematics::e << endl;
12 }
```

Listing 1.13 – Import « massif » de symboles dans l'espace de noms global.

```

1 #include <iostream>
2 namespace Earth {
3     float acceleration = 9.81;
4 }
5 namespace Moon {
6     float acceleration = 1.622;
7 }
8 int main(int , char *[])
{
9     using namespace Earth;
10    using namespace Moon;
11    std::cout << acceleration << std::endl; // Ambiguous!
12    return 0;
13 }
```

Listing 1.14 – Symbole ambigu suite à deux *using-directives*.

L'import est valable pour le reste du bloc dans lequel la directive (c.-à-d. `using namespace`) ou la déclaration (`using`) est placée. Si elle est placée en dehors de tout bloc, l'import est valable pour la suite de l'unité de compilation.

Conflits Dans certaines situations, des conflits de noms peuvent résulter d'imports réalisés par des *using-declarations* ou des *using-directives*. S'ensuivent alors des ambiguïtés au moment de l'utilisation des symboles (cf. listing 1.14). Toutefois, un symbole déclaré classiquement dans un bloc ou bien via une *using-declaration* est prioritaire sur le même symbole importé via une *using-directive* (cf. listing 1.15).

On tiendra compte de la mise en garde suivante :



La directive `using namespace` ne devrait jamais être utilisée dans les fichiers d'en-têtes en dehors de tout bloc. Pourquoi ?

```

1 #include <iostream>
2 namespace Earth {
3     float acceleration = 9.81;
4 }
5 namespace Moon {
6     float acceleration = 1.622;
7 }
8 namespace Sun {
9     int position = 0;
10}
11int main(int , char *[])
12{
13    using Moon::acceleration;
14    using namespace Earth;
15    std::cout << acceleration << std::endl; // Moon: 1.622
16
17    int position = 20;
18    using namespace Sun;
19    std::cout << position << std::endl; // 20, not 0
20    return 0;
21}

```

Listing 1.15 – Priorité d'un (*using-*)*declaration* sur une *using-directives*.

Un dernier exemple est donné dans le listing 1.16.

Quelques remarques finales :

- Les espaces de noms peuvent être imbriqués, avec modération. [...]
- Un espace de noms sans nom (ou anonyme) est toujours local à une unité de compilation. Il s'agit là de la version C++ des déclarations **static** de variables globales et de fonctions (cf. section 1.3.4).
- Les en-têtes <c*****> fournies avec le compilateur GNU g++ ne placent pas les symboles de la bibliothèque C de manière exclusive dans l'espace de noms **std**. Elles les placent aussi dans l'espace de nom global. C'est une liberté laissée par la norme.
- Concrètement, les bibliothèques du compilateur précédemment cité réalisent l'import inverse : espace de noms global vers l'espace **std**, à l'aide de *using-declarations*. La fonction **printf** est ainsi disponible à la fois dans l'espace global *et* dans l'espace **std**.

1.3 Portée et compilation séparée (C et C++)

1.3.1 Compatibilité des fichiers d'en-têtes

Les fichiers d'en-têtes standards du C++ (et pas du C, comme par exemple **iostream.h**) qui existaient dans les versions précédant la norme de 1998 sont conservés et permettent de régler le problème de la compatibilité ascendante. En effet, la différence entre un fichier inclus comme <**iostream**> et <**iostream.h**> est que ce dernier importe dans l'espace de noms global tous les symboles qui, dans <**iostream**>, se trouvent uniquement dans l'espace de noms **std**.

```

1 namespace Moon
2 {
3     const double acceleration = 1.622;
4     double mass2weight(double mass) {
5         return mass * acceleration;
6     }
7     // ...
8 } // namespace Moon
9 namespace Earth
10 {
11     const double acceleration = 9.81;
12     double mass2weight(double mass) {
13         return mass * acceleration;
14     }
15 } // namespace Earth
16 double my_weight = Earth::mass2weight(80.0);
17 double acc = Moon::acceleration;
18 using namespace Moon; // Or [...]
19 accel = acceleration;

```

Listing 1.16 – Exemple d'utilisation des espaces de noms.

Ainsi, la directive d'inclusion suivante :

```
#include <iostream.h>
```

est équivalente à

```
#include <iostream>
using namespace std;
```

Le fichier `<iostream>` étant semblable au listing 1.17.

```

1 #ifndef _GLIBCXX_IOSTREAM
2 #define _GLIBCXX_IOSTREAM 1
3 // ... Several include directives
4 namespace std
5 {
6     // Declarations [...]
7 }
8 #endif /* _GLIBCXX_IOSTREAM */

```

Listing 1.17 – Fichier `<iostream>`

1.3.2 Portées

Remarque 1.2 Une variable définie dans un fichier (c.-à-d.une unité de compilation) en dehors de toute fonction, bloc ou classe a une portée de fichier.

Remarque 1.3 *Un objet doit être défini au plus une fois dans un même programme. Ainsi, le code des listings 1.20 et 1.19 est incorrect. En effet, la variable g est définie deux fois et provoquera une erreur lors de l'édition de liens.*

```
1 double mass2weight(double);
```

Listing 1.18 – Gravity.h

```
1 #include "Gravity.h"
2
3 // ...
4
5 double g = 9.81;
6
7 double mass2weight(double) {
8     // ...
9 }
```

Listing 1.19 – Gravity.cpp

```
1 #include <iostream>
```

```
2 #include "Gravity.h"
3 using namespace std;
```

```
4
5 double g = 9.81;
```

```
6
7 int main() {
8     cout << mass2weight(80);
9     cout << endl;
10    return 0;
11 }
```

Listing 1.20 – Rocket.cpp

1.3.3 Mot-clé extern

Le mot-clé `extern` permet de déclarer, *sans le définir*, un objet défini dans une autre unité de compilation. Il permet, intuitivement, d'exprimer l'*existence* dans le programme complet d'un objet d'un type donné. L'exemple de la remarque 1.3 peut être compilé sans erreur si on remplace dans `Rocket.cpp` :

```
5 | double g = 9.81; // Declaration and definition.
```

par

```
5 | extern double g; // Declaration only.
```

Mais cette correction reste généralement insatisfaisante. Il est en effet logique dans ce cas que la déclaration `extern` soit faite dans le fichier `Gravity.h`, et uniquement dans celui-là.

1.3.4 Mot-clé static et espace de noms anonyme

Pour rappel, en langage C, le mot-clé `static` appliqué à un objet ayant une portée de fichier rend cet objet « réellement » local. Mais...

« Évitez ce mot-clé en dehors des fonctions et des classes. »

Bjarne Stroustrup, [9]

En effet, en C++ une construction beaucoup plus flexible existe : les espaces de noms anonymes. On pourra par exemple utiliser le code du listing 1.21 pour rendre la variable `g` réellement locale au fichier `Rocket.cpp` (comme l'aurait fait le mot-clé `static`). Dans ce cas, `g` ne peut plus entrer en conflit avec un symbole de même nom défini dans une autre unité de compilation.

```

1 #include <iostream>
2 #include "Gravity.h"
3 using namespace std;
4 namespace {
5     double g = 9.81;
6 }
7 int main() {
8     cout << mass2weight(80);
9     cout << endl;
10    return 0;
11 }
```

Listing 1.21 – Espace de noms anonyme.

Cette syntaxe est préférable car plus polyvalente : contrairement au mot-clé `static`, un espace de nom anonyme peut aussi s’appliquer à un type défini par l’utilisateur (classe, structure, `typedef`) pour rendre ce type local lui aussi.

1.4 Surcharge de fonctions

Principe Un même nom peut être utilisé pour définir plusieurs fonctions. Dans ce cas, chaque fonction se distingue par le type de ses arguments. Attention, le type retourné n’est donc pas pris en compte.

1.4.1 Exemples de déclarations

```

1 double inverse(double x);
2 Complex inverse(Complex z);
3
4 int abs(int i);
5 double abs(double x);
6 double abs(Complex z);
7
8 char succ(char c);
9 int succ(int i);
```

Listing 1.22 – Surcharge de fonctions : déclaration.

1.4.2 Appel de fonction surchargée

Au moment de l’appel d’une fonction surchargée, le compilateur doit choisir la bonne fonction. Il le fait en examinant les possibilités suivantes, classées par ordre de priorité :

- correspondance exacte des types ;
- application possible de la promotion (p. ex. : `short → int`) ;
- application possible des conversions standards (p. ex. : `int ↔ double`) ;
- application de conversions définies par le programmeur ;

- utilisation des points de suspension (...).

La première règle qui s'applique est utilisée, mais il peut arriver qu'une des cinq règles laisse plusieurs choix! [...] Dans ce cas, la compilation échoue et les règles qui suivent ne sont pas examinées.

Ainsi, en supposant que les déclarations du listing 1.22 sont présentes, on peut examiner le code suivant :

```

1 float s = 10.0f;
2 double w = 2.0;
3 float z;
4
5 z = inverse(w); // Call inverse(double)
6 z = inverse(s); // Cast s as double,
7 // then call inverse(double)
8 int i = succ(z); // Cast z as int, then call succ(int)

```

Listing 1.23 – Surcharge de fonction : l'appel.



La résolution de surcharge ne se fait qu'au sein d'une même portée (corps de fonction, classe, bloc) à l'exception des espaces de noms pour lesquels il est possible d'intervenir à l'aide de directives ou déclarations *using*.



1.5 Références

Les références constituent une notion nouvelle et très utile du langage. En termes simples, une référence est un synonyme pour un objet, on parle aussi parfois d'*alias*. Un des intérêts majeurs de cette notion est l'allègement de la syntaxe du passage d'arguments par adresses (§ 1.5.3).

1.5.1 Déclaration et initialisation dans un bloc

Étant donné un identificateur de type T , la syntaxe de déclaration d'une référence est la suivante :

T y;
T & x = y;

Dans ce cas l'opérateur n'initialise pas une *variable* x avec la valeur de y mais initialise x comme une *référence* à y. Les points suivants sont importants et à retenir :

- Une référence déclarée dans un bloc *doit* être initialisée lors de sa déclaration.
- Une référence déclarée et initialisé de cette façon est *définitive*.
- L'adresse d'une référence est celle de la variable référencée.

```

1 #include <iostream.h>
2
3 void linear_transform(float & x, float s) {
4     x *= s;
5 }
6
7 int main() {
8     float y = 2.0, z = 10;
9     cin >> y;
10    linear_transform(y, z);
11    cout << "y = " << y << endl;
12    return 0;
13 }
```

Listing 1.25 – Passage par adresse en C++.

Exemple

accel:	g:
	9.81
alpha:	0.6e-9
pg:	&g

```

1 {
2     double g = 9.81;
3     double & accel = g;
4     double alpha = 0.6e-9;
5     double * pg;
6     accel = 1.0; // g is 1.0
7     pg = &accel; // value of pg ?
8 }
```

Listing 1.24 – Déclaration et initialisation de références dans un bloc.

1.5.2 Références comme arguments de fonctions

Utilisées comme paramètres de fonctions, les références offrent les avantages du « passage par adresse » possible en C à l'aide de pointeurs, mais avec une notation allégée. Dans la fonction `trans_lineaire()` du listing 1.25, aucun espace n'est réservé sur la pile pour y stocker la variable `x` (comme c'est le cas, rappelons-le, pour `s` qui est une *copie* de la variable `z` donnée en argument). La référence `x` s'utilise comme une *lvalue* normale, et toute opération sur `x` porte en fait sur le `y` de la fonction `main`.

Exercice 1.1 Écrire l'équivalent en C du listing 1.25.



A l'appel, le passage par référence est moins explicite que le passage par adresse réalisé grâce aux pointeurs (cf. ligne 10 du listing 1.25). Attention donc aux noms des fonctions, et pensez aux commentaires !

1.5.3 Référence comme type de retour d'une fonction

Il est possible pour une fonction de retourner une référence. C'est intéressant car un appel de fonction peut ainsi être une *lvalue*. Cela permet aussi à une fonction de transmettre à la fonction appelante une référence qui a été reçue comme argument (cf. la surcharge de l'opérateur << par une fonction à deux arguments). Seule la syntaxe de la déclaration de fonction est différente par rapport à un type de retour classique (cf. listing 1.26). L'instruction **return** ne change pas et sert à désigner la *lvalue* sur laquelle porte la référence qui doit être retournée.

```

1 int & max(int & x, int & y) {
2     return (x >= y) ? x : y;
3 }
4
5 int main() {
6     int a = 10, b = 20;
7     max(a, b) += 10; // ?
8 }
```

Listing 1.26 – Fonction dont le type de retour est une référence.

Dans le cas du listing 1.26, les variables **x** et **y** sont elles mêmes des références, mais il pourrait s'agir par exemple de cellules particulières d'un paramètre de type tableau d'entiers. 

Première illustration Saisie clavier et opérateur **>>**.

L'opérateur **>>** surchargé pourrait avoir la déclaration suivante³ :

```
| ifstream & operator>>( ifstream & in, int & i );
```

Par conséquent, l'instruction de la ligne 2 suivante

```

1 Quaternion a, b;
2 cin >> a >> b;
```

est équivalente à :

```
| operator>>( operator>>( cin, a ), b );
```

Remarque 1.4 *L'opérateur >> est associatif de gauche à droite.*

Seconde illustration Premier élément non nul d'un tableau.

```

1 int & first_non_zero(int array[], int n) {
2     for (int i = 0; i < n; i++) {
3         if (array[i]) {
4             return array[i];
5         }
6     }
7     throw "No non-zero value in array";
8 }
```

Listing 1.27 – Référence sur le premier élément non nul d'un tableau.

3. La norme le définit en fait comme une méthode (cf. section 4.1).

Les instructions `return` de cet exemple renvoient une référence, pas une valeur. On peut donc écrire :

```
|| first_non_zero( an_array , 10 ) = 0;
```



On ne retourne jamais une référence à un objet local à une fonction, sauf si c'est une variable « `static` ». (Pourquoi ?)

1.5.4 Références comme données membres d'une classe

Ce sujet sera logiquement abordé une fois que les classes en C++ l'auront été.

1.5.5 Références constantes et objets temporaires

Il n'est pas possible d'initialiser une référence avec une constante littérale. Le code suivant n'est donc pas autorisé :

```
|| int & i = 45; // Forbidden , 45 is not an lvalue
```

Toutefois, l'initialisation d'une référence déclarée constante est dans ce cas possible :

```
|| const int & i = 45; // Correct
```

En effet, la ligne précédente peut être interprétée comme suit :

```
1 || int temporaryAndAnonymous = 45;
2 || const int & i = temporaryAndAnonymous;
```

Il y a donc création d'un objet temporaire et anonyme. Qui plus est, l'objet référencé peut alors être de type différent, et les règles de conversion s'appliquent. Ainsi,

```
1 || int i = 10;
2 || const double & phi = i;
```

est équivalent à

```
1 || int tmp1 = 10;
2 || double tmp2 = tmp1; // Implicit cast
3 || const double & phi = tmp2;
```

1.5.6 Objets temporaires

Soit la déclaration :

```
|| Complex z = x + v * r;
```

Un objet (structure ou instance de classe) temporaire est créé pour stocker le résultat de l'expression `v * r`. Un tel objet n'est pas une *lvalue* et ne peut donc pas être utilisé pour initialiser une référence, sauf si c'est une référence constante (cf. listing 1.28).

```

1 void f(double & x) { /* [...] */ }
2 void g(const double & x) { /* [...] */ }
3
4 double x = 1.0, y = 2.0;
5 f(x + y); // Error
6 f(3.14159); // Error
7 g(9.81); // OK

```

Listing 1.28 – Références à des objets temporaires.

Portée des objets temporaires

« Une variable temporaire utilisée dans l’initialisation d’une référence [constante] persiste jusqu’à la fin de la portée de sa référence. »

Bjarne Stroustrup, [9]

1.6 Allocation dynamique sur le tas

Les fonctions `malloc`, `calloc`, `realloc` et `free` de la bibliothèque C⁴ peuvent toujours être utilisées en C++ ; mais on doit leur préférer l’utilisation des deux nouveaux mots-clés `new` et `delete` dont la syntaxe est présentée dans cette section. En effet, comme il sera expliqué dans la section 2.1.4, le mot clé `delete` provoque, dans le cas d’une instance de classe, l’appel d’une méthode particulière appelée *destructeur*. La fonction `free` de la bibliothèque C n’offre pas cette possibilité.

1.6.1 Allocation

Pour toute dénomination de type T , l’expression :

`new T;`

alloue `sizeof(T)` octets et a pour valeur, en cas de succès, l’adresse du premier octet sous la forme d’un pointeur *typé* (de type T^*). Elle est implémentée à l’aide de l’opérateur pouvant être surchargé « `void *operator new(size_t)` ».

D’autre part, l’expression

`new T[n];`

alloue la mémoire nécessaire pour n objets de type T et vaut l’adresse du premier objet (si tout s’est bien passé). Bien entendu, n peut être une constante mais aussi une expression à valeur entière. Attention : il n’y a pas d’initialisation automatique de la mémoire allouée (sauf pour un type objet auquel cas le constructeur par défaut est appelé pour chacune des instances créées). Mais il est possible de demander l’initialisation à zéro, pour les types de base, par la syntaxe suivante :

`new T[n]() ;`

4. En-tête `<cstdlib>`.

Exemples

```

1 char * pc = new char [255];
2 int * pi;
3 double * alpha;
4 double * e = new double;           // *e is not initialized
5 int * array = new int [10]();     // initialization with zeros
6 pi = new int [314];
7 alpha = new int [10];            // Error (alpha is not int*)

```

Listing 1.29 – Exemples d’allocations dynamiques.

En cas d’erreur

Si l’allocation est impossible, l’opérateur `new` soulève une exception standard dont le type est `std::bad_alloc` (cf. chapitre 6). Cependant, deux alternatives à la capture de l’exception `bad_alloc` sont offertes : l’installation d’un gestionnaire d’erreur ou l’utilisation de la valeur spéciale 0.

Installation d’une fonction gestionnaire d’erreur Si l’allocation est impossible (?), `new` peut faire appel à une fonction utilisateur, précisée à l’aide de la fonction `set_new_handler()` déclarée dans l’en-tête `<new>` (cf. listing ci-dessous).

```

1 #include <cstdlib>
2 #include <new>
3 void memory_error() {
4     std :: exit (-1); // Rough
5 }
6
7 int main() {
8     std :: set_new_handler(&memory_error);
9     int * pi = new int [0xFFFFFFF];
10    std :: exit (0);
11 }

```

Listing 1.30 – Gestion des erreurs d’allocation dynamique.

Utilisation de l’allocator `nothrow` L’allocator `nothrow` permet de revenir au comportement que l’opérateur `new` avait avant l’introduction des exceptions dans le langage, à savoir retourner 0 en cas d’échec. (A l’instar des fonctions `calloc()` et `malloc()` de la bibliothèque C.) La syntaxe est la suivante :

```
new(std::nothrow) T;
new(std::nothrow) T[n];
```

Remarque 1.5 Une particularité importante de l’opérateur `new` le distingue des deux fonctions `malloc` et `calloc` : son type de retour. En effet, la fonction `malloc` retourne un pointeur « `void*` » qui peut donc être converti implicitement, en C, en tout type de pointeur. En C++, l’expression `new T[10]` retourne un pointeur de type `T*`. Il est donc impossible, sauf en utilisant

```

1 struct Complex {
2     // ...
3 };
4
5 void foo() {
6     Complex * pz = new Complex;
7     int n;
8     std::cin >> n;
9     float * array = new float [n];
10
11    delete pz;
12    delete [] array;
13    delete pz; // ?
14 }
```

Listing 1.31 – Exemples d’allocations dynamiques et les désallocations respectives.

une conversion explicite⁵, d’allouer un tableau d’entier pour le stocker dans un pointeur de flottants !

1.6.2 Désallocation

L’instruction

delete pointeur;

libère la mémoire occupée par l’objet pointé, qui a été précédemment alloué par **new**. Elle est implémentée à l’aide de l’opérateur **void operator delete(void *)**.

Remarque 1.6 On notera que :

- Les instructions « **delete 0;** » et « **delete nullptr;** » sont valides (et sans effet).
- L’effet de « **delete p;** » est indéfini si **p** n’est pas valide (par exemple si **p** a déjà été désalloué).
- Le cas d’un pointeur sur une instance de classe est particulier (cf. section 2.1.4).

De plus, l’instruction

delete[] pointeur;

libère la mémoire occupée par un tableau d’objets précédemment alloué par **new[]**. Le cas d’un tableau d’instances de classes est aussi un cas particulier qui sera précisé dans la section 2.1.4.

1.6.3 Exemple

Des exemples d’allocations (dynamiques) et désallocations sont donnés dans le listing 1.31.

5. À l’aide de l’opérateur **reinterpret_cast<>()**.

Remarque 1.7 `new` et `delete` sont des opérateurs standards qui peuvent être surchargés par l'utilisateur (§ 2.2).



Ne pas mélanger `new/delete` avec `malloc()/free()`. Il faut choisir...



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.

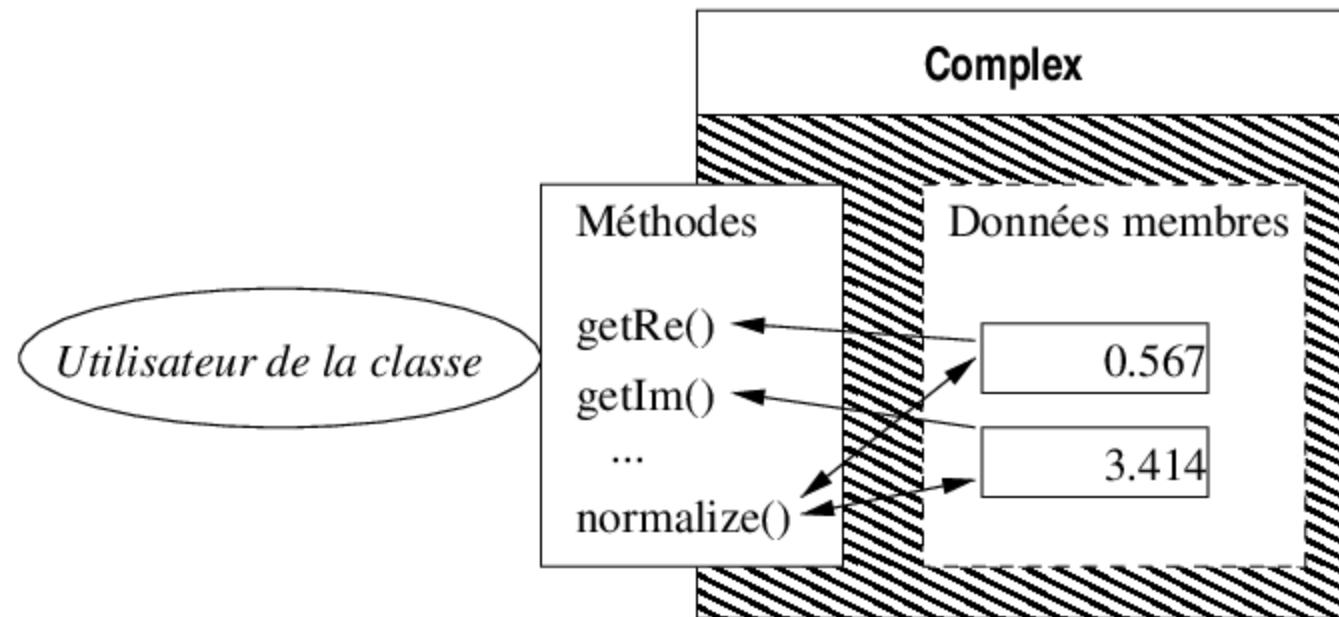


FIGURE 2.2 – Encapsulation

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.

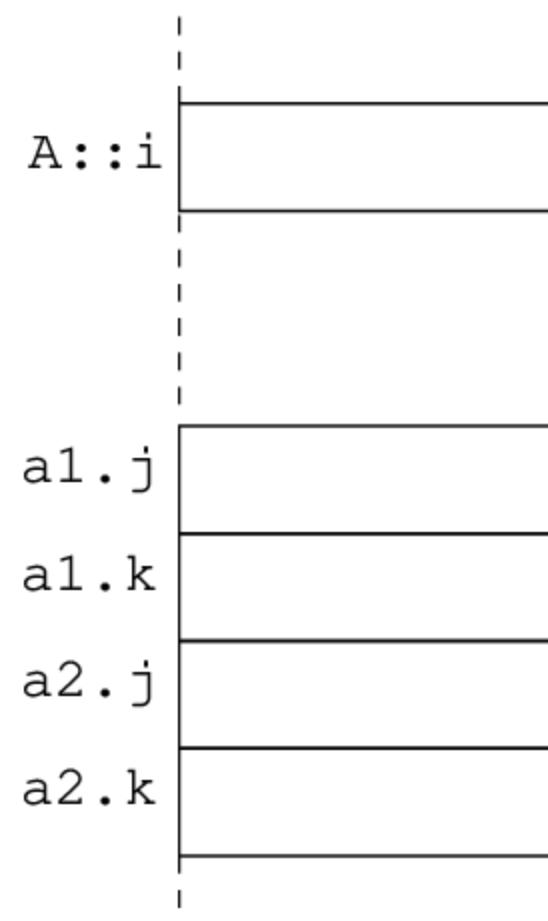
Les données membres statiques *constantes* de types *integrals* (char, bool, int) et flottants (float, double) font exception à cette règle : elles peuvent être initialisées, donc définies, dans le corps de la définition de classe.

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.

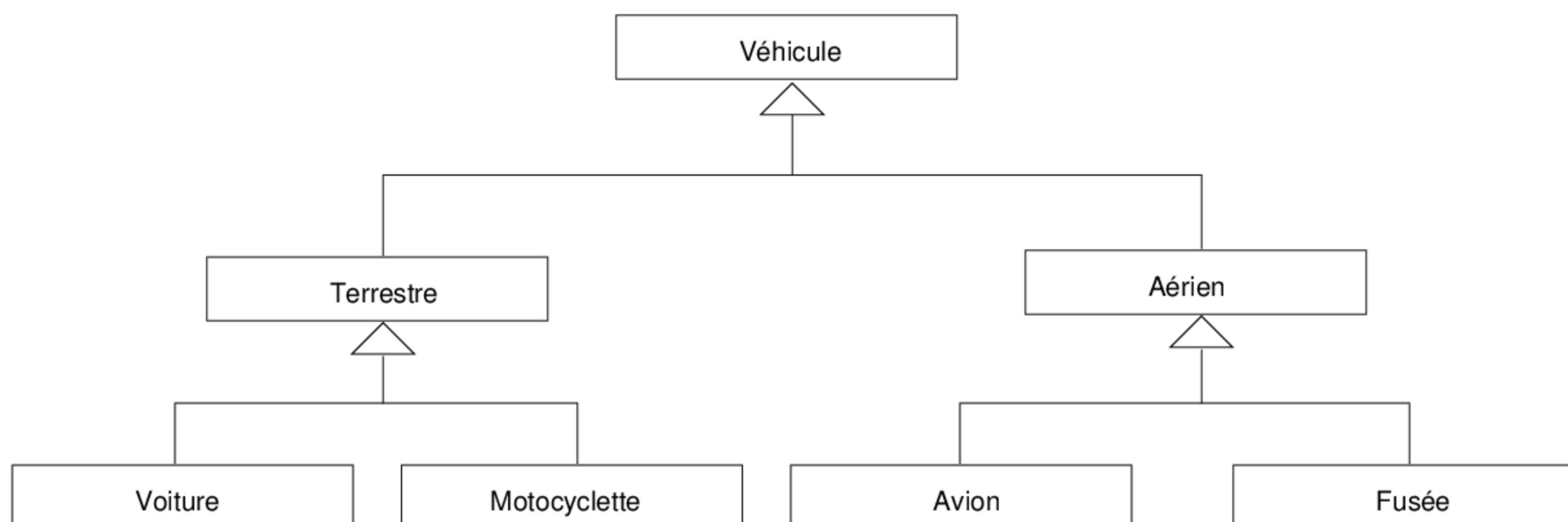


FIGURE 2.3 – Arbre d'héritage de types de véhicules.

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*.

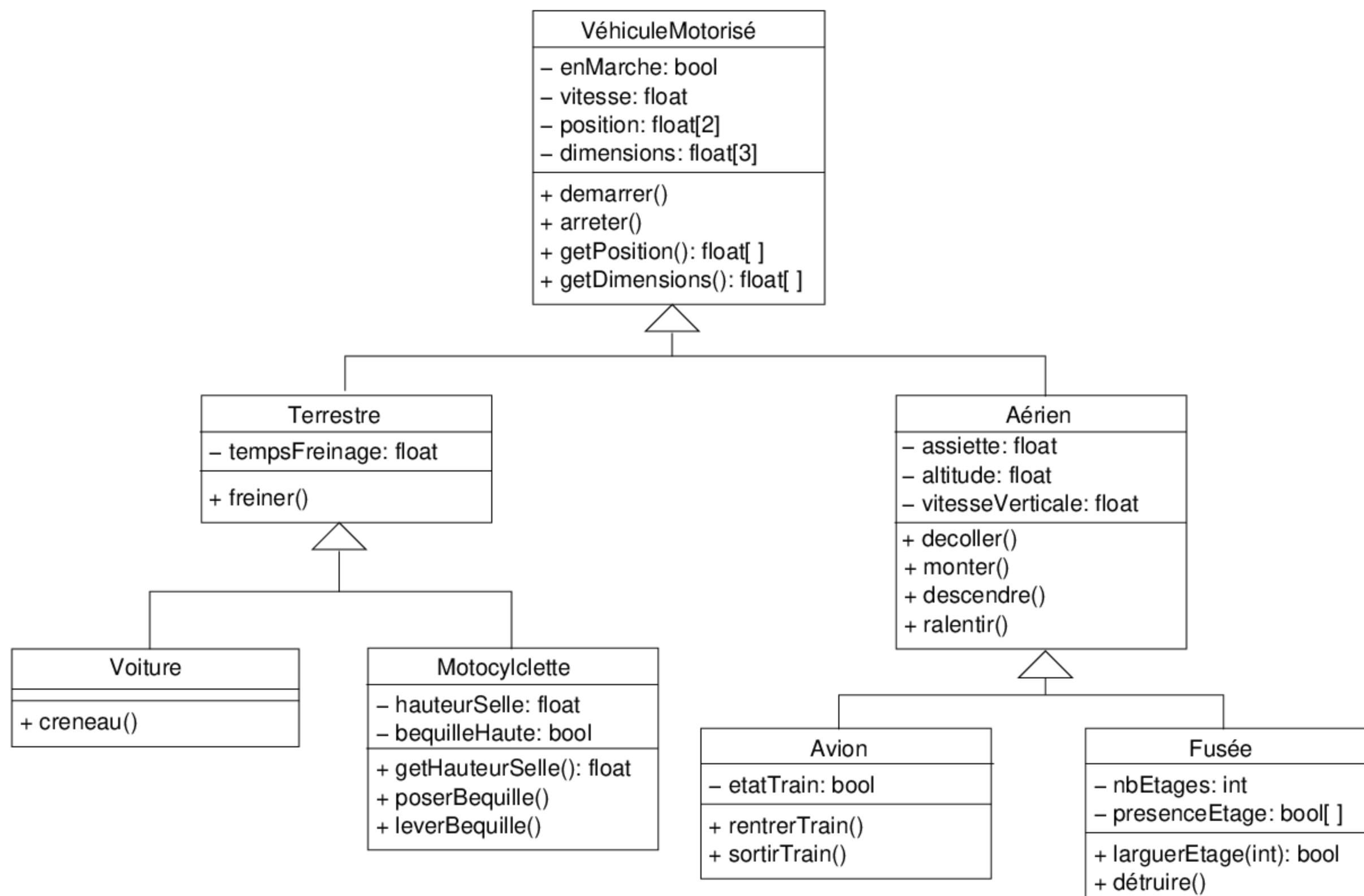
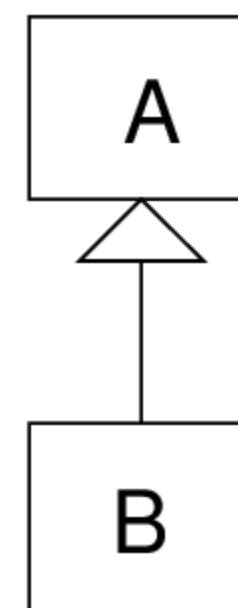


FIGURE 2.4 – Détail des classes modélisant les véhicules, avec héritage.

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.

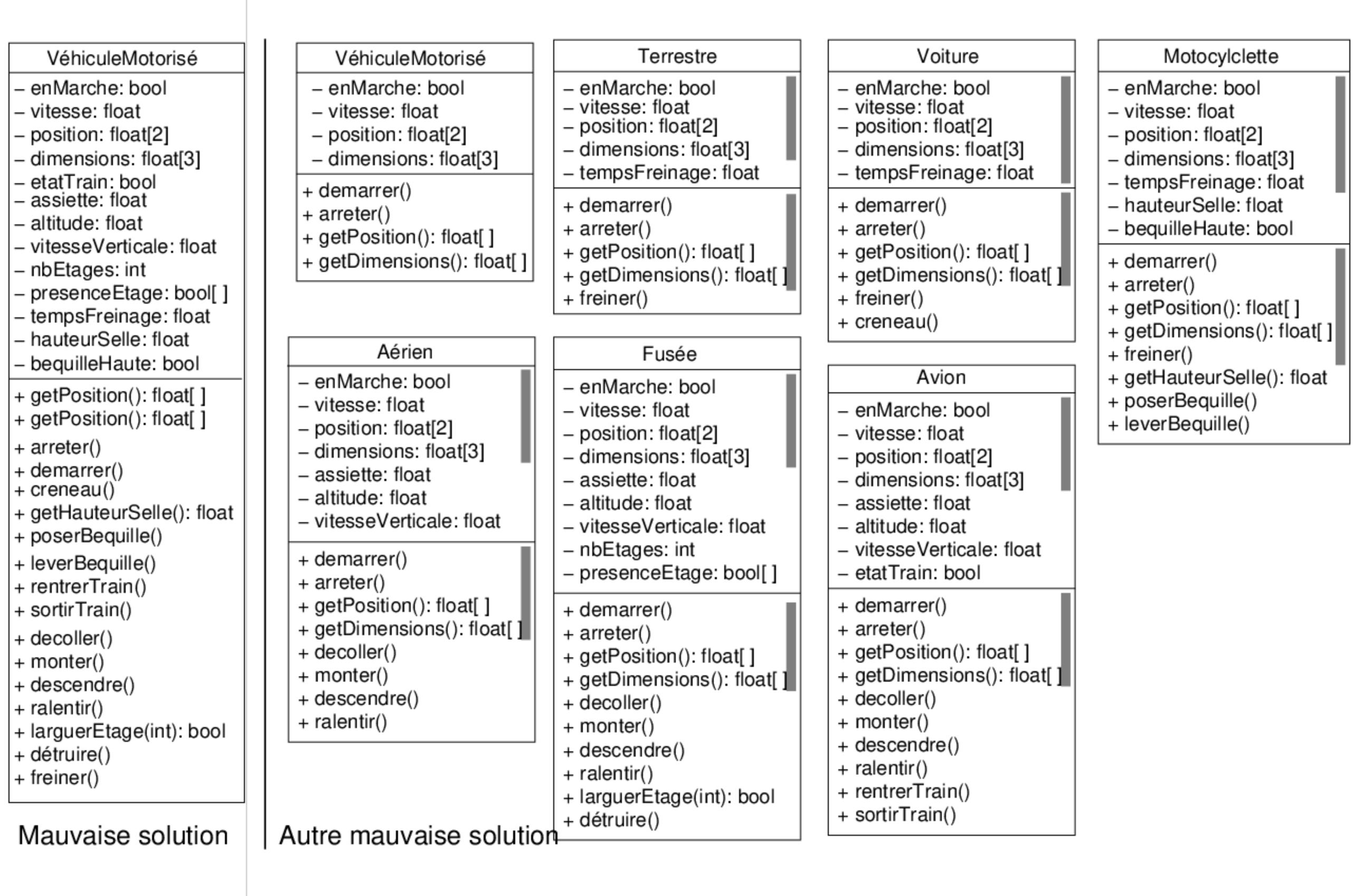


FIGURE 2.5 – Classes modélisant les véhicules, sans héritage.

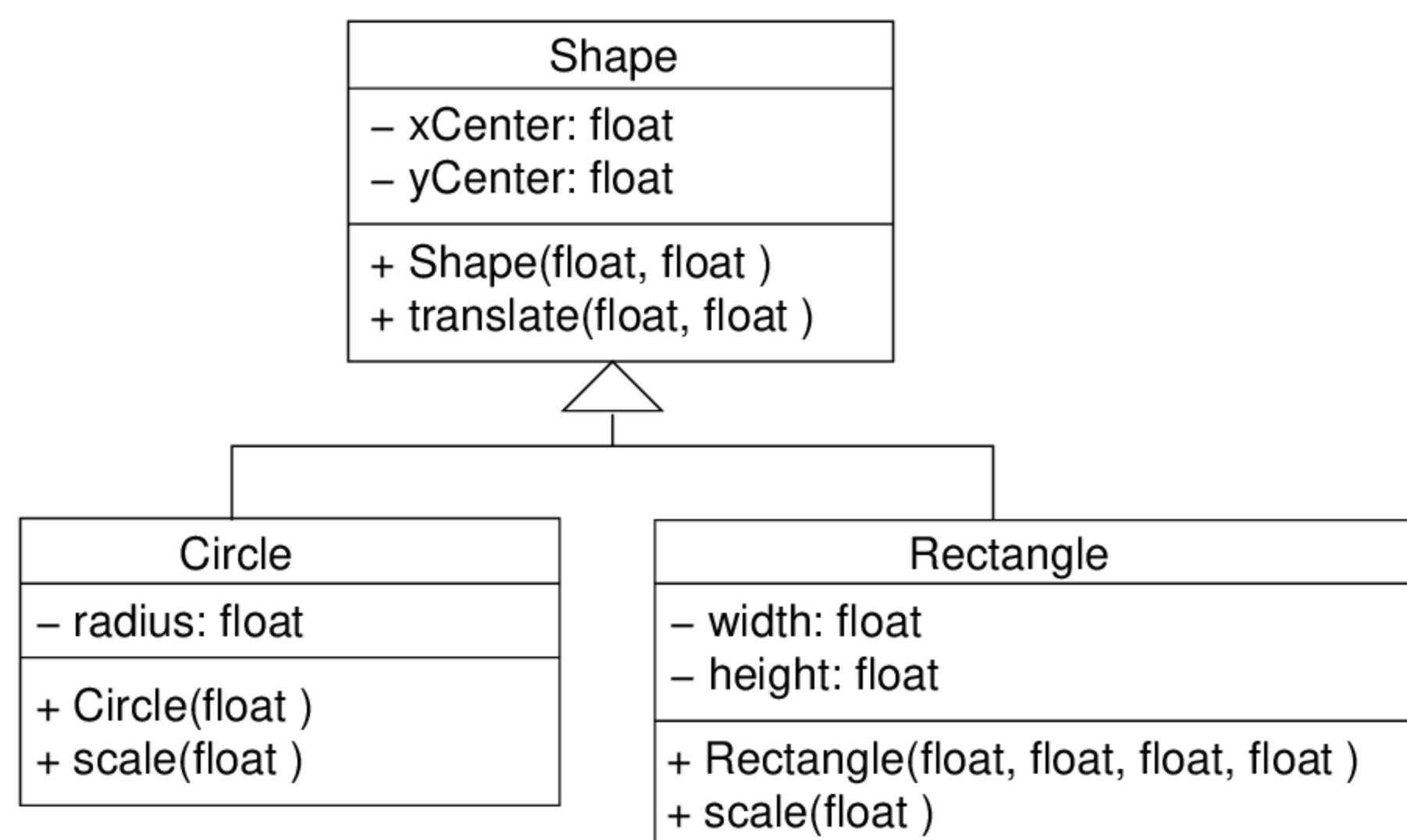


FIGURE 2.6 – Arbre d'héritage de formes géométriques (Diagramme de classes).

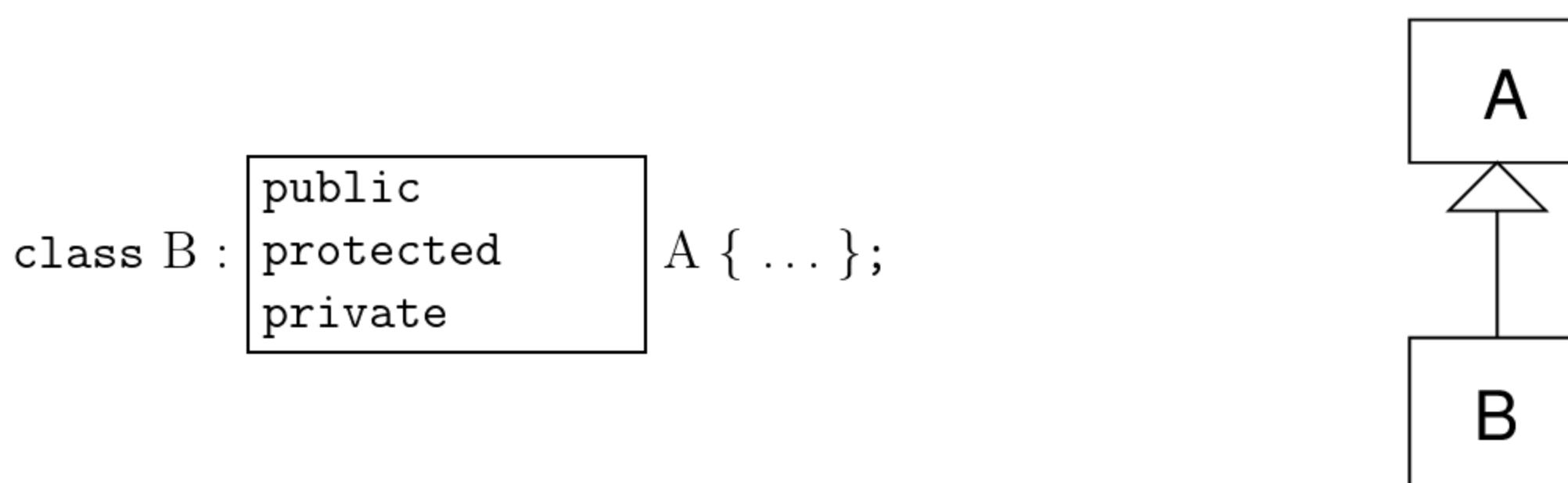
2.3.4 Exemple simple

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.

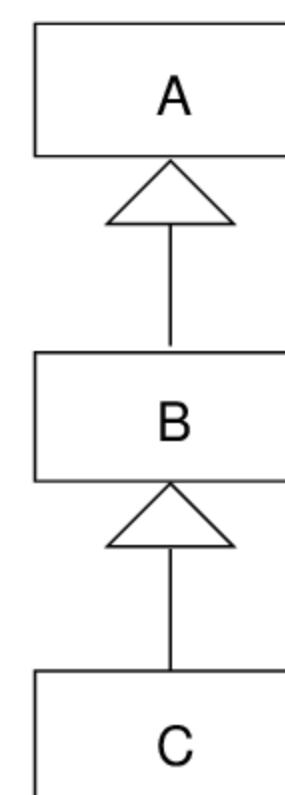
```

