



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

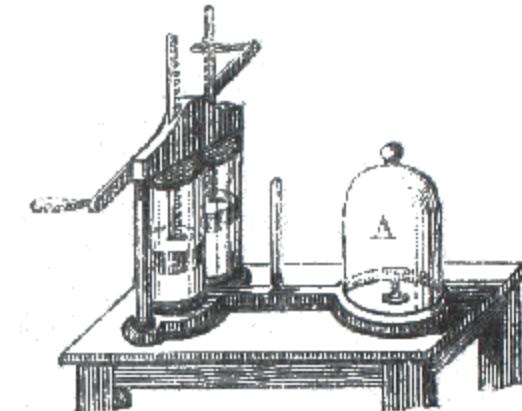


Fig. 122. — Les machines pneumatiques servent à retirer d'un vase A, non pas de l'eau, mais de l'air.

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.

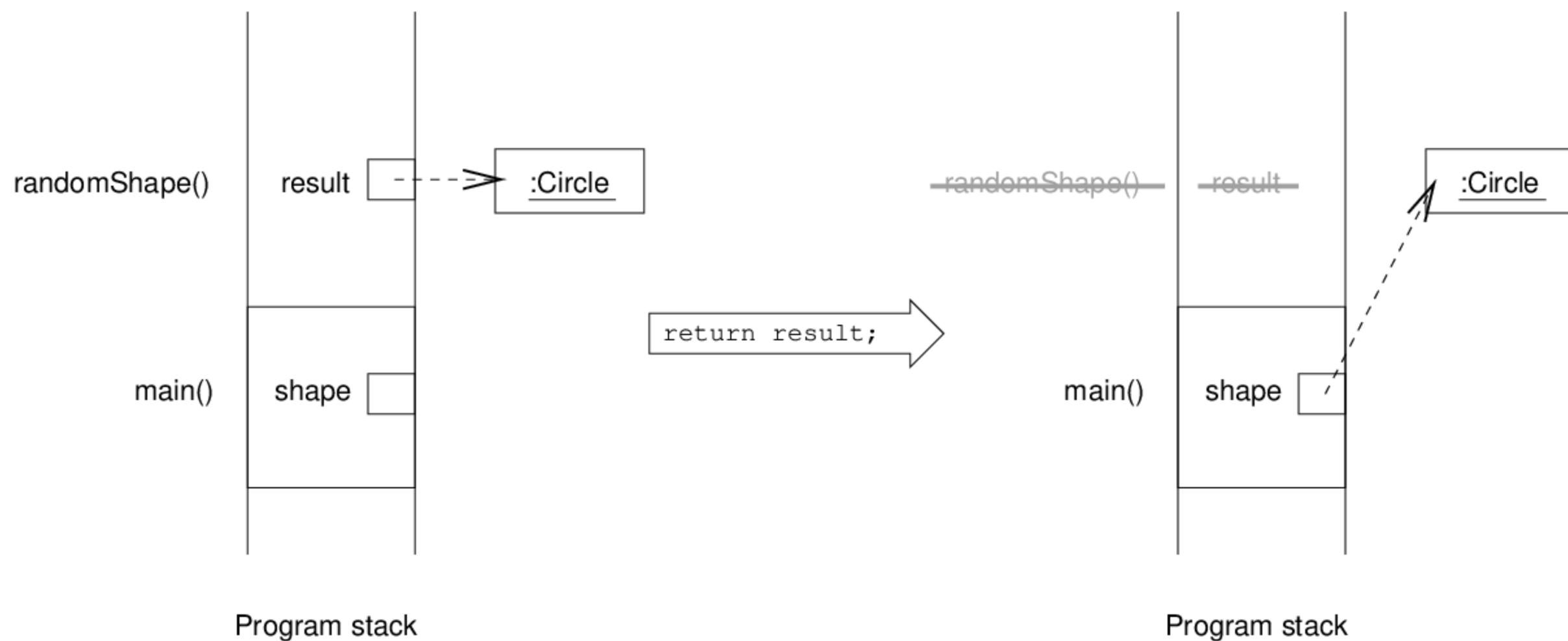


FIGURE 7.1 – Transfert de la propriété d'un objet entre fonctions (lignes 22 et 33 du listing 7.28). La fonction `main` est en charge de la désallocation de cet objet.

