Langage Ada 95 – 9 : Mémoire et pointeurs

par Yves Bailly

EN DEUX MOTS **Les possibilités offertes par le langage Ada pour manipuler la mémoire et les données qui s'y trouvent sont comparables à celles que l'on trouve en C/C++. Mais ces techniques sont proposées à l'aune des exigences de robustesse et de précision du langage, ce qui donne une approche assez originale.**

L'originalité apparaîtra notamment dans la manipulation des adresses mémoire. Mais ce qui est sans doute le plus agréable dans tout cela est qu'Ada ne fait pas appel à quelque symbole particulier présent sur le clavier, évitant ainsi les constructions syntaxiques parfois obscures si communes en C/C++.

Les types access

Voyons pour débuter un petit exemple élémentaire :

```
 1 with Text_IO ; use Text_IO ;
 2 procedure P1 is
 3    type P_Int is access all Integer ;
 4    a : aliased Integer := 1 ;
 5    pa: P_Int ;
 6 begin
 7    pa := a'Access ;
 8    pa.all := 2 ;
 9    Put_Line("a = " & Integer'Image(a)) ;
10 end P1 ;
```

Pour commencer, on déclare un type nommé `P_Int`, de type `access all Integer`, c'est-à-dire un type permettant d'accéder à des entiers. C'est ce que l'on appelle un « type access » ou encore un « type pointeur ». Disons dès maintenant que ce terme « pointeur » doit être pris avec précaution : contrairement aux langages C/C++, il ne s'agit pas forcément d'une adresse mémoire.

Il est possible que la représentation interne d'une instance de `P_Int` soit effectivement une adresse mémoire, mais cela n'a rien d'obligatoire, du moins si on se réfère à la lettre du standard Ada. Donc attention aux abus de langage. Ceci dit, en pratique, les compilateurs représentent généralement les types `access` par des adresses mémoire des plus conventionnelles, ce qui est en particulier le cas du compilateur Gnat.

Continuons. Ligne 4 est déclarée et initialisée une variable de type `Integer`, nantie du qualificatif `aliased`. Cela indique que cette variable pourra éventuellement être référencée par une autre variable d'un type `access` approprié.

En effet, par défaut en Ada, les variables ne sont pas « pointables », c'est-à-dire qu'il n'est normalement pas possible de référencer les données qu'une variable représente par un autre moyen qu'elle-même (à moins de passer par les adresses mémoire, ce que nous verrons plus loin). L'ajout de `aliased` dans la déclaration lève cette restriction et permet d'obtenir un pointeur (une instance d'un type `access`) sur cette variable.

C'est précisément ce que nous faisons ligne 7 : l'attribut `'Access` appliqué à une variable donne justement une instance d'un type `access` sur cette variable, ici stockée dans une variable `pa` de type `P_Int`. Donc `pa` « pointe » sur `a`, `pa` est un accès aux données contenues par `a`.

Le déréférencement d'un pointeur pour manipuler effectivement les données se fait en ajoutant `.all` au pointeur. La ligne 8 consiste donc à placer la valeur 2 dans la variable pointée par `pa`. L'affichage suivant permet de vérifier que l'on a bien indirectement modifié la valeur de `a` :

```
$ gnatmake p1 && ./p1
a =  2
```

Si on voulait écrire un programme C++ équivalent, voici à quoi il pourrait ressembler, en gardant bien à l'esprit que la nature des pointeurs en Ada n'est pas nécessairement équivalente à la nature des pointeurs en C++ :

```
 1 #include <iostream>
 2 int main(int argc, char* argv[])
 3 {
 4    typedef int* P_Int ;                    // 3
 5    int a = 1 ;                             // 4
 6    P_Int pa = &a ;                         // 5 et 7
 7    *pa = 2 ;                               // 8
 8    std::cout << "a = " << *pa << "\n" ;    // 9
 9    return 0 ;
10 }
```

Les commentaires en bout de ligne donnent la ligne correspondante dans le programme Ada. La ligne 6 regroupe déclaration et initialisation de la variable pointeur, ce qui aurait également pu être fait en Ada.