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 };
```

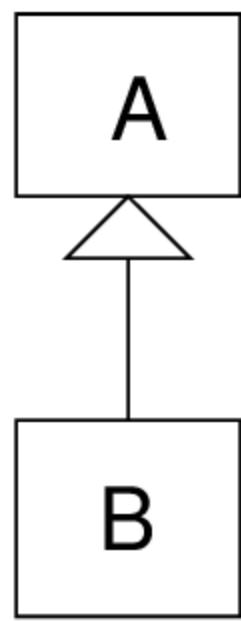
Listing 2.41 – Constructeurs et héritage.

```
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.

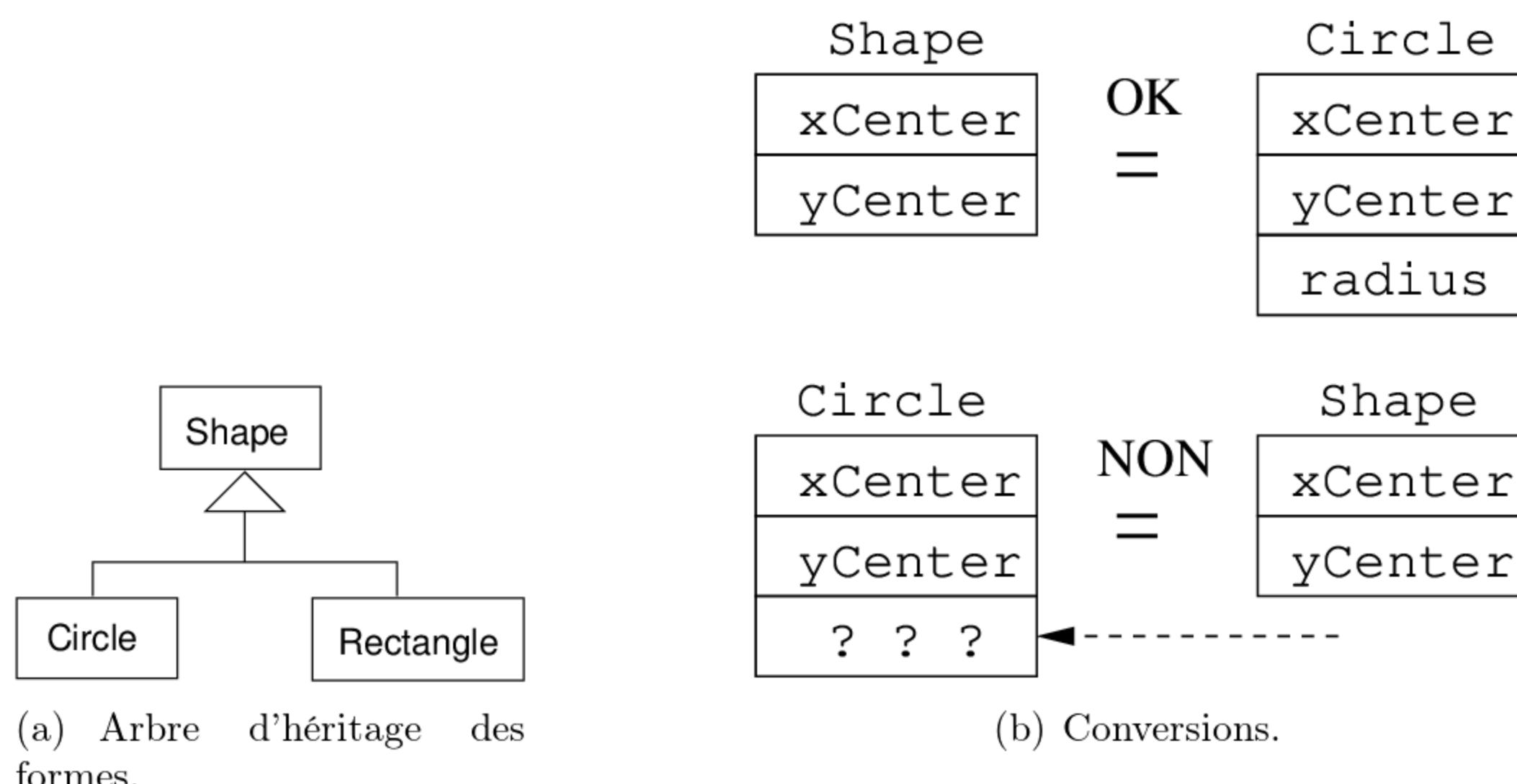


FIGURE 2.7 – Héritage et conversion.

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

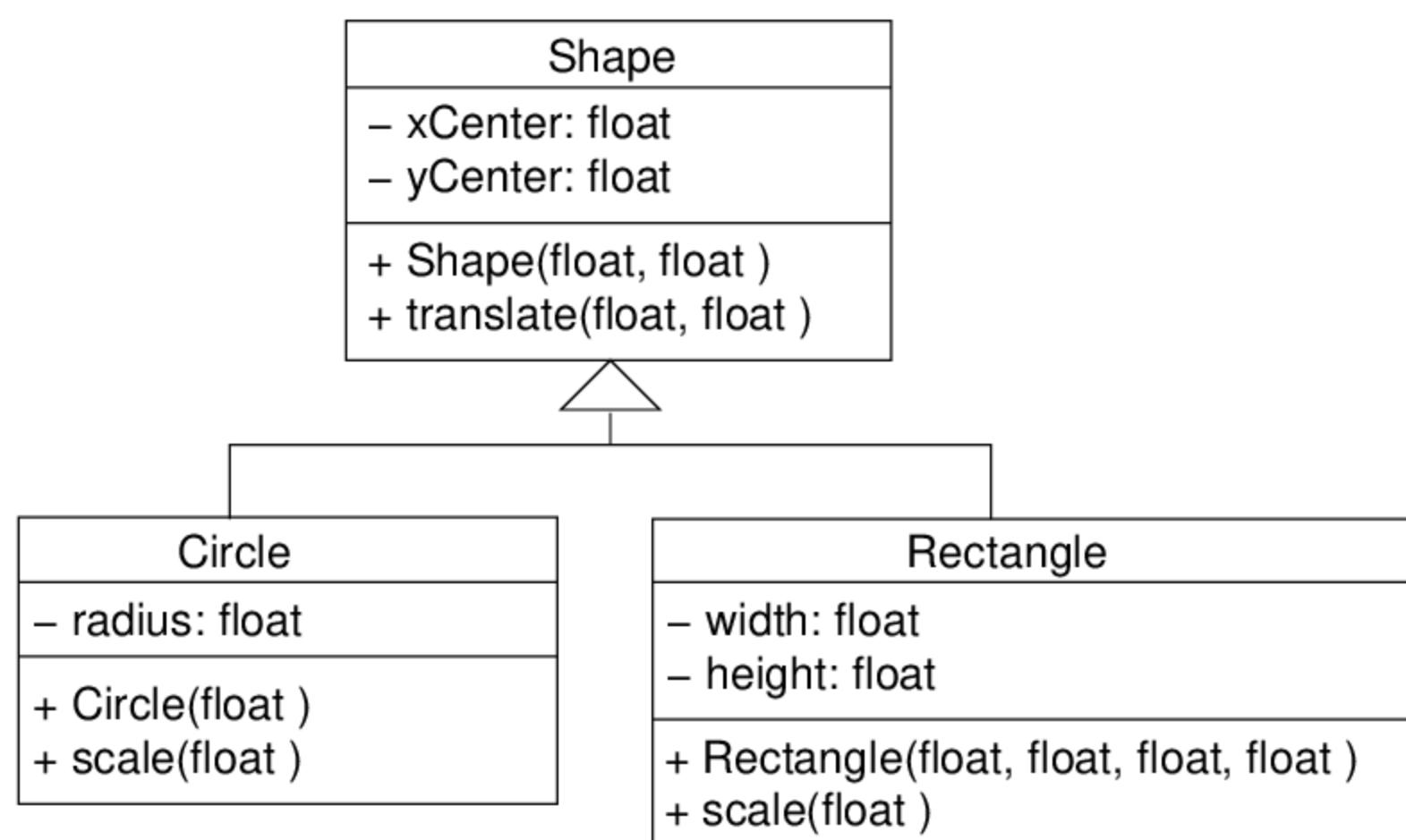


FIGURE 2.8 – Formes géométriques.

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.

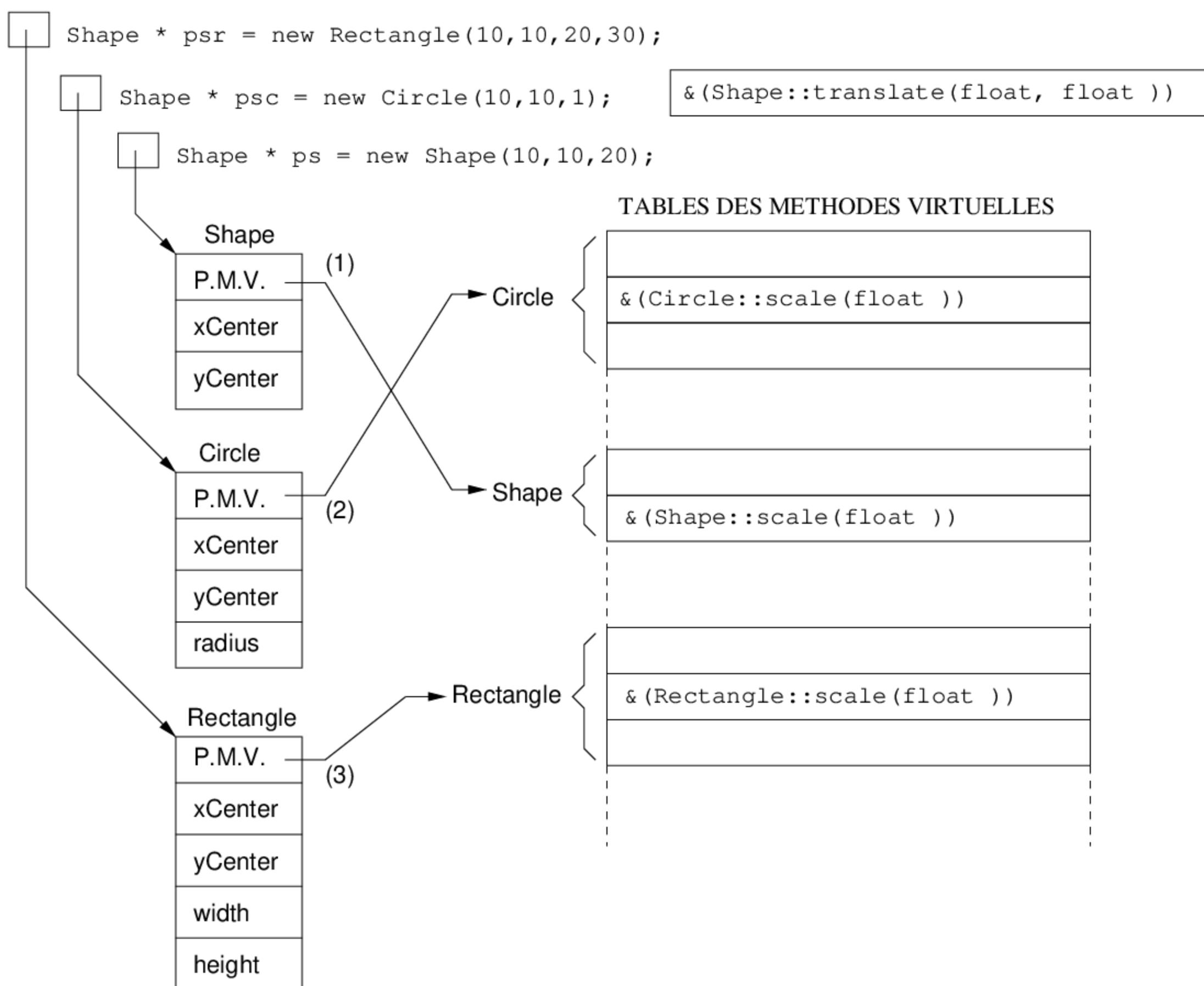


FIGURE 2.9 – Table des méthodes virtuelles.

[...]

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 précéder 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.

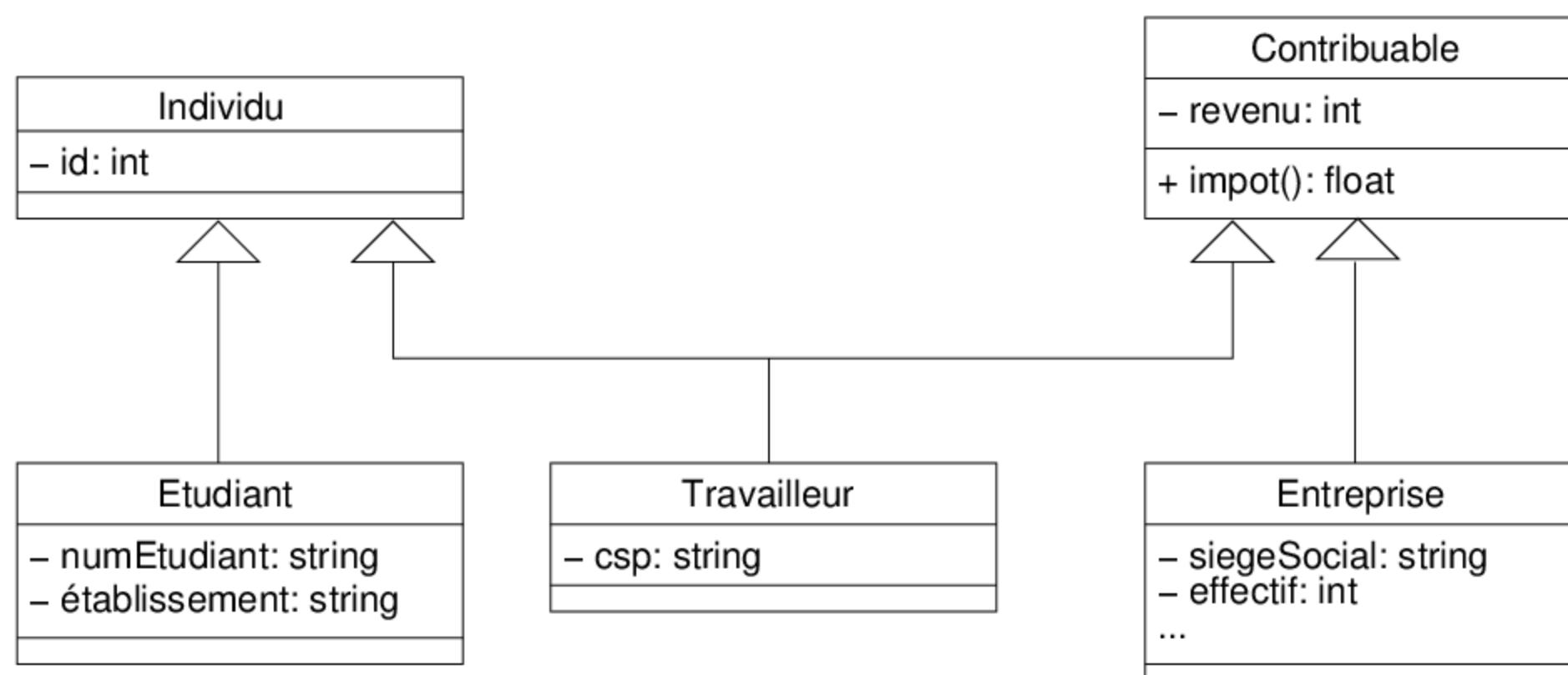


FIGURE 2.10 – Un travailleur est un individu *et* un contribuable.

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.

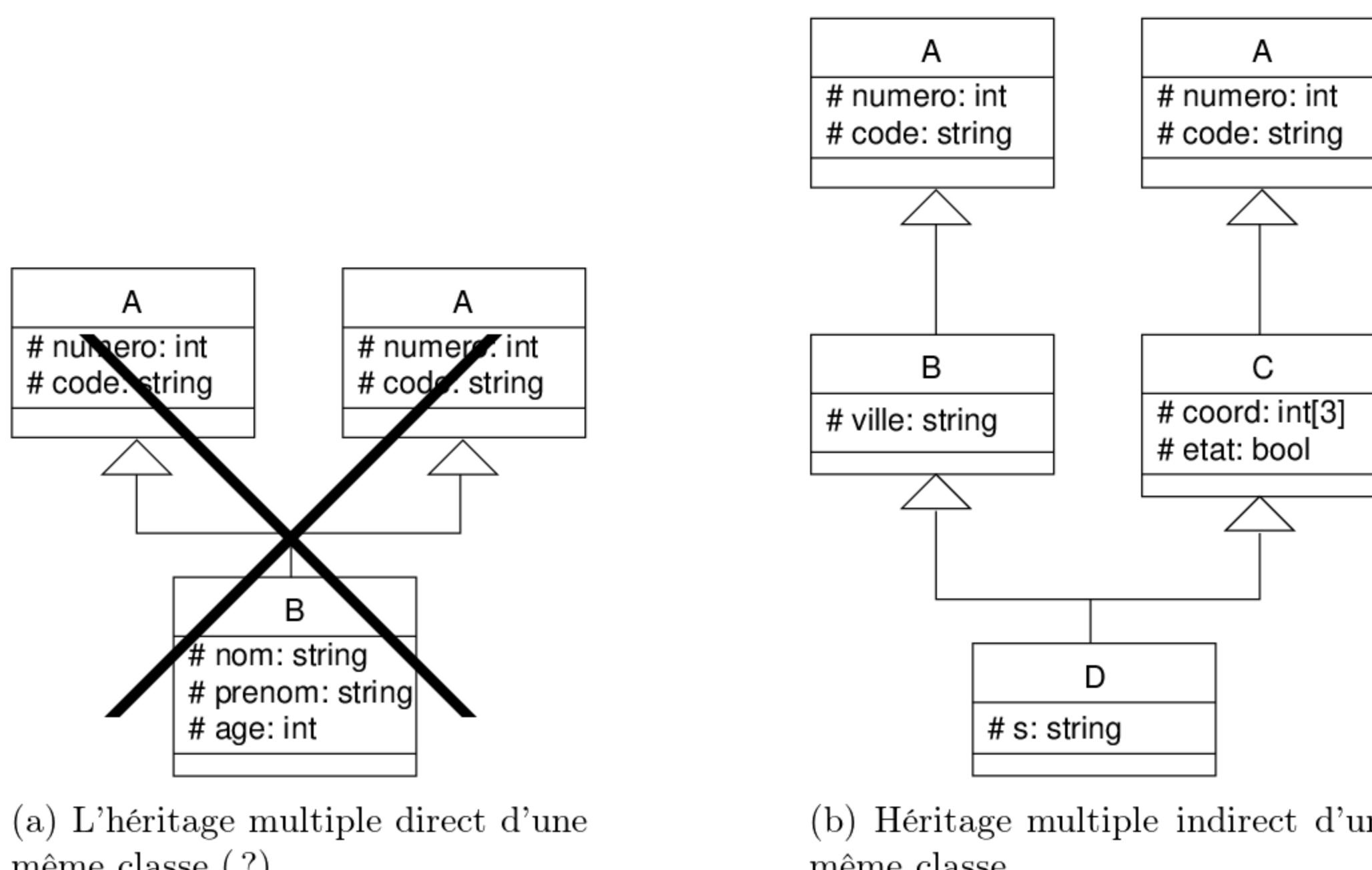


FIGURE 2.11 – Héritage multiple d'une même classe.

```
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).

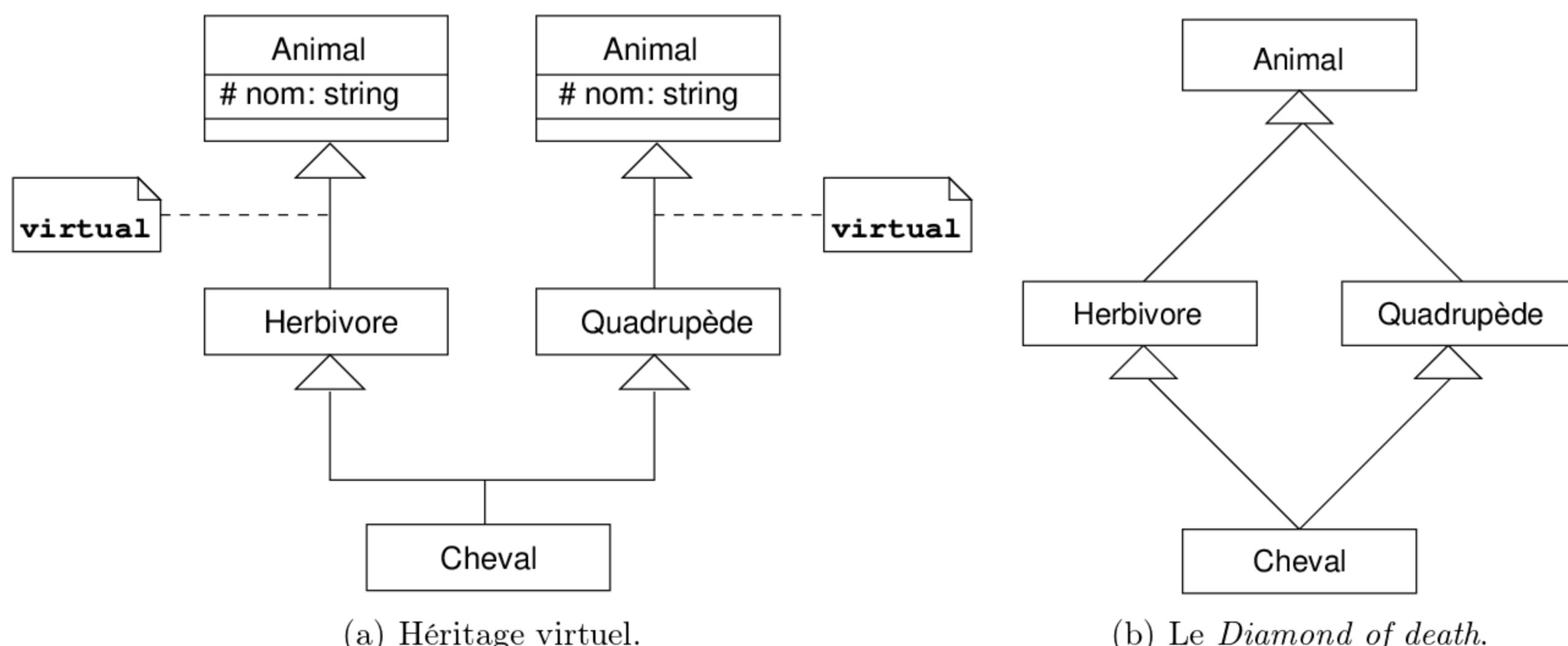


FIGURE 2.12 – Héritage multiple indirect sans duplication.

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.



2^e année – Spécialité Informatique

Année 2024 – 2025

Programmation orientée objet en C++

Notes de cours

Sébastien Fourey

Chapitre 3. Programmation générique grâce aux modèles

Version du 3 septembre 2024



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

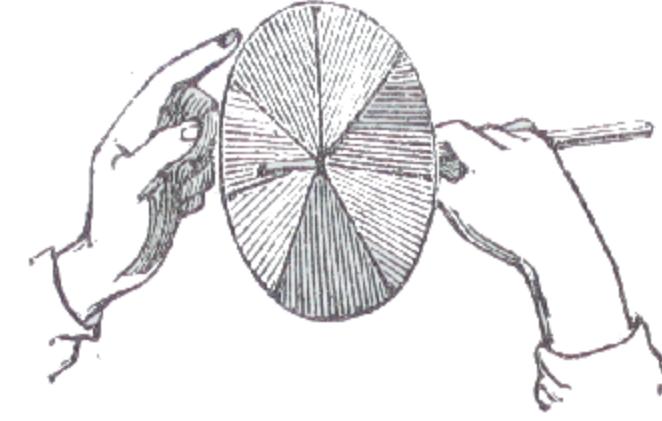


Fig. 46. — La lumière blanche est produite par la réunion des sept couleurs du spectre.

Chapitre 3

Programmation générique grâce aux modèles

3.1 Motivation

Il est très fréquent que des algorithmes complets ne diffèrent que par le type de certaines des données qu'ils manipulent. En terme d'implémentation dans un langage particulier, une fonction entière peut être applicable à des types de données différents, et ce à l'opérateur près. Il peut aussi arriver qu'en dehors des types utilisés, seulement quelques-unes des opérations réalisées soient dépendantes des types manipulés. Il semble alors justifié de vouloir réduire la quantité de code à écrire quand on souhaite utiliser ce genre d'algorithmes pour plusieurs types de données.

Le polymorphisme tel qu'il a été présenté dans le chapitre 2 peut s'avérer très utile pour répondre à ce besoin. Pensez par exemple à un algorithme qui ne manipule que des figures. Un tel algorithme, tant qu'il n'utilise que des opérations propres aux figures (et pas à ses types dérivés), peut en fait s'appliquer indifféremment à des cercles, rectangles ou autres.

D'autre part, ce qui est valable pour une fonction l'est aussi pour un ensemble de fonctions et donc pour une classe. Par exemple, quelle serait la différence entre la classe des ensembles d'entiers du listing 3.1 et une classe d'ensembles de nombres flottant ? 

Toutefois, il y a des situations pour lesquelles la solution apportée par le polymorphisme n'est pas satisfaisante, entre autres pour les quelques raisons qui suivent :

- Le polymorphisme, appelé dans ce cas *polymorphisme d'exécution*, constitue une solution *dynamique* à un problème qui peut parfois être résolu dès la compilation.
- Si tel est le cas, on peut juger que « l'artillerie » des méthodes virtuelles, avec les indirections et les contrôles dynamiques qu'elle implique, entraîne une perte d'efficacité dont on pourrait se passer.
- L'utilisation de l'héritage, comme moyen de regrouper au sein d'une même famille les classes d'objets qui *doivent pouvoir* être utilisées par un algorithme donné, peut devenir rapidement très arbitraire et contre-intuitif.
- Finalement, la lourdeur introduite est flagrante lorsqu'il s'agit de manipuler les types de base du langage. (Pensez à une classe *Integer*, comme en Java, et à un opérateur d'addition pour les types numériques qui serait une méthode virtuelle !)

Finalement, un programmeur (p. ex. en C) aura simplement pris l'habitude dans certains cas de dupliquer son code source pour produire des versions adaptées par de simples substitutions automatiques effectuées à l'aide de son éditeur favori. En première approximation, on peut dire que le C++ offre alors une solution analogue mais qui fait partie intégrante du langage : les modèles, sans doute aussi connus sous le nom de *templates*.

```

1 | class SetOfIntegers {
2 |     int * array;
3 |     unsigned int cardinality;
4 |     unsigned int capacity;
5 |
6 | public:
7 |     void SetOfIntegers(int capacity = 0);
8 |     void add(int);
9 |     bool contains(int);
10 |    // ...
11 | };

```

Listing 3.1 – Classe des ensembles d'entiers.

3.2 Modèle de classe

Un *modèle de classe* (ou *patron de classe*) est une définition de classe paramétrée par un ou plusieurs types. La définition d'une classe pour des types précis, à l'aide d'un modèle, porte le nom d'*instanciation de modèle*.

Avant de donner la syntaxe C++ utilisée, le diagramme de la figure 3.1 montre la notation UML correspondant aux modèles de classes.

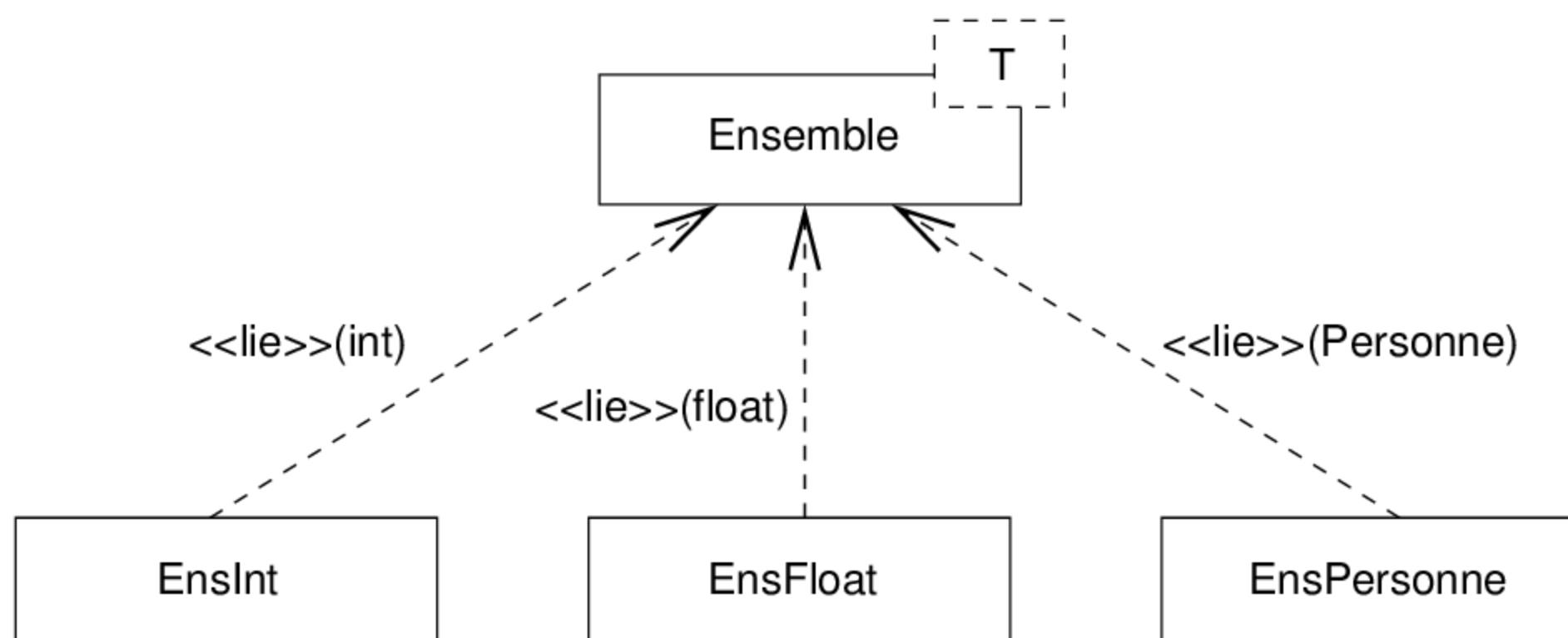


FIGURE 3.1 – Le modèle de classe Ensemble et trois de ses instantiations possibles.

3.2.1 Déclaration

Un modèle de classe, paramétré par un ou plusieurs types, est déclaré de l'une des manières suivantes :

```

template<class T> class NomMod {
    // Définition utilisant T comme un type classique.

```

```

};

template<class T, class U,... > class NomMod {
    // Définition utilisant T, U, ... comme des types classiques.
};

template<typename T> class NomMod {
    // Définition utilisant T comme un type classique.
};

template<typename T, class U,... > class NomMod {
    // Définition utilisant T, U, ... comme des types classiques.
};

```

D'un point de vue syntaxique, les mots-clés `typename` et `class` qui précèdent chaque paramètre du modèle sont équivalents. Le mot-clé `typename` peut être jugé plus lisible car il ne laisse pas supposer que le paramètre devrait être un nom de classe. En effet, un type de base ou tout autre type défini par l'utilisateur peut être utilisé pour instancier un modèle.

En fait, un paramètre de modèle peut aussi être une valeur constante d'un certain type, et pas uniquement un nom de type, comme le montre le listing 3.2. Les types autorisés sont cependant limités :

- types intégrals (int, char, etc.) ou énumérés ;
- pointeurs ;
- références.

Notamment, il n'est pas possible d'utiliser comme argument de template une constante de type float. De plus, dans le cas d'un argument de type pointeur (ou référence), l'adresse en question doit être connue à la compilation. Il ne peut donc pas s'agir d'une variable locale !

```

1 #include <iostream>
2
3 template <typename T, int size>
4 class Array {
5     T cells [size];
6
7 public:
8     Array ();
9     const T & operator [] (int) const;
10    T & operator [] (int);
11 }
12
13 int main () {
14     int s;
15     Array<int, 100> array; // OK
16     std :: cout << "Enter a size: " << std :: flush;
17     std :: cin >> s;
18     Array<int, s> dynamicArray; // Error!
19 }
```

Listing 3.2 – Utilisation d'une constante comme paramètre de type.

De plus, un paramètre de modèle peut avoir un *type* ou une *valeur* par défaut, comme illustré dans le listing 3.3.