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

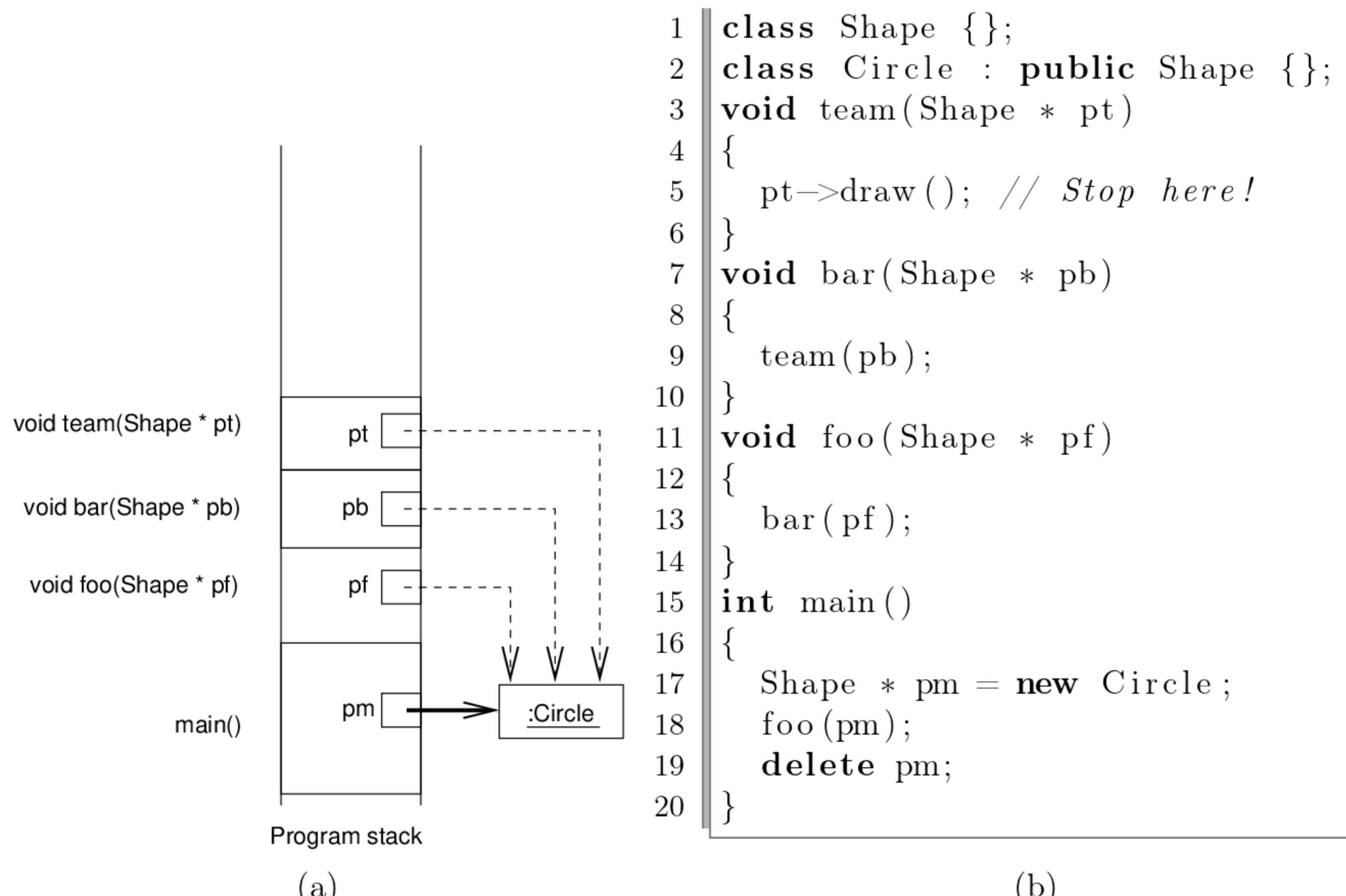


FIGURE 7.2 – Situation dans laquelle plusieurs pointeurs, dans des portées différentes, font référence à une même donnée. Seul un pointeur est propriétaire de cette donnée (flèche en trait plein). La pile d'appels représentée en (a) représente l'état du programme (b) juste avant l'exécution de l'instruction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