Allocation dynamique

Voyons maintenant comment réserver dynamiquement de la mémoire. Cela est fait au moyen du mot-clef `new`, qui n'est pas sans rappeler l'opérateur éponyme du C++ :

```
1 with Text_IO ; use Text_IO ;
2 procedure P2 is
3    type P_Int is access all Integer ;
4    pa: P_Int ;
5 begin
```

```
 6    pa := new Integer ;
 7    pa.all := 2 ;
 8    Put_Line("pa.all = " & Integer'Image(pa.all)) ;
 9    pa := new Integer'(3) ;
10    Put_Line("pa.all = " & Integer'Image(pa.all)) ;
11 end P2 ;
```

Reprenant le type précédent, une variable de type access sur Integer est déclarée ligne 4. À noter que contrairement aux types « normaux », les variables de types access sont automatiquement initialisées à la valeur null, un peu comme si tous les pointeurs déclarés dans un programme C++ étaient automatiquement et systématiquement initialisés à la valeur 0. Il va sans dire que tenter de déréférencer un pointeur ayant pour valeur null résulte en la levée immédiate d'une exception.

L'allocation dynamique est effectuée ligne 6, tout simplement avec le mot-clef new suivi du type voulu. La ligne 9 montre comment combiner allocation et initialisation : la valeur que l'on souhaite affecter doit être donnée sous la forme d'un agrégat (entre parenthèses) séparé du nom du type par une apostrophe, comme s'il s'agissait d'un attribut.

Le lecteur attentif aura remarqué que, d'une part, ce programme ne libère pas la mémoire qu'il réserve, d'autre part, que la mémoire réservée ligne 6 est définitivement perdue à la suite de l'allocation ligne 9, créant une fuite de mémoire. Nous verrons un peu plus bas comment libérer la mémoire réservée.

Pour ce qui est de la fuite de mémoire, sachez que le standard Ada autorise les implémentations (les compilateurs) à mettre en place un mécanisme de ramasse-miettes (*garbage collector*) qui libère automatiquement la mémoire réservée, mais qui n'est plus utilisé, mécanisme que l'on trouve dans des langages comme Python ou Java.

Mais ce n'est qu'une autorisation, pas une obligation. En l'occurrence, le compilateur Gnat n'implémente pas de telle technique. Si vous n'y prenez garde, un programme Ada peu donc « perdre » autant de mémoire que n'importe quel programme C.

Bon ! Réserver la mémoire pour un simple entier, c'est bien, mais ce n'est pas très intéressant. L'allocation dynamique est plus communément employée dans le cas de tableaux non contraints, par exemple :

```
 1 with Text_IO ; use Text_IO ;
 2 procedure P3 is
 3    type Tableau is
 4       array (Positive range <>) of Integer ;
 5    type Tableau_Ptr is access all Tableau ;
 6    procedure Put_Line(t: in Tableau_Ptr) is
 7    begin
 8       Put("(") ;
 9       for i in t'Range
10       loop
11          Put(Integer'Image(t(i))) ;
12       end loop ;
13       Put_Line(")") ;
14    end Put_Line ;
15    pt: Tableau_Ptr ;
16 begin
17    pt := new Tableau(1..10) ;
18    pt.all := (others => 1) ;
19    Put_Line(pt) ;
20    pt := new Tableau'(2..11 => 2) ;
21    Put_Line(pt) ;
22 end P3 ;
```

Le type Tableau déclaré lignes 3-4 est un tableau non contraint d'entiers indexés par des entiers strictement positifs. Comme nous l'avons vu dans un article précédent, il est normalement impossible de déclarer une variable de ce type, sans donner explicitement une limite aux bornes. Ligne 5, on déclare un type pointeur sur ce type Tableau.

Passons pour l'instant sur la procédure d'affichage Put_Line() et regardons l'allocation d'une variable pt, de type pointeur sur notre type Tableau, ligne 17.

Il est obligatoire de donner une contrainte pour pouvoir réserver la mémoire associée : l'écriture ressemble assez à ce qui serait utilisé pour déclarer une variable. La ligne 18 a pour effet d'initialiser les valeurs contenues dans le tableau à 1, au moyen d'un agrégat – remarquez le .all ajouté à la suite de la variable pointeur.

L'initialisation peut se faire au moment de la réservation, comme montré ligne 20. On ne donne plus les bornes du tableau attachées au type, mais dans l'agrégat qui suit l'apostrophe : c'est lui qui fixe les limites. Il y a là une petite subtilité dans l'écriture. Prenez garde toutefois à cette forme d'initialisation des tableaux dynamiques : l'agrégat est en fait créé sur la pile, ce qui peut être problématique si le tableau est de grande taille (essayez en remplaçant le 11 par 11_000_000).