```

1 #include <iostream>
2 template <typename T, int columns, int rows = 1>
3 class Matrix {
4     T cells [rows] [columns];
5
6 public:
7     Matrix();
8     const T & operator()(int, int) const;
9     T & operator()(int, int);
10 };
11
12 int main() {
13     int s;
14     Matrix<double, 100, 100> matrixA; // Matrix 100 by 100
15     Matrix<double, 100> matrixX;       // Row vector
16 }
```

Listing 3.3 – Modèle de classe dont un paramètre possède une valeur par défaut.

Enfin, point important, un modèle peut hériter d'autres modèles. Comme on le verra dans la section 3.3, deux instances d'un même modèle étant en fait deux classes bien distinctes, il est possible que la spécialisation d'un modèle hérite d'une instanciation du modèle en question.

3.2.2 Définition d'une méthode hors de la déclaration du modèle

La définition d'une méthode inline à l'intérieur de la définition de la classe est immédiate, mais le mot-clé *template* doit être réutilisé lors d'une définition à l'extérieur à la classe.

```

template<typename T> class NomModèle {
    [...]
    typeRetour idMéthode(T arg1, ...); // Simple déclaration
};

template<typename T> typeRetour NomModèle<T>::idMéthode(T arg1, ... ) {
    // Code de la méthode
}
```

D'une manière générale, on peut se rappeler que le nom d'un modèle de classe ne peut apparaître qu'affublé des symboles < >. L'instanciation d'un modèle dont tous les paramètres ont une valeur par défaut ne fait pas exception à la règle [...]. La mise en garde suivante va dans le même sens.



Un modèle de classe ne peut pas être surchargé par une définition de classe. (Mais des spécialisations sans paramètres sont possibles.)

3.2.3 Instanciation d'un modèle : vérifications et variantes

Comme cela a été présenté en introduction, l'instanciation d'un modèle résulte en la définition d'une classe C++ habituelle, et donc la génération du code correspondant. En effet, un modèle

présent dans un fichier source mais qui n'est pas instancié ne produit aucun code. La phase de vérification syntaxique et sémantique est plus que succincte pour un modèle tant qu'il n'est pas instancié. Ce n'est que lorsque tous les types sont connus et que le code d'une classe est généré que ce dernier peut être vérifié. Cette phase, l'instanciation, est provoquée dès que le nom du modèle apparaît avec des paramètres totalement renseignés. Elle peut prendre plusieurs formes, qui sont énumérées ici.



Quand un modèle est utilisé pour des types particuliers dans un programme, il faut s'assurer que le code correspondant *peut* être généré (c.-à-d., le compilateur dispose de la définition complète du modèle) ou bien qu'il a été généré dans une autre unité de compilation et sera présent lors de l'édition de liens. Ceci correspond à des niveaux de générericité bien distincts [...].

Première méthode La méthode d'instanciation de modèle la plus concise passe par la déclaration d'un objet (y compris comme paramètre d'une fonction) :

```
NomModèle<type,...> nom_objet;
NomModèle<type,...> nom_objet(arg1, ..., argN);
```

Deuxième méthode Le mot-clé `template` peut aussi être utilisé pour déclencher l'instanciation, mais en ne définissant cette fois ci aucun objet. Cette méthode est préférée dans le cas d'une bibliothèque pour laquelle il serait dommage de devoir créer des variables (globales) inutilisées.

```
template class NomModèle<type,...>;
```

Ces différentes méthodes d'instanciation d'un modèle de classe sont illustrées par le code suivant :

```
1 List<int> primes;
2 class Student {
3     // ...
4 };
5 List<Student> my_group;
6 template class List<Student>;
```

Listing 3.4 – Différents types d'instanciation d'un modèle de classe.

Exercice 3.1 (Question d'examen) Écrivez un code (très court) définissant un modèle de classe rudimentaire. Ensuite, donnez une ligne de code réalisant l'instanciation de votre modèle sans aucune déclaration ni construction de variable.

3.2.4 C++11 : Empêcher l'instanciation

En C++11, il est possible d'empêcher l'instanciation d'un modèle dans une unité de compilation. En effet, l'instanciation à plusieurs reprises d'un même modèle dans différentes unités de compilation pose un problème d'efficacité (de la compilation) non négligeable. On peut donc empêcher cette instantiation grâce au mot-clé `extern` comme dans l'exemple du listing 3.5.

```
1 | extern template class List<Student>;
```

Listing 3.5 – Instruction **extern** pour un template.

3.2.5 Exemple

Le listing 3.6 montre le « squelette » d'un modèle de classe complet qui a la particularité, néanmoins classique, de posséder un modèle de fonction amie. La syntaxe, mais aussi l'ordre des déclarations mérite que le lecteur s'y attarde. En effet, pareille définition d'un modèle de fonction amie se fait en pas moins de 4 étapes.

3.3 Spécialisations définies par l'utilisateur

Motivation

Les modèles permettent d'écrire des algorithmes génériques avec une syntaxe intégrée au langage. Mais il arrive parfois que cette générericité aille à l'encontre de l'efficacité. En effet, certains types peuvent posséder des particularités qui, si on les utilise, permettent d'accélérer sensiblement l'exécution d'un programme.

Il peut aussi arriver, plus simplement, qu'une opération utilisée dans un modèle n'existe pas pour une type donné. Dans ce cas comme dans le précédent, on peut écrire une ou plusieurs variantes du modèle adaptées à des types précis. On parle alors de *spécialisation(s)* du modèle.

```
1 | Vector<int> v;
2 | Vector<Shape> vs;
3 | Vector<Shape *> vps; // What for?
4 | Vector<Object *> vo;
5 | Vector<Node *> vn;
```

Listing 3.7 – Différentes instanciations d'un modèle de classe Vector.

Dans le cas des déclarations du listing 3.7 on peut se poser par exemple une question au sujet de la destruction d'un vecteur d'entier, par rapport à celle d'un vecteur de pointeurs sur des formes. En effet, le vecteur dans ce dernier cas doit-il s'occuper de la désallocation individuelle des formes pointées ? Comme mentionné dans le paragraphe précédent, nous serions alors dans la situation où l'opérateur **delete** n'a effectivement pas de sens pour un entier, alors qu'on souhaite l'utiliser dans le cas d'un vecteur de pointeurs. Il est alors utile de pouvoir définir deux modèles, l'un pour les types *pointeur sur quelque chose* et un autre pour tous les autres types. Cet exemple est très courant. Qui plus est, une solution élégante s'applique à ce cas précis : Elle consiste à utiliser l'héritage afin de minimiser le code supplémentaire à écrire. En effet, seul le destructeur est particulier dans le cas des pointeurs alors que tout le reste peut être défini pour le type **void*** comme pour un **int**. On réalise donc :

1. Une spécialisation *complète* du modèle pour le type **void***, puis
2. une spécialisation *partielle* du modèle pour le type **T***.

Toute ceci est illustré par le listing 3.8 qui montre au passage la syntaxe de la spécialisation, qui peut être schématisée de la manière suivante :

```

1 class Complex;
2
3 // (1) Forward declaration of Matrix
4 template <typename T> class Matrix;
5
6 // (2) Forward declaration of the operator template
7 template <typename T>
8 Matrix<T> operator+(const Matrix<T> &, const Matrix<T> &);
9
10 template <typename T>
11 class Matrix {
12     T ** data;
13     unsigned int rows, columns;
14     T ** allocate(unsigned long, unsigned long){};  

15 public:
16     Matrix() {
17         data = 0;
18     }
19     Matrix(const Matrix &);
20     Matrix(unsigned long l, unsigned long c)
21         : rows(l), columns(c) {
22         data = allocate(l, c);
23     }
24     ~Matrix();
25     Matrix & operator=(const Matrix &);
26     Matrix operator*(const Matrix &);
27     // (3) friend function template
28     friend Matrix<T> operator+<>(const Matrix<T> &,
29                                         const Matrix<T> &);
30     T * operator[](unsigned long);
31 };
32
33 template <typename T> // (4) template definition
34 Matrix<T> operator+(const Matrix<T> & a,
35                         const Matrix<T> & b) {
36     // ...
37 }
38
39 int main() {
40     Matrix<int> m1(10, 2);
41     Matrix<float> m2(12, 12);
42     Matrix<Complex> m3(12, 23);
43 } // ...

```

Listing 3.6 – Exemple de modèle de classe avec une fonction amie.

Modèle général

```
template<typename T, typename U,... > class NomModèle {
    //...
};
```

Spécialisation complète

```
template<> class NomModèle<idTypePourT,idTypePourU,...> {
    //...
};
```

Spécialisation partielle

```
template<typename U,... > class NomModèle<idTypeSpecialisePourT,U,...> {
    //...
};
```

(Viennent après le mot-clé *template* tous les noms de paramètres de types qui font que la spécialisation n'est pas complète. Ils sont aussi répétés à droite du nom du modèle.)

Définition d'une méthode dans le cas d'une spécialisation complète

```
typeRetour NomModèle<Type1,Type2,...>::idMéthode(Type1 arg1,...) {
    // Code de la méthode
}
```

Notez l'absence du mot-clé `template<>` au début du prototype. Il s'agit alors en effet d'une simple définition.

3.4 Les modèles de fonctions

De même qu'il est possible de définir des modèles de classes (donc des modèles de méthodes), on peut définir des modèles de fonctions. Comme pour les classes, un tel modèle permet de générer automatiquement le code d'une fonction adaptée à un ou des types particuliers à partir d'un *patron* de fonction.

Un modèle est instancié (c.-à-d., du code est généré) lorsque le compilateur rencontre un appel de fonction modèle avec des arguments de types définis. La syntaxe de cet appel est en tout point similaire à un appel de fonction classique, sauf lorsque le type des arguments du modèle ne peut être déduit. C'est le cas des fonctions sans arguments, mais aussi lorsqu'aucun paramètre de type n'est utilisé dans la liste d'arguments, ou encore lorsqu'il y a ambiguïté en raison de l'application possible de conversions implicites multiples.

3.4.1 Syntaxe

Un modèle de fonction est défini de la manière suivante :

```

1 template <typename T>
2 class Vector {
3     // ...
4 };
5
6 template <>
7 class Vector<void *> {
8     // ...
9     void * & operator[](int i);
10};
11
12 Vector<void *> v;
13
14 template <typename T>
15 class Vector<T *> : private Vector<void *> {
16     typedef Vector<void *> Base;
17     // ...
18     T * & operator[](int i) {
19         return reinterpret_cast<T * &>(Base::operator[](i));
20     }
21 };

```

Listing 3.8 – Spécialisations complète et partielle d'un modèle de classe.

```

template<typename T0, typename T1,...> T0 idFonction( T1 arg1, [...] ) {
    [...]
};

template<typename T0,...> type idFonction( T0 arg1, [...] ) {
    [...]
};

template<typename T0, type1 arg,...> type idFonction( T0 arg1, [...] ) {
    [...]
};

template<typename T0, type1 arg,...> type idFonction( [...] ) {
    Utilisation des types ou valeurs paramètres.
    [...]
};

```

L'appel peut être totalement transparent (c.-à-d., aucune différence par rapport à une fonction classique), il peut aussi rendre explicite le type ou les valeurs des paramètres du modèle :

```

idFonction(arg1, [...] );
idFonction<type1,type2,...>();
idFonction<type1,type2, constante1,...>(arg1, [...] );

```

La surcharge de fonctions autorise enfin une forme de spécialisation...

3.4.2 Surcharge

Contrairement aux modèles de classes, la surcharge des modèles de fonctions par une fonction classique est possible. Au moment de l'appel, si les types ne sont pas précisés (entre <>) et qu'il y a correspondance stricte des types, la priorité est donnée aux fonctions ordinaires. Ensuite, les spécialisations les plus précises sont recherchées. (Un paramètre de type déduit ne peut faire ensuite l'objet d'une promotion, d'une conversion standard, ou d'une conversion définie par l'utilisateur.)

3.5 Remarques finales

- Une classe peut posséder des modèles de méthodes ;
- Un modèle de classe peut aussi posséder des modèles de méthodes (c.-à-d., qui ont leurs propres paramètres de types).



2^e année – Spécialité Informatique

Année 2024 – 2025

Programmation orientée objet en C++

Notes de cours

Sébastien Fourey

Chapitre 4. La bibliothèque standard

Version du 3 septembre 2024



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

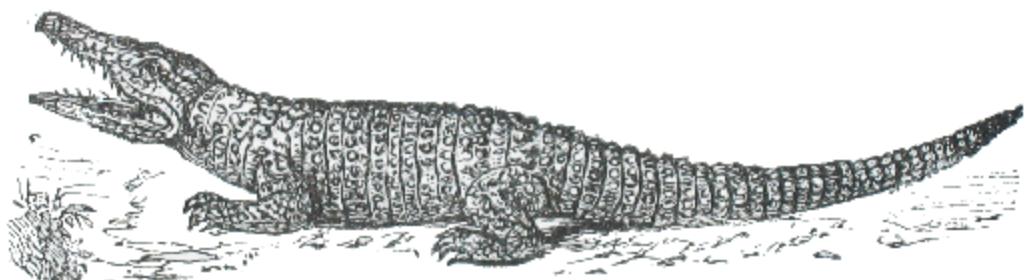


Fig. 145. — Crocodile. Il y en a qui atteignent 8 mètres de long.

Chapitre 4

La bibliothèque standard

4.1 Flux d'entrée/sortie

4.1.1 Hiérarchie des flux d'E/S en C++

En C++ les entrées/sorties sont implémentées par une hiérarchie relativement complexe de classes dont deux au moins sont connues du débutant : les classes `istream` et `ostream` dont les deux instances respectives `cin` et `cout` sont déclarées dans le fichier d'en-tête `<iostream>`.

La hiérarchie de ces classes est donnée par la figure 4.1 dans laquelle on trouve :

- Les classes les plus générales regroupant les propriétés communes à n'importe quel flux (`ios_base` et `ios`, cf. sections 4.1.2 et 4.1.3) ;
- Les classes spécialisées des flux d'entrée (`istream`) et des flux de sortie (`ostream`) ;
- La classe spécialisée des flux d'entrée ou sortie (`iostream`) ;
- Les classes spécialisées des flux de fichiers (`ifstream`, `ofstream` et `fstream`) ;
- Les classes spécialisées des flux de chaînes (`istringstream`, `ostringstream` et `stringstream`).

Le lecteur aura compris que la dernière catégorie de la liste précédente permet de « mimer » avec une syntaxe C++ les manipulations possibles avec les fonctions `sprintf` et `sscanf` de la bibliothèque C.

En fait, la bibliothèque standard C++ ne définit pas directement les classes d'entrée/sorties. Ces dernières ne sont effet que des instanciations de modèles de classes standards. Le paramètre des modèles de flux est le type de *caractère* utilisé. Les modèles standards qui correspondent aux classes d'entrées sorties sont représentés dans la figure 4.2.

Exercice 4.1 Quelle relation peut on faire entre les classes `ostream` et `istream` que vous connaissez déjà, associées à la surcharge possible des opérateurs de flux `<<` et `>>` ; et les méthodes `toString()` du langage Java ?

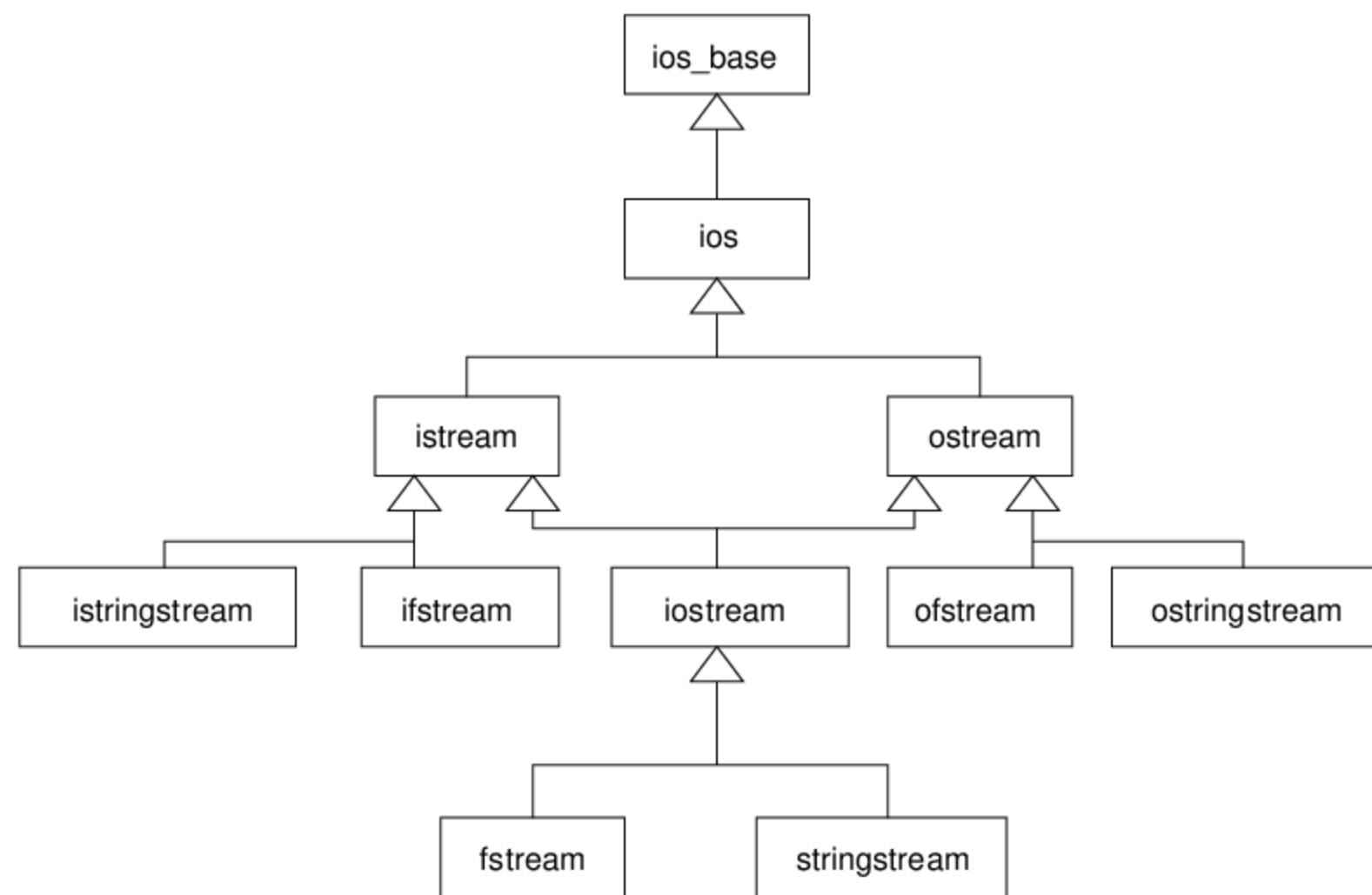


FIGURE 4.1 – Les classes de flux d'E/S en C++.

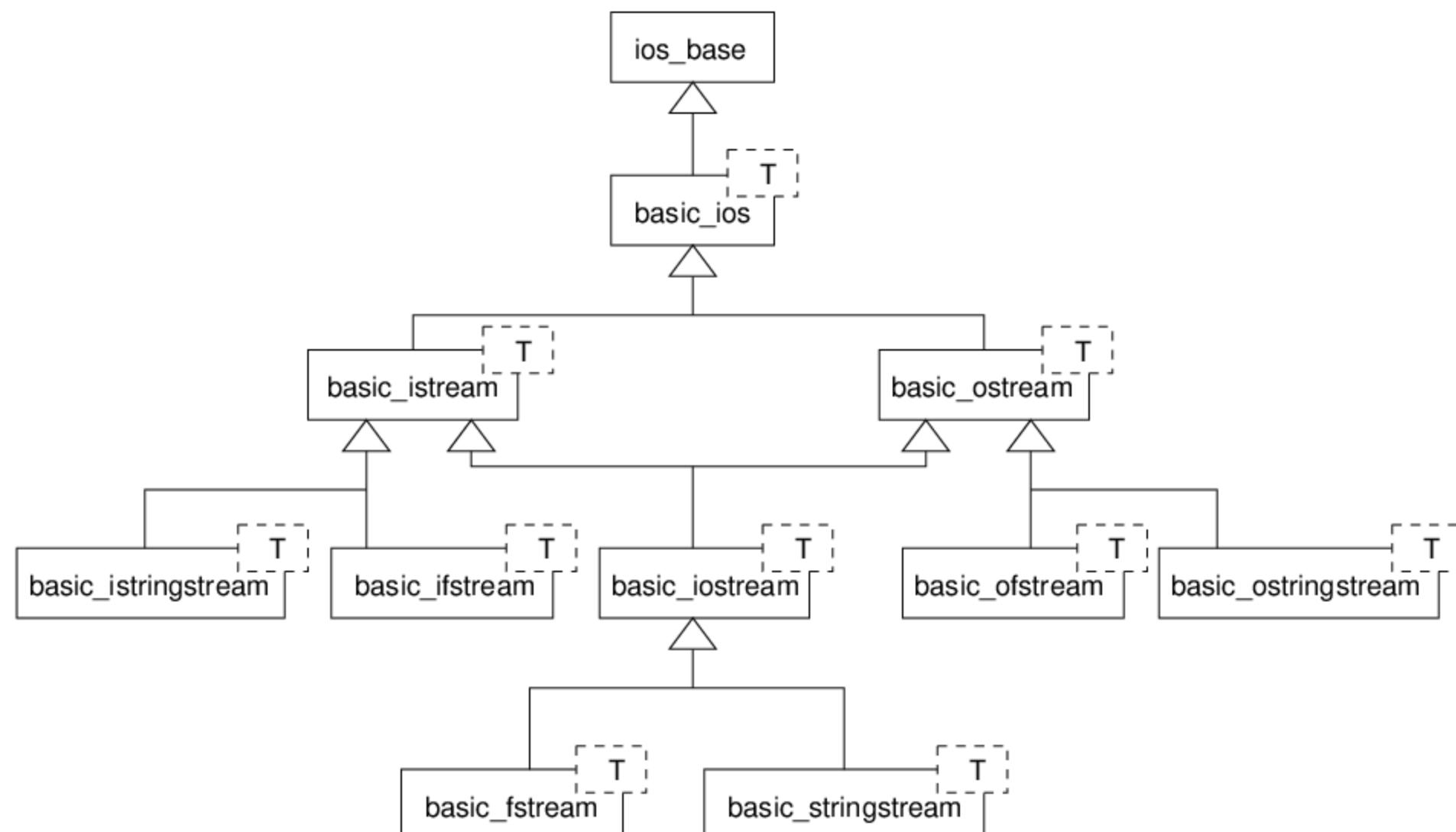


FIGURE 4.2 – Les modèles de classes de flux en C++.

4.1.2 La classe `ios_base`

Cette classe mère de toutes les classes de flux est déclarée dans l'en-tête standard `<iostream>`. Elle définit un grand nombre de constantes (cf. tableau 4.1) et quelques méthodes (cf. tableau 4.2) qui sont utilisées dans toutes les classes dérivées.

4.1.3 La classe `ios`

Elle fournit les services de gestion de tampons aux classes dérivées (grâce à un objet de la classe `streambuf`) ainsi qu'un ensemble de méthodes permettant de connaître l'état d'un flux. Les méthodes usuelles de cette classe sont données dans le tableau 4.3.

Format (<code>ios::fmtflags</code>)			Modes d'ouvertures (<code>ios::openmode</code>)		
boolalpha	left	uppercase	in	out	binary
hex	right	initbuf	trunc	app	ate
oct	internal	skipws	Positionnement (<code>ios::seekdir</code>)		
dec	showbase	adjustfield	beg	cur	end
fixed	showpoint	basefield	État du flux (<code>ios::iostate</code>)		
scientific	showpos	floatfield	eofbit	goodbit	badbit
					failbit

TABLE 4.1 – Constantes définies par la classe `ios_base`. (Leurs types respectifs.)

Formats	
<code>fmtflags</code>	<code>flags() const;</code>
<code>fmtflags</code>	<code>flags(fmtflags);</code>
<code>fmtflags</code>	<code>setf(fmtflags);</code>
<code>fmtflags</code>	<code>setf(fmtflags f, fmtflags mask);</code>
<code>void</code>	<code>unsetf(fmtflags);</code>
<code>streamsize</code>	<code>precision() const;</code>
<code>streamsize</code>	<code>precision(streamsize);</code>
<code>streamsize</code>	<code>width() const;</code>
<code>streamsize</code>	<code>width(streamsize);</code>

TABLE 4.2 – Méthodes usuelles définies par la classe `ios_base`.

État	
<code>operator void*() const;</code>	
<code>bool operator!() const;</code>	
<code>iostate rdstate() const;</code>	
<code>void clear(iostate status = goodbit);</code>	
<code>void setstate(iostate status);</code>	
<code>bool eof();</code>	
<code>bool fail();</code>	
<code>bool good();</code>	
<code>bool bad();</code>	

Remplissage	
<code>char_type fill();</code>	
<code>char_type fill(char_type);</code>	

Synchronisation	
<code>ostream* tie(ostream *);</code>	

TABLE 4.3 – Méthodes usuelles définies par la classe `ios`

4.1.4 La classe `ostream`

Elle est déclarée dans l'en-tête standard `<ostream>` et définit comme son nom l'indique l'ensemble des méthodes propres aux flux de sortie. Les méthodes essentielles sont rappelées dans le tableau 4.4.

Constructeur	
<code>ostream(streambuf*);</code>	
Sortie et position	
<code>ostream& operator<<(T);</code>	
<code>ostream& put(char_type);</code>	
<code>ostream& write(const char_type *, streamsize);</code>	
<code>ostream& flush();</code>	
<code>pos_type tellp();</code>	
<code>ostream& seekp(pos_type p);</code>	
<code>ostream& seekp(off_type p, ios_base::seekdir d);</code>	
Gestion des manipulateurs	
<code>ostream& operator<<(ostream& (*pf)(ostream &));</code>	
<code>ostream& operator<<(ios& (*pf)(ios &));</code>	
<code>ostream& operator<<(ios_base& (*pf)(ios_base &));</code>	

TABLE 4.4 – Les principales méthodes de la classe `ostream`.

Des manipulateurs (cf. tableau 4.5) sont déclarés dans l'en-tête `<iomanip>`. Grâce à une surcharge de l'opérateur `<<`, ils permettent d'agir sur les flux de sortie « à la volée », c.-à-d. avec la même syntaxe que celle qui est utilisée pour envoyer des données dans un flux.

Exercice 4.2 Combien de fonctions ou méthodes sont appelées dans l'instruction ci-dessous ?

```
std::cout << "Hello world" << std::endl;
```

Manipulateurs simples		
<code>endl</code>	<code>hex</code>	<code>showbase</code>
<code>ends</code>	<code>oct</code>	<code>noshowbase</code>
<code>flush</code>	<code>dec</code>	<code>showpoint</code>
<code>boolalpha</code>	<code>fixed</code>	<code>noshowpoint</code>
<code>noboolalpha</code>	<code>scientific</code>	<code>showpos</code>
<code>left</code>	<code>uppercase</code>	<code>noshowpos</code>
<code>right</code>	<code>nouppercase</code>	<code>unitbuf</code>
<code>internal</code>		<code>nounitbuf</code>
Manipulateurs avec paramètre		
<code>setw(int)</code>	<code>setbase(int)</code>	<code>setfill(char_type)</code>
<code>setprecision(int)</code>		
<code>resetiosflags(ios_base::formatflags)</code>		
<code>setiosflags(ios_base::formatflags)</code>		

TABLE 4.5 – Les manipulateurs utilisables avec les `ostream`.

4.1.5 La classe istream

C'est l'analogie de la classe `ostream` pour les flux d'entrée. Elle est déclarée dans l'en-tête standard `<iostream>`. Ses principales méthodes sont données dans le tableau 4.6.

Constructeur	
istream(streambuf *);	
Entrée et position	
istream&	operator>>(T &); T : type de base
int_type	get();
istream&	get(char_type &);
int_type	peek();
istream&	put_back(char_type);
istream&	unget();
istream&	read(char_type *, streamsize);
streamsize	readsome(char_type *, streamsize);
istream&	get(char_type *, streamsize);
istream&	get(char_type *, streamsize, char_type delm);
istream&	getline(char_type *, streamsize);
istream&	getline(char_type *, streamsize, char_type delm);
istream&	ignore(streamsize n, int_type d=traits::eof);
int	sync();
pos_type	tellg();
streamsize	gcount() const;
istream&	seekg(pos_type p);
istream&	seekg(off_type p, ios_base::seekdir d);

TABLE 4.6 – Les principales méthodes de la classe `istream`.

Les manipulateurs suivants peuvent agir sur un `istream` :

hex oct dec boolalpha noboolalpha skipws noskipws ws

La gestion de ces manipulateurs passe par la définition des méthodes suivantes dans la classe `istream`.

```
istream& operator>>( istream& (*pf)(istream &) );
istream& operator>>( ios& (*pf)(ios &) );
istream& operator>>( ios_base& (*pf)(ios_base &) );
```

4.1.6 Fichiers

Le fichier d'en-tête `<fstream>` fournit les déclarations des classes `fstream`, `ifstream` et `ofstream`. Quelques méthodes propres aux fichiers (ouverture à partir d'un nom, fermeture, état) sont données dans le tableau 4.7.



Attention aux flux de fichiers passés comme arguments de fonctions. [...]

fstream

```
fstream();
fstream(const char*, ios_base::openmode m=ios_base::out|ios_base::in);
void open(const char *p, ios_base::openmode m=ios_base::in|ios_base::out);
bool is_open() const;
void close();
```

ofstream

```
ofstream();
ofstream(const char *p, ios_base::openmode m=ios_base::out);
void open(const char *p, ios_base::openmode m=ios_base::out);
bool is_open() const;
void close();
```

ifstream

```
ifstream();
ifstream(const char *p, ios_base::openmode m=ios_base::in);
void open(const char *p, ios_base::openmode m=ios_base::in);
bool is_open() const;
void close();
```

TABLE 4.7 – Les principales méthodes des classes *fstream.

4.1.7 Les flux chaînes de caractères

Ils sont définis dans l'en-tête standard `<sstream>`. Il s'utilisent bien entendu comme des flux génériques, et les fonctions utiles qui permettent les conversions avec des chaînes standard sont données dans le tableau 4.8.

stringstream

```
stringstream(ios_base::openmode m=out|in);
explicit stringstream(string &s, openmode m=out|in);
string str() const;
void str(const string &) const;
```

ostringstream