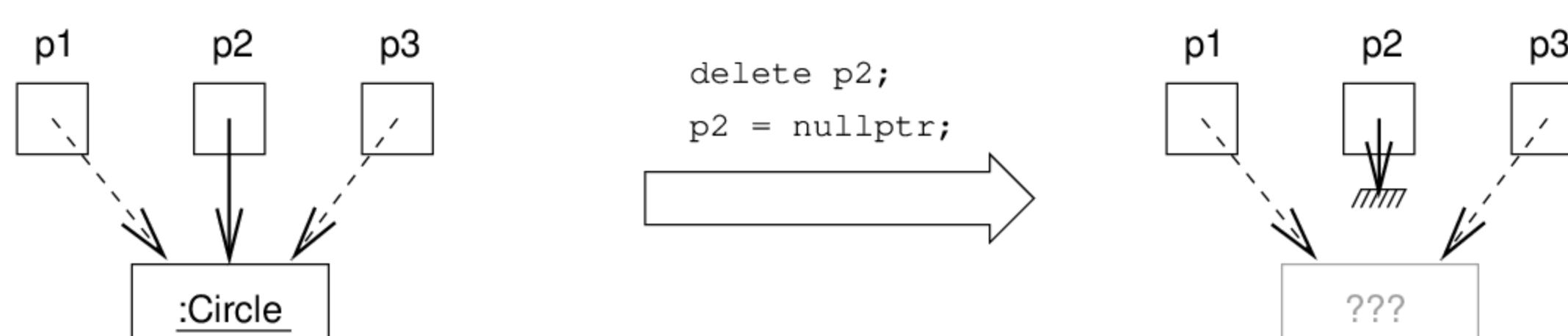


FIGURE 7.3 – Pointeurs pendouillants (*dangling pointers*). Le pointeur `p2` est propriétaire de la donnée de type `Circle` (flèche en trait plein), c'est donc via ce pointeur que cette donnée est libérée. Les pointeurs `p1` et `p3` pointent toujours vers une donnée qui a été libérée.

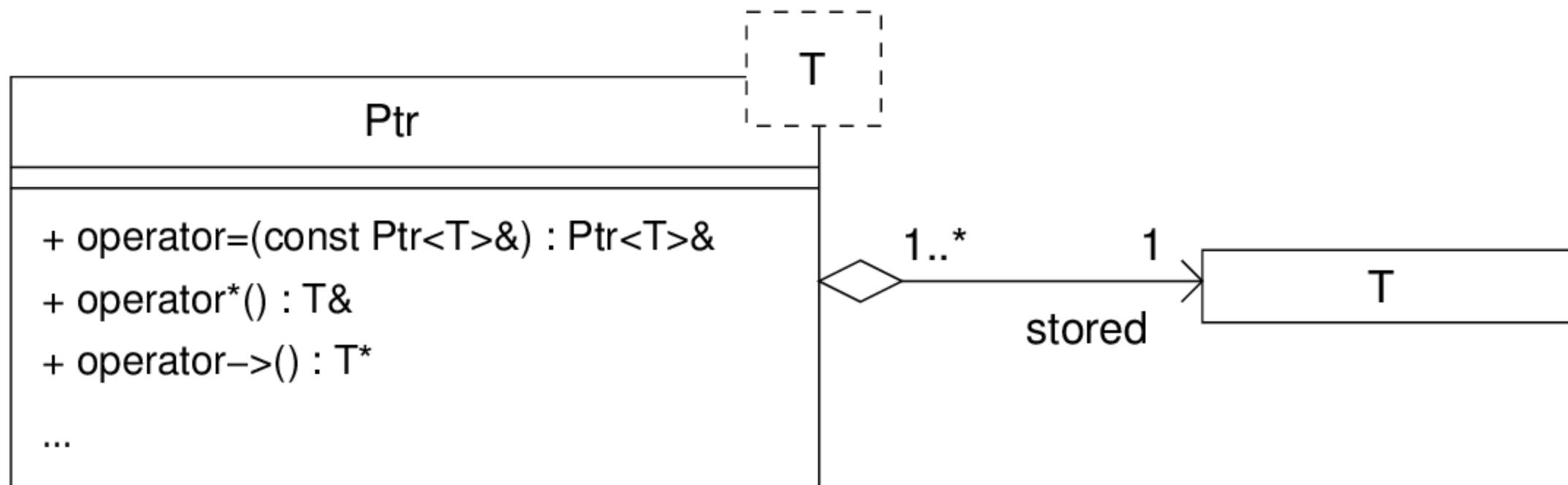


FIGURE 7.4 – Diagramme de classes d'un pointeur intelligent vu comme un *Proxy*.

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 permute 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.