Jetons maintenant un œil à la procédure d'affichage, lignes 6 à 14. Elle prend en paramètre une instance de Tableau_Ptr, donc un pointeur sur un tableau non contraint. Remarquez que le tableau est utilisé « normalement », comme si le paramètre t était du type tableau : le suffixe .all est facultatif dans ces situations.

Il y a toutefois une différence de taille : le paramètre étant un type access, bien qu'il soit passé en mode in il est parfaitement possible de modifier le contenu du tableau au sein de la procédure. Ce qui n'est pas modifiable est la valeur du pointeur.

Par ailleurs, répétons-le, un type access n'est pas forcément une adresse mémoire. Cela signifie en pratique que l'arithmétique sur les pointeurs si commune en C/C++ n'existe tout simplement pas en Ada, du moins avec ces types. Par exemple, dans le programme précédent, une instruction comme pt :=

`pt + 1 ;` sera refusée par le compilateur. Cela ne signifie pas pour autant qu'il est impossible d'obtenir un accès aux données individuelles contenues dans le tableau : il suffit d'ajouter le qualificatif `aliased` dans la déclaration du type tableau. Voyez par exemple ce morceau de code :

```
1 -- ...
2    type P_Int is access all Integer ;
3    type Tableau is
4       array(Positive range <>) of aliased Integer ;
5 -- ...
6    t  : Tableau(1..10) ;
7    p_i: P_Int ;
8 -- ...
9    p_i := t(5)'Access ;
```

Remarquez le `aliased` dans la définition du type `tableau`, ligne 4. Cela nous permet d'appliquer l'attribut `'Access` à un élément du tableau. Il faut bien comprendre qu'obtenir un accès au premier élément du tableau, par exemple avec `t(t'First)'Access`, n'est pas équivalent à obtenir un accès sur le tableau lui-même, qui s'obtiendrait avec `t'Access` (ce qui ne fonctionnerait pas dans l'exemple, car la variable `t` n'a pas été qualifiée par `aliased`, ligne 6). Si par hasard cela n'avait pas déjà été précisé, un type `access` n'est pas un type pointeur au sens du C/C++.

Libérer la mémoire allouée

Ada permet une gestion sophistiquée de la réservation de mémoire. Cette sophistication a un prix : une certaine complexité pour effectuer des tâches apparemment simples, comme la libération de la mémoire réservée. Celle-ci s'effectue au moyen de l'instanciation d'une procédure générique du paquetage `Ada`, nommément `Ada.Unchecked_Deallocation()`. Par exemple :

```
1 with Text_IO ; use Text_IO ;
2 with Ada.Unchecked_Deallocation ;
3 procedure P5 is
4    type Tableau is
5       array (Positive range <>) of Integer ;
6    type Tableau_Ptr is access all Tableau ;
7    procedure Free is
8       new Ada.Unchecked_Deallocation(Object => Tableau,
9                                      Name   => Tableau_Ptr) ;
10   pt: Tableau_Ptr ;
11 begin
12   Free(pt) ;
13   pt := new Tableau(1..10_000_000) ;
14   if pt /= null
15   then
16      Put_Line("Initialisation...") ;
17      pt.all := (others => 0) ;
18   end if ;
19   Free(pt) ;
20   if pt = null
21   then
22      Put_Line("Mémoire libérée.") ;
23   end if ;
24 end P5 ;
```

On reprend le type tableau et le type `access` associé du troisième exemple de cet article, lignes 4 à 6. Puis, on crée une instance de `Ada.Unchecked_Deallocation()` spécialisée pour ces deux types, lignes 7-9. Remarquez la clause `with` ligne 2, nécessaire pour pouvoir effectuer cette instanciation. Le premier paramètre générique, nommé `Object`, doit recevoir le type pour lequel on souhaite réaliser la libération de mémoire.

Le deuxième, `Name`, est le type pointeur associé. Cette longue instruction nous crée une nouvelle procédure, nommée conventionnellement `Free()` (mais cela n'a rien d'obligatoire), capable de libérer la mémoire réservée pour une instance du type `Tableau` et « pointée » par une instance du type `Tableau_Ptr`. La ligne 12 a pour seul but de montrer que l'invocation de notre procédure de libération en lui passant un pointeur nul (les types `access` sont automatiquement initialisés à la valeur `null`) est sans conséquence.

