Le langage Ada : Types limités et objets contrôlés

Yves Bailly

EN DEUX MOTS **Jusqu'ici nous avons presque toujours déclaré et défini nos enregistrements dans la partie publique des paquetages.**

Cela présente l'avantage de la simplicité, mais l'inconvénient de permettre à l'utilisateur de faire ce qu'il veut avec le contenu de ces structures.

Les déclarations privées, vues dans le sixième article de cette série, ne sont pas toujours suffisantes pour assurer une encapsulation correcte.

Imaginons que nous déclarions un type décrivant un polygone, dont le nombre de points peut varier. Nous devrons très certainement faire appel à la mémoire dynamique. Cela pourrait ressembler à ceci :

```
 1 package Polygones is
 2    type Point is
 3    record
 4       x: Float := 0.0 ;
 5       y: Float := 0.0 ;
 6    end record ;
 7    function To_String(p: in Point) return String ;
 8    type TabPoints is array (Positive range <>) of Point ;
 9    type TabPoints_Ptr is access all TabPoints ;
10    type Polygone is tagged private ;
11    procedure Init(p     : in out Polygone ;
12                   taille: in     Natural) ;
13    procedure Detruire(p: in out Polygone) ;
14    function To_String(p: in Polygone'Class) return string ;
15 private
16    type Polygone is tagged
17    record
18       pts: TabPoints_Ptr ;
19    end record ;
20 end Polygones ;
```

On commence par la déclaration d'un type représentant un point (lignes 2 à 6), suivi d'une fonction facilitant son affichage, puis d'un type tableau de points avec un type pointeur associé. Notre type `Polygone` n'arrive que ligne 10.

Il est déclaré `tagged` pour en faire un type objet et qualifié `private` pour que l'utilisateur ne puisse pas accéder directement à son contenu. Sage précaution dès lors que l'on manipule de la mémoire dynamique, n'est-ce pas ? Pourtant cela ne suffit pas.

Voyez ce programme d'exemple :

```
 1 with Text_IO ; use Text_IO ;
 2 with Polygones ; use Polygones ;
 3 procedure Test_Poly is
 4    p1: Polygone ;
 5    p2: Polygone ;
 6 begin
 7    p1.Init(2) ;
 8    Put_Line(p1.To_String) ;
 9    p2 := p1 ;
10    p2.Detruire ;
11    Put_Line(p1.To_String) ;
12 end Test_Poly ;
```

Comme vous vous en doutez, la procédure `Init()` réserve de l'espace pour le tableau de points du polygone, tandis que la procédure `Detruire()` libère la mémoire précédemment réservée. Compilez et exécutez ce programme :

```
$ gnatmake test_poly && ./test_poly
gcc -c test_poly.adb
gcc -c polygones.adb
gnatbind -x test_poly.ali
gnatlink test_poly.ali
[ ( 0.00000E+00, 0.00000E+00 ) -> ( 0.00000E+00, 0.00000E+00 ) ]

raised CONSTRAINT_ERROR : polygones.adb:47 index check failed
```

Pas de chance : le programme plante par la levée de l'exception `CONSTRAINT_ERROR`. Ce qui était en fait parfaitement prévisible.

Dans le programme, deux polygones sont déclarés lignes 4 et 5. Le premier est initialisé ligne 7 pour contenir deux points, il est correctement affiché ligne 8. Mais que se passe-t-il ligne 9 ?

L'affectation de `p1` à `p2` effectue une copie membre à membre, c'est-à-dire que le membre `pts` de `p2` prend la valeur du membre éponyme de `p1`. Or ce membre est un pointeur : on obtient donc deux pointeurs référençant la même zone de mémoire, celle réservée ligne 7.

Ici Ada ne fait pas mieux que C/C++ : la destruction demandée ligne 10 libère la mémoire préalablement réservée... mais `p1.pts` pointe toujours dessus ! Il est donc logique que la demande d'affichage de `p1` ligne 11 lève une exception, puisque nous tentons d'accéder à une zone mémoire qui a été libérée.

Une telle situation est évidemment intolérable. Prévoir une méthode destinée à effectuer la copie d'un polygone, par exemple une procédure `Copier()`, n'apporte qu'une solution de façade : rien n'oblige l'utilisateur à utiliser cette procédure, rien de l'empêche d'écrire

un code incorrect comme le programme précédent.