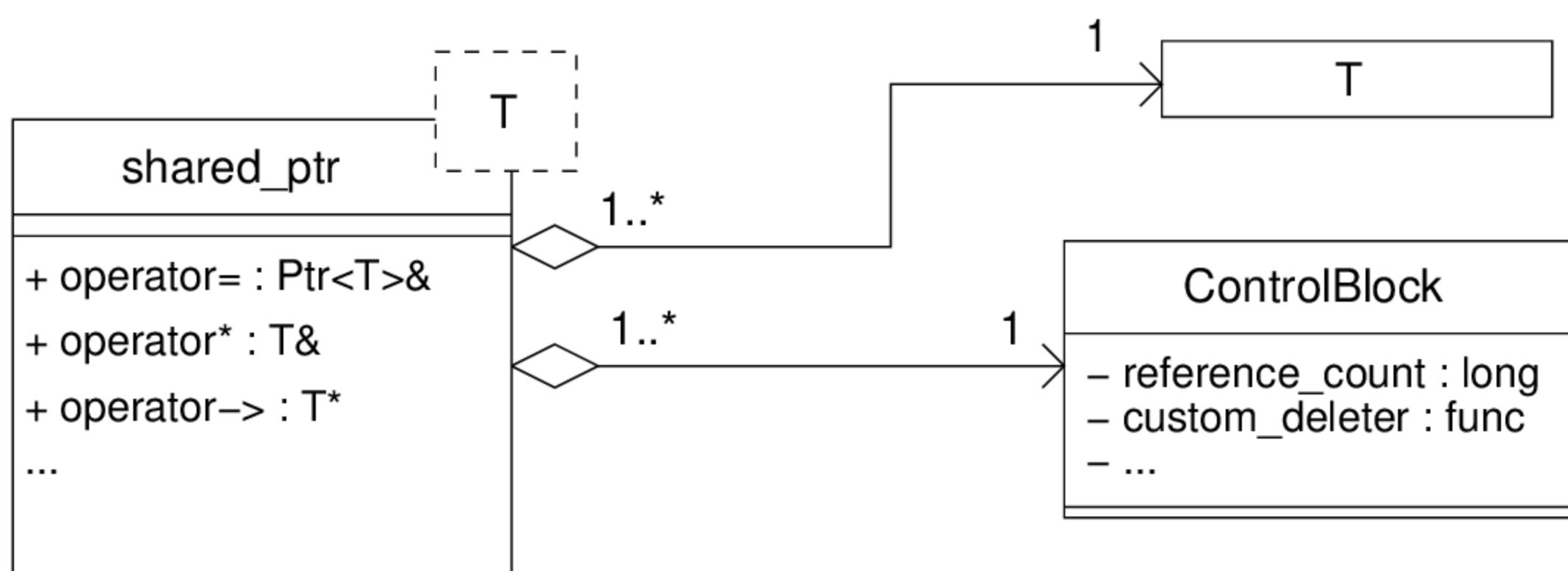


FIGURE 7.5 – Diagramme de classes d'un `shared_ptr`.

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 :