Puis nous réservons un tableau de 10 millions d'entiers ligne 13. Dès lors la variable `pt` n'est plus nulle : si l'allocation a réussi, le test ligne 14 est vrai et le contenu du tableau est initialisé ligne 17. Cette mémoire réservée est libérée ligne 19. Conséquence intéressante, la variable pointeur `pt` est alors automatiquement remise à la valeur `null`, contrairement à ce qui ce passe en C/C++ : le test ligne 20 est donc toujours vrai. L'affichage du programme est donc :

```
$ gnatmake p5 && ./p5
Initialisation...
Mémoire libérée.
```

S'il paraît bien compliqué, ce mécanisme offre malgré tout l'avantage d'être un peu mieux « sécurisé » que l'équivalent en C/C++ : une variable pointeur est par défaut initialisée à `null` et retrouve cette valeur lorsqu'elle est libérée. Mais il n'y a pas de miracle : il est toujours possible de provoquer un plantage simplement en dupliquant le pointeur, puis en libérant l'un, tout en utilisant le second.

Voyez cet extrait :

```
10   pt : Tableau_Ptr ;
11   pt2: Tableau_Ptr ;
12 begin
13   pt := new Tableau(1..10) ;
14   pt2 := pt ;
15   Free(pt) ;
16   if pt2 /= null
17   then
18      pt2.all := (others => 0) ; -- crash
19   end if ;
```

Le pointeur vers la mémoire réservée ligne 13 est dupliqué ligne 14, le plus simplement du monde. Le premier pointeur est libéré ligne 15 et vaut donc `null` à partir de ce point, mais la valeur du deuxième n'a pas changé : le test ligne 16 est vrai.

On tente alors d'initialiser (ligne 18) la mémoire réservée puis libérée... ce qui provoque un crash du programme. Ou plus précisément, cela provoque la levée d'une exception nommée `Storage_Error` : il est ainsi théoriquement possible d'intercepter le problème et d'éviter le crash. Une telle réaction est tout simplement impossible en C/C++.

Pointeurs sur sous-programmes

Il est évidemment possible en Ada d'obtenir un pointeur sur un sous-programme (procédure ou fonction) et de le placer dans une variable pour utilisation ultérieure. Voici un exemple :

```
 1 with Text_IO ; use Text_IO ;
 2 procedure P8 is
 3    procedure Pr_1(int: in Integer) is
 4    begin
 5       Put_Line("(1) " & Integer'Image(int)) ;
 6    end Pr_1 ;
 7    procedure Pr_2(int: in Integer) is
 8    begin
 9       Put_Line("(2) " & Integer'Image(int)) ;
10    end Pr_2 ;
11    function F_1 return Integer is
12    begin
13       return 11 ;
14    end F_1 ;
15    function F_2 return Integer is
16    begin
17       return 22 ;
18    end F_2 ;
```

Deux procédures de même signature sont déclarées, pour afficher de deux manières différentes l'entier qu'elles reçoivent en paramètre. Également deux fonctions similaires, sans paramètre et retournant un entier.

```
20    type Pr_Ptr is access procedure(a: in Integer) ;
21    type F_Ptr is access function return Integer ;
```

Ces deux instructions déclarent deux types pointeur sur sous-programmes. Ces types sont construits très simplement : il suffit d'écrire la signature voulue, sans donner de nom de sous-programme.

```
23    proc_ptr: Pr_Ptr ;
24    func_ptr: F_Ptr ;
25 begin
26    proc_ptr := Pr_1'Access ;
27    func_ptr := F_1'Access ;
28    proc_ptr(func_ptr.all) ;
29    func_ptr := F_2'Access ;
30    proc_ptr(func_ptr.all) ;
31    proc_ptr := Pr_2'Access ;
32    proc_ptr(func_ptr.all) ;
33 end P8 ;
```

Simple exemple d'utilisation. L'attribut `'Access` s'applique également aux sous-programmes. Les lignes 28, 30 et 32 ont pour effet d'invoquer la procédure pointée par `proc_ptr`, en lui passant en paramètre le résultat de la fonction pointée par `func_ptr`. Petite subtilité, le suffixe .all est facultatif dans le cas du pointeur sur procédure du fait de la présence de paramètres ; par contre, il est indispensable pour le pointeur sur fonction, car la signature donnée ligne 21 ne prend aucun paramètre.