Les types limités

C'est là qu'interviennent les types limités. Un type enregistrement qualifié par le mot-clef `limited` interdit explicitement la copie d'une instance dans une autre. Modifions ainsi notre paquetage `Polygones` :

```
  ...
10   type Polygone is tagged limited private ;
  ...
16   type Polygone is tagged limited
17   record
  ...
```

Comme vous pouvez le constater, les modifications sont réellement mineures : simplement l'ajout de `limited` dans la déclaration et la définition du type `Polygone`.

Mais cela suffit à empêcher notre programme de compiler : la ligne 9, qui effectue une copie entre deux instances de `Polygone`, est devenue illégale et donc rejetée par le compilateur. Désormais, une procédure dédiée à la copie d'un polygone dans un autre prend véritablement un sens.

Notez que le caractère limité d'un type est indépendant de son caractère privé : vous pouvez parfaitement déclarer un type limité tout en laissant libre accès à son contenu.

L'exemple suivant est parfaitement légal, sauf naturellement la ligne 10 qui provoquera une erreur de compilation :

```
 1 procedure Test is
 2    type T is limited
 3    record
 4       a: Integer ;
 5    end record ;
 6    t1: T ;
 7    t2: T ;
 8 begin
 9    t1.a := 1 ; -- OK
10    t1 := t2 ;  -- erreur !
11 end Test ;
```

Toutefois, dans la pratique, ce genre de déclarations présente rarement un grand intérêt.

Types contrôlés

Les restrictions imposées par le qualificatif `limited` sont parfois un peu trop gênantes.

Il peut arriver que l'on souhaite autoriser l'affectation entre instances d'un type, sans pour autant abandonner tout contrôle sur l'opération. C'est précisément ce que permettent les types contrôlés, par l'intermédiaire du paquetage `Ada.Finalization`, ainsi que de maîtriser les séquences de création et de destruction des objets.

Il suffit pour cela de faire dériver vos objets du type `Controlled`, un type objet abstrait déclaré dans `Ada.Finalization`. Ensuite surdéfinissez les procédures `Initialize()` (pour la création), `Finalize()` (pour la destruction) et `Adjust()` (pour d'éventuelles actions à l'issue d'une copie). Voici ce que devient alors notre paquetage de polygones :

```
 1 with Ada.Finalization ;
 2 package Polygones is
 3    type Point is
 4    record
 5       x: Float := 0.0 ;
 6       y: Float := 0.0 ;
 7    end record ;
 8    function To_String(p: in Point) return String ;
 9    type TabPoints is array (Positive range <>) of Point ;
10    type TabPoints_Ptr is access all TabPoints ;
11    type Polygone is tagged private ;
12    procedure Init(p     : in out Polygone ;
13                   taille: in     Natural) ;
14    procedure Detruire(p: in out Polygone) ;
15    procedure Def_Point(poly: in out Polygone ;
16                        ind : in     Positive ;
17                        pt  : in     Point) ;
18    procedure Initialize(p: in out Polygone) ;
19    procedure Adjust(p: in out Polygone) ;
20    procedure Finalize(p: in out Polygone) ;
21    function To_String(p: in Polygone'Class) return string ;
22 private
23    type Polygone is new Ada.Finalization.Controlled with
24    record
25       pts: TabPoints_Ptr ;
26    end record ;
27 end Polygones ;
```

Les trois procédures surdéfinies apparaissent lignes 18 à 20, tandis que la ligne 23 réalise la dérivation du type `Polygone` depuis `Controlled`. À titre d'exemple, voici les contenus de ces trois procédures :

```
45    procedure Initialize(p: in out Polygone) is
46    begin
47       Put_Line("Initialize " & Integer_Address'Image(To_
Integer(p'Address))) ;
48    end Initialize ;
```

La procédure `Initialize()` est invoquée lors de la création d'une instance du type `Polygone`. Ce n'est pas équivalent à la procédure `Init()` déclarée plus haut, dont le rôle est plutôt d'initialiser cette instance et qui doit être appelée explicitement.

Pour faire une analogie, on pourrait dire que `Initialize()` est l'équivalent d'un constructeur par défaut en C++. Dans notre situation, on se contente d'afficher un message contenant l'adresse de la variable en cours de création.