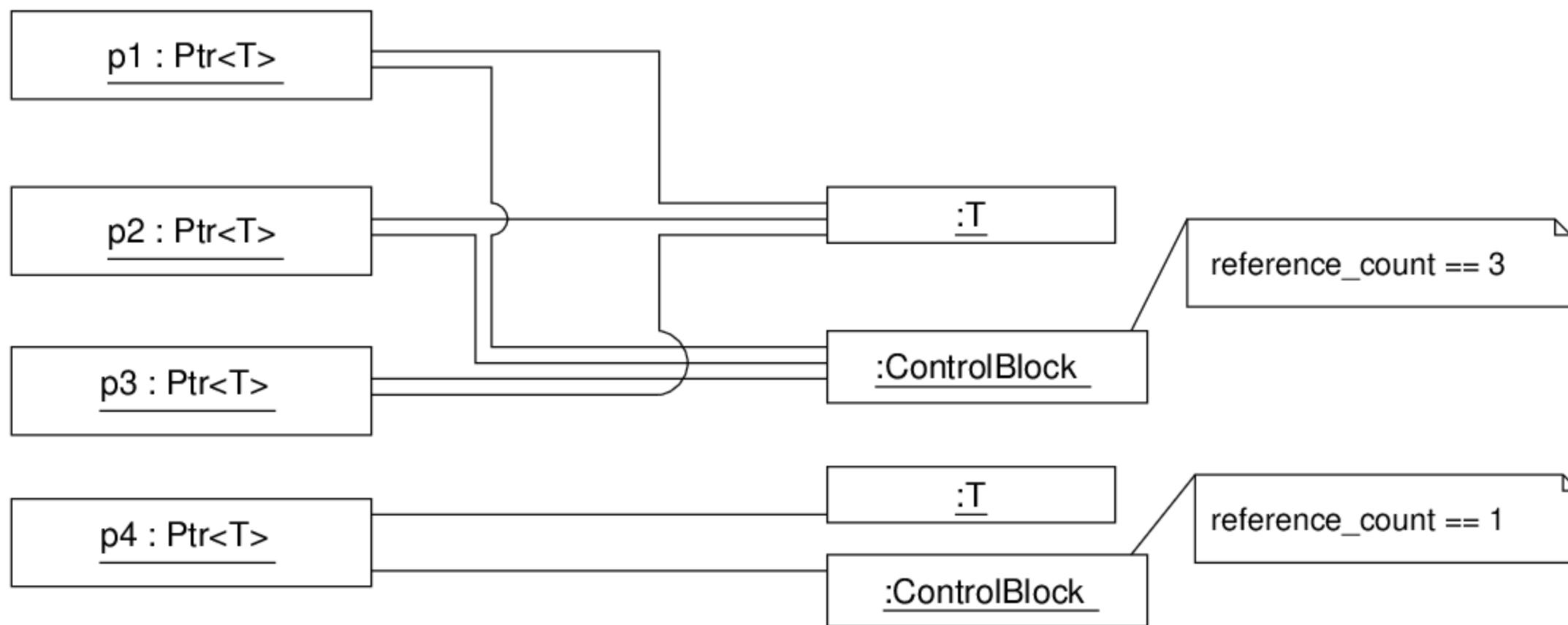
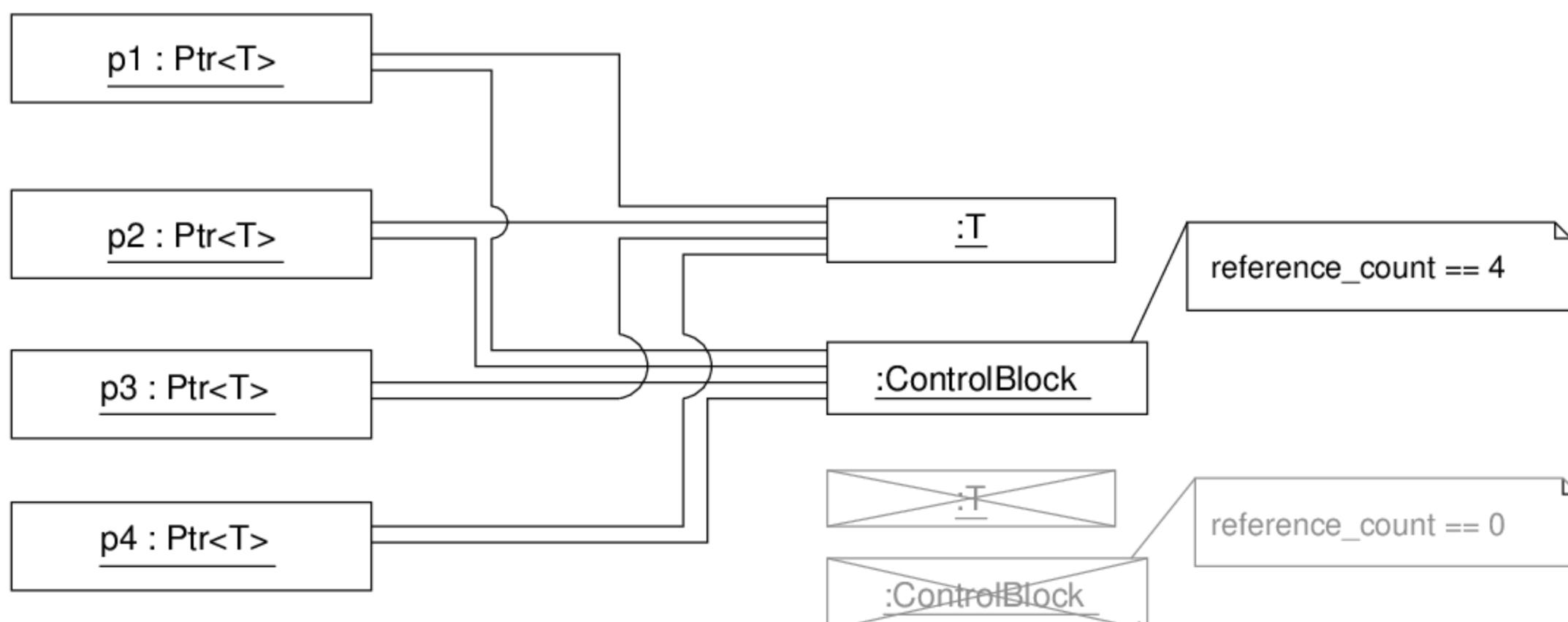


FIGURE 7.6 – Diagramme d'instances mettant en jeu des pointeurs intelligents.