Le résultat :

```
$ gnatmake p8 && ./p8
(1) 11
(1) 22
(2) 22
```

D'une manière générale, le suffixe .all n'est pas nécessaire quand l'entité pointée possède des éléments « complémentaires ». Par exemple, si on a les déclarations suivantes :

```
type Struct is
record
   a: Integer ;
   b: Float ;
end record ;
type Struct_Ptr is access all Struct ;
-- ...
ptr: Struct_Ptr := new Struct'(1, 1.1) ;
```

...alors le premier élément de l'enregistrement peut être référencé simplement par `ptr.a`, l'écriture `ptr.all` représentant l'enregistrement pris dans sa globalité.

Le retour des adresses

Peut-être est-il temps de rassurer ceux qui tiennent absolument à manipuler les adresses mémoire : celles-ci n'ont pas disparu, Ada est parfaitement capable de les utiliser. L'adresse d'un élément de mémoire, le plus souvent un octet (mais cela dépend du matériel sous-jacent), est représentée par le type `Address` du paquetage `System`. Grossièrement, on peut dire que ce type est l'équivalent du type `void*` en C/C++ : cela représente véritablement une adresse mémoire, et rien de plus – notamment, une valeur de type `System.Address` ne « contient » aucune information sur le type de donnée figurant à cette adresse. On ne peut donc pas vraiment le comparer, par exemple, au type `char*`. Ce paquetage `System` contient une foule d'informations liées au système sur lequel fonctionne le programme, comme les plus grands et plus petits entiers, la précision des nombres à virgule flottante, etc. Pour ce qui est de la gestion de la mémoire, on trouve notamment (Tableau I)

Les valeurs indiquées sont celles obtenues sur un processeur de type x86 à partir du 80386. D'autres matériels peuvent donner des résultats différents. Une arithmétique minimaliste est possible sur les adresses mémoire au moyen d'opérateurs + et - définis dans le paquetage `System.Storage_Elements`. Ces opérateurs combinent des valeurs de type `System.Address` et de type `System.Storage_Elements.Storage_Offset`, ce dernier représentant un décalage en mémoire, pour produire des valeurs de type `System.Address` – sauf la différence de deux adresses, qui donne un décalage. Notez bien que l'on parle d'arithmétique sur adresses

Éléments dans le paquetage System		
Nom	Description	Valeur
Null_Address	L'adresse nulle	0
Storage_Unit	Nombre de bits par élément de mémoire	8
Word_Size	Nombre de bits par mot mémoire	32
Memory_Size	Taille de la mémoire, ce qui peut revêtir diverses interprétations. Le plus souvent, il s'agira de la mémoire adressable.	4294967296
type Bit_Order is (High_Order_First, Low_Order_First);	Type décrivant l'ordre des bits, soit respectivement *big endian* ou *little endian*.	
Default_Bit_Order	Ordre naturel des bits du matériel.	Low_Order_First

Tableau 1

mémoire, par sur pointeurs : ces deux notions sont bien distinctes en Ada, alors qu'elles sont confondues dans de nombreux autres langages. Il est toutefois possible de convertir des adresses en pointeurs (c'est-à-dire, en types `access`) par l'utilisation de fonctions du paquetage générique `System.Address_To_Access_Conversions`, le paramètre générique étant le type de base dont on veut manipuler des adresses ou des pointeurs. Normalement, les habitués des manipulations de haut vol sur les pointeurs en C/C++ doivent terminer de pâlir, tout cela devant leur paraître d'une absurde complexité. Voyons un exemple, dont l'objet est de modifier une valeur dans un tableau en passant par les adresses.

```
1 with Text_IO ;
2 use  Text_IO ;
3 with System ;
4 use  System ;
5 with System.Storage_Elements ;
6 use  System.Storage_Elements ;
7 with System.Address_To_Access_Conversions ;
```

Pour commencer, les paquetages que nous allons utiliser. Oui, l'opération apparemment simple que nous allons effectuer nécessite tout cela...

```
 8 procedure P7 is
 9    package A2A_Conv is
10       new System.Address_To_Access_Conversions(Integer) ;
11    use A2A_Conv ;
```