```
ostringstream();
explicit ostringstream(string &);
```

istringstream

```
istringstream();
explicit istringstream(string &);
```

TABLE 4.8 – Les principales méthodes des classes *stringstream.

4.2 La classe string des chaînes de caractères

Elle est définie dans le fichier d'en-tête `<string>` et possède les méthodes listées dans le tableau 4.9.

Quelques méthodes

	<code>string();</code>
	<code>string(const char *);</code>
	<code>string(const char *, size_type);</code>
<code>const char&</code>	<code>at(size_type);</code>
<code>char&</code>	<code>at(size_type);</code>
<code>const char&</code>	<code>operator[](size_type);</code>
<code>char&</code>	<code>operator[](size_type);</code>
<code>string&</code>	<code>operator=(const char *);</code>
<code>bool</code>	<code>empty() const;</code>
<code>size_t</code>	<code>length() const;</code>
<code>size_t</code>	<code>size() const;</code>
<code>const char*</code>	<code>c_str() const;</code>
<code>size_type</code>	<code>copy(char*, size_type, size_pos p=0) const;</code>
<code>int</code>	<code>compare(char *) const;</code>
<code>int</code>	<code>compare(string &) const;</code>
<code>string&</code>	<code>operator+=(const string &);</code>
<code>string&</code>	<code>operator+=(const char *);</code>
<code>string&</code>	<code>insert(size_type, const string &);</code>
<code>string&</code>	<code>insert(size_type, const char *);</code>
<code>size_type</code>	<code>find(const string &, size_type i=0);</code>
<code>size_type</code>	<code>find(const char *, size_type i=0);</code>
<code>string</code>	<code>substr(size_type i, size_type n);</code>

Fonctions et opérateurs

<code>bool</code>	<code>operator==(const string &, const string &);</code>
<code>bool</code>	<code>operator==(const string &, const char &);</code>
<code>bool</code>	<code>operator==(const char *, const string &);</code>
<code>string</code>	<code>operator+(const string &, const string &);</code>

TABLE 4.9 – Les principales méthodes de la classe `string` et quelques fonctions associées.



2^e année – Spécialité Informatique

Année 2024 – 2025

Programmation orientée objet en C++

Notes de cours

Sébastien Fourey

Chapitre 5. Conteneurs et algorithmes de la bibliothèque standard

Version du 3 septembre 2024



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

Chapitre 5

Conteneurs et algorithmes de bibliothèque standard

La bibliothèque standard du langage C fournit des fonctions que l'on peut qualifier de bas niveau en comparaison des possibilités offertes, par exemple, par les kits de développement du langage Java. La bibliothèque standard C++ se place à un niveau intermédiaire. Elle reste très rudimentaire, comparée à l'API¹ de Java, mais elle offre une batterie de modèles de classes et de fonctions qui dispensent le programmeur de la redéfinition de structures de données classiques, mais aussi de la réécriture d'algorithmes usuels portant sur ces structures. A titre d'exemple, nous donnons ici une liste non exhaustive de structures et d'algorithmes qu'un programmeur C++ ne devrait jamais être amené à récrire :

- tableau de taille évolutive ;
- liste chaînée ;
- tas ;
- tableau associatif ;
- ensemble et opérations ensemblistes usuelles ;
- fusion de deux séquences ordonnées ;
- remplacement des éléments d'une séquence qui vérifient une condition ;
- etc.

5.1 Présentation

Comme son nom le rappelle, tous les symboles de la bibliothèque standard se trouvent dans l'espace de noms `std`. Ce préfixe est volontairement omis dans tout ce chapitre. La figure 5.1 montre la hiérarchie des modèles de classes définis par la bibliothèque. Ce sont les classes appelées *classes conteneurs*, les *adaptateurs* et les *itérateurs*. On distingue généralement deux catégories pour les classes conteneurs :

- Les *conteneurs séquentiels* dans lesquels les éléments sont classés en fonction de leur ordre d'insertion ou bien la position qui a été précisée lors de cette insertion. Les éléments ne sont pas nécessairement stockés de manière contiguë en mémoire, même si cette propriété

1. Application Programming Interface

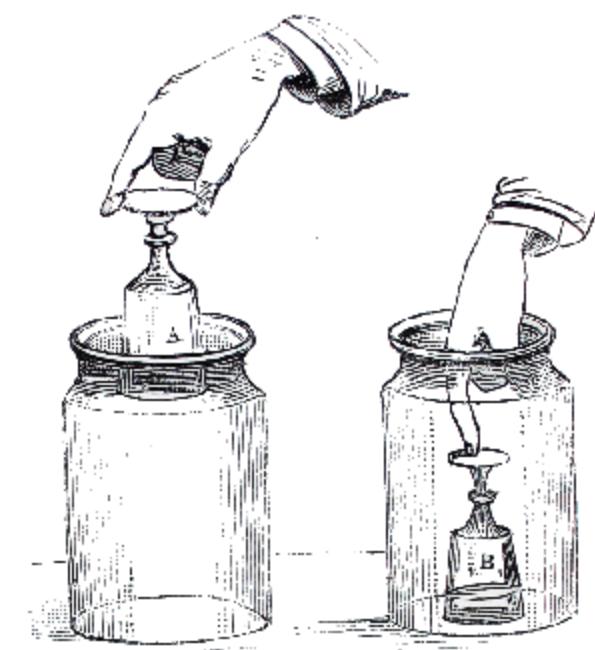


Fig. 8. — Au moment où il touche l'eau, le petit verre est plein d'air en A.

Fig. 9. — Quand le petit verre est au fond, l'air se resserre, se comprime, en B.

est vraie pour le modèle `vector<>` par exemple².

- Les conteneurs ordonnés pour lesquels l’organisation des données en mémoire tire profit d’une relation d’ordre sur les éléments qu’ils contiennent (cf. cours d’algorithmique au sujet de la recherche dichotomique, de la structure de tas, des B-arbres, des arbres de recherche équilibrés, etc.)

Une troisième catégorie concerne les modèles qui *utilisent* des conteneurs pour offrir les services d’autres structures classiques : les piles, les files, et les files de priorité (5.1.3).

Une liste des méthodes essentielles spécifiques ou partagées par les différents types de conteneurs sera donnée dans la section 5.2. La section 5.5 dresse la liste des algorithmes génériques qui reposent en majorité sur l’utilisation des itérateurs (§ 5.4) et aussi pour certains sur la notion d’objet fonction (§ 5.6).

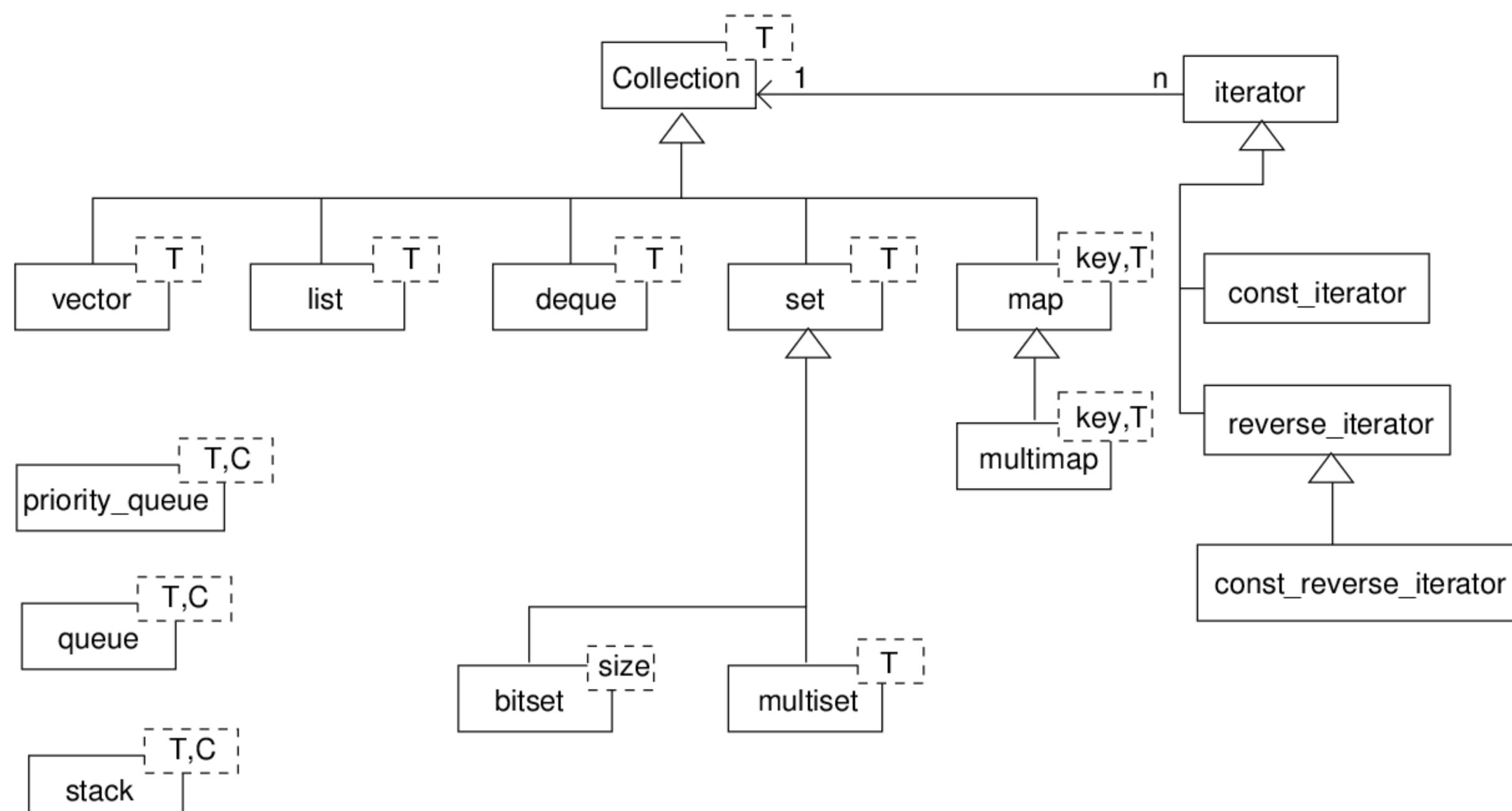


FIGURE 5.1 – Diagramme des quelques modèles de classes définis dans la bibliothèque standard (C++ 2003). Le type `Collection` représenté ici n’est pas un type de la bibliothèque ; il n’existe dans ce diagramme qu’à des fins d’illustration.

5.1.1 Les conteneurs séquentiels

Ils préservent l’ordre dans lequel les éléments sont insérés.

- **deque** (en-tête `<deque>`)
Modèle qui implémente une queue à double entrée. L’insertion et la suppression est faite en temps constant ($O(1)$) en début et fin de séquence. Ces deux opérations s’exécutent en $O(n)$ ailleurs, où n est le nombre d’éléments. L’accès direct à une position donnée se fait en temps constant ($O(1)$).
- **list** (en-tête `<list>`)
Modèle qui implémente une liste doublement chainée. L’insertion et la suppression est effectuée en $O(1)$ n’importe où. L’accès direct est impossible.

2. C'est d'ailleurs là une propriété, essentielle, précisée par la norme de 2003, qui était absente de la norme de 1998 !

- **vector** (en-tête `<vector>`)

Modèle des tableaux dont la taille peut varier en fonction des besoins. Les opérations d'insertion et de suppression ont un coût faible en fin de tableau, elle sont plus longues ailleurs. L'accès direct est possible. Cette classe doit être utilisée pour les tableaux là où on aurait recours en C à la fonction `realloc`. En effet, la norme stipule que les éléments sont stockés de façon contiguë en mémoire dans un vecteur. Il est donc possible de considérer l'adresse du premier élément comme un tableau C classique.

5.1.2 Les conteneurs ordonnés

Tous les conteneurs de cette catégorie stockent leurs éléments en tirant avantage d'une relation d'ordre qui doit être définie. La relation d'ordre utilisée par défaut correspond à l'opérateur `<` (³), éventuellement surchargé. Il est important de noter que pour les conteneurs ordonnés, deux éléments a et b sont considérés égaux si $\neg(a < b) \wedge \neg(b < a)$. Les conteneurs ordonnés sont au nombre de quatre :

- **set** et **multiset** (en-tête `<set>`)

Ces deux modèles définissent de manière générique les ensembles. La particularité d'un **multiset** est qu'il peut contenir plusieurs éléments égaux au sens de la relation d'ordre utilisée.

- **map** et **multimap** (en-tête `<map>`)

Les éléments sont des paires (clé, valeur)⁴ et la rapidité d'accès aux éléments par leur clé est obtenue grâce à l'utilisation d'une relation d'ordre portant uniquement sur ces clés, et pas sur les valeurs associées. Enfin, une **multimap** peut contenir plusieurs paires ayant une même clé, au contraire d'une **map**.

Les conteneurs **set** et **map** effectuent une insertion, une suppression ou une recherche en $O(\log(n))$, si n est le nombre d'éléments présents.

5.1.3 Les adaptateurs

Ces trois modèles (file de priorité, file et pile) sont en plus paramétrés par le type de conteneur qu'ils utilisent pour stocker leurs éléments. On conçoit en effet aisément qu'un modèle de file peut être implémenté à l'aide d'une liste ou bien d'une queue à double entrée. Notez qu'il s'agit bien d'adaptateurs au sens du *design pattern* de même nom : ils n'offrent pas tous les services communs aux autres conteneurs, mais ils ont leur propres interfaces. Par exemple, l'utilisation d'un itérateur sur une pile n'a pas de sens ; le modèle n'en définit donc pas.

- **priority_queue** (en-tête `<queue>`)

Implémente un tas. (Composition avec une **deque** ou un **vector** pour l'accès direct.)

- **queue** (en-tête `<queue>`)

Implémente une file FIFO. (Composition avec une **deque** ou une **list**.)

- **stack** (en-tête `<stack>`)

Implémente une pile. (Composition avec une **deque**, une **list** ou un **vector**.)

Exercice 5.1 Où est le haut de la pile lorsque qu'une instance de **stack** est composée avec un **vector** ?

3. Attention, `<` n'est pas une relation d'ordre au sens mathématique. L'appellation est ici un léger abus de langage.

4. Voir section 5.2.5.

5.1.4 C++11 : conteneurs additionnels

Le C++11 apporte son lot de nouveaux conteneurs :

- **array** (en-tête `<array>`)

Définit un conteneur séquentiel, similaire à un tableau, *dont la taille est connue à la compilation*. Il peut être parcouru comme un conteneur séquentiel à l'aide d'un *bidirectional iterator* (section 5.4), gère les débordements, possède une sémantique de copie par valeur et des méthodes usuelles (`size()`, etc.). Le listing 5.1 illustre son utilisation.

- **forward_list** (en-tête `<forward_list>`)

Type similaire à `std::list` à ceci près qu'il n'est itérable qu'en avant à l'aide d'un *forward iterator* (cf. section 5.4). Implémenté par une liste simplement chainée, il est de ce fait un peu plus efficace qu'une `std::list` en terme d'occupation mémoire.

- **unordered_set** (en-tête `<unordered_set>`)

Ensemble dont les éléments ne sont pas nécessairement comparables, basé sur une table de hachage. La recherche, l'insertion et la suppression d'éléments se fait en moyenne en $O(1)$.

- **unordered_multiset** (en-tête `<unordered_set>`)

Idem que précédemment mais une même valeur peut apparaître plusieurs fois.

- **unordered_map** (en-tête `<unordered_map>`)

Tableau associatif pour lequel les clés ne sont pas nécessairement comparables (une fonction de hachage est utilisée). La recherche, l'insertion et la suppression d'éléments se fait en moyenne en $O(1)$.

- **unordered_multimap** (en-tête `<unordered_map>`)

Idem que précédemment mais une même clé peut apparaître plusieurs fois.

```

1 #include <array>
2 #include <iostream>
3 using namespace std;
4
5 int main() {
6     array<int, 5> numbers{1, 2, 3, 4, 5};
7
8     cout << numbers[10] << endl; // Dangerous!
9     try {                                // Better
10        cout << numbers.at(10) << endl;
11    } catch (const out_of_range & e) {
12        cerr << "Out of range: " << e.what() << "\n";
13    }
14
15    for (int x : numbers) {
16        cout << x << endl;
17    }
18 }
```

Listing 5.1 – Utilisation d'un `array`.

5.2 Les conteneurs

Nous énumérons dans cette section les méthodes spécifiques à chaque type de conteneur mais aussi celles qui sont communes à plusieurs d'entre eux. Au préalable, il est nécessaire de connaître un certain nombre de synonymes de types qui sont définis *dans* les modèles de classes conteneurs.

<code>value_type</code>	Synonyme du type des éléments.
<code>size_type</code>	Type des indices et des tailles.
<code>difference_type</code>	Type des différences entre itérateurs.
<code>iterator</code>	Itérateur ($\simeq \text{value_type}^*$).
<code>const_iterator</code>	Itérateur ($\simeq \text{const value_type}^*$).
<code>reference</code>	$\simeq \text{value_type} \ \&$.
<code>const_reference</code>	$\simeq \text{const value_type} \ \&$.
<code>value_compare</code>	Type du comparateur. (Conteneurs ordonnés.)
<code>key_type</code>	Type des clés. (Conteneurs associatifs.)
<code>mapped_type</code>	Type des valeurs associées. (Conteneurs associatifs.)
<code>key_compare</code>	Type du comparateur de clés. (Conteneurs associatifs.)

5.2.1 Méthodes communes à tous les conteneurs

Elles sont données dans le tableau 5.1. Notez que si tous les conteneurs permettent la suppression d'un élément ou d'un intervalle désigné à l'aide d'itérateurs (méthodes `erase()`), mais aussi l'insertion d'un élément à une position donnée ; la complexité de ces opérations reste liée au type de conteneur. On rappelle par exemple que l'insertion est faite en temps constant n'importe où dans une liste, elle est relativement plus lente au sein d'un vecteur. (Elle peut en effet dans ce dernier cas nécessiter une réallocation, suivie de la recopie de tous les éléments situés après le point d'insertion.)

Tous les conteneurs

```

        conteneur(const conteneur &);
        ~conteneur();
        void clear();
        size_type size();
        bool empty() const;
        iterator begin() et end();
        iterator rbegin() et rend();
        iterator insert(iterator, const T &);
        iterator erase(iterator);
        iterator erase(iterator f, iterator l);
        const conteneur& operator=(const conteneur &);
    
```

TABLE 5.1 – Méthodes des conteneurs standard.

5.2.2 Méthodes spécifiques des conteneurs séquentiels

Elles sont détaillées dans le tableau 5.2. Notez que les vecteurs et les queues à double entrée surchargent l'opérateur [] pour l'accès direct aux éléments d'indices 0 à `size()-1`. Une méthode

`at()` offre le même service mais diffère de l'opérateur dans le cas d'un dépassement d'indice. En effet, le comportement de l'opérateur `[]` est indéfini dans ce cas alors que la méthode `at()` lève une exception de type `std::out_of_range` (cf. chapitre 6).

La capacité d'un vecteur (`capacity()`), qu'il ne faut pas confondre avec sa taille, est le nombre d'éléments qu'il pourra contenir sans qu'il y ait réallocation de mémoire. En effet, une insertion en fin de vecteur ne provoque qu'occasionnellement une réallocation. La plupart du temps, un vecteur n'occupe qu'une partie de la zone mémoire qui lui est réellement allouée.

Tous les conteneurs séquentiels
<code>conteneur(size_type, const value_type &)</code>
<code>vector</code>
<code>size_type capacity()</code> <code>void reserve(size_type)</code>
<code>deque vector</code>
<code>/const/ value_type& at(size_type n) /const/ ; {out_of_range}</code> <code>/const/ value_type& operator[](size_type n) /const/ ;</code>
<code>deque list vector</code>
<code>/const/ value_type& back() /const/ ;</code> <code>/const/ value_type& front() /const/ ;</code> <code>void pop_back()</code> <code>void push_back(const T &);</code> <code>void resize(size_type n, T t = T());</code> <code>iterator insert(iterator, const T &);</code>
<code>deque list</code>
<code>void push_front(const T &);</code> <code>void pop_front();</code>

TABLE 5.2 – Méthodes des conteneurs séquentiels.

5.2.3 Méthodes spécifiques des conteneurs ordonnés

Les principales méthodes de ces conteneurs sont données dans le tableau 5.3. Mis à part le constructeur, toutes les méthodes de ce tableau sont rendues efficaces grâce à l'utilisation de la relation d'ordre définie entre les éléments. Par exemple, la recherche d'un élément dans un ensemble à partir de sa valeur, comme la recherche d'une paire dans un tableau associatif à partir d'une valeur de clé, est réalisée avec une complexité en $O(\log(n))$.

5.2.4 Les adaptateurs

Les opérations classiques portant sur les piles, les files et les tas sont présentées dans le tableau 5.4. La figure 5.2 montre que le modèle `stack` est une classe paramétrée par le type `C` de conteneur utilisé pour stocker les éléments.

5.2.5 Le modèle de classe pair (couples)

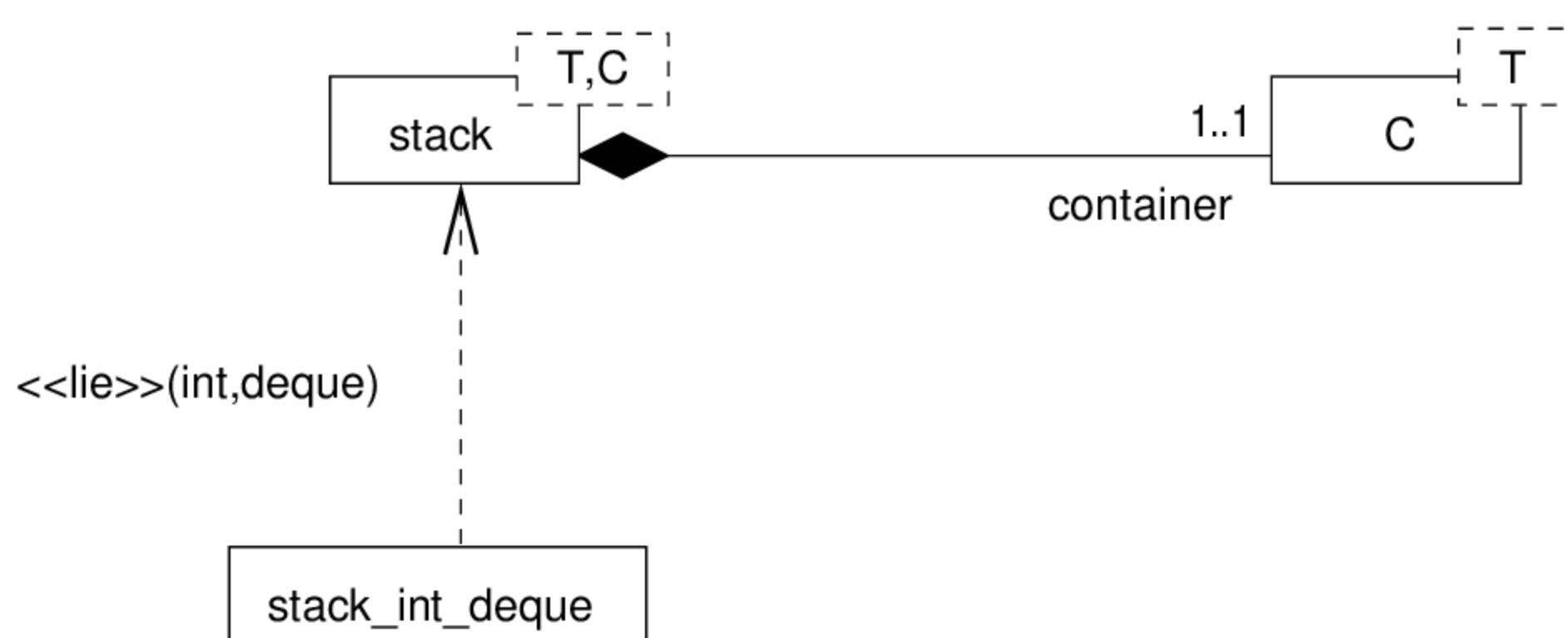
Il est défini dans dans `<utility>`. Le listing 5.2 donne un aperçu de sa définition.

Tous les conteneurs ordonnés
<i>conteneur(key_compare)</i>
map multimap set multiset
<pre>/const/_iterator find(const key_type &) /const; /const/_iterator lower_bound(const key_type &) /const; /const/_iterator upper_bound(const key_type &) /const; pair<it.,it.> equal_range(const key_type &); size_type erase(const key_type &) size_type count(const key_type &) pair<it., bool> insert(const value_type &)</pre>
map
<pre>mapped_type& operator[](const key_type&);</pre>

TABLE 5.3 – Méthodes des conteneurs ordonnés.

Tous les adaptateurs
<i>size_type size();</i>
stack
<pre>/const/ value_type& top() /const/; bool empty() const; void pop(); void push(const value_type &);</pre>
queue
<pre>bool empty() const; /const/ value_type& back() /const/; /const/ value_type& front() /const/; void pop(); void push(const value_type &);</pre>
priority_queue
<pre>bool empty() const; void pop(); void push(const value_type &); const value_type& top() const;</pre>

TABLE 5.4 – Méthodes des classes d'adaptateurs.

FIGURE 5.2 – Diagramme de classes du modèle **stack**.

```

1 template <typename T1, typename T2>
2 struct pair {
3     T1 first;
4     T2 second;
5     pair(const T1 & x, const T2 & y);
6     // ...
7 };

```

Listing 5.2 – Extrait de la définition du modèle `pair<>` de la bibliothèque standard.

Le modèle de fonction `make_pair` est aussi défini (cf. listing 5.3). Il permet, grâce à la syntaxe d’instanciation des modèles de fonctions en utilisant les `<>`, de créer des paires temporaires sans ambiguïté sur les types.

```

1 template <class T1, class T2>
2 pair<T1, T2> std::make_pair(const T1 &, const T2 &);

```

Listing 5.3 – Prototype du modèle de fonction `make_pair`.

5.2.6 C++11 : le modèle de classe `tuple` (*n*-uplet)

Le modèle de classe `tuple`, défini dans l’en-tête `<tuple>`, permet de représenter des séquences ordonnées et de taille fixe, de données *hétérogènes*. Autrement dit, il implémente la notion de *n*-uplet. Il peut être vu comme une structure (c.-à-d. `struct`) sans nom.

Comme pour le type `pair`, il existe un modèle de fonction `make_tuple` permettant de créer des *n*-uplets par déduction automatique des types utilisés comme arguments.

Un exemple d’utilisation est donné dans le listing 5.4.

Comme illustré par la ligne 21 du listing 5.4, il est possible de comparer des `tuple` pour peu que les types des éléments soient eux aussi tous comparables.

Remarque 5.1 *Le nombre maximum d’éléments dans un tuple dépend de l’implémentation.*

Pour finir, signalons que le modèle `tuple` ne peut exister que grâce à une nouveauté du C++11 : les modèles à liste de paramètres variables (cf. [12], *Variadic Templates*).

5.3 Exemples d’utilisations

5.3.1 Instanciations

Le listing ci-dessous montre quelques exemples d’instanciations de modèles de classes conteneurs.

```

1 std::stack<int> pile;
2 std::map<string, int> ages;
3 std::map<float, float> f;
4 std::map<double, pair<double, double> > courbe;

```

```
1 #include <iostream>
2 #include <string>
3 #include <tuple>
4 using namespace std;
5
6 void display(const std::tuple<string, string, int> & person) {
7     cout << "Firstname : " << get<0>(person) << endl;
8     cout << "Lastname : " << get<1>(person) << endl;
9     cout << "Age : " << get<2>(person) << endl;
10 }
11
12 int main() {
13     std::tuple<string, string, int> author{"Sebastien", "Fourey", 20};
14     int age = get<2>(author);
15     display(author);
16     display(std::tuple<string, string, int>{"John", "Doe", 33});
17     string firstname{"John"};
18     string lastname{"Doe"};
19     display(make_tuple(firstname, lastname, 33));
20
21     cout << (make_tuple(1, 2, 3) > make_tuple(1, 1, 1)) // 1
22         << endl;
23 }
```

Listing 5.4 – Utilisation du modèle tuple.

5.3.2 Premier exemple

Cet exemple montre l'utilisation d'une queue à double entrée et d'un itérateur sur ce conteneur. (La notion d'itérateur fait l'objet de la section 5.4.)

```

1 #include <deque>
2 #include <iostream>
3 using std :: cout ;
4 using std :: endl ;
5
6 int main() {
7     std :: deque<char> a_string ;
8     std :: deque<char>::iterator it , end ;
9
10    cout << a_string . max_size () << endl ;
11
12    a_string . push_back( '@' );
13    a_string . push_back( 'b' );
14    a_string . push_back( 'o' );
15    a_string . push_back( 'n' );
16    a_string . push_back( 'j' );
17    a_string . push_back( 'o' );
18    a_string . push_back( 'u' );
19    a_string . push_back( 'r' );
20    a_string . pop_front () ;
21
22    end = a_string . end () ;
23    for ( it = a_string . begin () ; it != end ; ++it ) {
24        cout << *it ;
25    }
26    cout << endl ;
27 }
```

Listing 5.5 – Exemple d'utilisation d'un modèle de classe conteneur de la bibliothèque standard.

5.3.3 Deuxième exemple

```

1 #include <algorithm>
2 #include <iostream>
3 #include <iterator>
4 #include <list>
5 #include <vector>
6 using std :: cout ;
7 using std :: endl ;
8
9 int main() {
10    std :: vector<int> v(10) ;
11    std :: vector<int>::iterator it ;
```

```

13  for (int i = 0; i < 10; i++) {
14      v[i] = i;
15  }
16
17  v.push_back(10);
18  v.insert(v.begin(), 5); // No push_front method
19
20  it = v.begin() + 5;
21  v.erase(it);
22
23  for (it = v.begin(); it != v.end(); ++it) {
24      cout << *it << " ";
25  }
26  cout << endl;
27  // 5 0 1 2 3 5 6 7 8 9 10
28 }
```

Listing 5.6 – Exemple d'utilisation du modèle de classe `vector`.

5.3.4 Troisième exemple

Ce dernier exemple illustre l'utilisation d'un tableau associatif `map`. La syntaxe courante d'accès en consultation comme en modification utilise l'opérateur `[]`. Dans ce listing, on montre toutefois qu'une `map` n'est autre qu'un ensemble de paires (cf. l'utilisation de la méthode `insert()` et du modèle de fonction `make_pair<>()`).



Dans une `map`, l'insertion d'une paire à l'aide la méthode `insert()` suit la règle qui est valable pour les ensembles : si la clé est déjà présente, aucune insertion ni modification n'est faite ! (Il vaut mieux, dans certains cas, utiliser les crochets.)

```

1 #include <iostream>
2 #include <map>
3 #include <string>
4 using namespace std;
5
6 int main() {
7
8     map<int, string> v;
9     map<int, string>::iterator i;
10    pair<map<int, string>::iterator, bool> p;
11
12    v.insert(make_pair(1, string("OK")));
13    v.insert(make_pair(2, string("Coral")));
14    v.insert(make_pair(3, string("Tango")));
15    p = v.insert(make_pair(3, string("Tango2"))); // void
16
17    cout << p.second << endl;
18
19    v.insert(make_pair(4, string("Charly")));
20    v[5] = string("Bravo");
```

```

21   v[5] = string("Daddy");
22
23   i = v.find(2);
24   v.erase(i);
25
26   for (i = v.begin(); i != v.end(); i++) {
27     cout << "(" << i->first << "," << i->second << ")";
28   }
29   cout << endl;
30 }
```

Listing 5.7 – Exemple d'utilisation d'un modèle de classe conteneur de la bibliothèque standard.

Le programme du listing 5.7 produit la sortie suivante :

```

0
(1,OK)(3,Tango)(4,Charly)(5,Daddy)
```

5.4 Les itérateurs

Les itérateurs sont la clé de voûte des algorithmes que la bibliothèque standard met à la disposition du programmeur. Ils y sont définis d'une façon légèrement différente de celle qui a été vue dans le cadre des *design patterns* (cf. figure 5.3(b)). En C++, un itérateur est plus proche d'une abstraction de la notion de pointeur⁵. Notamment, les itérateurs de la bibliothèque standard ne disposent pas de méthode `first()` ni de méthode `isDone()`. En effet, un itérateur n'a que très peu de données sur le conteneur qui lui est associé. De ce fait :

- C'est le conteneur qui fournit un itérateur *pointant* sur son premier élément via sa méthode `begin()` (car un itérateur ne sait pas se positionner de lui-même sur le début) ;
- Un conteneur peut construire un itérateur *pointant* sur un élément virtuel qui succède au dernier élément, c'est l'itérateur retourné par la méthode `end()` (car un itérateur ne sait pas signaler de lui-même qu'il a atteint la fin du conteneur).

Le second point, ajouté à la possibilité de comparer deux itérateurs à l'aide de l'opérateur `==`, offre finalement l'équivalent de la méthode `isDone()` du *design pattern*. (Voir la boucle `for` du listing 5.5).

Une autre différence, qui est essentiellement syntaxique, réside dans l'utilisation des opérateurs d'incrémentation (`++`) et de déréférencement (`*`) qui remplacent respectivement les méthodes `next()` et `currentItem()`.

On peut distinguer plusieurs familles d'itérateurs (cf. figure 5.3) :

- Les `input iterator` et les `output iterator` servent de base pour les itérateurs de *flux*.
- Les `forward iterator` combinent les fonctionnalités des deux précédentes familles. Ils peuvent donc être utilisés partout où l'une des deux est attendue.
- Les `bidirectional iterator` servent pour les conteneurs associatifs et les listes.
- Les `random iterator` servent pour les *vecteurs* et les *queues*.



Les itérateurs n'offrent que peu de sécurité, à l'instar des pointeurs. Par exemple, rien n'empêche d'incrémenter un itérateur alors qu'il est déjà positionné à la fin d'un conteneur...

5. A tel point qu'un itérateur est parfois effectivement un simple pointeur.

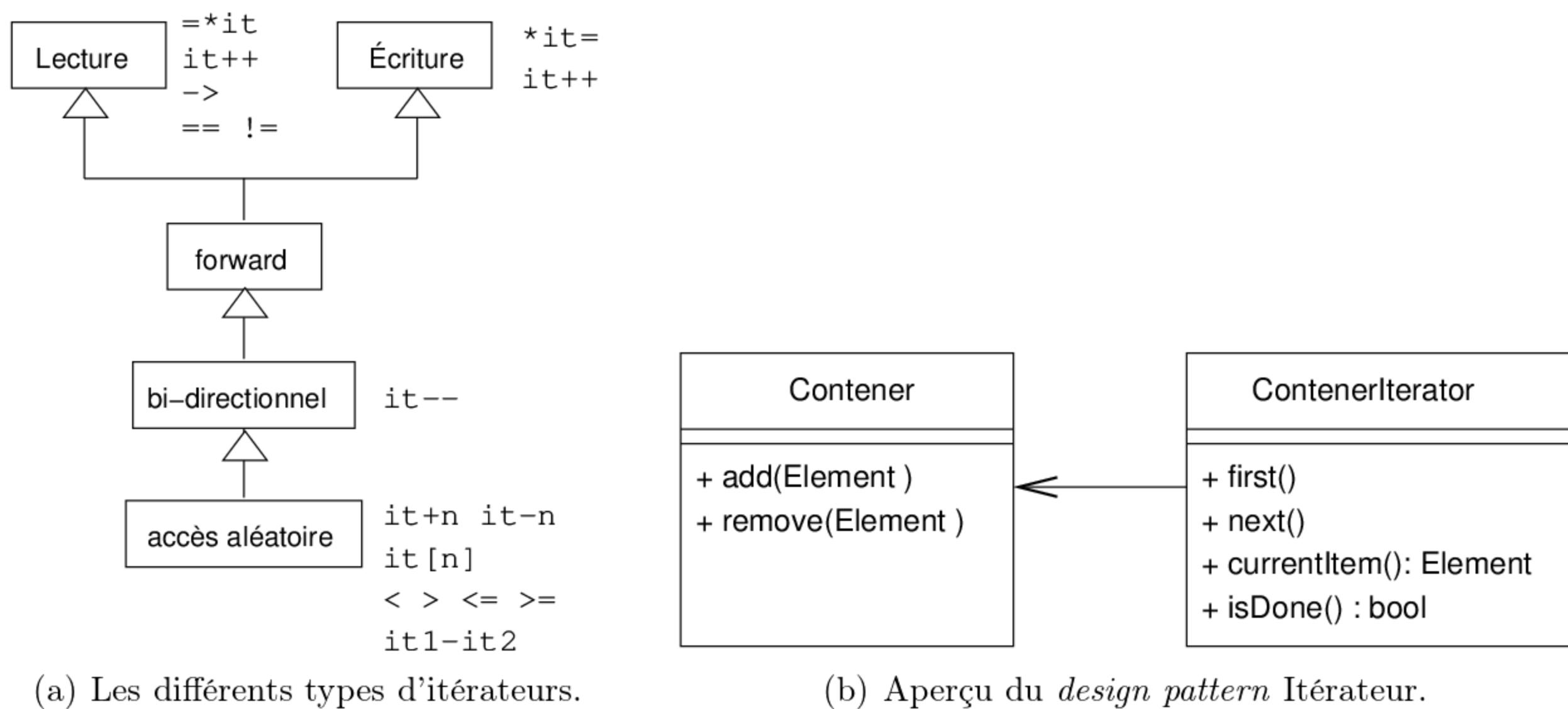


FIGURE 5.3 – Iterateurs

5.4.1 Le modèle iterator_traits

Ce modèle (un *trait*) permet d'écrire des algorithmes génériques en utilisant une syntaxe valable pour des itérateurs qui sont des classes mais aussi pour ceux qui sont de simples pointeurs. (Il existe un modèle partiellement spécialisé `iterator_traits<T*>`.) Le listing 5.8 donne un aperçu des synonymes de types définis dans ces traits.

```

1 | template <typename I>
2 | struct iterator_traits {
3 |     typedef typename I::iterator_category iterator_category;
4 |     typedef typename I::value_type value_type;
5 |     typedef typename I::difference_type difference_type;
6 |     typedef typename I::pointer pointer;
7 |     typedef typename I::reference reference;
8 | };
  
```

Listing 5.8 – Les *traits* d'un itérateur.

L'utilité de ce trait est illustrée par le modèle de fonction `distance()` donné dans la section suivante.

5.4.2 Deux modèles de fonctions sur les itérateurs

Pour illustrer le principe des algorithmes génériques définis dans la bibliothèque standard (Section 5.5), qui utilisent intensivement les itérateurs ; nous donnons ici deux exemples très simples de modèles de fonctions. (Il sont définis dans l'en-tête `<iterator>`.)

La fonction `distance()`

Le listing suivant (5.9) est une écriture possible de l'algorithme `distance`. Notez cependant qu'il peut être spécialisé pour donner une réponse en temps constant, et non pas linéaire, si le conteneur le permet (p. ex. `std::vector`).

```
1 template <typename I>
2 typename iterator_traits<I>::difference_type
3 distance(I first, I last) {
4     typename iterator_traits<I>::difference_type d = 0;
5     while (first++ != last) {
6         d++;
7     }
8     return d;
9 }
```

Listing 5.9 – Le modèle de fonction `distance<>()`.

Il est important de noter que sans le trait `iterator_traits<>` précédemment défini, il serait impossible de définir correctement et de manière générique le type de retour de cette fonction (c.-à-d. `difference_type`). Le même problème se pose, et se résoud, si on a besoin d'utiliser le type des éléments « pointés » lorsque l'on ne dispose que du paramètre de type **I** (pour itérateur).

La fonction `advance()`

Ci-dessous, la version « basique » de cette fonction. En réalité, elle est spécialisée selon le type d’itérateur.