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.

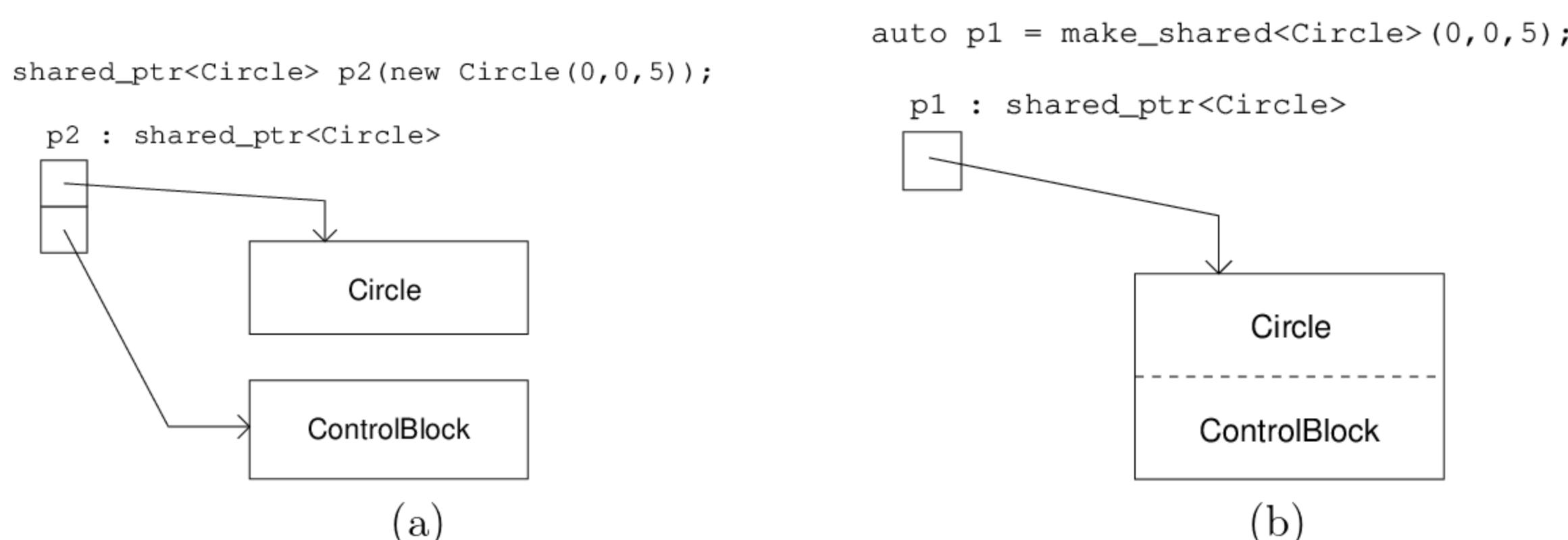


FIGURE 7.8 – Un `shared_ptr` (a) obtenu à partir d'un pointeur brut comparé à (b) celui fourni par `make_shared` en (b).

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...