On commence par instancier le paquetage générique `System.Address_To_Access_Conversions`, afin de pouvoir effectuer des conversions entre des adresses mémoire et des pointeurs sur des valeurs de type `Integer`. Au fait, avons-nous déjà précisé qu'en Ada, la notion de « pointeur » n'est pas forcément équivalente à celle d'« adresse mémoire » ? Continuons...

```
12    type P_Int is access all Integer ;
13    type Tableau_1 is array(1..10) of Integer ;
14    pragma Convention(C, Tableau_1) ;
15    type Tableau_2 is array(1..10) of aliased Integer ;
16    pragma Convention(C, Tableau_2) ;
```

Voici justement définis un type pointeur sur entiers et deux types tableau. Remarquez le `aliased` ligne 15 : cela nous permettra d'obtenir un pointeur sur un élément du tableau. Les clauses `pragma` lignes 14 et 16, qui sont définies par le langage, indiquent que les types `Tableau_1` et `Tableau_2` doivent respecter les conventions du langage C pour ce qui est de leur représentation interne. Cela empêche le compilateur de prendre quelques libertés avec la manière dont les données sont agencées en mémoire. Ainsi, nous aurons des résultats prévisibles.

```
17    int_adr: Address ;
18    int_ptr: P_Int ;
19    t1: Tableau_1 := (others => 1) ;
20    t2: Tableau_2 := (others => 2) ;
21    t3: Tableau_1 := (others => 3) ;
22    a : Integer ;
23    for a'Address use t3(3)'Address ;
```

Déclaration des variables que nous allons utiliser. Le type `Address` ligne 17 n'est pas préfixé par le nom du paquetage `System` grâce à la clause `use` ligne 4. Trois tableaux sont créés, le premier ne contenant que des 1, le deuxième que des 2, le troisième que des 3. La ligne 22 déclare une variable de type `Integer` apparemment des plus normales. Mais la ligne suivante apporte une précision importante : c'est une clause de localisation. Elle indique que l'adresse en mémoire de la variable `a` (donnée par `a'Address`) doit être (`use`) la valeur `t3(3)'Address`, c'est-à-dire l'adresse de l'élément d'indice 3 dans le tableau `t3`. Par chance, `a` et `t3(3)` sont du même type `Integer`, mais cela n'a rien d'obligatoire : on peut ainsi « forcer » l'adresse d'une variable à l'adresse de n'importe quelle entité du programme, par exemple l'adresse de la procédure principale `P7()`, ce qui pourrait donner des résultats curieux... La seule contrainte est que l'adresse puisse être connue à la compilation. Cela peut même être une valeur numérique, comme 16#BFF83388# (adresse 32bits donnée sous forme hexadécimale).

```
24 begin
25    --
26    Put_Line("t1(3) = " & Integer'Image(t1(3))) ;
27    int_adr := t1(1)'Address ;
```

```
28    int_adr := int_adr + (t1'Component_Size/Storage_Unit)*2 ;
29    int_ptr := P_Int(To_Pointer(int_adr)) ;
30    int_ptr.all := 11 ;
31    Put_Line("t1(3) = " & Integer'Image(t1(3))) ;
```

Première manipulation. D'abord est affichée la valeur du troisième élément de `t1`, pour référence. Ligne 27, on stocke dans `int_adr` l'adresse du premier élément de `t1` : l'attribut `'Address` retourne l'adresse mémoire de n'importe quelle entité du programme. La ligne suivante modifie cette adresse ainsi :

- L'attribut `'Component_Size`, qui ne s'applique qu'aux tableaux, donne la taille d'un élément du tableau en bits (et non pas en octets) ;
- Cette valeur est divisée par `Storage_Unit` (issue de `System`) pour obtenir le nombre d'éléments de mémoire correspondants, typiquement le nombre d'octets ;
- Le résultat est doublé, pour obtenir un décalage en mémoire correspondant à la taille de deux éléments du tableau – cette correspondance n'étant garantie que par la clause `pragma` ligne 14 ;
- Ce décalage est ajouté à `int_adr`, l'opérateur `"+"` venant du paquetage `System.Storage_Elements`.