```

1 template <typename Iterator, typename Distance>
2 void advance(Iterator & i, Distance n) {
3     while (n--) {
4         ++i;
5     }
6 }
```

Listing 5.10 – Le modèle de fonction `advance<>()`.

5.5 Les algorithmes (<algorithm>)

Dans cette section, les principaux algorithmes définis dans la bibliothèque standard sont énumérés sous la forme de tableaux, avec les notations données ci-après. Tous ces algorithmes s’appliquent sur des séquences qui sont spécifiées à l’aide d’itérateurs. D’une manière générale, toute séquence d’entrée peut être précisée sous la forme de deux itérateurs : son début et sa fin. Une séquence de sortie est souvent déterminée sans ambiguïté par son début lorsqu’une séquence d’entrée a été précisée. (Les algorithmes `fill_n` et `generate_n` font parties des exceptions.)

Notez que beaucoup d’algorithmes utilisent aussi, en plus des itérateurs, des objets fonctions. Plusieurs modèles sont définis (dans `<functional>`) pour faciliter l’utilisation de ce type de classe. Ils seront présentés dans la section 5.6.

Les notations suivantes sont utilisées dans les tableaux de cette section.

r random access iterator	V valeur	p prédict unaire
b bidirectionnal iterator	R référence	p2 prédict binaire
f forward iterator	CR référence constante	c fonction de comparaison
i input iterator	P paire	F fonction unaire
o output iterator	B booléen	F2 fonction binaire
		n compteur

5.5.1 Opérations sans modification

Ces algorithmes (tableau 5.5) ne modifient pas les séquences données comme paramètres.

5.5.2 Opérations avec modification

Ces algorithmes (tableau 5.6) modifient la séquence donnée en paramètre lorsqu’elle est unique. Si deux séquences sont données, les deux peuvent être modifiées ou bien uniquement la seconde.

Nom	Retour	Arguments	Description
for_each	F	i,i,F	Applique F sur chaque élément
find	i	i,i,V	Trouve V dans la séquence
find_if	i	i,i,p	Trouve le premier élément satisfaisant p
find_first_of	f	f,f,f,f[,p2]	Trouve le premier élément d'une séquence ayant une correspondance dans une autre
adjacent_find	f	f,f[,p2]	Recherche une paire de valeurs contigües correspondantes
count	diff. type	i,i,V	Compte le nombre d'occurrence de V
count_if	diff. type	i,i,p	Compte le nombre d'éléments satisfaisant p
equal	B	i,i,i[,p2]	Test d'égalité des éléments deux à deux
mismatch	pair<i,i>	i,i,i[,p2]	Recherche la première paire d'éléments différents
search	f	f,f,f,f[,p2]	Recherche la deuxième séquence dans la première
find_end	f	f,f,f,f[,p2]	Recherche la dernière occurrence de la seconde séquence dans la première
search_n	f	f,f,n,V[,p2]	Recherche une séquence de n valeurs correspondantes

TABLE 5.5 – Algorithmes en « lecture seule ».



La convention, sur l'ordre des arguments, choisie dans la bibliothèque standard pour les algorithmes qui ont une source et une destination (comme `copy()`) est l'inverse de celle utilisée dans les fonctions de la bibliothèque C (`memcpy()`, `strcpy()`, `memmove()`, etc.).

Nom	Retour	Arguments	Description
copy	o	i,i,o	Copie une séquence vers un itérateur de sortie
transform	o	i,i,o,F	Produit une transformation de son entrée par F
transform	o	i,i,i,o,F2	Transformation de deux séquences par une fonction binaire
unique	f	f,f[,p2]	Réordonne la séquence en plaçant les éléments uniques en tête
unique_copy	o	i,i,o[,p2]	Copie en éliminant les doublons
replace	void	f,f,V,V	Remplace une valeur par une autre
replace_if	void	f,f,p,V	Remplace les valeurs satisfaisant p
replace_copy	o	i,i,o,V,V	
replace_copy_if	o	i,i,o,p,V	
remove	f	f,f,V	Supprime les éléments égaux à V (mis en fin de séquence)
remove_if	f	f,f,p	Supprime les éléments satisfaisant p
remove_copy	o	i,i,o,V	Copie tout sauf V
remove_copy_if	o	i,i,o,p	Copie ce qui ne satisfait pas p
fill	void	f,f,V	Remplit la séquence avec V
fill_n	void	o,n,V	Remplit avec n copies de V
generate	void	f,f,g()	Remplit avec le générateur g()
generate_n	void	o,n,g()	remplit avec n appels à g()
reverse	void	b,b	Renverse la séquence
reverse_copy	void	b,b,o	Crée une copie renversée
rotate	void	f,f,f	(begin,middle,last) Effectue une rotation de sorte que middle se retrouve en first
rotate_copy	void	f,f,f,o	Rotation avec recopie
random_shuffle	void	r,r[,g()]	Mélange les éléments
swap_ranges	f	f,f,f	Échange deux séquences

TABLE 5.6 – Algorithmes modifiant une ou plusieurs séquences.

5.5.3 Opérations sur les séquences triées

En dehors des algorithmes dont les noms contiennent le mot « `sort` » (tableau 5.7), tous ces modèles de fonctions portent sur des séquences qui sont supposées *triées*.

Nom	Retour	Arguments	Description
<code>sort</code>	<code>void</code>	<code>r,r[,c]</code>	Trie une séquence
<code>stable_sort</code>	<code>void</code>	<code>r,r[,c]</code>	Préserve l'ordre relatif des éléments égaux
<code>partial_sort</code>	<code>void</code>	<code>r,r,r[,c]</code>	Trie une partie de séquence
<code>partial_sort_copy</code>	<code>void</code>	<code>i,i,r,r[,c]</code>	Tri avec recopie pour remplir la deuxième séquence
<code>nth_element</code>	<code>void</code>	<code>r,r,r[,c]</code>	(first,nth,last) Trie jusqu'à ce que l'élément nth soit le bon
<code>binary_search</code>	<code>bool</code>	<code>f,f,V[,c]</code>	Recherche dichotomique
<code>lower_bound</code>	<code>f</code>	<code>f,f,V[,c]</code>	Premier élément d'une sous-séquence d'éléments égaux
<code>upper_bound</code>	<code>f</code>	<code>f,f,V[,c]</code>	Fin d'une sous-séquence d'éléments égaux
<code>equal_range</code>	<code>pair<f,f></code>	<code>f,f,V[,c]</code>	Intervalle d'éléments égaux
<code>merge</code>	<code>o</code>	<code>i,i,i,i,o[c]</code>	Fusionne deux séquences triées
<code>inplace_merge</code>	<code>void</code>	<code>b,b,b[,c]</code>	Fusionne deux séquences consécutives
<code>partition</code>	<code>b</code>	<code>b,b,p</code>	Partitionne en fonction de p
<code>stable_partition</code>	<code>b</code>	<code>b,b,p</code>	Partitionne et maintient l'ordre relatif

TABLE 5.7 – Algorithmes sur les séquences triées.

Exercice 5.2 Ecrire un programme qui crée un vecteur de 500000 entiers, initialisé avec des nombres aléatoires à l'aide de l'algorithme `generate_n<>()`. Une copie du vecteur sera faite dans un tableau classique. Ensuite, mesurer et comparer les temps d'exécution des opérations suivantes :

- tri du vecteur à l'aide de l'algorithme `sort<>()` ;
- tri du tableau à l'aide de la fonction `qsort()`.

5.5.4 Opérations ensemblistes

Ces algorithmes (tableau 5.8) ne sont utilisables qu'avec des conteneurs ordonnés ou des séquences triées.

Nom	Retour	Arguments	Description
includes	bool	i,i,i,i[c]	Verifie l'inclusion de la deuxième séquence dans la première
set_union	o	i,i,i,i,o[,c]	Union ensembliste
set_intersection	o	i,i,i,i,o[,c]	Intersection ensembliste
set_difference	o	i,i,i,i,o[,c]	Différence entre la première et la deuxième séquence
set_symmetric_difference	o	i,i,i,i,o[,c]	Différence symétrique

TABLE 5.8 – Algorithmes ensemblistes.

5.5.5 Opérations sur les tas

Ces algorithmes (tableau 5.9) permettent de manipuler un conteneur séquentiel comme un tas.

Nom	Retour	Arguments	Description
push_heap	void	r,r[,c]	Ajoute au tas
pop_heap	void	r,r[,c]	Retire la tête du tas
make_heap	void	r,r[,c]	Transforme la séquence en tas
sort_heap	void	r,r[,c]	Transforme le tas en séquence triée

TABLE 5.9 – Algorithmes de tas.

5.5.6 Comparaisons et bornes

Nom	Retour	Arguments	Description
min	V	V,V[,c]	Minimum de deux valeurs
max	V	V,V[,c]	Maximum de deux valeurs
min_element	f	f,f[,c]	Minimum d'une séquence (première occurrence)
max_element	f	f,f[,c]	Maximum d'une séquence (première occurrence)
lexicographical_compare	bool	i,i,i,i[,c]	Comparaison lexicographique

TABLE 5.10 – Algorithmes sur les bornes.

5.5.7 Permutations

Les deux modèles suivants (tableau 5.11) permettent de générer sur place les permutations lexicographiques qui précèdent ou succèdent à une séquence donnée. Elles retournent le booléen `true` s'il reste des séquences à suivre. Si la séquence donnée en argument est la dernière dans l'ordre, alors la plus petite permutation (toujours au sens lexicographique) est produite et la fonction retourne `false`.

Nom	Retour	Arguments	Description
next_permutation	bool	b,b[,c]	Prochaine permutation
prev_permutation	bool	b,b[,c]	Permutation précédente

TABLE 5.11 – Algorithmes de permutations.

Exercice 5.3 Écrire un programme affichant tous les mots de 8 lettres qui sont formés des lettres de « ENSICAEN ».

5.5.8 Algorithmes numériques

Ces algorithmes (tableau 5.12) sont définis dans l'en-tête `<numeric>`.

Nom	Retour	Arguments	Description
accumulate	V	i,i,V[,F2]	Somme d'une valeur et tous les éléments d'une séquence
adjacent_difference	o	i,i[,F2]	Pour la séquence <code><a,b,c,d></code> , retourne <code><a,b-a,c-b,d-c></code>
inner_product	V	i,i,i,V[,F2,F2]	Produit scalaire entre deux séquences, ou accumulation par une fonction des résultats d'une deuxième fonction appliquée terme à terme.
partial_sum	o	i,i,o[,F2]	Calcule les sommes (ou accumulations) partielles

TABLE 5.12 – Algorithmes numériques.

5.5.9 C++11 : nouveaux algorithmes

Les nouveaux algorithmes de la bibliothèque standard sont décrits dans le tableau 5.13.

De plus, quelques fonctions utiles basées sur les listes d'initialisation sont définies dans l'en-tête `<algorithm>`. Elles sont décrites par leur prototype dans le listing 5.11 et un exemple d'utilisation de la fonction `min` « étendue » est donné dans le listing 5.12.

5.6 Les types d'objets fonctions

Comme cela apparaît dans la section précédente, la bibliothèque standard fait un usage important des objets fonctions. Des modèles de structures, classes et fonctions sont logiquement définis pour faciliter la manipulation de ce type d'objet.

Nom	Retour	Arguments	Description
all_of	B	i,i,p	Vérifie que p est vrai pour tout élément
any_of	B	i,i,p	Vérifie que p est vrai pour un des éléments
none_of	B	i,i,p	Vérifie que p n'est vrai pour aucun des éléments
find_if_not	i	i,i,p	Trouve le premier élément qui ne satisfait pas p
copy_if	o	i,i,o,p	Copie uniquement les éléments qui vérifient p
copy_n	o	i,n,o	Copie n éléments
move	o	i,i,o	Déplace un séquence
move_backward	o	i,i,o	Déplace un séquence en l'inversant
partition_copy	pair<o,o>	i,i,o,o,p	Partitionne une séquence en deux groupes (vrai/faux) selon un prédicat
partition_point	i	i,i,p	Premier élément d'une partition qui ne vérifie pas p
is_sorted	B	i,i[,c]	Vérifie qu'une séquence est triée
is_sorted_until	i	i,i[,c]	Trouve la fin de la plus grande sous-séquence triée
is_heap	B	r,r[,c]	Vérifie qu'une séquence a la structure d'un tas
is_heap_until	i	r,r[,c]	Trouve la fin de la plus grande sous-séquence correspondant à un tas
minmax_element	pair<i,i>	i,i[,c]	Recherche les éléments min et max d'une séquence
minmax	pair<CR,CR>	CR,CR[,c]	Retourne les min et max de deux valeurs
iota		f,f,V	Remplit une séquence avec les valeurs successives commençant à V

TABLE 5.13 – Nouveaux algorithmes de la bibliothèque C++11.

```

1 template <typename T>
2 T min( initializer_list<T> t );
3 template <typename T>
4 T min( initializer_list<T> t , Compare comp );
5
6 template <typename T>
7 T max( initializer_list<T> t );
8 template <typename T>
9 T max( initializer_list<T> t , Compare comp );
10
11 template <typename T>
12 pair<const T &, const T &> minmax( initializer_list<T> t );
13 template <typename T>
14 pair<const T &, const T &>
15 minmax( initializer_list<T> t , Compare comp );

```

Listing 5.11 – Min et max à partir de listes d'initialisation.

```

1 #include <algorithm>
2 #include <iostream>
3
4 int main() {
5     int i, j, k, l;
6     std :: cin >> i >> j >> k >> l;
7     std :: cout << std :: min({i, j, k, l}) << std :: endl;
8 }
```

Listing 5.12 – Calcul du minimum de 4 variables.

5.6.1 Structures de base pour les classes d'objets fonctions

Des modèles de structures pour des objets fonctions à un ou deux arguments sont prévus dans la bibliothèque standard. Ils servent de bases aux classes *foncteurs* (c.-à-d. classes d'objets fonctions) en définissant simplement des synonymes pour les types des arguments et valeurs de retour.

Le type unary_function

```

1 template <class Arg, class Result>
2 struct unary_function {
3     typedef Arg argument_type;
4     typedef Result result_type;
5 };
```

Le type binary_function

```

1 template <class Arg1, class Arg2, class Result>
2 struct binary_function {
3     typedef Arg1 first_argument_type;
4     typedef Arg2 second_argument_type;
5     typedef Result result_type;
6 };
```

5.6.2 Exemples

Le modèle de classe d'objets fonctions « plus ».

```

1 template <class T>
2 struct plus : public binary_function<T, T, T> {
3     T operator()(const T & x, const T & y) const {
4         return x + y;
5     }
6 };
```

La classe d'objets fonctions « sinus ».

```
1 | struct sinus : public unary_function<double, double>
2 | {
3 |     double operator ()( double x) {
4 |         return sin (x);
5 |     }
6 | };
```

Un objet fonction accumulateur

Le listing 5.13 donne un exemple d'objet fonction de type « accumulateur » permettant de calculer la somme des éléments sur lesquels il est appliqué.

```

1 #include <iostream>
2 #include <vector>
3
4 template <typename T>
5 class Sum {
6     T value;
7
8 public:
9     Sum(T x = 0) : value(x) {
10 }
11     void operator()(T x) {
12         value += x;
13     }
14     operator T() const {
15         return value;
16     }
17     const T & operator=(const T & x) {
18         value = x;
19     }
20 };
21
22 int main() {
23
24     std::vector<float> v;
25     v.push_back(1.0);
26     v.push_back(0.414);
27
28     Sum<float> s, r;
29     r = std::for_each(v.begin(), v.end(), s);
30
31     std::cout << "The sum is " << r << std::endl;
32     std::cout << "The sum is not " << s << std::endl;
33 }
```

Listing 5.13 – Calcul de la somme des éléments d'un conteneur à l'aide d'un accumulateur.

5.6.3 Les prédictats

Un certain nombre de modèles de classes d'objets fonctions correspondant à des opérateurs du langage est défini dans l'en-tête `<functional>`. Ils sont énumérés dans le tableau 5.14.

Type	Arité	Équivalent
equal_to	Binaire	<code>arg1 == arg2</code>
not_equal_to	Binaire	<code>arg1 != arg2</code>
greater	Binaire	<code>arg1 > arg2</code>
less	Binaire	<code>arg1 < arg2</code>
greater_equal	Binaire	<code>arg1 >= arg2</code>
less_equal	Binaire	<code>arg1 <= arg2</code>
logical_and	Binaire	<code>arg1 && arg2</code>
logical_or	Binaire	<code>arg1 arg2</code>
logical_not	Unaire	<code>!arg</code>

TABLE 5.14 – Les prédictats.

5.6.4 Opérations arithmétiques

Les modèles d'objets fonctions du tableau 5.15 sont définis dans l'en-tête `<functional>`.

Type	Arité	Équivalent
plus	Binaire	<code>arg1 + arg2</code>
minus	Binaire	<code>arg1 - arg2</code>
multiplies	Binaire	<code>arg1 * arg2</code>
divides	Binaire	<code>arg1 / arg2</code>
modulus	Binaire	<code>arg1 % arg2</code>
negate	Unaire	<code>-arg</code>

TABLE 5.15 – Les classes d'objets fonctions pour les opérations arithmétiques de base.

5.6.5 L'éditeur de liaison bind2nd

Ce modèle permet d'utiliser un objet fonction à deux arguments là où on attend une fonction unaire, en précisant une valeur (constante) pour le deuxième argument. On pourra par exemple écrire :

```
1 transform( src.begin(),  
2             src.end(),  
3             dst.begin(),  
4             bind2nd( plus<int>(), 42 ) );
```

Une définition possible de ce modèle est donnée dans le listing 5.14.

```

1 template <typename Operation>
2 class binder2nd
3   : public unary_function<typename Operation::first_argument_type,
4                             typename Operation::result_type> {
5 protected:
6   Operation op;
7   typename Operation::second_argument_type value;
8
9 public:
10  binder2nd(const Operation & x,
11             const typename Operation::second_argument_type & y)
12    : op(x), value(y) {
13    }
14
15  typename Operation::result_type
16  operator()
17  (const typename Operation::first_argument_type & x) const {
18    return op(x, value);
19  }
20  };
21
22 template <typename Operation, typename T>
23 inline binder2nd<Operation>
24 bind2nd(const Operation & op, const T & x) {
25   typedef typename Operation::second_argument_type arg2_type;
26   return binder2nd<Operation>(op, arg2_type(x));
27 }
```

Listing 5.14 – Définition du modèle de fonction bind2nd.

```

1 void f( std :: list<Shape *> & figure ) {
2     for_each( figure .begin() ,
3             figure .end() ,
4             std :: mem_fun(&Shape :: draw ) );
5 }
```

Listing 5.16 – Utilisation de l'appelant de méthode.

5.6.6 L'appelant de méthode `mem_fun`

Il est courant de devoir appeler une méthode particulière pour tous les objets d'un conteneur. L'algorithme `for_each` rend la chose aisée mais la syntaxe peut devenir fastidieuse car elle oblige à définir une fonction accessoire (cf. listing 5.15).

```

1 void draw( Shape * f ) {
2     f->draw();
3 }
4
5 void f( list<Shape *> & lf ) {
6     for_each( lf.begin() , lf.end() , draw );
7 }
```

Listing 5.15 – Motivation pour un appelant de méthode.

Une solution élégante consiste à créer un objet temporaire « appelant de méthode » à l'aide des modèles de fonctions `mem_fun` ou `mem_fun_ref` (cf. listing 5.16). L'objet créé par `mem_fun` à partir d'un pointeur de méthode s'applique sur les pointeurs d'objets ; alors que celui qui est produit par le modèle `mem_fun_ref` s'applique à un objet (par référence).

Exercice 5.4 Réécrire le code du listing 5.16 en utilisant une lambda fonction (voir § 7.4).

5.6.7 Le *composeur* d'objets fonctions

On rappelle le type `unary_function` :

```

1 template <typename Arg , typename Result>
2 struct unary_function {
3     typedef Arg argument_type;
4     typedef Result result_type;
5 };
```

Listing 5.17 – Le modèle de structure `unary_function`.

Le modèle de classe `composer1` est le modèle des foncteurs qui sont la composition de deux autres objets fonctions. La définition de ce modèle ainsi que celle du modèle de fonction `compose1` est donnée dans le listing suivant.

```

1 template <typename OperationF , typename OperationG>
2 class composer1
3     : public unary_function<typename OperationG :: argument_type ,
```

```

4           typename OperationF :: result_type> {
5   protected:
6     OperationF f;
7     OperationG g;
8
9   public:
10    composer1(const OperationF & f,
11               const OperationG & g) : f(f), g(g) {
12      // ...
13    }
14
15   typename OperationF :: result_type
16   operator()(const typename OperationG :: argument_type x) const {
17     return f(g(x));
18   }
19 }
20
21 template <typename OperationF, typename OperationG>
22 inline composer1<OperationF, OperationG>
23 compose1(const OperationF & f, const OperationG & g) {
24   return composer1<OperationF, OperationG>(f, g);
25 }
```

Listing 5.18 – Modèles de classe et de fonction pour la fabrication des objets « fonction composée ».

5.6.8 Le fabriquant d'objets fonctions ptr_fun

Ce modèle permet de construire des objets fonctions à partir d'une fonction classique. Il est par exemple utile pour utiliser des fonctions avec les outils fournis par la bibliothèque standard et qui s'appliquent uniquement sur des foncteurs. Le listing 5.19 donne un exemple d'utilisation pour la composition de fonctions.

5.6.9 C++11 : Les pointeurs de fonction généralisés

Le C++11 étend largement la notion de fonction avec les *lambda functions* mais aussi le type `std::function`. Ces notions, décrites au chapitre 7, rendent potentiellement obsolètes celles qui sont décrites dans les sections 5.6.5 à 5.6.8. Pour le moins, elles en offrent une alternative intéressante.

5.6.10 Les itérateurs d'insertion

La bibliothèque standard définit des itérateurs « améliorés » qui permettent d'appliquer des algorithmes portant sur une séquence *source* et une séquence *destination* et agissant terme à terme dans essentiellement deux situations particulières (la première est en fait un cas particulier de la seconde) :

- la séquence destination est vide ;

```
1 template <typename Arg, typename Result>
2 pointer_to_unary_function<Arg, Result>
3 ptr_fun(Result (*x)(Arg));
4
5 template <typename Arg1, typename Arg2, typename Result>
6 pointer_to_binary_function<Arg1, Arg2, Result>
7 ptr_fun(Result (*x)(Arg1, Arg2));
8
9 vector<double> v(20);
10 // ...
11
12 transform(v.begin(),
13             v.end(),
14             v.begin(),
15             compose1(negate<double>, ptr_fun(fabs)));
```

Listing 5.19 – Usage du modèle de fonction `ptr_fun`

— on souhaite *ajouter* (c.-à-d. insérer) la séquence résultat à la fin d'un conteneur. En fait, ces itérateurs ne se contentent pas d'offrir le parcours ou l'accès aux éléments des conteneurs auxquels ils sont associés, il sont capables de modifier ces conteneurs en *y insérant* des éléments.

A titre d'exemple, une utilisation possible d'un objet de type `back_insert_iterator<>` est donnée dans le listing 5.20. Ces itérateurs particuliers peuvent être créés simplement à l'aide du modèle de fonction `back_inserter<>()`.

```
1 #include <algorithm>
2 #include <iostream>
3 #include <iterator>
4 #include <list>
5 #include <vector>
6 using namespace std;
7
8 template <typename T>
9 class AddValue {
10     T value;
11
12 public:
13     AddValue(T value) {
14         AddValue<T>::value = value;
15     }
16     T operator()(T x) {
17         return x + value;
18     }
19 };
20
21 int main() {
22     vector<int> v(10);
23     list<int> res;
24     list<int>::iterator i;
25
26     for (int i = 0; i < 10; i++) {
27         v[i] = i;
28     }
29
30     AddValue<int> add_ten(10);
31
32     transform(v.begin(), v.end(), back_inserter(res), add_ten);
33
34     for (i = res.begin(); i != res.end(); i++) {
35         cout << *i << " ";
36     }
37     cout << endl;
38 }
```

Listing 5.20 – Exemple d'utilisation du modèle de classe des itérateurs d'insertion.



2^e année – Spécialité Informatique

Année 2024 – 2025

Programmation orientée objet en C++

Notes de cours

Sébastien Fourey

Chapitre 6. Les Exceptions

Version du 3 septembre 2024



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

Chapitre 6

Les Exceptions

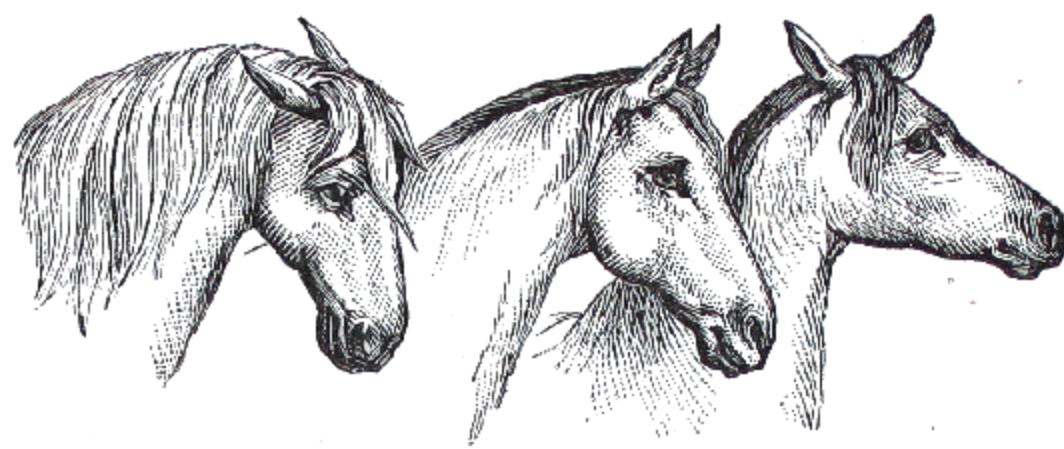


Fig. 51. — Le cheval tourne l'oreille du côté du bruit.

Les exceptions autorisent une forme de gestion des erreurs qui a déjà été présentée dans le cadre du cours de Java. En dehors des quelques particularités propres au C++ ce chapitre est donc un rappel. Il faut noter par exemple qu'il n'y a pas en C++ de notion d'exception *sous contrôle*, c.-à-d. qui provoque une erreur de compilation si elle n'est pas gérée. Les exceptions du C++ sont donc les `RunTimeException` du langage Java. La syntaxe de capture d'une exception quelconque (§ 6.2.2) est aussi différente car il n'y a pas en C++ de classe mère de toutes les exceptions.

6.1 Motivation

Les exceptions sont une caractéristique utile du C++ pour plusieurs raisons :

- Elle permettent une gestion des erreurs plus souple :
 - L'utilisation des valeurs de retour des fonctions est entièrement laissée à d'autres fins. Elles évitent ainsi les conflits de « domaines ».
 - La gestion des erreurs, riche en particularités, peut être faite dans des parties du code bien identifiées et moins éparses.
- Dans le cas des constructeurs, qui par définition n'ont pas de valeur de retour, lever une exception est le meilleur moyen pour gérer les situations où la construction échoue [7]. Notamment, cela permet de mettre fin « proprement » à l'allocation dynamique qui précède la construction d'un objet via l'utilisation du mot-clé `new` (cf. Listing 6.1).
- Enfin, elles facilitent l'utilisation de bibliothèques.

Un principe de l'utilisation des exceptions est le suivant : Une fonction qui détecte un problème qu'elle n'est pas censée gérer se contente d'envoyer (*throw*) le problème à sa fonction appelante. Une exception émise et qui n'est pas capturée provoque l'arrêt de l'exécution du programme.

En C++ une exception est une variable d'un type quelconque, prédéfini ou non, dès lors que celui-ci possède une sémantique de recopie. En particulier, il est aussi possible d'organiser les exceptions en hiérarchies de classes. Un exemple simple de classe d'exceptions est donné dans le listing 6.2.

```
1 #include <iostream>
2 #include <new>
3
4 class MemoryBuffer {
5 public:
6     MemoryBuffer(unsigned long size) {
7         try {
8             _array = new char[size];
9         } catch (std::bad_alloc) {
10             throw "Buffer construction failure.";
11         }
12     }
13
14 private:
15     char * _array;
16 };
17
18 int main() {
19     MemoryBuffer tm(18446744073709551615ul);
20     // On failure, tm has not been constructed
21
22     MemoryBuffer * ptm = nullptr;
23     try {
24         ptm = new MemoryBuffer(18446744073709551615ul);
25         // On failure, no memory allocation has been done
26         // by the 'new' => No memory leak
27     } catch (const char * str) {
28         std::cerr << str;
29     }
30 }
```

Listing 6.1 – Exception levée par un constructeur.

```

1  class Error {
2      int code;
3      const char * message;
4
5  public:
6      Error(int, const char *);
7      int getCode();
8      const char * getMessage();
9  };
10
11 class DiskError : public Error {
12     // ...
13 };
14
15 void formatDisk(const char * path) throw DiskError {
16     // ...
17
18     if ( /* ... */ ) {
19         throw DiskError(10, "No disk in the reader");
20     }
21 }
```

Listing 6.2 – Exemple de définition d'une classe d'exceptions.

6.2 Éléments de syntaxe

6.2.1 Déclencher, ou émettre, une exception (`throw`)

Le déclenchement d'une exception se fait à l'aide du mot-clé `throw`. Ainsi, l'instruction

```
throw uneException;
```

envoie l'exception *uneException* de blocs en blocs, puis de fonction appelante en fonction appelante jusqu'à ce qu'elle soit capturée par un *gestionnaire d'exceptions* (cf. sous-section suivante).

6.2.2 Capturer et gérer une exception : `try` et `catch`

La syntaxe des gestionnaires d'exceptions est illustrée par ce qui suit.

```

try {
    // Code susceptible de déclencher l'exception
} catch ( TypeException1 /idObjet1/ ) {
    // Code de gestion des exceptions de type TypeException1
}
catch ( TypeException2 /idObjet2/ ) {
    // Code de gestion des exceptions de type TypeException2
} // etc.
```

Les points de suspension sont utilisés pour capturer toutes les exceptions selon la syntaxe :

```
catch (...) {
    // Code de gestion
}
```

Les exceptions précisées dans les blocs `catch` peuvent l'être par valeur, par pointeur ou par référence. Dans tous les cas, les règles de conversions implicites s'appliquent.

Enfin, il est important de noter que l'ordre des gestionnaires est significatif et que les conversions automatiques éventuelles s'appliquent en priorité.

En reprenant les types définis dans le listing 6.2, on utilisera typiquement les gestionnaires d'exceptions suivants :

```
1 | try {
2 |     file.readData();
3 | } catch (DiskError & ed) {
4 |     // If a disk error occurred...
5 | } catch (Error & e) {
6 |     // A more general error occurred (not a disk error)
7 | } catch (...) {
8 |     // Any other kind of error
9 | }
```

Listing 6.3 – Capture d'exceptions bien ordonné.

6.2.3 Redéclencher une exception (throw)

L'instruction « `throw;` », quand elle est placée dans un gestionnaire d'exception, redéclenche l'exception en cours de gestion.

6.2.4 Spécifier des exceptions en C++ 98 (throw)

La déclaration de fonction :

```
typeRetour idFonction( arguments... ) throw ( E1, E2, [...] );
```

spécifie que la fonction « `idFonction` » est susceptible d'envoyer *uniquement* les exceptions de type `E1`, `E2`, etc. Il s'agit d'une garantie qui n'est vérifiée que lors de l'exécution du programme. La fonction ne pourra alors émettre aucune autre exception. (Sinon, la fonction `std::unexpected()` sera appelée, provoquant l'interruption du programme.)

Finalement, il est possible de spécifier qu'une fonction ne lève *aucune* exception par la syntaxe `throw()`.

6.2.5 Spécifier des exceptions à partir de C++ 11

La vérification des exceptions spécifiées étant effectuée à l'exécution, elle n'offre aucune garantie lors de la compilation et reste donc d'un intérêt limité pour le programmeur. De plus, cette

vérification a un coût, en code supplémentaire et en temps d'exécution. C'est entre autres pour ces raisons que le C++ 11 rend *deprecated* la spécification des exceptions, telle que présentée en 6.2.4, pour ne laisser que deux possibilités : une fonction peut par défaut lever n'importe quel type d'exception, ou bien le mot-clé `noexcept` peut être utilisé pour garantir qu'aucune exception ne sera levée (syntaxe ci-dessous).

```
typeRetour idFonction( arguments... ) noexcept;
```

6.2.6 Exemple

Le listing 6.4 donne un exemple de définition et d'utilisation (émission et capture) d'une exception.

6.3 Les exceptions standard

La figure 6.1 montre la hiérarchie des classes d'exceptions de la bibliothèque standard dont l'utilisation est résumée par la tableau 6.1.

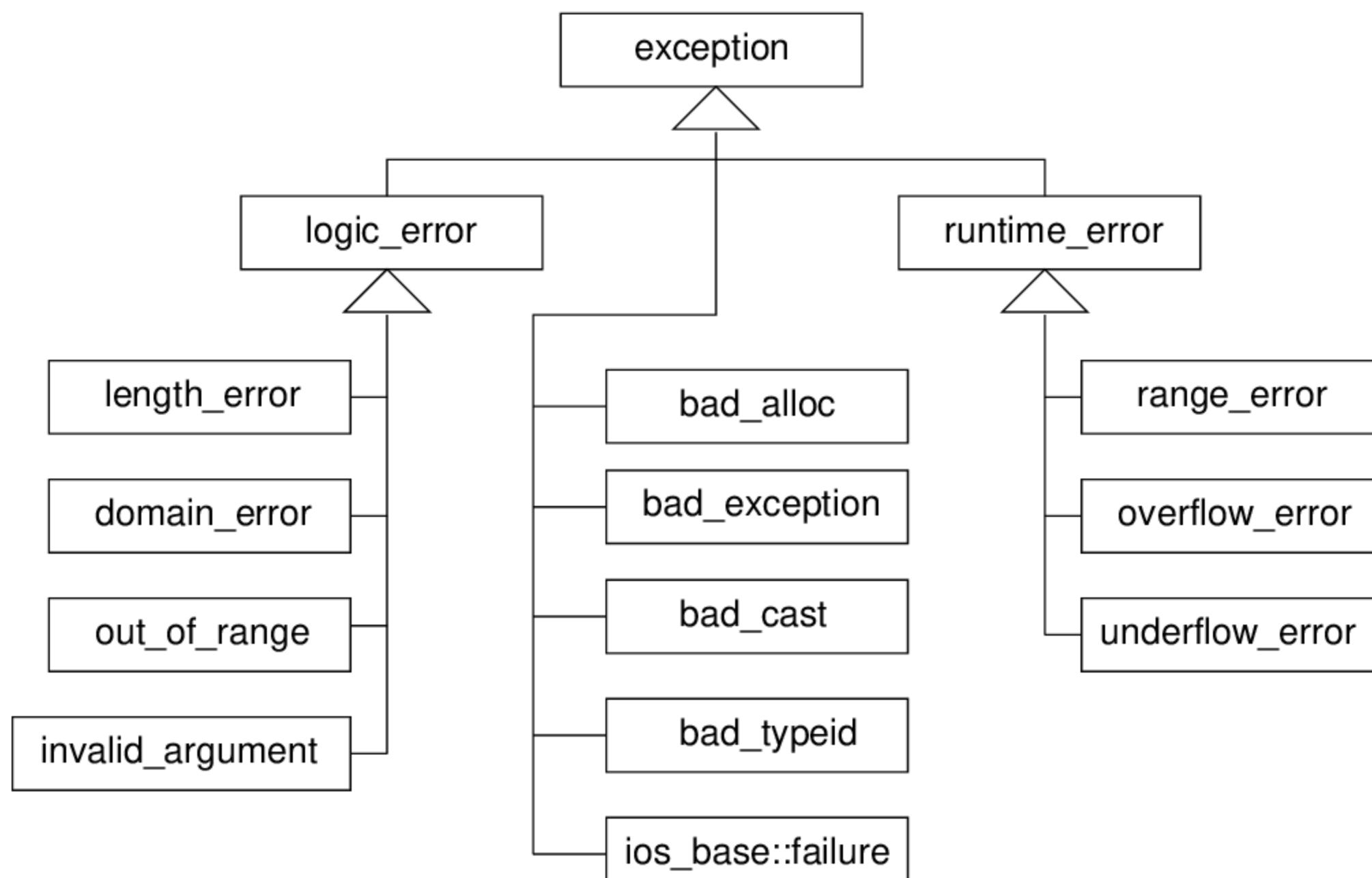


FIGURE 6.1 – Hiérarchie des exceptions de la bibliothèque standard

```
1 #include <iostream>
2 using namespace std;
3
4 struct Error {
5     int code;
6     const char * message;
7 };
8 struct FatalError : public Error {};
9
10 void f() {
11     Error e = {1, "Disk is out of order"};
12     throw e;
13 }
14
15 void g() {
16     // ...
17     FatalError ef;
18     throw ef; // Or: throw FatalError();
19 }
20
21 void h() {
22     g();
23 }
24
25 int main() {
26     try {
27         g();
28         // ...
29     } catch (const FatalError &) {
30         cout << "Error fatal" << endl;
31         return 0;
32     } catch (const Error & e) {
33         cout << "Error non fatal :" ;
34         cout << e.message << endl;
35         return 1;
36     }
37     return 0;
38 }
```

Listing 6.4 – Exemple de définition et gestion d'une exception.

Nom	Déclenchée par	En-tête
bad_alloc	new	<new>
bad_cast	dynamic_cast	<typeinfo>
bad_typeid	typeid	<typeinfo>
bad_exception	spécification	<exception>
out_of_range	at()	<stdexcept>
	bitset<>::operator[]()	<stdexcept>
invalid_argument	constructeur de bitset	<stdexcept>
overflow_error	bitset<>::to_ulong()	<stdexcept>
ios_base::failure	ios_base::clear()	<iostream>

TABLE 6.1 – Quelques unes des exceptions de la bibliothèque standard.



2^e année – Spécialité Informatique

Année 2024 – 2025

Programmation orientée objet en C++

Notes de cours

Sébastien Fourey

Chapitre 7. C++11

Version du 3 septembre 2024



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

Chapitre 7

C++11

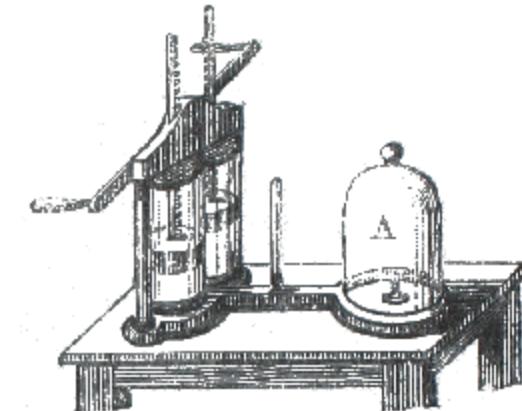


Fig. 122. — Les machines pneumatiques servent à retirer d'un vase A, non pas de l'eau, mais de l'air.

Cette section est consacrée à plusieurs nouveautés apparues avec la norme de 2011 du langage, et qui n'ont pas trouvé leur place dans d'autres sections du document.

7.1 Listes d'initialisation et initialisation uniforme

En C comme en C++, il est possible d'initialiser un tableau lors de sa définition à la manière du code ci-dessous :

```
1 int main() {  
2     int t[] = {1, 2, 1, 5, 3, 8, 21, 13, 55, 34};  
3 }
```

Listing 7.1 – Initialisation d'un tableau C lors de sa définition.

En comparaison, avant la norme de 2011, l'initialisation des éléments d'un conteneur de la STL pouvait être laborieuse (listing 7.2).

```
1 #include <vector>  
2 using std::vector;  
3 int main() {  
4     vector<int> v;  
5     v.push_back(1);  
6     v.push_back(2);  
7     v.push_back(1);  
8     v.push_back(5);  
9     v.push_back(3);  
10    v.push_back(8);  
11    v.push_back(21);  
12    v.push_back(13);  
13    v.push_back(55);  
14    v.push_back(34);  
15 }
```

Listing 7.2 – Initialisation d'un vecteur en C++03.

Exercice 7.1 Écrivez une variante du code du listing 7.2 pour initialiser le vecteur *v* en utilisant un tableau (comme dans le listing 7.1) ainsi que l'algorithme `std::copy`.

Le C++11 apporte une nouvelle forme d'initialisation applicable aux conteneurs de la STL mais aussi à tous les types définis par l'utilisateur (classes et structures) : les listes d'initialiseurs. Pour cela, un type dédié et itérable comme un conteneur classique est défini dans la bibliothèque standard : `std::initializer_list<T>`.

Afin d'offrir cette nouvelle syntaxe d'initialisation à une classe, il suffit de l'équiper d'un constructeur ayant pour argument ce nouveau type, comme illustré dans le listing 7.3.

```

1 #include <algorithm>
2 #include <initializer_list>
3 template <typename T>
4 class Array {
5 public:
6     Array() : _data(nullptr), _count(0) {
7     }
8     Array(std::initializer_list<T> l) {
9         if (l.size()) {
10             _count = l.size();
11             _data = new T[_count];
12             std::copy(l.begin(), l.end(), _data);
13         } else {
14             _data = nullptr;
15             _count = 0;
16         }
17     }
18
19 private:
20     T * _data;
21     unsigned long _count;
22 };
23
24 int main() {
25     Array<int> t = {1, 2, 1, 5, 3, 8, 21, 13, 55, 34};
26 }
```

Listing 7.3 – Utilisation d'une liste d'initialiseurs

Remarque 7.1 Le paramètre du constructeur est bien passé par valeur et pas par référence constante, comme on pourrait juger plus efficace de le faire. Mais ceci s'explique par le fait que le type `initializer_list` est « intrinsèquement » constant au sens où ses itérateurs sont assimilables à des pointeurs vers des valeurs constantes ; mais aussi parce que ce type possède une sémantique de copie par référence : il n'y a pas duplication de données quand on copie une liste d'initialiseurs.

En fait, plusieurs syntaxes sont possibles pour appeler ce même constructeur (listing 7.4). Mais l'usage des accolades en C++11 n'est pas limité aux listes d'initialiseurs. En effet, elle permettent

```
1 #include <algorithm>
2 #include <initializer_list>
3 template <typename T>
4 class Array {
5     // ...
6 };
7
8 int main() {
9     Array<int> t1 = {1, 2, 1, 5, 3, 8, 21, 13, 55, 34};
10    Array<int> t2({1, 2, 1, 5, 3, 8, 21, 13, 55, 34});
11    Array<int> t3{1, 2, 1, 5, 3, 8, 21, 13, 55, 34};
12 }
```

Listing 7.4 – Trois syntaxes équivalentes pour utiliser une liste d'initialiseurs.