Le type `Integer_Address` appartient au paquetage `System.Storage_Elements`, lequel fournit également la fonction `To_Integer()` permettant de transformer une adresse mémoire en un entier affichable.

```
62    procedure Finalize(p: in out Polygone) is
63    begin
64       Put_Line("Finalize " & Integer_Address'Image(To_
Integer(p'Address))) ;
65       if p.pts /= null
66       then
67          Free(p.pts) ;
68       end if ;
69    end Finalize ;
```

À l'inverse, la procédure `Finalize()` est invoquée lors de la destruction d'une instance. Pour poursuivre dans l'analogie, on pourrait l'assimiler à un destructeur en C++.

En plus d'un message, nous prenons soin ici de libérer la mémoire qui avait été réservée pour le polygone en question, évitant ainsi la fuite de mémoire qui existait dans le premier programme de démonstration de cet article.

```
50    procedure Adjust(p: in out Polygone) is
51       ptr: TabPoints_Ptr ;
52    begin
53       Put_Line("Adjust " & Integer_Address'Image(To_
Integer(p'Address))) ;
54       if p.pts /= null
55       then
56          ptr := new TabPoints(p.pts.all'Range) ;
57          ptr.all := p.pts.all ;
58          p.pts := ptr ;
59       end if ;
60    end Adjust ;
```

Enfin, `Adjust()` est invoquée juste après qu'une opération de copie ait eu lieu. En réalité, l'affectation d'une instance dans une autre se déroule en trois étapes :

- 1. L'instance cible de l'affectation (le membre de gauche de l'opérateur `:=`) est finalisée, sans pour autant être détruite : `Finalize()` est invoquée.
- 2. Ensuite les membres sont copiés de la source dans la destination.
- 3. Puis la procédure `Adjust()` est invoquée, afin d'ajuster éventuellement le résultat de la copie.

Dans notre cas, le rôle de `Adjust()` va être d'effectivement dupliquer le tableau de points (lignes 57-58) en réservant la mémoire nécessaire (ligne 56). Voyons tout cela sur un exemple :

```
1 with Text_IO ; use Text_IO ;
2 with Polygones ; use Polygones ;
3 procedure Test_Fin_Poly is
4    p1: Polygone ;
5    p2: Polygone ;
6 begin
7    Put_Line("--- Début") ;
8    p1.Init(2) ;
9    p1.Def_Point(1, (1.0, 2.0)) ;
10   p1.Def_Point(2, (11.0, 22.0)) ;
11   Put_Line(p1.To_String) ;
12   Put_Line(p2.To_String) ;
13   Put_Line("--- p2 := p1...") ;
14   p2 := p1 ;
15   Put_Line("--- p1.Detruire...") ;
16   p1.Detruire ;
17   Put_Line(p1.To_String) ;
18   Put_Line(p2.To_String) ;
19   Put_Line("--- Fin") ;
20 end Test_Fin_Poly ;
```

Voici ce que cela donne à l'exécution :

```
$ ./test_fin_poly
Initialize  3213236128
Initialize  3213236108
--- Début
[ ( 1.00000E+00, 2.00000E+00 ) -> ( 1.10000E+01, 2.20000E+01 ) ]
[]
--- p2 := p1...
Finalize  3213236108
Adjust  3213236108
--- p1.Detruire...
[]
[ ( 1.00000E+00, 2.00000E+00 ) -> ( 1.10000E+01, 2.20000E+01 ) ]
--- Fin
Finalize  3213236108
Finalize  3213236128
```

Vous pouvez constater que la création des instances `p1` et `p2` a lieu avant le début du programme. L'affectation de `p1` dans `p2` donne bien lieu à la finalisation, puis à l'ajustement de cette dernière. Enfin les « destructeurs » sont invoqués après la dernière instruction du programme. Signalons pour terminer qu'il existe également dans `Ada.Finalization` le type `Limited_Controlled`, offrant les procédures `Initialize()` et `Finalize()` pour les types limités. Par contre `Adjust()` n'est pas disponible, les types limités interdisant les copies entre eux, cette procédure est sans objet.

Conclusion

Voilà pour cette présentation des types limités et contrôlés. La prochaine fois, nous aborderons les facilités offertes par Ada pour s'interfacer avec d'autres langages de programmation, notamment les langages C et C++, mais également ce bon vieux Fortran.

Yves Bailly,