À l'issu de la ligne 28, `int_adr` contient donc (normalement) l'adresse du troisième élément du tableau `t1`. Notez que toutes ces manipulations passent outre les sécurités propres à Ada sur les bornes des tableaux. L'adresse `int_adr` est ensuite convertie en un pointeur (ligne 29), lequel est stocké dans `int_ptr`. La fonction `To_Pointer()` est issue de l'instanciation effectuée lignes 9-10, rendue accessible sans préfixe grâce à la clause `use` ligne 11. Les raisons du transtypage en `P_Int` apparaîtront dans la manipulation suivante. Enfin, la valeur pointée par `int_ptr` est modifiée (ligne 30) et on affiche à nouveau la valeur du troisième élément de `t1`. L'affichage résultant de l'exécution des lignes 26 à 31 est :

```
t1(3) =  1
t1(3) =  11
```

Où l'on vérifie bien que l'élément `t1(3)` a été modifié, sans pour autant y faire référence. Continuons.

```
32    --
33    Put_Line("t2(3) = " & Integer'Image(t2(3))) ;
34    int_ptr := t2(1)'Access ;
35    int_adr := To_Address(Object_Pointer(int_ptr)) ;
36    int_adr := int_adr + (t2'Component_Size/Storage_Unit)*2 ;
37    int_ptr := P_Int(To_Pointer(int_adr)) ;
38    int_ptr.all := 22 ;
39    Put_Line("t2(3) = " & Integer'Image(t2(3))) ;
```

On part cette fois d'un pointeur sur le premier élément du tableau `t2`, obtenu ligne 34, ce qui n'est possible que par la présence de `aliased` dans la déclaration du type `tableau` ligne 15. Ce pointeur est transformé en adresse mémoire au moyen de la fonction `To_Adress()` présente dans `System.Address_To_Access_Conversions`. Remarquez le transtypage de `int_ptr`, de type `P_Int` en un type `Object_Pointer`. Ce type est également issu du paquetage de conversion, il est simplement défini comme étant un type `access` sur le type générique passé lors de l'instanciation. `Object_Pointer` et `P_Int` sont donc tous deux du « genre » `access all Integer`, mais comme ils ont été définis séparément et que Ada n'effectue (presque) aucune conversion automatique entre types, ce transtypage est nécessaire. C'est également la raison du transtypage ligne 29 évoqué plus haut : `To_Pointer()` retourne un `Object_Pointer`. Les lignes suivantes sont similaires à celles que nous avons déjà étudiées. L'affichage donne :

```
t2(3) =  2
t2(3) =  22
```

Ce qui est bien le résultat attendu. Voici enfin le dernier exemple, qui par rapport aux précédents confine à la trivialité :

```
40    --
41    Put_Line("t3(3) = " & Integer'Image(t3(3))) ;
42    a := 33 ;
43    Put_Line("t3(3) = " & Integer'Image(t3(3))) ;
44 end P7 ;
```

Rappelez-vous, la variable `a` déclarée ligne 22 a été « ancrée » à l'adresse occupée par le troisième élément du tableau `t3` (ligne 23). Logiquement, modifier l'une est équivalent à modifier l'autre : `a` et `t3(3)` désignent véritablement la même zone mémoire, sans pour autant avoir recours à un quelconque type pointeur. Comme on peut s'y attendre, l'affichage est :

```
t3(3) =  3
t3(3) =  33
```

Ouf ! Tout cela peut vous paraître épouvantablement compliqué, mais avec un peu de pratique cela vient assez naturellement. Cette approche des manipulations d'adresses mémoire reflète bien les principes fondamentaux qui ont régi la conception du langage Ada, notamment qu'il doit rester souple (on peut faire vraiment n'importe quoi) tout en offrant un maximum de sécurités.

Conclusion

Voilà pour ces quelques mots concernant la gestion de la mémoire en Ada. Bien que relativement longue, cette présentation passe sous silence deux aspects assez importants : le mode de passage de paramètres `access` aux sous-programmes, les possibilités offertes pour maîtriser soi-même l'espace de stockage des données (ce que l'on désigne par le terme de *storage pools* en anglais) et les soucis de visibilité entre paquetages qui apparaissent parfois. Ces notions apparaîtront par la suite lorsque le besoin s'en fera sentir. Le mois prochain, nous étudierons le sujet souvent évoqué de la gestion des exceptions en Ada.

Yves Bailly,

http://www.kafka-fr.net