```
1 struct Item {
2     std::string description;
3     float price;
4 };
5
6 int main() {
7     Item a{"cauliflower", 1.5};
8 }
```

Listing 7.5 – Initialisation des données membres d'une structure.

```

1 #include <string>
2 class Item {
3 public:
4     Item( std :: string description )
5         : _description( description ), _price( 0.0 ) {
6     }
7
8 private:
9     std :: string _description;
10    float _price;
11 };
12
13 int main() {
14     Item pdt{ "Potatoes" };
15 }
```

Listing 7.6 – Syntaxe d'appel du constructeur avec des accolades.

```

1 #include <initializer_list>
2 void foo( char c , std :: initializer_list<int> l ) {
3 }
4
5 int main() {
6     int k = 100;
7     foo( 'a' , { 1 , 1 , k , 3 , 5 , 8 , k } );
8 }
```

Listing 7.7 – Liste d'initialiseur comme paramètre de fonction.

d'initialiser les champs d'une structure ou d'une classe dont toutes les données membres sont publiques, et ce en l'absence de constructeur (cf. listing 7.5).

Enfin, si un constructeur « classique » (c.-à-d. dont les paramètres ne se réduisent pas à une liste d'initialiseurs) est défini et comporte le bon nombre de paramètres, il peut aussi être appelé avec cette même syntaxe (listing 7.6). L'utilisation des accolades provoque simplement en priorité l'appel du constructeur à partir d'une `initializer_list<>`, et, s'il n'y en a pas, l'appel d'un constructeur classique. Enfin, en l'absence de constructeur, c'est une initialisation membre à membre qui est réalisée si tous les attributs sont publics. Au vu de ces nombreux usages, l'utilisation des accolades constitue en C++11 une syntaxe dite d'initialisation uniforme.

Paramètre de fonction

Une liste d'initialisation peut être un paramètre de fonction, comme dans l'exemple donné dans le listing 7.7.

```

1 struct Item {
2     std::string description;
3     float price;
4 };
5
6 Item bestSell(int year) {
7     if (year < 2010) {
8         return {"Computer", 2000.0f};
9     } else {
10        return {"Smartphone", 300.0f};
11    }
12 }
```

Listing 7.8 – Initialisation de la valeur de retour d'une fonction.

Retour de fonction

Il est possible d'utiliser les accolades pour spécifier la valeur de retour d'une fonction lorsqu'il s'agit d'une structure ou d'une classe (équipée ou non d'un constructeur).

Autres usages

Finalement, les listes d'initialisation peuvent être utilisées dans les contextes énumérés ci-dessous. Les lignes indiquées font référence aux exemples donnés dans le listing 7.9.

- initialisation d'une variable lors de sa définition (lignes 29 et 30) ;
- initialisation lors d'une allocation par `new` (ligne 31) ;
- dans une instruction `return` (ligne 21) ;
- comme « indice » d'un opérateur `[]` (ligne 34) ;
- comme argument d'une fonction (ligne 32) ;
- comme syntaxe d'appel à un constructeur (ligne 35) ;
- comme initialisation d'une donnée membre non-statique (ligne 12) ;
- dans une liste d'initialiseurs de données membres (ligne 6) ;
- comme opérande droite d'un opérateur d'affectation (ligne 37).

Perte de précision (*narrowing*)

Un avantage des listes d'initialisation est qu'elles n'autorisent pas, en théorie, les pertes de précision, comme le montre le listing 7.10.



Contrairement à ce que dit la norme, à savoir qu'une perte de précision provoque une erreur, `g++` dans sa version 4.8.3 se contente d'émettre des *warnings*.

```
1 #include <initializer_list>
2 #include <list>
3 #include <string>
4
5 struct Item {
6     Item() : description{"N/A"}, price{0.0f} {
7     }
8     void operator[](std::initializer_list<int>) {
9     }
10    std::string description;
11    float price;
12    float VATRate{18.6};
13 };
14
15 struct Size {
16     int age;
17     float height;
18 };
19
20 Size foo() {
21     return {20, 1.74f};
22 }
23 void bar(Size) {
24 }
25 void team(Item) {
26 }
27
28 int main() {
29     float x{2.5f}; // (a)
30     float y = {2.5f}; // (b)
31     std::list<int> * pl = new std::list<int>{1, 2, 3, 4, 5};
32     bar({18, 1.70f});
33     Item a;
34     a[{1, 2, 3}];
35     team(Item{});
36     Size t;
37     t = {20, 1.60f};
38 }
```

Listing 7.9 – Diverses utilisations des listes d’initialisation et de l’initialisation uniforme.

```

1 #include <initializer_list>
2
3 void foo(std::initializer_list<int>) {
4 }
5
6 int main() {
7     int i{1.5}; // Error (double -> int)
8     int j{10.0}; // Error (double -> int)
9     double x = 9.81;
10    int a{x}; // Error (double -> int)
11    char k{300}; // Error
12    char l{10}; // OK!
13    foo({10, 10.5}); // Error (10.5 -> int)
14 }
```

Listing 7.10 – Liste d'initialiseur et perte de précision

```

1 void foo(const std::vector<int> & v) {
2     std::vector<int>::const_iterator it = v.begin();
3     while (it != v.end()) {
4         std::cout << (*it++) != 0;
5     }
6 }
```

Listing 7.11 – Exemple de boucle utilisant un itérateur de la STL.

7.2 Inférence de type

La STL est parfois critiquée pour son caractère « verbeux » résultant de l'imbrication de (modèles de) types, parfois nombreux. C'est le cas par exemple avec la boucle, pourtant simple, du listing 7.11.

La norme 2011 du C++ offre une solution appelée « inférence de type » basée sur l'utilisation de deux nouveaux mots-clés : `auto` et `decltype`. Le mot-clé `auto` peut ainsi être utilisé dans une définition de variable (initialisée) à la place du type. Le type sera alors déduit par le compilateur comme étant celui du résultat de l'expression d'initialisation. Le listing 7.12 donne un cas d'utilisation possible.

Le mot-clé `decltype`, quant à lui, permet d'utiliser le type d'une variable déjà définie ou bien

```

1 void foo(const std::vector<int> & v) {
2     auto it = v.begin();
3     while (it != v.end()) {
4         std::cout << (*it++) != 0;
5     }
6 }
```

Listing 7.12 – Utilisation du mot-clé `auto`.

```

1 #include <vector>
2
3 void foo(const std::vector<int> & v) {
4     auto it = v.begin();
5     decltype(it) itShift = it + 1;
6     // ...
7 }
8
9 int main() {
10    std::vector<int> v(10);
11    int i = 0, j = 1;
12    decltype(i) k = 2;
13    decltype(i + j) l;      // int l;
14    decltype(v[2]) cell2; // int & cell2;
15 }
```

Listing 7.13 – Utilisation du mot-clé `decltype`.

d'une expression. Cela peut être utile par exemple si le type d'une variable a été lui-même déduit, ou simplement pour éviter de répéter un type un peu long (listing 7.13).

Remarque 7.2 *L'utilisation du mot-clé `auto`, si elle permet de rendre une code plus concis, peut aussi avoir comme inconvénient de le rendre moins explicite, voire moins lisible. En effet, le type d'une variable doit être déduit non seulement par le compilateur, mais aussi par toute personne qui lit le code ! Par exemple, le programmeur qui a écrit la ligne 2 du listing 7.11 a clairement tenu compte du fait que la version `const` de la méthode `begin` est appelée depuis la référence constante `v`. C'est bien pour cela que l'itérateur `it` est un itérateur en lecture seule (`const_iterator`). Avec le mot-clé `auto` (listing 7.12), tout ceci est un peu moins explicite. Notez que la STL, en C++11, équipe les conteneurs itérables de méthodes `cbegin()` et `cend()` pour rendre plus explicite le caractère `const` de ces méthodes.*

7.3 Boucle `for` basée sur les itérateurs

Le parcours séquentiel de tous les éléments d'un conteneur ou d'un tableau (de taille connue à la compilation pour les tableaux) peut se faire à l'aide d'une forme dédiée de la boucle `for`. Des exemples sont donnés dans le listing 7.14.

7.4 Les fonctions anonymes ou fonctions *lambda*

Les fonctions anonymes permettent de définir et d'utiliser une fonction à la volée, sans lui donner nécessairement un nom. Les fonctions ainsi créées sont compatibles avec le type *pointeur de fonction*. Un premier exemple en est donné dans le listing 7.15.

Bien entendu, une fonction anonyme peut avoir des arguments et un type de retour. Ce dernier peut être déduit à partir du type des expressions fournies aux instructions `return` (si elles sont

```
1 #include <iostream>
2 #include <vector>
3 using namespace std;
4
5 void foo( int array [] ) {
6     for ( int & i : array ) { // ERROR! The size of the array
7         std :: cout << i // is unknown
8             << std :: endl;
9     }
10 }
11
12 int main() {
13     char arrayOfChar [] = { 'a', 'b', 'c' };
14     for ( char c : arrayOfChar ) { // OK
15         cout << c;
16     }
17     for ( char & r : arrayOfChar ) { // OK
18         r = '-';
19     }
20     for ( auto a : arrayOfChar ) { /* OK but be careful */ /*
21         a = '/';
22     }
23     for ( auto & a : arrayOfChar ) { a = '/'; } // OK
24
25     vector<int> v = { 1, 2, 3, 4 };
26     const vector<int> w = { 1, 2, 3, 4 };
27     for ( int i : v ) { // OK
28         cout << i;
29     }
30     for ( int & r : v ) { // OK
31         r = 0;
32     }
33     for ( int & r : w ) { // ERROR: w is const
34         r = 0;
35     }
36 }
```

Listing 7.14 – Exemple de boucles **for** basées sur les intervalles.

```

1 #include <iostream>
2 using namespace std;
3
4 typedef void (*func)();
5
6 void callTwice(func f) {
7     f();
8     f();
9 }
10
11 int main() {
12     callTwice([] { cout << "Hello\n"; });
13 }
```

Listing 7.15 – Un usage possible de fonctions anonymes.

```

1 #include <algorithm>
2 #include <vector>
3 using namespace std;
4
5 int main() {
6     // sum returns an int
7     auto sum = [](int a, int b) { return a + b; };
8
9     std::vector<int> v = {1, 2, 3, 4, 5};
10    replace_if(v.begin(),
11                 v.end(),
12                 [](int i) -> bool { return i % 2; },
13                 0);
14 }
```

Listing 7.16 – Fonctions anonymes.

toutes du même type), auquel cas il n'a pas à être précisé. Sinon, la syntaxe utilisée dans le listing 7.16 s'impose.

Comme une fonction classique, une fonction anonyme n'a accès qu'aux variables globales et à ses paramètres. Mais il est possible de *capturer*, par valeur ou par référence, des variables présentes dans le contexte où la fonction anonyme est créée. Ceci est à rapprocher de la notion de clôture (*closure*) en programmation fonctionnelle. Un exemple de capture par valeur est donné dans le listing 7.17, la syntaxe générale étant donnée ci-après :

- [] Seules les variables globales sont capturées (par référence).
- [x,&y] La variable x est capturée par valeur, y est capturée par référence.
- [&] Toutes les variables disponibles, si elle sont utilisées, sont capturées par référence.
- [=] Toutes les variables disponibles, si elle sont utilisées, sont capturées par valeur.
- [&,x] La variable x est capturée par valeur, les autres variables le sont par référence.
- [=,&y] La variable y est capturée par référence, les autres variables le sont par valeur.
- [this] La variable this est toujours capturée par valeur.



Une variable capturée par valeur est constante dans la fonction anonyme (listing 7.18).

```

1 #include <algorithm>
2 #include <vector>
3 using namespace std;
4
5 void initVector(vector<int> & v, int value) {
6     for_each(v.begin(),
7               v.end(),
8               [value](int & x) { x = value; });
9 }
10
11 int main() {
12     std::vector<int> v{1, 2, 3, 4, 5};
13     initVector(v, 10);
14 }
```

Listing 7.17 – Exemple de fonction anonyme avec clôture (ligne 8).

```

1 int main() {
2     int y;
3     auto foo = [y](int x) { y = x; }; // ERROR: y is const here
4     foo(30);
5 }
```

Listing 7.18 – Capture par valeur.

7.4.1 Pointeur de fonction généralisé

En C++11, on peut répertorier différents « objets » qui se comportent comme des fonctions :

- les fonctions classiques ;
- les pointeurs de fonction ;
- les fonctions anonymes ;
- les méthodes ;
- les instances de classes munies d'un opérateur () .

La bibliothèque standard fournit un modèle de classe qui *généralise* la notion de pointeur de fonction à tous les cas cités ci-dessus : `std::function` (défini dans l'entête `<functional>`). Il est ainsi possible d'instancier ce modèle, puis à son tour la classe obtenue et affecter à l'objet (instance de classe) qui en résulte l'un des « objets » précédents. La syntaxe d'instanciation de ce modèle, un peu particulière, est toutefois très naturelle. Ainsi, la notion de « pointeur de fonction » prenant comme arguments deux entiers et retournant un `float` s'écrit simplement :

```
std::function<float(int,int)>
```

Ce type peut alors être utilisé pour créer des variables, des paramètres de fonction ou encore comme type de retour de fonction.

```
1 #include <cstdlib>
2 #include <functional>
3
4 class Sum {
5 public:
6     int operator()(int a, int b) const {
7         return a + b;
8     }
9 };
10
11 int zero(int, int) {
12     return 0;
13 }
14
15 /*
16 * randomFunc() returns a function
17 */
18 std::function<int(int, int)> randomFunc() {
19     switch (rand() % 3) {
20     case 0:
21         return Sum(); // Function object (functor)
22         break;
23     case 1:
24         return zero; // C-like function pointer
25         break;
26     default:
27         return [] (int a, int b) { // lambda function
28             return a * b;
29         };
30         break;
31     }
32 }
33
34 int main() {
35     Sum sum;
36     std::function<int(int, int)> sumBis = Sum();
37     std::function<int(int, int)> z = zero;
38     std::function<int(int, int)> operation = randomFunc();
39     operation(1, 2);
40 }
```

Listing 7.19 – Pointeur de fonction généralisé.

Un avantage non négligeable est l'uniformisation et la relative simplicité de la syntaxe utilisée. Le listing 7.19 donne un exemple de fonction retournant un pointeur de fonction généralisé.

La notion de pointeur de méthode est elle aussi quelque peu simplifiée : un tel pointeur est vu comme une fonction avec comme (premier) argument supplémentaire un pointeur ou une référence sur une instance de la classe. Ainsi,

```
std::function< float( Matrix *, int ) >
ou
std::function< float( Matrix &, int ) >
```

peut désigner deux types de fonctions, finalement similaires :

- Une fonction prenant comme arguments un pointeur (ou une référence) vers une matrice et un entier, retournant un **float**.
- Une méthode de la classe **Matrix** ayant un argument de type entier et qui retourne un **float**.

Le listing 7.20 fournit un exemple d'utilisation de pointeur de méthode à l'aide du modèle **std::function**.

7.4.2 Le générateur d'adaptateurs à la volée **std::bind**

Parfois, par exemple lors de l'utilisation d'un algorithme de la STL, on a besoin d'utiliser un objet fonction (au sens général) mais il ne convient pas exactement à l'usage prévu par l'algorithme. Un exemple classique est celui de l'algorithme **std::transform** (listing 7.21) dont le dernier argument est un foncteur unaire. Si on souhaite, à l'aide de cet algorithme, ajouter une valeur donnée à tous les éléments d'une séquence, il sera nécessaire de définir une fonction unaire qui ajoute cette valeur à son argument et retourne le résultat (listing 7.22).

Finalement, c'est une solution un peu longue si on considère que la STL fournit le foncteur **plus** (listing 7.23) et qu'il suffirait de spécifier une valeur pour l'un des deux arguments en créant un autre foncteur, adaptateur de **plus**, n'ayant qu'un argument. En fait, tout ceci est possible, à la volée et éventuellement sans même donner de nom au foncteur ainsi créé ; à l'aide du modèle de fonction **std::bind** (entête `<functional>`). Ce « générateur » permet en effet de créer un foncteur à partir d'un autre en spécifiant des valeurs pour une partie des arguments, voire tous. Il est aussi possible de réordonner les arguments du foncteur qui est adapté. La syntaxe générale est illustrée dans l'exemple ci-dessous :

```
auto f2 = std::bind( fonc, 'a', _2, 10, _1 );
```

Ici, on a créé un adaptateur basé sur le foncteur **fonc**. Le foncteur créé possède deux paramètres (**_1** et **_2**) et a le même type de retour que **fonc**. L'appel

```
f2("Hello", 200);
```

revient, par l'intermédiaire de l'objet **f2**, à l'appel

```
fonc('a', 200, 10, "Hello");
```

```
1 #include <functional>
2
3 class Counter {
4     int val = 0;
5
6 public:
7     void add(int n) {
8         val += n;
9     }
10    void subtract(int n) {
11        val -= n;
12    }
13};
14
15 void callAdd(Counter * c, int n) {
16     c->add(n);
17 }
18
19 int main() {
20     std::function<void(Counter *, int)> op;
21     Counter c;
22     op = &Counter::add; // op = method pointer
23     op(&c, 10);
24     op = &Counter::subtract;
25     op(&c, 5);
26     op = callAdd; // op = function
27     op(&c, 10);
28
29     std::function<void(Counter &, int)> opRef;
30     opRef = &Counter::add;
31     opRef(c, 10); // Note the absence of &
32 }
```

Listing 7.20 – Pointeur de méthode avec `std::function`.

```

1  namespace std
2  {
3      template <typename IN, typename OUT, typename FUNC>
4      void transform(IN begin, IN end, OUT out, FUNC func) {
5          while (begin != end) {
6              *out = func(*begin);
7              ++begin;
8              ++out;
9          }
10     }
11 } // namespace std

```

Listing 7.21 – Algorithme `transform` de la STL.

Finalement, un objet fonction équivalent à une instance de la classe `AddVal<int>` (listing 7.22) peut s'écrire :

```
std::bind( plus<int>(), _1, 10 )
```

pour aboutir au code du listing 7.24, plus concis !



Les *placeholders* (~ substituts, remplaçants) `_1`, `_2`, `_3`, etc. sont définis dans l'espace de noms `std::placeholders`. L'utilisation d'une directive `using` a par exemple été choisie dans le listing 7.24.

Notez que l'inférence de type (section 7.2) est souvent utile pour gérer simplement le type d'objet retourné par `std::bind`. Enfin, il est important de remarquer que la génération d'adaptateur est basée sur la notion de pointeur de fonction généralisé (section 7.4.1), et offre donc le degré de générativité attendu (c.-à-d. utilisable avec la grande famille des objets de type « fonction »).

Remarque 7.3 *Les fonctions surchargées peuvent être utilisées avec `std::bind` mais l'ambiguïté doit être levée, comme illustré dans le listing 7.25.*

7.5 Type de retour suffixé

Il est des situations où la spécification du type de retour d'une fonction est difficile à faire simplement, notamment dans le cas d'un modèle de fonction. Par exemple, dans le cas du listing 7.26, le type de retour dépendra des types effectifs `U` et `V` utilisés pour instancier le modèle : si `U` et `V` sont `int`, ce sera `int` alors que si `U==int` et `V==float` ce sera `float`.

```

1  template <typename U, typename V>
2  /* ? */ product(U u, V v) {
3      return u * v;
4  }

```

Listing 7.26 – Type de retour de fonction inconnu.

```

1 #include <algorithm>
2 #include <vector>
3
4 template <typename T>
5 struct AddValue {
6     T value;
7     AddValue( int value ) : value( value ) {
8     }
9     T operator()( const T & x ) const {
10         return x + value;
11     }
12 };
13
14 int main() {
15     std :: vector<int> v{1, 2, 3, 4, 5, 6};
16     std :: transform( v.begin(),
17                     v.end(),
18                     v.begin(),
19                     AddValue<int>(10));
20     // v == {11,12,13,14,15,15}
21 }
```

Listing 7.22 – Utilisation de l’algorithme `transform`.

```

1 template <typename Arg, typename Result>
2 struct unary_function {
3     typedef Arg argument_type;
4     typedef Result result_type;
5 };
6
7 template <typename T>
8 struct plus : unary_function<T, T> {
9     T operator()( const T & a, const T & b) const {
10         return a + b;
11     }
12 };
```

Listing 7.23 – Foncteur `plus` de la STL.

```

1 #include <algorithm> // For std::transform
2 #include <functional> // For std::bind and std::plus
3 #include <vector>
4
5 using std::placeholders::_1;
6
7 int main() {
8     std::vector<int> v{1, 2, 3, 4, 5, 6};
9     std::transform(v.begin(),
10                 v.end(),
11                 v.begin(),
12                 std::bind(std::plus<int>(), _1, 10));
13     // v == {11,12,13,14,15,15}
14 }
```

Listing 7.24 – Adaptation de `std::plus` à l'aide de `std::bind`.

```

1 #include <functional>
2
3 int sum(int p, int q) {
4     return p + q;
5 }
6
7 float sum(float x, float y) {
8     return x + y;
9 }
10
11 using std::placeholders::_1;
12
13 int main() {
14     auto opA = std::bind(sum, 8, _1);           // ERROR: Ambiguous
15     auto opB = std::bind((int (*)(int, int))sum, 8, _1); // OK
16     int i = opB(100);
17 }
```

Listing 7.25 – `std::bind` et les fonctions surchargées.

En fait, il existe une solution un peu complexe à base de *traits* pour résoudre ce problème. En C++11, il est une solution intermédiaire basée sur le mot-clé `decltype`, qui reste insatisfaisante et inélégante. Mais finalement, le C++11 apporte une nouvelle syntaxe qui, combinée à `decltype`, permet de spécifier le type de retour sous forme de suffixe comme le montre le listing 7.27.

```

1 template <typename U, typename V>
2 auto product(U u, V v) -> decltype(u * v) {
3     return u * v;
4 }
```

Listing 7.27 – Type de retour suffixe.

Notez que la précision du type par `decltype` arrive en fin de signature justement parce que la portée des variables `u` et `v` commence à la déclaration de la liste des paramètres. C'est ce qui rendrait l'utilisation de `decltype`, à la place de `auto` dans l'exemple précédent, difficile.

7.6 Les pointeurs intelligents

Il est indéniable que l'absence de ramasse-miettes (*garbage collector*) en C++ ne facilite pas la vie du programmeur. Mais il ne faut pas oublier que le C++ se veut être un langage efficace et que la gestion fine de la mémoire, comme par exemple le déterminisme de la libération de cette ressource, est une caractéristique souhaitée. Cette section présente néanmoins trois modèles de classes offerts par la bibliothèque standard qui peuvent être d'une aide précieuse. Deux d'entre eux (`unique_ptr` et `shared_ptr`) permettent de gérer automatiquement la désallocation, le dernier (`weak_ptr`) permet quant à lui la détection des pointeurs pendouillants, ou *dangling pointers*. Avant de présenter chacun de ces modèles, il faut revenir sur la notion de *propriété* d'un pointeur. Cette notion est simple à énoncer, mais d'un point de vue pratique elle est on ne peut plus critique.

7.6.1 Notion de propriété d'un pointeur

Lorsqu'une allocation dynamique sur le tas est réalisée à l'aide de l'opérateur `new`, une question se pose très souvent : qui « possède » ou « est propriétaire de » (*owns, has ownership of*) la donnée allouée ? Autrement dit, *qui est responsable de la libération de cette donnée* ? Attention, dans la suite de cette section, c'est bien cette signification qui est donnée au verbe posséder et à la notion de propriété.

Ainsi, dans le cas du listing 7.28 la méthode `ShapeFactory::randomShape` effectue une allocation dynamique sur le tas, mais elle ne conserve pas la propriété du pointeur obtenu. Au contraire, elle le retourne à la fonction appelante (ici `main()`) qui en devient propriétaire et se charge donc de sa libération (ligne 36). Ceci est illustré par la figure 7.1. À l'instar d'une fonction, une structure ou une classe peuvent bien entendu aussi avoir la *propriété* d'un pointeur, puisque leur destructeur (ou une des méthodes) peut se charger de sa libération. C'est le cas de l'instance `owner` créée à la ligne 35. C'est en effet lorsque cette instance est elle-même détruite à la fin de la fonction `main()` (ligne 37) que l'objet pointé par `shape` est détruit (ligne 28). Autrement dit, à la ligne 35, la propriété du pointeur retourné par `randomShape()` a été transférée à l'objet `owner`.

On peut faire plusieurs remarques sur cette notion de propriété.

```
1 #include <cstdlib>
2
3 class Shape {
4 public:
5     virtual void draw() = 0;
6     virtual ~Shape();
7 }
8
9 class Circle : public Shape;
10
11 class Rectangle : public Shape;
12
13 class ShapeFactory {
14 public:
15     static Shape * randomShape() {
16         Shape * result;
17         if (rand()%2) {
18             result = new Circle;
19         } else {
20             result = new Rectangle;
21         }
22         return result;
23     }
24 };
25
26 struct ShapeOwner {
27     Shape * shape;
28     ~ShapeOwner() { delete shape; }
29 };
30
31 int main()
32 {
33     Shape * shape = ShapeFactory::randomShape();
34     shape->draw();
35     ShapeOwner owner{ShapeFactory::randomShape()};
36     delete shape;
37 }
```

Listing 7.28 – Illustration de la notion de propriété.

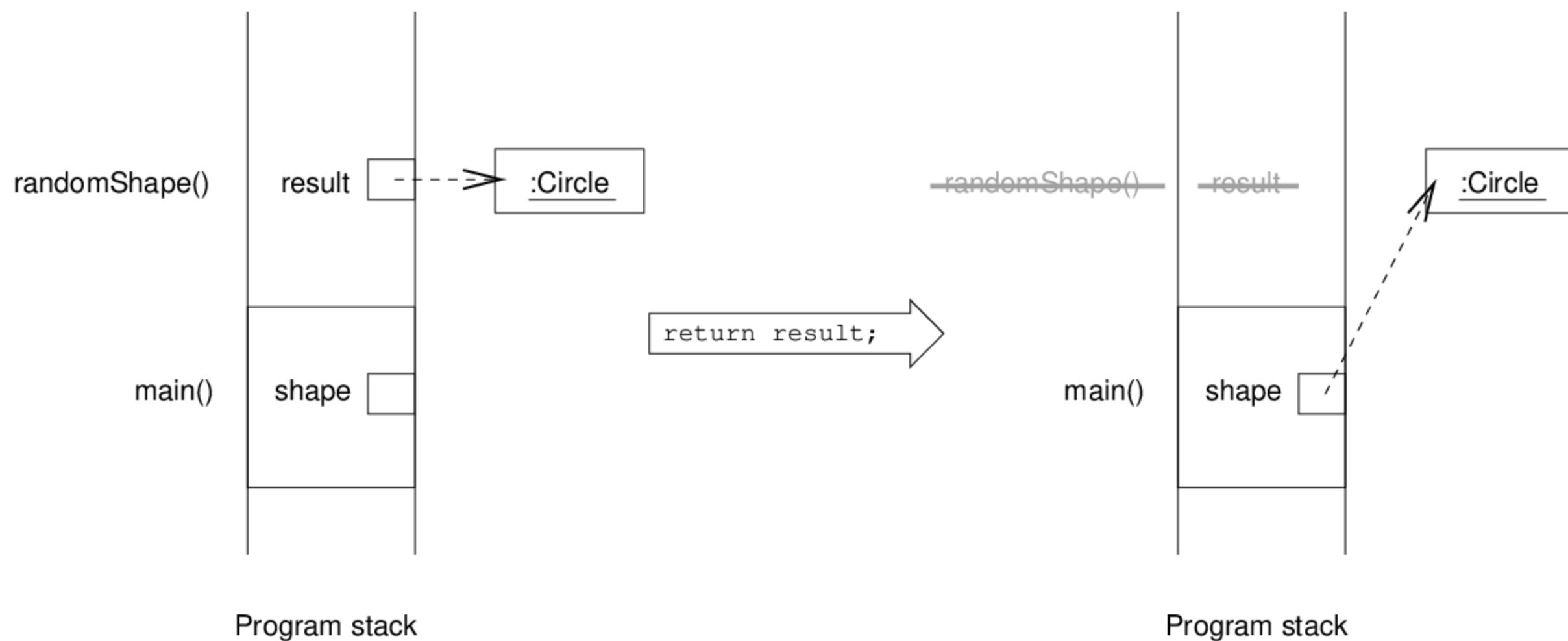


FIGURE 7.1 – Transfert de la propriété d'un objet entre fonctions (lignes 22 et 33 du listing 7.28). La fonction `main` est en charge de la désallocation de cet objet.

Remarque 7.4 (La propriété est exclusive) *Une donnée allouée dynamiquement sur le tas ne devrait être possédée à un instant donné que par une unique fonction ou instance (de structure ou classe). Dans le cas contraire, plusieurs destructions d'un même pointeur à l'aide du mot clé `delete` seront réalisées, ce qui amène à un comportement indéterminé (undefined behavior).*

Remarque 7.5 (Propriété transférable) *Celui qui est propriétaire d'une donnée allouée a deux possibilités : transférer la propriété à une autre entité ou bien, le moment venu, désallouer cette donnée. C'est ce comportement qui garantit que la donnée finira bien par être désallouée, au moment opportun.*

Remarque 7.6 (Pointeur pendouillant) *Bien entendu, plusieurs pointeurs peuvent faire référence à une donnée que seul l'un d'entre eux possède. C'est par exemple le cas dans la situation décrite par la figure 7.2. Parfois, cela pose cependant un gros problème : lorsque le propriétaire décide de désallouer la donnée, les autres pointeurs, s'ils existent toujours, se retrouvent dans un état dit « pendouillant » (dangling pointers). Ils font référence à une donnée qui n'existe plus ! C'est la situation décrite par la figure 7.3. Inutile de dire que l'utilisation de ces pointeurs serait alors une grave erreur.*

7.6.2 Un pointeur intelligent est un *Proxy*

Les trois modèles qui seront présentés par la suite sont des implémentations du patron de conception « Procuration » (*Proxy*). En effet, leur but est de se substituer à un pointeur classique afin de gérer sa recopie et la désallocation de la donnée pointée. Cette substitution, plus ou moins complète, s'entend au sens syntaxique grâce à la surcharge d'opérateurs (affectation, déréférencement, opérateur `->`, etc.). Finalement, c'est cette procuration qui permettra d'étendre et de gérer finement la notion de propriété.

Ainsi, le diagramme de classes de la figure 7.4 illustre le fait que le type `Ptr<T>` se comporte comme un pointeur vers le type `T` auquel il est associé mais il contrôle totalement l'utilisation de ce pointeur et sa recopie. Dans la suite, on désigne par pointeur *stocké* (*stored pointer*) le

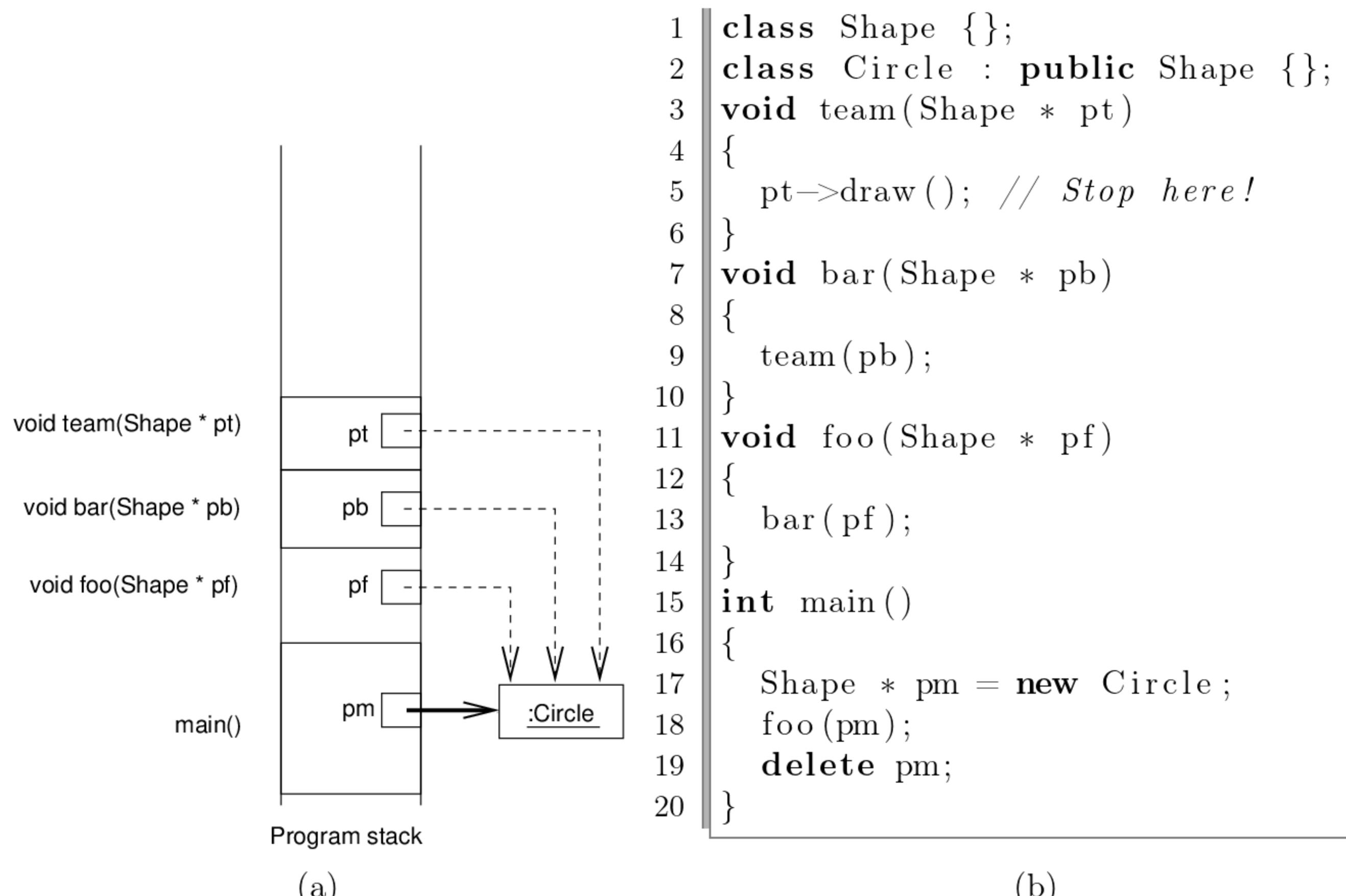


FIGURE 7.2 – Situation dans laquelle plusieurs pointeurs, dans des portées différentes, font référence à une même donnée. Seul un pointeur est propriétaire de cette donnée (flèche en trait plein). La pile d'appels représentée en (a) représente l'état du programme (b) juste avant l'exécution de l'instruction de la ligne 5.

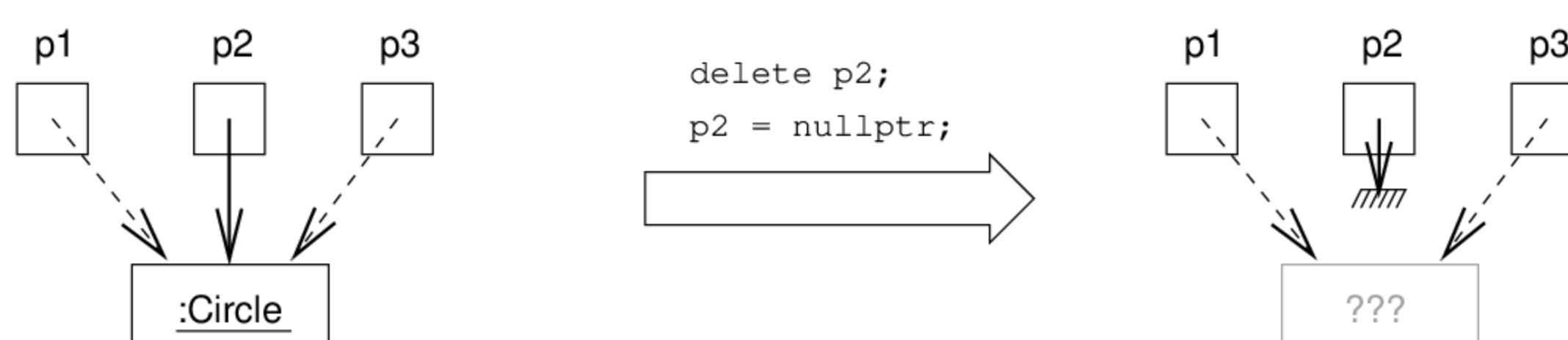


FIGURE 7.3 – Pointeurs pendouillants (*dangling pointers*). Le pointeur `p2` est propriétaire de la donnée de type `Circle` (flèche en trait plein), c'est donc via ce pointeur que cette donnée est libérée. Les pointeurs `p1` et `p3` pointent toujours vers une donnée qui a été libérée.

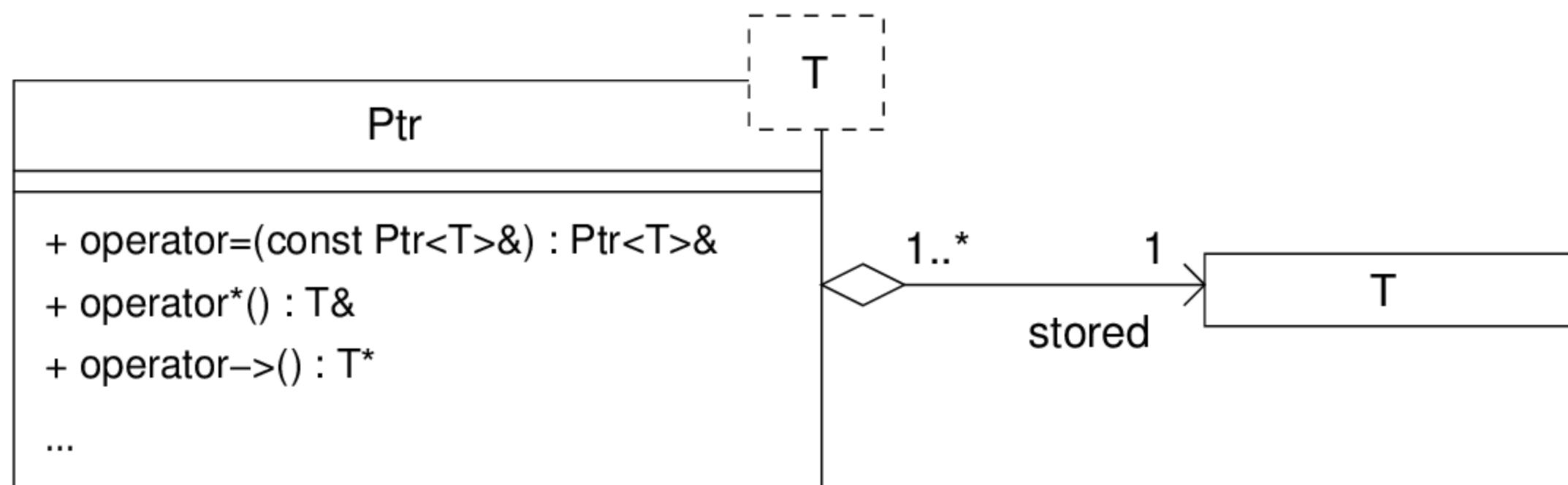


FIGURE 7.4 – Diagramme de classes d'un pointeur intelligent vu comme un *Proxy*.

pointeur vers `T` auquel se substitue le proxy, ce qui correspond bien à l'agrégation illustrée dans cette figure.

7.6.3 Le modèle `unique_ptr`

Le modèle `unique_ptr` permet de garantir strictement la notion de propriété en traitant les trois remarques de la section 7.6.1 de la manière suivante :

- Un pointeur sur une donnée allouée doit être immédiatement stocké dans un `unique_ptr`, pour ne plus être utilisé directement.
- Quand un `unique_ptr` est déplacé dans un autre `unique_ptr`¹, il perd la propriété de la donnée et prend un état équivalent à celui du pointeur `nullptr` (nouvelle valeur du pointeur stocké). Ainsi :
- Seul un objet issu du déplacement d'un objet qui était propriétaire avant lui peut avoir la propriété de la donnée. La propriété est donc bien exclusive (remarque 7.4). Sur le diagramme de la figure 7.4, cela revient à dire que la multiplicité à gauche de l'agrégation devient `0..1` (et non pas `1..*`).
- Il est impossible d'obtenir un `unique_ptr` pendouillant (remarque 7.6). Seul celui qui possède la donnée peut avoir une valeur non nulle.

En fait, un `unique_ptr` ne possède pas de sémantique de recopie mais seulement une sémantique de déplacement. C'est même ce qui garantit son bon fonctionnement. La propriété ne peut pas être dupliquée, elle ne peut qu'être transférée (déplacée).

- Quand un `unique_ptr` est lui-même détruit, si son pointeur stocké est non nul, alors la donnée pointée est aussi libérée. Ce comportement est une réponse à la remarque 7.5. Il garantit que le seul propriétaire de la donnée assurera automatiquement sa libération.

Dans le listing 7.28, les lignes 33 et 36 correspondent à la situation d'une donnée allouée dans une fonction mais dont la propriété revient à la fonction appelante, qui se charge donc de la libération de cette ressource. Cette situation est un cas d'usage typique du modèle `unique_ptr` sur lequel nous reviendrons plus loin.

Avant de rentrer dans le vif du sujet par une description de l'interface du modèle `unique_ptr` et des exemples d'utilisation, il faut préciser que ce modèle possède en réalité deux paramètres de type : le type pointé `T` mais aussi un type de « supprimeur personnalisé » qui sera utilisé pour libérer la donnée en lieu et place de l'opérateur `delete` (qui est bien entendu le supprimeur par défaut). Par souci de simplification, ce second paramètre ne sera pas utilisé ici.

1. Mais aussi dans un `shared_ptr` comme nous le verrons plus loin.

Interface d'un unique_ptr

L'interface quasi-complète du modèle `unique_ptr` est donnée dans le listing 7.29. Cette interface est commentée dans ce qui suit.

- Un `unique_ptr` nul peut être créé via le constructeur par défaut (ligne 6) ou s'il est initialisé avec `nullptr` (ligne 7).
- Un `unique_ptr` peut être construit et initialisé à l'aide d'un pointeur sur `T` (ligne 8).
- En dehors des constructions évoquées précédemment, un `unique_ptr` peut être construit par déplacement uniquement (ligne 9). En effet, il n'est pas possible de construire un tel pointeur par recopie (constructeur marqué `delete`, ligne 12).
- De même un `unique_ptr` ne peut être affecté qu'à partir de `nullptr` ou par déplacement (ligne 14), donc dans le cas d'un transfert de propriété. L'opérateur d'affectation par recopie est en effet supprimé (ligne 18).
- Un `unique_ptr` peut être réinitialisé à nul de différentes façons, *auquel cas il détruit la donnée pointée* :
 - il peut être affecté à `nullptr` (ligne 15) ;
 - par appel de la méthode `release` (ligne 23) qui retourne aussi le pointeur stocké ;
 - par appel de la méthode `reset` sans argument (ligne 24) ;
 - par échange de pointeur avec un `unique_ptr` nul via la méthode `swap` (ligne 25).
- Construction et affectation par déplacement sont possibles à partir d'autres `unique_ptr` qui pointent sur des types compatibles avec `T` (lignes 10/11 et 16/17).
- La méthode `get` et l'opérateur `->` donnent accès au pointeur stocké. L'opérateur flèche n'étant disponible que dans le cas d'une donnée unique (et non un tableau).
- Le test « pointeur nul » est rendu aisément par la surcharge de l'opérateur de conversion en un type `bool` (ligne 22).
- La méthode `reset` (ligne 24) permet de modifier la valeur du pointeur stocké. La donnée qui était anciennement pointée, s'il y en avait une, est préalablement détruite.
- La méthode `swap` (l. 25) permet de permuter les pointeurs stockés par deux `unique_ptr`.
- L'opérateur unaire de déréférencement est défini pour les objets uniques (ligne 26).
- L'opérateur `[]` est défini pour les pointeurs vers des tableaux d'objets (ligne 27).

Remarque 7.7 Il est important de noter que le constructeur d'un `unique_ptr` à partir d'un pointeur de type `T` est déclaré *explicit*. Ce sera d'ailleurs aussi le cas pour le type `shared_ptr`. Il est en effet utile que la création d'un objet de ce type soit toujours demandée explicitement à partir d'un pointeur brut et ne puisse être le résultat d'une conversion implicite. Si c'était possible, l'appel de la fonction `foo()` à la ligne 12 du listing 7.30 serait correct et il ne serait pas du tout clair que l'objet `Circle` sera détruit par la fonction ! Par contre, un appel comme celui de la ligne 13 ne laisse aucun doute sur le fait que la fonction appelante perd la propriété de la donnée.

Exemples d'utilisation

Dans le listing 7.31, on montre le transfert de la propriété du `Circle` alloué par la fonction `makeCircle` vers un `unique_ptr` `p` (ligne 18). C'est le pendant de la même opération réalisée avec un pointeur brut à la ligne 33 du listing 7.28. Toujours dans le listing 7.31, la propriété de la donnée de type `Circle` est ensuite transférée par déplacement au `unique_ptr` `pc` (ligne 23). Le `unique_ptr` `p`, dont le pointeur stocké est alors `nullptr`, ne réalise donc aucune