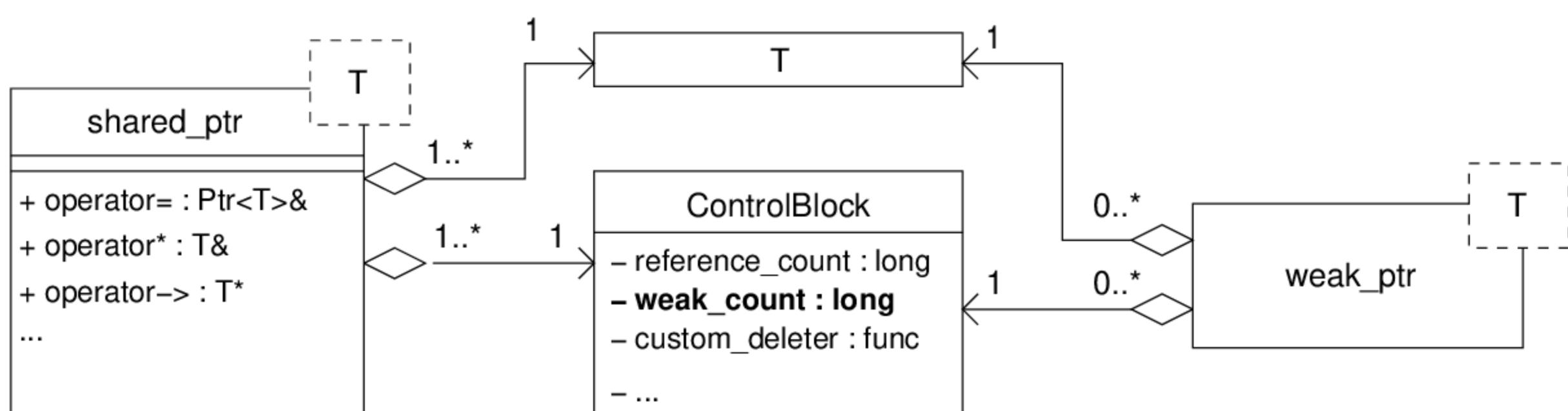


FIGURE 7.9 – Diagramme de classes des `shared_ptr` et `weak_ptr`.

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.