```

1 template <typename T, typename Deleter = default_delete<T>>
2 class unique_ptr {
3     // Define 'pointer' as a synonym for 'T*'
4     typedef __unique_ptr_trait<T>::pointer pointer;
5 public:
6     constexpr unique_ptr() noexcept;
7     constexpr unique_ptr(nullptr_t) noexcept;
8     explicit unique_ptr(pointer p) noexcept;
9     unique_ptr(unique_ptr && x) noexcept;
10    template <typename T2, typename D2>
11        unique_ptr(unique_ptr<T2,D2> && ) noexcept;
12        unique_ptr(const unique_ptr&) = delete;
13        ~unique_ptr();
14        unique_ptr & operator=(unique_ptr && ) noexcept;
15        unique_ptr & operator=(nullptr_t) noexcept;
16        template <typename U, typename E>
17            unique_ptr & operator=(unique_ptr<U,E> && ) noexcept;
18            unique_ptr & operator=(const unique_ptr & ) = delete;
19
20        pointer get() const noexcept;
21        pointer operator->() const noexcept; // Non-array only
22        explicit operator bool() const noexcept;
23        pointer release() noexcept;
24        void reset(pointer p = pointer()) noexcept;
25        void swap(unique_ptr& x) noexcept;
26        T & operator*() const; // Non-array only
27        T & operator[](size_t i) const; // For arrays only
28    };
29
30    // Operator that apply on the stored pointer (and nullptr)
31
32    template <typename T1, typename D1, typename T2, typename D2>
33    bool operator==(const unique_ptr<T1,D1> & lhs,
34                      const unique_ptr<T2,D2> & rhs);
35    template <typename T, typename D>
36    bool operator==(const unique_ptr<T,D> & lhs, nullptr_t) noexcept;
37    template <typename T, typename D>
38    bool operator==(nullptr_t, const unique_ptr<T,D>& rhs) noexcept;
39
40    // Same for != < <= > >=

```

Listing 7.29 – Interface (quasi-complète) d'un `unique_ptr`.

```

1 #include <memory>
2 using std::unique_ptr;
3 #include "Circle.h"
4 void foo(unique_ptr<Circle> pc)
5 {
6     pc->show();
7 }
8
9 int main(int , char *[])
10 {
11     Circle * pc = new Circle;
12     foo(pc); // Error: implicit conversion is disabled
13     foo(unique_ptr<Circle>(pc)); // Correct
14     foo(unique_ptr<Circle>(new Circle)); // Correct
15     return 0;
16 }
```

Listing 7.30 – Construction nécessairement explicite d'un `unique_ptr`.

désallocation lorsque lui-même disparaît. Par contre, la destruction de `pc` à la fin de la fonction `main` provoquera automatiquement celle du `Circle` pointé par ce dernier.

Dans le listing 7.32, on montre d'utilisation d'un `unique_ptr` comme attribut de classe en remplacement d'un pointeur *vers un tableau d'objets*. Le tableau dynamique dont la taille est donnée à la construction d'un `IntVector` a vocation à disparaître en même temps que l'objet auquel il est lié. De plus, le pointeur n'étant pas destiné à être dupliqué, l'usage d'un `unique_ptr` offre la destruction automatique du tableau sans qu'il soit nécessaire de l'expliciter par l'usage du mot clé `delete[]` dans un destructeur !

Enfin, le listing 7.33 illustre le fait qu'un `unique_ptr` ne peut être recopié, mais seulement déplacé.

En résumé

Un `unique_ptr` offre :

1. La propriété unique, exclusive, d'un pointeur vers un objet (au sens large) ou vers un tableau d'objets.
2. Il possède une sémantique de déplacement. Il ne permet pas la recopie.
3. Lorsqu'un `unique_ptr` disparaît, il libère la donnée pointée.

7.6.4 Le modèle `shared_ptr`

Un `shared_ptr` est un pointeur intelligent au sens le plus classique du terme. Il apporte une réponse aux problèmes soulevés par les remarques 7.4, 7.5 et 7.6 (section 7.6.1) d'une façon radicale : la notion de propriété perd son caractère d'exclusivité ou d'unicité (pourtant fondamentale) et devient *partagée*.

Contrairement à un `unique_ptr`, un `shared_ptr` peut en effet être recopié sans que cela ne

```
1 #include <memory>
2 #include <utility>
3 using std::unique_ptr;
4
5 struct Circle {
6     void draw() {}
7 };
8
9 unique_ptr<Circle> makeCircle() {
10     unique_ptr<Circle> pc(new Circle);
11     return pc;
12 }
13
14 int main()
15 {
16     unique_ptr<Circle> pc;
17     {
18         unique_ptr<Circle> p = makeCircle();
19         // p owns the Circle
20         // pc = p; // Forbidden (deleted operator)
21
22         // Transfer ownership from p to pc
23         pc = std::move(p);
24
25         // p will be destroyed (end of scope) but its
26         // stored pointer is nullptr, so nothing happens
27     }
28     (*pc).draw();
29 } // pc is destroyed, so is the Circle it refers to
```

Listing 7.31 – Exemple d'utilisation d'un `unique_ptr`.

```
1 #include <memory>
2
3 class IntVector {
4 public:
5     IntVector(unsigned size);
6     int & operator[](unsigned index);
7 private:
8     unsigned _size;
9     std::unique_ptr<int[]> _data;
10};
11
12 IntVector::IntVector(unsigned size)
13 : _size(size),
14   _data(new int[size])
15{
16}
17
18 int & IntVector::operator[](unsigned index)
19{
20    return _data[index];
21}
22
23 int main(int , char *[])
24{
25    IntVector v(1024);
26    return 0;
27}
```

Listing 7.32 – Utilisation d'un `unique_ptr` pour la simple désallocation automatique.

```

1 #include <memory>
2 #include <utility>
3 using std::unique_ptr;
4
5 struct Circle {
6     void draw() {}
7 };
8
9 void foo(unique_ptr<Circle> pc)
10 {
11     pc->draw();
12 } // *pc will be destroyed
13
14 int main()
15 {
16     unique_ptr<Circle> p(new Circle);
17     // foo(p); // Forbidden. Copy constructor is deleted.
18     foo(std::move(p)); // Take p as an rvalue.
19                         // Ownership is transferred to 'pc'
20                         // Here, p's stored pointer is nullptr.
21 } // Nothing to be destroyed here. 'foo' did the job!

```

Listing 7.33 – Illustration du fait que la construction par recopie d'un `unique_ptr` est impossible.

pose de problème. Ne retombe-t-on pas alors dans les problèmes évoqués par les remarques 7.4 et 7.5 ? La donnée ne va-t-elle pas être détruite plusieurs fois, ou au contraire jamais détruite car on se saura plus qui a la responsabilité de le faire ?

La parade est simple : connaître (donc mettre à jour) en permanence le nombre de pointeurs qui font référence à la donnée. C'est le rôle d'un bloc de contrôle associé au pointeur et dans lequel on trouve un compteur de références, comme indiqué dans le diagramme de classes de la figure 7.5.

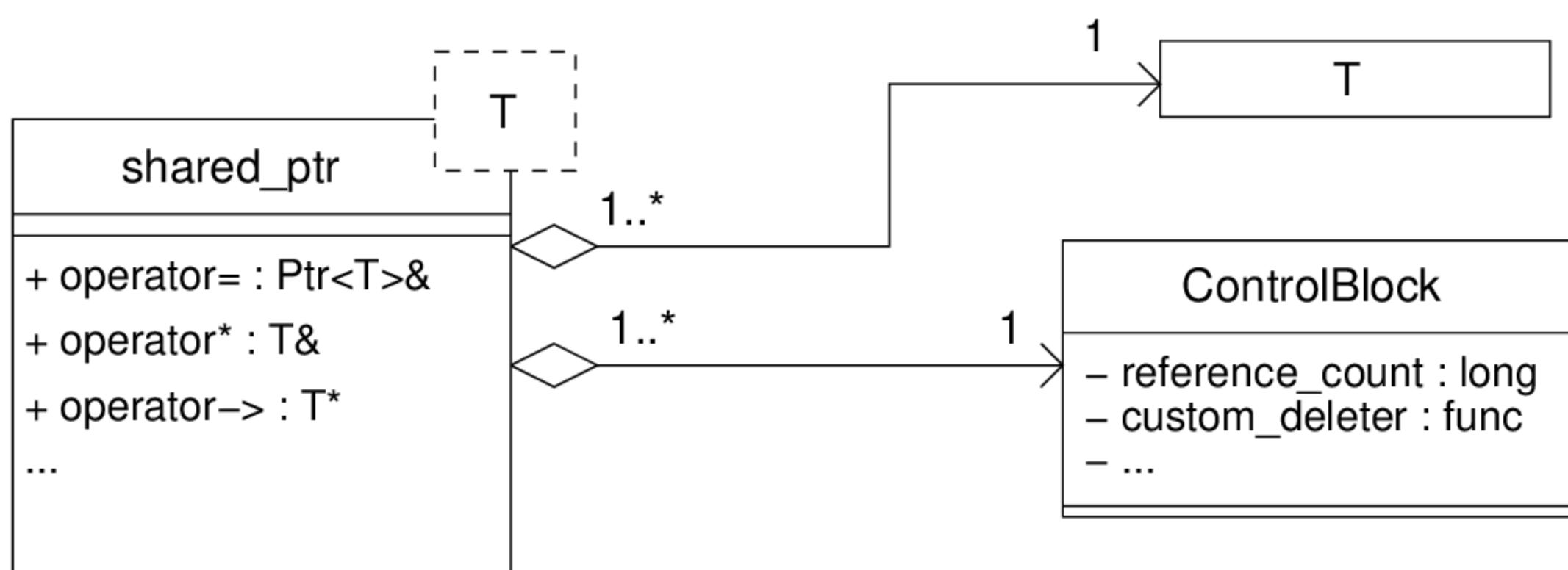


FIGURE 7.5 – Diagramme de classes d'un `shared_ptr`.

On constate bien dans ce diagramme que plusieurs `shared_ptr<T>` peuvent être liés à un même couple (`T`, bloc de contrôle). Simplement, chaque recopie ou destruction d'un pointeur intelligent

de type `shared_ptr` se traduira par une modification du compteur de référence contenu dans le bloc de contrôle.

Ainsi, en cas de recopie d'un `shared_ptr`, le compteur de référence est simplement incrémenté. Dans le cas de la destruction d'un `shared_ptr`, ce compteur est décrémenté. S'il atteint la valeur zéro, le `shared_ptr` qui effectue la décrémentation doit alors détruire la donnée puisqu'il était le dernier à y faire référence. C'est ce qui garantit la destruction automatique (et unique) de la donnée pointée.

La figure 7.6 présente une situation dans laquelle trois pointeurs (`p1`, `p2` et `p3`) partagent une ressource et un quatrième pointeur en possède une autre. La figure 7.7 représente la situation obtenue après l'instruction :

```
p4 = p3;
```

Dans ce cas, le compteur de référence associé initialement à `p4` est tout d'abord décrémenté. Il passe donc à zéro ce qui provoque la destruction de la donnée pointée. Ensuite le compteur de référence de `p3` (et donc du « nouveau » `p4`) est incrémenté.

Remarque 7.8 *Un `shared_ptr` ne peut pas être pendouillant (remarque 7.6). S'il n'est pas nul, c'est que sa donnée n'a pas été désallouée. (S'il pointe dessus, le compteur de référence est au moins égale à 1.)*

Interface d'un `shared_ptr`

L'interface quasi-complète du modèle `shared_ptr` est donnée dans le listing 7.34. Pour une plus grande clarté et concision, les modèles de méthodes sont simplement marqués par un commentaire. Ainsi, la ligne suivante :

```
5 || explicit shared_ptr(Y * ); /*<Y>*/
```

doit être lue comme :

```
5 || template<typename Y> explicit shared_ptr(Y * );
```

Cette interface est commentée dans ce qui suit.

- Un `shared_ptr` nul peut être créé via le constructeur par défaut (ligne 3) ou s'il est initialisé avec `nullptr` (ligne 4).
- Un `shared_ptr` peut être construit et initialisé à l'aide d'un pointeur sur `Y`, à condition que `Y` soit convertible en `T` (ligne 5). Un *supprimeur personnalisé* peut aussi être précisé à la construction dans ce cas (ligne 6).
- Contrairement à un `unique_ptr`, un `shared_ptr` peut être construit et affecté *par recopie* depuis un autre `shared_ptr` de type identique ou compatible (lignes 7, 8, 13 et 14).
- Un `shared_ptr` peut être construit et affecté par déplacement (lignes 9, 10, 15 et 16).
- Un `shared_ptr` peut être construit ou affecté par *déplacement* d'un `unique_ptr` (lignes 12 et 17). Il acquière donc dans ce cas la propriété de la donnée, qui est perdue par l'`unique_ptr`.
- Un `shared_ptr` peut être réinitialisé à nul de différentes façons, auquel cas il perd la propriété de la donnée, le compteur de références étant alors décrémenté :

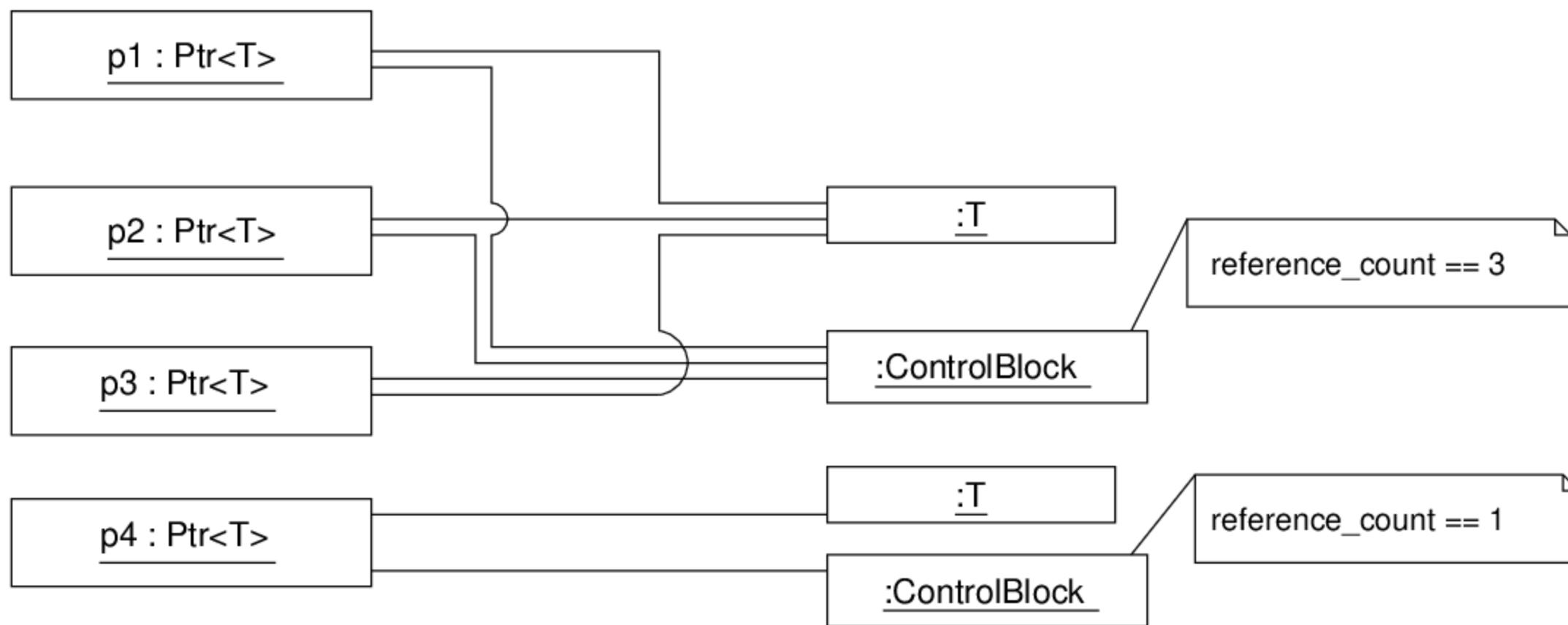
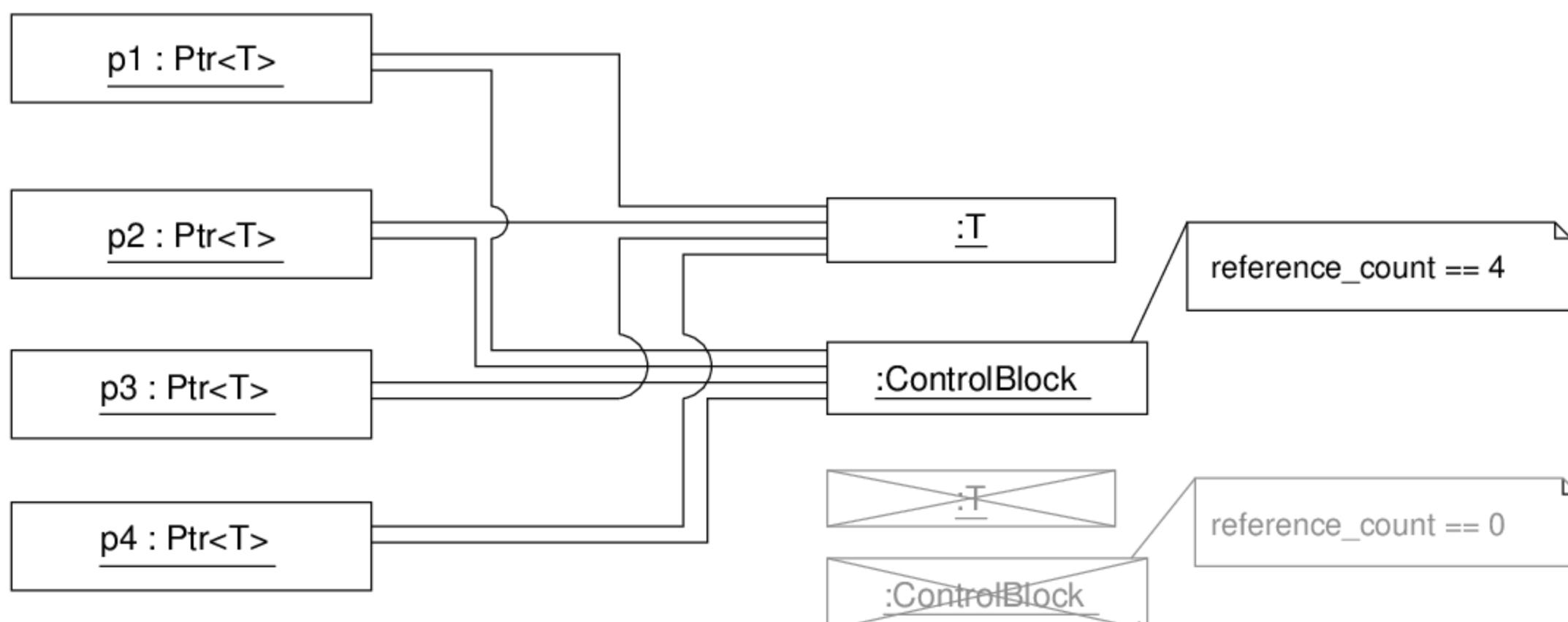


FIGURE 7.6 – Diagramme d'instances mettant en jeu des pointeurs intelligents.

FIGURE 7.7 – Résultat obtenu après l'affectation `p4 = p3` à partir de la situation de la figure 7.6.

- il peut être affecté à `nullptr` (ligne 4, par construction « implicite » d'un `shared_ptr` temporaire nul) ;
- par appel de la méthode `reset` sans argument (ligne 18) ;
- par échange de pointeur avec un `shared_ptr` nul via la méthode `swap` (ligne 21).
- La méthode `get` et l'opérateur `->` donnent accès au pointeur stocké (lignes 22 et 24). L'opérateur flèche n'étant disponible que dans le cas d'une donnée unique (et non un tableau).
- L'opérateur unaire de déréférencement est défini pour les objets uniques (ligne 23).
- L'opérateur `[]` est défini pour les pointeurs vers des tableaux d'objets (ligne 25).
- Le test « pointeur nul » est rendu aisément par la surcharge de l'opérateur de conversion en un type `bool` (ligne 27).
- La méthode `reset` permet de modifier la valeur du pointeur stocké à partir d'un pointeur brut (lignes 19 et 20). Le compteur de références de l'ancien contenu est décrémenté, et une libération intervient le cas échéant.
- La méthode `swap` permet de permutez les pointeurs stockés par deux `shared_ptr` (ligne 21).
- La méthode `use_count` permet de connaître la valeur du compteur de références (ligne 26).
- Enfin, les `shared_ptr` peuvent être comparés, au sens du pointeur stocké (ligne 30 et suivantes).

Enfin, notez qu'il est possible de construire un `shared_ptr` par recopie (ligne 11) d'un troisième type de pointeur intelligent, le `weak_ptr` qui est décrit dans la section suivante (7.6.5). Ce constructeur étant *explicit* il ne définit cependant pas de conversion implicite depuis ce type.

Le modèle de fonction `make_shared`

La construction d'un `shared_ptr` peut se faire de la manière suivante :

```
1 || shared_ptr<Circle> pc1(new Circle(0,0,5));
2 || auto pc2 = shared_ptr<Circle>(new Circle(0,0,5));
```

Listing 7.35 – Deux exemples d'initialisation d'un `shared_ptr`.

Toutefois, on peut faire deux reproches à cette façon de faire :

1. Le type de la donnée (ici `Circle`) doit être répété dans les deux cas, ce qui est laborieux, inélégant et peut être source d'erreur.
2. Chose qui peut sembler à la fois évidente et inévitable, deux objets sont en fait alloués : un `Circle` et un bloc de contrôle (voir figure 7.5, ou plus schématiquement encore dans la figure 7.8(a)).

La bibliothèque standard apporte une réponse à ces deux reproches avec le modèle de fonction `make_shared`. Grâce à ce modèle, le code du listing 7.35 peut être réécrit :

```
1 || shared_ptr<Circle> pc1 = make_shared<Circle>(0,0,5);
2 || auto pc2 = make_shared<Circle>(0,0,5);
```

Ce modèle, combiné à la déduction de type, évite l'utilisation redondante du type de la donnée pointée. Les paramètres du constructeur sont alors passés comme arguments de la fonction. La déclaration du modèle de fonction `make_shared` est la suivante :

```
1 || template<typename T, typename ... Args>
```

```

1 template <typename T>
2 class shared_ptr {
3     constexpr shared_ptr() noexcept;
4     constexpr shared_ptr(nullptr_t) noexcept;
5     explicit shared_ptr(Y * ); /*<Y>*/
6     shared_ptr(Y * , Deleter ); /*<Y, Deleter>*/
7     shared_ptr(const shared_ptr & ) noexcept;
8     shared_ptr(const shared_ptr<Y> & ) noexcept; /*<Y>*/
9     shared_ptr(shared_ptr && ) noexcept;
10    shared_ptr(shared_ptr<Y> && ) noexcept; /*<Y>*/
11    explicit shared_ptr(const weak_ptr<Y> & ); /*<Y>*/
12    shared_ptr(unique_ptr<Y, Deleter> && ); /*<Y, Deleter>*/
13    shared_ptr & operator=(const shared_ptr & ) noexcept;
14    shared_ptr & operator=(const shared_ptr<Y> & ) noexcept; /*<Y>*/
15    shared_ptr & operator=(shared_ptr && ) noexcept;
16    shared_ptr & operator=(shared_ptr<Y> && ) noexcept; /*<Y>*/
17    shared_ptr & operator=(unique_ptr<Y, Deleter> && ); /*<Y, Deleter>*/
18    void reset() noexcept;
19    void reset(Y * ); /*<Y>*/
20    void reset(Y* , Deleter ); /*<Y, Deleter>*/
21    void swap(shared_ptr & ) noexcept;
22    T * get() const noexcept;
23    T & operator*() const noexcept;
24    T * operator->() const noexcept;
25    T & operator[](ptrdiff_t) const; // C++ 17
26    long use_count() const noexcept;
27    explicit operator bool() const noexcept;
28    ~shared_ptr();
29 };
30 template <typename T, typename U>
31 bool operator==(const shared_ptr<T> & ,
32                   const shared_ptr<U> & ) noexcept;
33 template<typename T>
34 bool operator==(const shared_ptr<T> & ,
35                   nullptr_t ) noexcept;
36 // Same for != <> <=>
```

Listing 7.34 – Interface (quasi-complète) d'un `shared_ptr`.

```
2 | shared_ptr<T> make_shared( Args &&... args );
```

Mais l'intérêt du modèle de fonction `make_shared` est surtout la solution qu'il apporte au second reproche formulé précédemment : une seule allocation (au lieu de deux) sera effectuée pour stocker l'objet pointé ainsi que le bloc de contrôle. Ainsi, un `shared_ptr` créé à l'aide de cette fonction correspondra en mémoire à la configuration décrite dans la figure 7.8(b), alors qu'une initialisation par le constructeur classique correspond au schéma de la figure 7.8(a).

Comme les durées de vie de l'objet pointé et du bloc de contrôle sont a priori² les mêmes, l'optimisation apportée par le modèle `make_shared` est tout à fait judicieuse et il n'y a pas lieu de s'en priver.

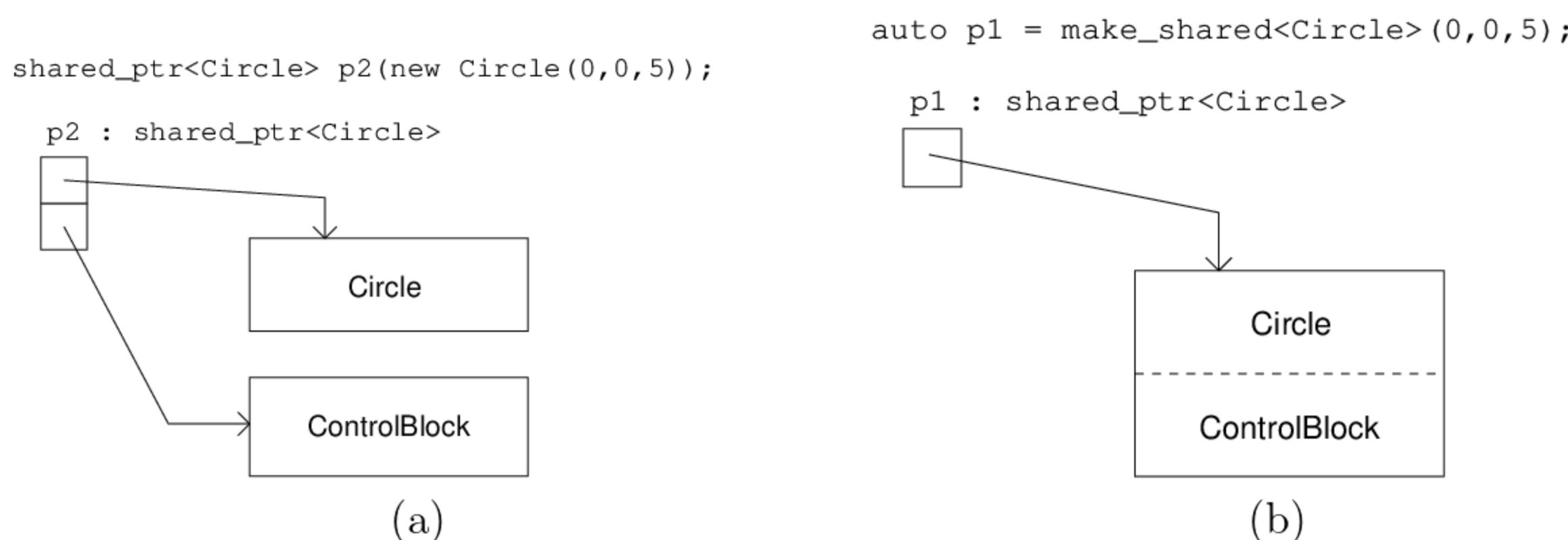


FIGURE 7.8 – Un `shared_ptr` (a) obtenu à partir d'un pointeur brut comparé à (b) celui fourni par `make_shared` en (b).

Exemples d'utilisation

Le listing 7.36 donne quelques exemples d'utilisation d'un `shared_ptr`. Ainsi, à la ligne 18 l'affectation à `nullptr` d'un pointeur qui est le seul propriétaire d'un `Circle` provoque immédiatement la destruction de cet objet. Ce comportement est à comparer au non-déterminisme de pareille situation en langage Java.

À la ligne 20, le pointeur `pc2` est recopié dans un paramètre `circle` local à la fonction `foo`. Le compteur de références associé à la fois à `pc2` et `circle` prend alors la valeur 2. Il est ramené à 1 par l'appel de la méthode `reset` (ligne 9) qui retire à `circle` la possession du pointeur initialement stocké dans `pc2`. L'objet `circle` stocke alors le pointeur `nullptr` et, par convention, son compteur de références vaut 0 (lignes 10 et 11). La valeur du compteur de référence de `pc2` après la ligne 10 est lui affiché à la ligne 21.

On peut insister notamment sur le fait que l'objet `circle` de type `shared_ptr<Circle>` qui est détruit à la sortie de la fonction `foo()`, n'étant plus associé à une donnée, ne provoquera aucune autre destruction que sa propre libération. Enfin, il faut souligner si besoin que ce code ne provoque aucune fuite de mémoire malgré l'absence d'utilisation (explicite) du mot-clé `delete` !

Le listing 7.37 montre l'usage d'un `shared_ptr` pour un tableau d'objet. Notez l'importance du paramètre de type `T[]` (ici `Circle[]`).

2. En fait, ce n'est pas vrai du fait de l'existence des **weak ptr** ...

```

1 #include <iostream>
2 #include <cassert>
3 #include <memory>
4 #include "Circle.h"
5
6 void foo( std :: shared_ptr<Circle> circle )
7 {
8     std :: cout << circle .use_count() << std :: endl ; // 2
9     circle .reset ();
10    std :: cout << circle .use_count() << std :: endl ; // 0
11    assert( circle .get () == nullptr ); // OK
12 }
13
14 int main()
15 {
16     // Explicit constructor
17     std :: shared_ptr<Circle> pc( new Circle(0,0,5));
18     pc = nullptr; // Circle ::~Circle()
19     auto pc2 = std :: make_shared<Circle>(10,10,5);
20     foo( pc2 );
21     std :: cout << pc2 .use_count() << std :: endl ; // 1
22 }
```

Listing 7.36 – Exemple d'utilisation du modèle `shared_ptr`.

```

1 #include <memory>
2 #include "Circle.h"
3 int main()
4 {
5     std :: shared_ptr<Circle> spc1( new Circle[10]); // Wrong!
6     std :: shared_ptr<Circle[]> spc2( new Circle[10]); // Correct
7     spc2 [0].draw();
8     // spc2->draw(); // Error (no such operator)
9     // (*spc2).draw(); // Error (no such operator '*')
10    spc2 = nullptr; // Circle ::~Circle() (10 times)
11 }
```

Listing 7.37 – Un `shared_ptr` faisant référence à un tableau (C++17).

En résumé

1. Un `shared_ptr` peut être déplacé et recopié.
2. Un `shared_ptr` n'est pas adapté aux tableaux en C++11, il l'est seulement à partir de C++17.
3. La notion de propriété n'est plus unique et exclusive, au contraire elle est partagée (d'où le nom).
4. La libération est automatique lorsque plus aucun pointeur n'est propriétaire de la donnée.

7.6.5 Le modèle `weak_ptr`

Ce troisième type de pointeur intelligent vient compléter les possibilités offertes par les deux précédents. En première approximation, on peut dire qu'il apporte une réponse à la remarque 7.6 sur les pointeurs pendouillants (section 7.6.1). Dans certaines situations, il peut en effet être souhaitable de disposer de plusieurs pointeurs sur une donnée partagée mais que certains de ces pointeurs ne soient pas pour autant *propriétaires* de cette donnée. Autrement dit, la destruction de cette donnée ne sera pas de leur responsabilité. Il s'ensuit que ces pointeurs peuvent pendouiller puisque les autres pointeurs, propriétaires eux, peuvent disparaître à tout moment et donc libérer la donnée.

Un `weak_ptr` est ainsi obtenu par recopie (ou déplacement depuis C++14) d'un `shared_ptr`, ou d'un autre `weak_ptr`. Il pointe alors sur le même objet *sans en partager la propriété*. Ainsi, le compteur de références associé au `shared_ptr` n'est pas modifié lors d'une telle opération. La donnée pointée peut être libérée lorsque plus aucun `shared_ptr` n'y fait référence. Là où un pointeur brut serait alors pendouillant, le `weak_ptr` passe alors dans l'état dit « périme » (*expired*, ligne 20 du listing 7.38). Autrement dit, il sait que le pointeur stocké n'est plus valide car la donnée a été détruite par le dernier `shared_ptr` qui en était propriétaire.

Pour expliquer comment un `weak_ptr` est capable de détecter qu'il est périme, il convient ici de préciser son implémentation, et celle des `shared_ptr` par la même occasion. Tout d'abord, la raison pour laquelle un `weak_ptr` peut détecter qu'il est périme est simple : il est associé au même bloc de contrôle que le `shared_ptr` dont il est issu, comme indiqué dans le diagramme de la figure 7.9. Contrairement à un `shared_ptr` il n'agira pas sur le compteur de référence, mais une valeur nulle de ce dernier indiquera qu'il est périme. Le fait que le bloc de contrôle existe toujours malgré cette valeur nulle – ce qui peut sembler paradoxal – est expliqué plus loin...

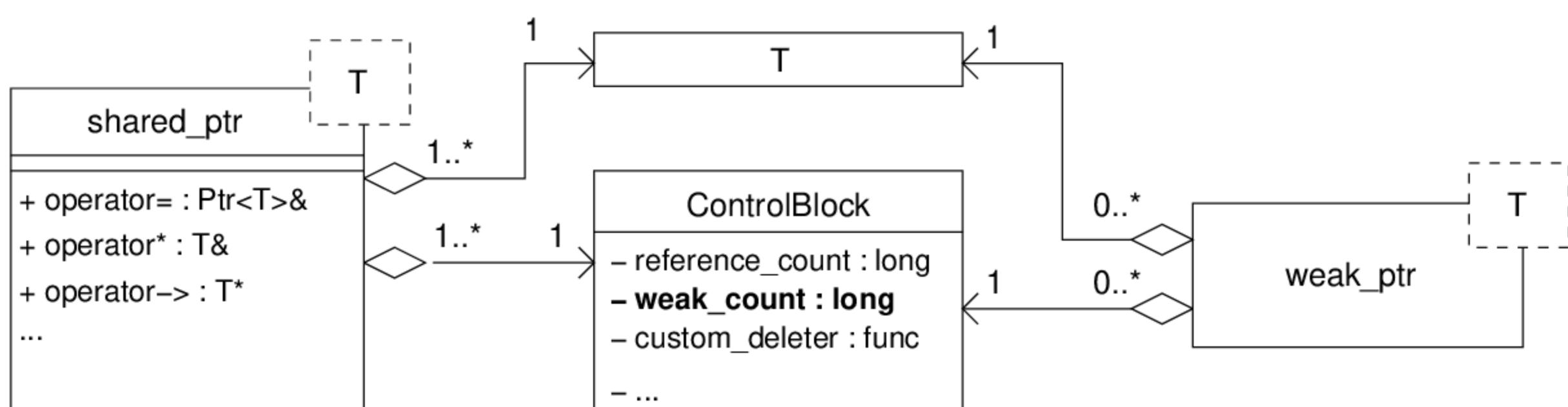


FIGURE 7.9 – Diagramme de classes des `shared_ptr` et `weak_ptr`.

Contrairement aux modèles `unique_ptr` et `shared_ptr`, un `weak_ptr` n'est pas utilisable comme un pointeur. Il n'en possède pas l'interface : déréférencement, méthode `get()`, opérateurs,

etc. Ce point est abordé dans la prochaine sous-section.

Interface d'un `weak_ptr`

L'interface quasi-complète du modèle `weak_ptr` est donnée dans le listing 7.38, avec la même convention de commentaire que celle utilisée précédemment pour les méthodes *templates*.

On remarque tout de suite qu'un `weak_ptr` ne possède pas l'interface d'un pointeur. Avant d'être utilisé comme tel, il doit en effet être converti en un `shared_ptr` ! La raison en est simple. Une utilisation naïve serait en effet la suivante :

```

1 | extern weak_ptr<Circle> wpc;
2 | void foo()
3 | {
4 |     if (!wpc. expired()) {
5 |         wpc->draw();           // There is no such operator (->)
6 |     }
7 | }
```

Or, dans le cas d'une programmation *multi-thread*, entre le test de la ligne 4 et l'utilisation du pointeur à la ligne 5, ce dernier peut avoir expiré ! Afin de simplifier l'utilisation d'un `weak_ptr`, la bibliothèque impose donc qu'il soit convertit en un `shared_ptr` qui devient temporairement (le temps de sa durée de vie) propriétaire de la donnée si le `weak_ptr` n'était pas périmé, permettant ainsi l'utilisation de cette donnée sans risque qu'elle ne soit libérée. Cette conversion peut être réalisée de deux façons :

- Via la méthode `lock()` (listing 7.38, ligne 21) qui retourne un `shared_ptr` nul si le `weak_ptr` est pendouillant/périmé et un `shared_ptr` non nul sinon.
- Via le constructeur explicite de `shared_ptr` à partir d'un `weak_ptr` (listing 7.34, ligne 11) qui lève une exception `std::bad_weak_ptr` si le pointeur est périmé.

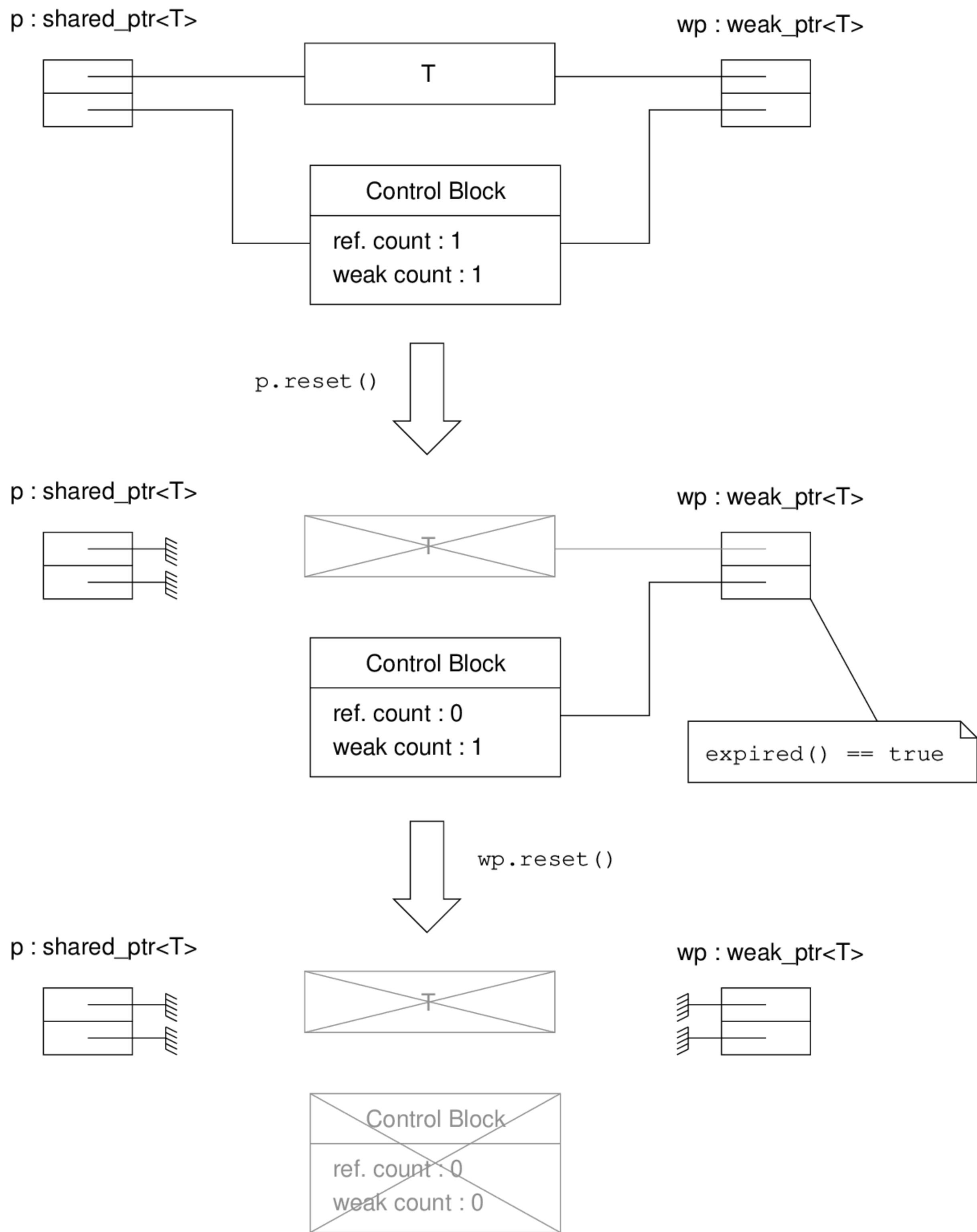
Implémentation

Revenons ici sur un paradoxe apparent : un `weak_ptr` détecte qu'il pendouille car le compteur du bloc de référence auquel il est associé vaut zéro. Mais dans ce cas, le dernier `shared_ptr` qui a décrémenté ce compteur n'est-il pas censé avoir détruit la donnée *et le bloc de contrôle* ? En pratique, si des `weak_ptr` lui sont associés, la réponse est non !

En effet, le bloc de contrôle utilisé par les `shared_ptr` contient en fait un autre compteur qui n'a pas été présenté dans la figure 7.5 : le compteur faible (*weak count*). Il a été ajouté dans la figure 7.9. Ce compteur reflète le nombre de `weak_ptr` qui portent sur le couple (T, bloc de contrôle) et il est mis à jour par ces derniers lors de leurs recopies et destructions. Ainsi, quand un `shared_ptr` décrémente à zéro le compteur de référence :

- si le compteur faible vaut zéro, alors la donnée et le bloc de contrôle sont libérés ;
- si le compteur faible est non nul, alors seule la donnée est libérée, le bloc de contrôle est préservé (mais son compteur de référence est bien mis à zéro).

Dans le deuxième cas, il faudra attendre que le dernier `weak_ptr` associé soit détruit pour qu'il libère le bloc de contrôle. Ce comportement est illustré dans la figure 7.10. Bien entendu, le comportement obtenu par l'utilisation des méthodes `reset()` est en tout point similaire à celui correspondant aux destructeurs des objets p et wp dont il est question dans cette illustration.

FIGURE 7.10 – Illustration de la collaboration entre `shared_ptr` et `weak_ptr`.

```

1 template <typename T>
2 class weak_ptr {
3 public:
4     constexpr weak_ptr() noexcept;
5     weak_ptr(const weak_ptr &) noexcept;
6     weak_ptr(const weak_ptr<Y> &) noexcept; /*<Y>*/
7     weak_ptr(const shared_ptr<Y> &) noexcept; /*<Y>*/
8     weak_ptr(weak_ptr &&) noexcept; /* C++14 */
9     weak_ptr(weak_ptr<Y> &&) noexcept; /*<Y> C++14*/
10    ~weak_ptr();
11
12    weak_ptr & operator=(const weak_ptr &) noexcept;
13    weak_ptr & operator=(const weak_ptr<Y> &) noexcept; /*<Y>*/
14    weak_ptr & operator=(const shared_ptr<Y> &) noexcept; /*<Y>*/
15    weak_ptr & operator=(weak_ptr &&) noexcept; /* C++14 */
16    weak_ptr & operator=(weak_ptr<Y> &&) noexcept; /*<Y> C++14*/
17    void reset() noexcept;
18    void swap(weak_ptr &) noexcept;
19    long use_count() const noexcept;
20    bool expired() const noexcept;
21    shared_ptr<T> lock() const noexcept;
22 };

```

Listing 7.38 – Interface partielle du modèle `weak_ptr`.

Remarque 7.9 *Du fait de ce qui vient d’être décrit, il est à noter que dans le cas de l’utilisation du modèle `make_pair`, le bloc de contrôle et l’instance de `T` occupent le même bloc de mémoire. Par conséquent, c’est ce bloc complet qui est susceptible de persister malgré la libération du dernier `shared_ptr`. Si le type `T` est de taille conséquente, ceci peut poser un problème d’efficacité.*

Exemples d’utilisation

Le listing 7.39 illustre le rôle d’un `weak_ptr` dans l’évolution du compteur de référence et du compteur faible associés à un `shared_ptr`. L’acquisition temporaire d’un `weak_ptr` est mise en œuvre dans le listing 7.40.

Enfin, un exemple typique d’utilisation d’un `weak_ptr` intervient en cas de référence cyclique, ce qui peut aboutir à une fuite de mémoire. Considérez en effet la situation du listing 7.41 illustrée par la figure 7.11. Dans pareille situation, aucun des deux objets de types A et B n’est plus accessible. Il sera donc impossible de libérer B via le pointeur `A::pb`, ni de libérer A via le pointeur `B::pa`. Remplacer le pointeur `B::pa` par un `weak_ptr` apporte une solution à ce problème puisque dans ce cas, rien n’empêche la destruction automatique de l’objet A lorsque `main_a` est réinitialisé.

En résumé

- Un `weak_ptr` peut faire référence à un objet possédé par des `shared_ptr`, sans en être lui-même propriétaire. Il n’est donc pas responsable de la libération de cette donnée.

```

1 #include <memory>
2 #include <iostream>
3 using namespace std;
4 #include "Circle.h"
5
6 int main()
7 {
8     shared_ptr<Circle> psc = make_shared<Circle>(0,0,5);
9     // ref_count == 1, weak_count == 0
10    weak_ptr<Circle> pwc = psc;
11    // ref_count == 1, weak_count == 1
12    cout << boolalpha << pwc.expired() << endl; // false
13    psc.reset();
14    // ref_count == 0, weak_count == 1
15    // Circle is destroyed, control block remains
16    cout << boolalpha << pwc.expired() << endl; // 1 (true)
17 }

```

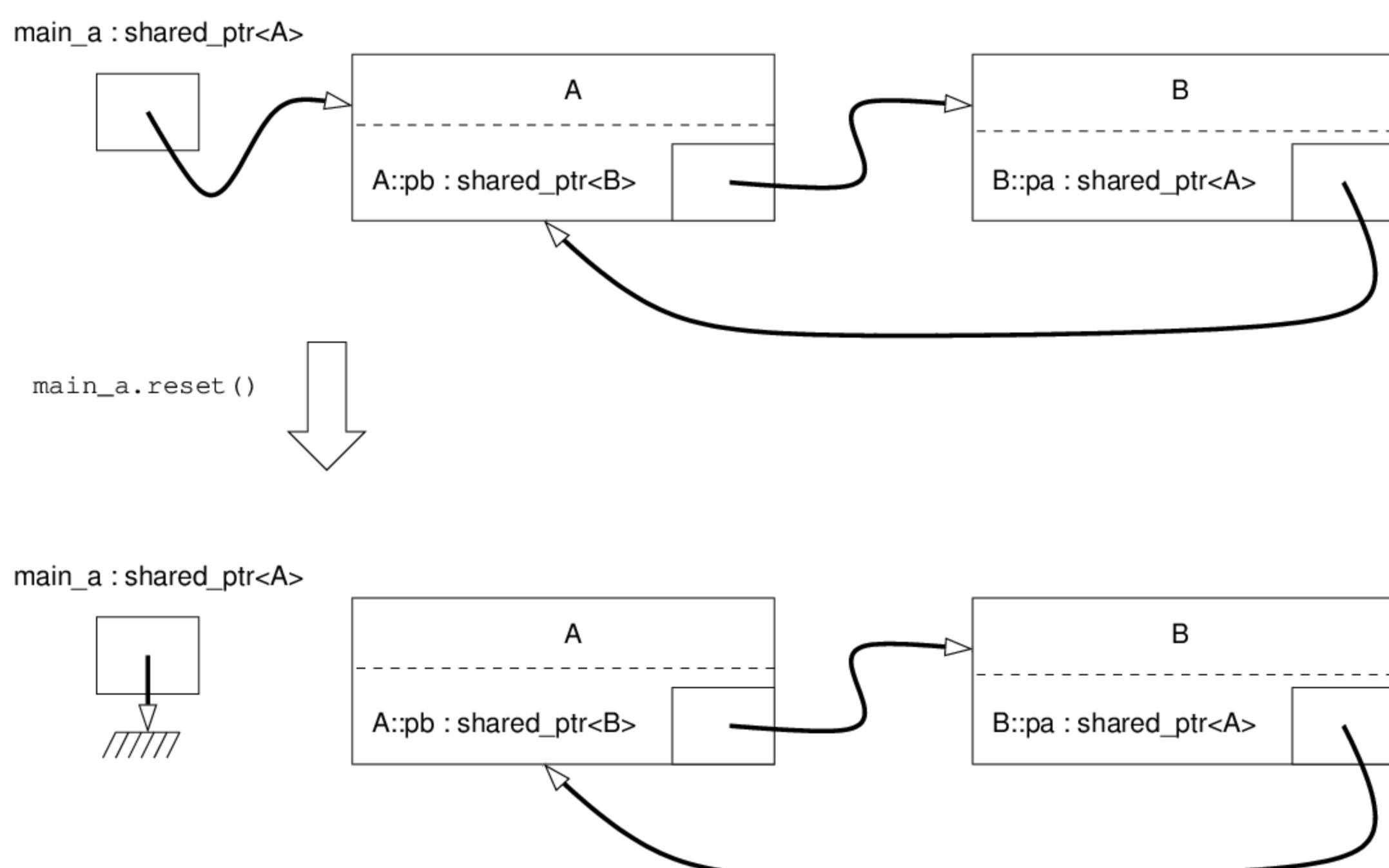
Listing 7.39 – Exemple d'utilisation d'un `weak_ptr`.

FIGURE 7.11 – Référence cyclique

```
1 #include <memory>
2 #include <iostream>
3 using namespace std;
4 #include "Circle.h"
5
6 int main()
7 {
8     shared_ptr<Circle> psc = make_shared<Circle>(0,0,5);
9     weak_ptr<Circle> pwc = psc;
10    {
11        // Take ownership
12        shared_ptr<Circle> tmp = pwc.lock();
13        cout << psc.use_count() << endl; // 2
14    }
15    cout << psc.use_count() << endl; // 1
16    psc.reset();
17    cout << boolalpha << pwc.expired() << endl; // true
18    {
19        // Try to take ownership (failing)
20        shared_ptr<Circle> tmp = pwc.lock();
21        if (tmp) {
22            tmp->draw();
23        } else {
24            cerr << "Circle has been destroyed" << endl;
25        }
26    }
27 }
```

Listing 7.40 – Acquisition temporaire de la propriété via un `weak_ptr`.

```
1 #include <iostream>
2 #include <memory>
3 using namespace std;
4
5 struct B;
6 struct A {
7     shared_ptr<B> pb;
8 };
9 struct B {
10    shared_ptr<A> pa;
11 };
12
13 int main(int , char *[])
14 {
15     auto main_a = make_shared<A>();
16     main_a->pb = make_shared<B>();
17     main_a->pb->pa = main_a;
18     main_a.reset();           // Memory leak!
19     return 0;
20 }
```

Listing 7.41 – Cas de référence cyclique avec des `shared_ptr`.

- Un `weak_ptr` est capable de détecter qu'il pendouille. Il est alors dit périme.
- Ce n'est pas un pointeur utilisable comme tel. Il doit d'abord être converti pour permettre l'accès à la donnée.
- Il peut servir à résoudre des problèmes de référence cyclique.

7.7 Autres nouveautés

D'autres nouveautés, apparues avec la norme de 2011, sont présentées dans ce document là où leur mention s'avère pertinente car elles sont en rapport immédiat avec des notions de la norme précédente. Le tableau 7.1 recense ces notions avec leur position dans le document.

	Page
Le type <code>long long</code>	10
La constante <code>nullptr</code>	12
La délégation de constructeur	39
Initialisation des données membres	39
Constructeur par déplacement	48
<i>rvalue reference</i>	47
Affectation par déplacement	48
<code>default</code> et <code>delete</code>	50
Opérateur de conversion explicite	63
Héritage explicite des constructeurs	71
<code>override</code> et <code>final</code>	85
<code>extern template</code>	97
Nouveaux conteneurs de la STL	114
Le modèle <code>tuple</code>	118
Nouveaux algorithmes de la STL	130
Mot-clé <code>noexcept</code>	144

TABLE 7.1 – Nouveautés du C++11 présentées au fil des pages.



2^e année – Spécialité Informatique

Année 2024 – 2025

Programmation orientée objet en C++

Notes de cours

Sébastien Fourey

Chapitre 8. Épilogue, Bibliographie, Index

Version du 3 septembre 2024



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

Chapitre 8

Épilogue



Fig. 5. — Haricot qui a germé *en pleine lumière*. Il est vert, son poids a augmenté : il a vécu sur le carbone qu'il a pris à l'acide carbonique de l'air.

Nous terminons ce document par la présentation du moyen d'obtenir, en C++, des informations propres à une implémentation particulière. Une deuxième courte section est consacrée à des mots-clés qui existent pour des raisons essentiellement historiques et que nous qualifions ici d'exotiques.

8.1 Données propres à l'implémentation

En C++ des données concernant les types de base sont fournies par l'en-tête `<limits>`. Pour ce faire, chaque implémentation définit ses propres *spécialisations* d'un modèle de structure, le modèle `numeric_limits<>`. Celui-ci définit finalement les *traits*¹ des types de base du langage (voir listing 8.1). donne un aperçu du modèle général.

Exemples

Avec le compilateur GNU g++ (GCC) 3.2.1 sur une architecture SPARCTM :

L'instruction

```
|| cout << numeric_limits<unsigned long long>::max();
```

affiche :

```
|| 18446744073709551615
```

D'autre part, l'instruction

```
|| cout << numeric_limits<unsigned long long>::digits;
```

affiche

```
|| 64
```

Exercice 8.1 Quelle est la relation entre les deux entiers qui viennent d'être « affichés » ?

1. La notion de *trait* (c.-à-d. caractéristique), basée sur la spécialisation de modèle de classe ou de structure, est normalement abordée par l'exemple en cours, dans le cadre de la programmation générique (§ 3.3).

```

1 template <typename T>
2 struct numeric_limits {
3     static const bool is_specialized;
4     static const int digits = ? ? ? ;
5     static const bool is_signed = ? ? ? ;
6     static const bool is_integer = ? ? ? ;
7     static const bool is_exact = ? ? ? ;
8     static const int radix = ? ? ? ;
9     static const int digits = ? ? ? ;
10    static const int digits10 = ? ? ? ;
11    static const bool is_modulo = ? ? ? ;
12    static const bool has_infinity;
13    inline static T min_exponent = ? ? ? ;
14    inline static T max_exponent = ? ? ? ;
15    inline static T min() throw();
16    inline static T max() throw();
17    inline static T epsilon() throw();
18    inline static T infinity() throw();
19    // ...
20 };

```

Listing 8.1 – Extrait de la définition du modèle `numeric_limits<>`.

Notez que le type « long long » est apparu avec la norme de 2011 du langage, même s'il était disponible auparavant avec certains compilateurs comme g++.

8.2 Exotisme, les synonymes d'opérateurs

Plusieurs opérateurs du langage ont des synonymes sous la forme de mots-clés (qui sont donc des noms réservés).

Opér.	Mot-clé	Opér.	Mot-clé	Opér.	Mot-clé
<code>&&</code>	<code>and</code>	<code>&</code>	<code>bitand</code>	<code>&=</code>	<code>and_eq</code>
<code> </code>	<code>or</code>	<code> </code>	<code>bitor</code>	<code> =</code>	<code>or_eq</code>
<code>!</code>	<code>not</code>	<code>~</code>	<code>compl</code>	<code>~=</code>	<code>not_eq</code>
		<code>^</code>	<code>xor</code>	<code>^=</code>	<code>xor_eq</code>

TABLE 8.1 – Synonymes d'opérateurs.

Le listing 8.2 montre que ces mots-clés, qui restent d'un usage marginal, mériteraient peut être d'être utilisés plus souvent puisqu'ils apportent, pour un lecteur peu familier du langage, un certain gain d'« intelligibilité ». Il existe aussi des équivalents pour certaines unités lexicales (cf. tableau 8.2). Il ne s'agit pas d'opérateurs, et à la différence des synonymes précédents leur utilisation a tendance à obscurcir le code pour le programmeur non averti.

Symbol	Équiv.	Symbol	Équiv.
{	<%	}	%>
[<:]	:>
#	%:	##	%:%:

TABLE 8.2 – Synonymes d'unités lexicales.

```

1 struct Knowledge {};
2 struct Teacher {
3     void increases(Knowledge & k);
4 };
5 struct Language {
6     bool isGoodToKnow() {
7         return true;
8     }
9     Language & operator++();
10    Language operator++(int);
11 };
12 struct Reading {
13     bool isGood() {
14         return true;
15     }
16 };
17 Reading theLecture() {
18     return Reading();
19 }
20 bool youKnow(Language language);
21 namespace Your
22 {
23     int Help;
24 }
25
26 int main() {
27     Knowledge yourKnowledge;
28     Teacher sebastienFoureys;
29     Language c;
30
31     if ((youKnow(c++) or theLecture().isGood())
32         and (c++.isGoodToKnow())) {
33         using Your::Help;
34         sebastienFoureys.increases(yourKnowledge);
35     }
36 }
```

Listing 8.2 – Un code correct et très lisible, mais plutôt vers la fin.

Remerciements

Les personnes citées ci-après ont contribué à l'amélioration de la qualité de ce support par leurs suggestions de corrections ou simples modifications.

- Nicolas Signolle, attaché temporaire à l'enseignement et à la recherche à l'ENSICAEN en 2007-2008 ;
- Christine Porquet, maître de conférences à l'ENSICAEN ;
- Loïc Simon, maître de conférences à l'ENSICAEN ;
- Yoran Le Bagousse, étudiant en spécialité informatique à l'ENSICAEN (promotion 2014) ;
- Charles Chaudet, étudiant en spécialité informatique à l'ENSICAEN (promotion 2019).
- Bastien Hubert, étudiant en spécialité informatique à l'ENSICAEN (promotion 2021).
- Julien Zaïdi, étudiant en spécialité informatique à l'ENSICAEN (promotion 2022).

Tout élève est invité à mettre en défaut le contenu de ce document pour voir apparaître son nom dans les versions futures. En effet, l'auteur ne peut prétendre qu'appocher (modestement) de très loin le niveau d'exactitude du créateur du langage. D'ailleurs, Bjarne Stroustrup lui-même commet des erreurs comme le confirme la page d'errata de la 3^e édition du livre de référence [8] dont l'URL est donnée ci-dessous.

cf. <http://www.informit.com/content/images/0201889544/errata%5C833.pdf>

Conclusion

Beaucoup de sujets n'ont pas été abordés dans ce document. Certains sont traités lors des cours magistraux, d'autres sont rencontrés à l'occasion des travaux pratiques. Les exercices proposés dans les différents chapitres devraient aussi avoir initié quelques recherches personnelles. À titre de document de référence complet sur le langage, et en dehors de la norme elle-même [4], le livre de Bjarne Stroustrup [11] est sans doute le plus complet.

Le support s'achèvera par une citation de M. Clouard, enseignant-chercheur en informatique à l'ENSICAEN depuis 1999.

« Si tu veux savoir programmer en C++, il faut que tu pratiques en permanence ; y compris les week-ends car sinon le lundi, tu ne sais plus rien faire avec. »

Régis Clouard²

2. Entretien avec l'auteur, Été 2006.

Bibliographie

- [1] Paul BERT. *La première année d'enseignement scientifique (Sciences naturelles et physiques)*. Librairie classique Armand Colin et Cie, Paris, 1882. 550 gravures.
- [2] Michel DESVIGNES. *Programmation objet en C++, notes de cours*. ENSICAEN, M^{me} Mullois Édition, 2002.
- [3] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. *Programming languages – C++*. International standard ISO/IEC 14882:2003, Octobre 2003. Remplace 14882:1998.
- [4] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. *Information technology – Programming languages – C++*. International standard ISO/IEC 14882:2011, Mars 2011. Remplace 14882:2003.
- [5] Ray LISCHNER. *STL – Précis & concis*. O'Reilly, 2004.
- [6] Scott MEYERS. *Programmer efficacement en C++*. Dunod, 2016.
- [7] STANDARD C++ FOUNDATION. C++ FAQ, Exceptions and Error Handling [Page Web]. <https://isocpp.org/wiki/faq/exceptions#ctors-can-throw>, 2016. [visité en octobre 2022].
- [8] Bjarne STROUSTRUP. *The C++ programming language, third edition*. Addison Wesley Publishing Company, 1997.
- [9] Bjarne STROUSTRUP. *C++*. CampusPress France, 1999. Troisième édition du livre de Bjarne Stroustrup, traduit de l'américain par Christine Eberhardt.
- [10] Bjarne STROUSTRUP. Sibling Rivalry: C and C++. AT&T Labs – Research Technical Report TD-54MQZY, http://www.stroustrup.com/sibling_rivalry.pdf, Janvier 2002. [visité en octobre 2022].
- [11] Bjarne STROUSTRUP. *The C++ Programming Language, fourth edition*. Pearson Education, 2013.
- [12] Bjarne STROUSTRUP. Bjarne Stroustrup's C++11 FAQ [Page Web]. <http://www.stroustrup.com/C++11FAQ.html>, 2014. [visité en octobre 2022].
- [13] Bjarne STROUSTRUP. Bjarne Stroustrup's FAQ [Page Web]. http://www.stroustrup.com/bs_faq.html#really-say-that, 2014. [visité en octobre 2022].

Index

() surcharge, 61
-, ++ surcharge, 59
-> surcharge, 61
:: espaces de noms, 15
résolution de portée, 14
= opérateur d'affectation, 44
opérateur d'affectation par déplacement, 48
[] surcharge, 59

algorithmes, *voir* STL, algorithmes
allocation dynamique sur le tas, 25
arguments
 valeurs par défaut, 13
arité des opérateurs et surcharge, 57

Bibliothèque standard, 111
 algorithmes, **125**
 C++11, 130
 modification de séquence, 125
 numériques, 130
 opérations ensemblistes, 129
 séquences triées, 128
 conteneurs ordonnés, 116
 exemples d'utilisation, 118
 iterator_traits, 123
 itérateurs, 122
 d'insertion, 138
 objets fonctions, 130
 prédicats, 135
 binary_function, 132
 bind2nd, éditeur de liaison (bibliothèque standard),
 135
boucle **for**, 14

C++11
 =, affectation par déplacement, 48
 algorithmes, nouveaux ~, 130

std::bind, 161
 boucle **for** et itérateurs, 156
 constructeur par déplacement, 47
 conteneurs, 114
 default et **delete**, 50
 délegation de constructeur, 39
 extern template, 97
 fonctions anonymes, 156
 fonctions *lambda*, 156
 std::function, 159
 héritage explicite des constructeurs, 71
 inférence de type, 155
 initialisation des données membres, 39
 initialisation uniforme, 149
 liste d'initialiseurs, 149
 long long, 10
 new et initialisation, 40
 noexcept, 144
 opérateur de conversion explicite, 63
 override et **final**, 85
 pointeur de fonction généralisé, 159
 pointeurs intelligents, 166
 rvalue reference, 47
 shared_ptr, 173
 tuple, 118
 type de retour suffixé, 163
 unique_ptr, 170
 weak_ptr, 183
 catch, *voir* exceptions
 classe
 abstraite, 84
 amie, *voir friend*
 commentaires, 10
 composer1, **compose1**, 137
 const
 méthodes constantes, 54
 références constantes, 24
 constructeur, 33
 ~ et conversion implicite, 37
 constructeur par déplacement, 47
 constructeur par recopie, 46

délégation de ~, 39
explicit, 37
héritage, 71
héritage explicite des ~, 71
conteneurs, *voir* Bibliothèque standard
conversion
 explicite
 dynamic_cast, 79
 reinterpret_cast, 79
 static_cast, 78
 implicite
 grâce au constructeur, 37
 opérateur de conversion, 61
copy (algorithme de la bibliothèque standard), 127
default
 modificateur, 50
delete
 désallocation, 27
 modificateur, 50
destructeur, 40
 héritage, 71
dynamic_cast, 79
en-têtes standard
 syntaxe, 17
 <fstream>, 107
 <ios>, 104
 <iostream>, 107
 <ostream>, 106
 <sstream>, 108
 <string>, 108
espaces de noms, 15
 anonymes, 19
exceptions, 141
 spécifier des ~, 144
 spécifier des ~ en C++11, 144
 sim standard, 145
 try, catch, 143
explicit
 constructeur, 37
 opérateur de conversion ~, 63
extern, 19
final, 85
fonctions
 amies, 54
 anonymes, 156
 std::function, 159, 161
lambda, 156
modèles de ~, 100
for
 boucle, 14
 nouvelle syntaxe C++11, 156
friend
 classe amie, 56
 fonction amie, 54
héritage, 65
 ~ et conversions, 71
 explicite des constructeurs, 71
 multiple, 86
 et dérivation virtuelle, 90
 inférence de type, 155
 initialisation uniforme, 149
itérateurs, *voir* Bibliothèque standard, itérateurs
less, prédicat de la bibliothèque standard, 135
limites, *voir* **numeric_limits**
liste d'initialiseurs, 149
long long, 10
lvalue, 14
mem_fun, appelant de méthode (bibliothèque standard), 137
modèles, 93
 de classes, 94
 et fonctions amies, 98
 spécialisation, 98
 de fonctions, 100
 surcharge, 102
méthodes
 constantes, 54
 implémentation, 74
 statiques, 33
 virtuelles, 82
 virtuelles pures, 84
namespace, 15
new, 25
numeric_limits, 11, 191
objets
 constructeur par recopie, 46
 contrôle d'accès, 31
 et héritage, 68
 destructeur, 40
 instances de classe, 29
 objets temporaires

création, 38
 références sur les `~`, *voir* réf. constantes
opérateurs
 synonymes d'`~`, 192
opérateur
 `=`, affectation, 44
 `=`, affectation par déplacement, 48
 chaînage (exemple avec `operator>>`), 23
 `~` de conversion, 61
opérateur de conversion explicite, 63
override, 85

pair, modèle des paires d'éléments, 116
 partage de données, 44
 pointeur
 notion de propriété, 166
 pointeurs intelligents, 166
private
 dérivation, 68
 spécificateur d'accès, 31
protected
 dérivation, 68
 spécificateur d'accès, 31
 prédictats, *voir* Bibliothèque standard, prédictats
ptr_fun, fabricant d'objets fonctions, 138
public
 dérivation, 68
 spécificateur d'accès, 31

reinterpret_cast, 79
rvalue, 14
rvalue reference
 affectation par déplacement, 48
 constructeur par déplacement, 47
 `std::move`, 49
 règle des trois, 47
 références, **21**
 argument de fonction, 22
 retour de fonction, 23
 `~` constantes et objets temporaires, 24

shared_ptr, 173
static
 donnée membre, 33
 et unité de compilation, 19
 méthode, 33
static_cast, 78
std::move, 49
STL
 adaptateurs, 116

 conteneurs séquentiels, 115
 surcharge
 d'un modèle de classe (impossible), 96
 d'un modèle de fonction, 102
 de fonctions, 20
 des opérateurs, 57
 des opérateurs
 listes des opérateurs surchargeables, 57
 synonymes d'opérateurs, 192

 tailles des types de base, 10
template, *voir* modèles
this, 74
throw, 143–145
traits
 `iterator_traits`, 123
trois
 règle des `~`, 47
try, *voir* exceptions
tuple, 118
 type de retour suffixé, 163
typeid, 77

unary_function, 132
unique_ptr, 170
using, déclaration, 15
using namespace, directive, 15

weak_ptr, 183