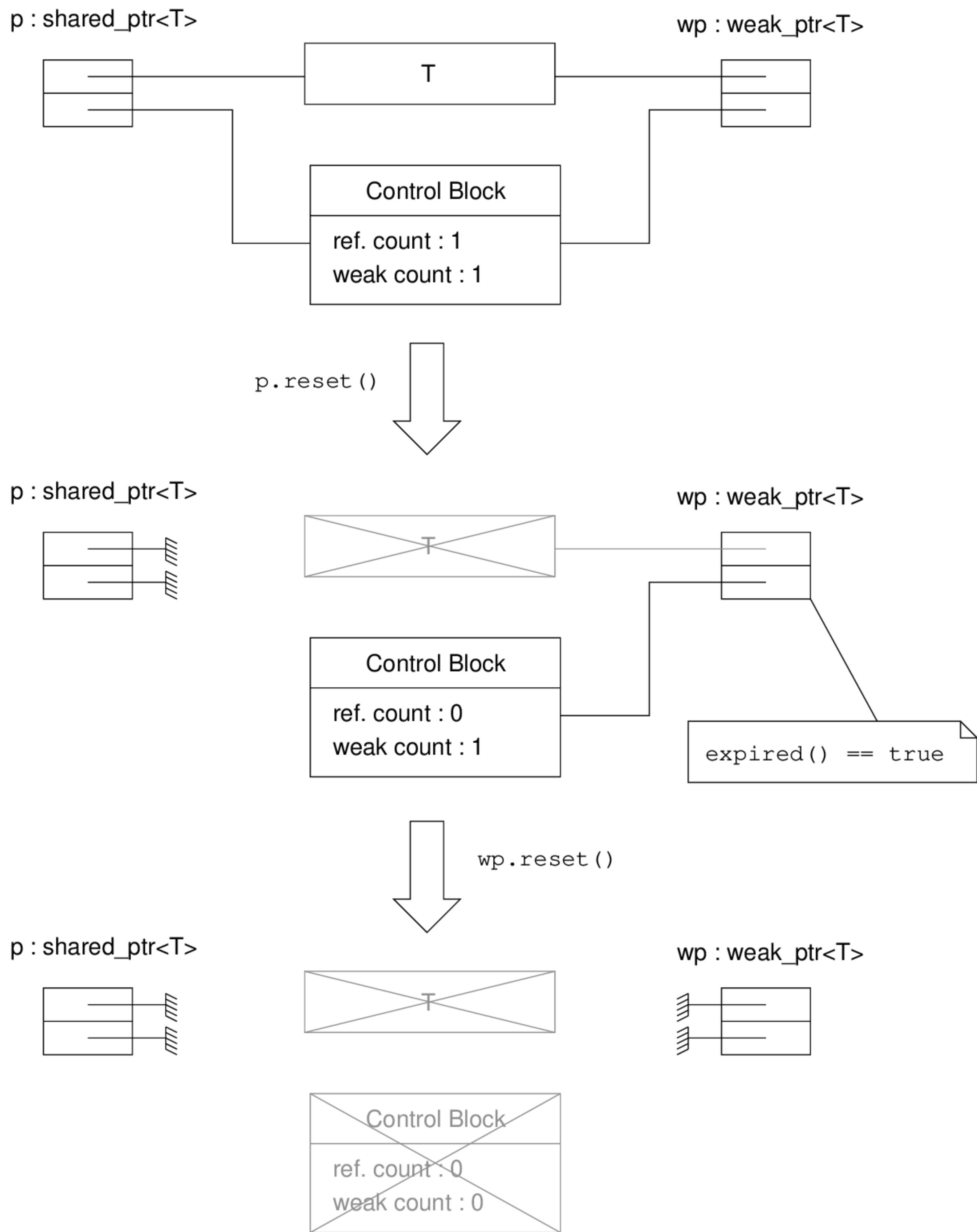
Implémentation

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 }

```

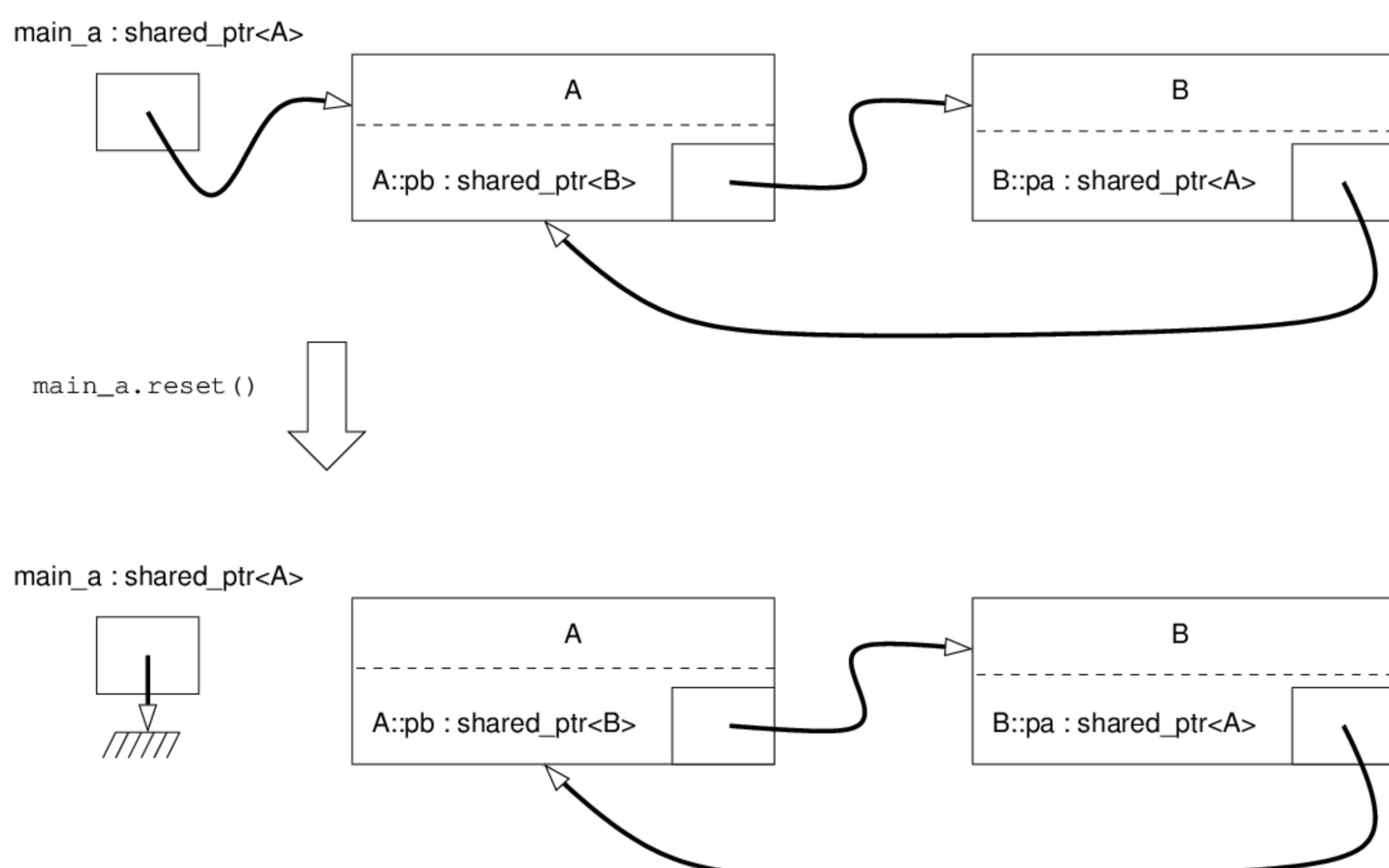
Listing 7.39 – Exemple d'utilisation d'un `weak_ptr`.

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 périmé.
- 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.