Par :: Yves Bailly :: http://www.kafka-fr.net ::

Ada : présentation d'un langage pas comme les autres

Il est des langages de programmation qui semblent surgir d'un passé lointain, marqués d'une aura étrange, parfois un peu inquiétants. Le langage de programmation Ada fait partie de ceux-ci. Méconnu, parfois redouté, souvent dénigré, je vous propose de découvrir un langage puissant et moderne utilisé dans les situations critiques où une défaillance du logiciel est inacceptable.

Lady Augusta Ada Byron, comtesse de Lovelace [1], naquit à Londres le 10 décembre 1815. Effrayée à l'idée qu'elle puisse suivre l'exemple de son père, le grand poète Lord Byron, la mère d'Augusta Ada décida de lui enseigner les sciences en général et les mathématiques en particulier.

En 1933, Lady Ada rencontre Charles Babbage, concepteur de la machine analytique universelle, un assemblage extraordinaire de rouages et engrenages généralement considéré comme le premier ordinateur, même si la machine ne fut jamais achevée. Ada travailla avec l'inventeur à la mécanisation et au perfectionnement des métiers à tisser.

Durant les années 1840, elle proposa une méthode pour calculer les nombres de Bernoulli avec la machine de Babbage : historiquement, il s'agit là du premier programme informatique, même s'il ne portait pas encore ce nom (certains historiens estiment par ailleurs que le programme fut en réalité écrit par Babbage, et « seulement » corrigé par Ada). À ce titre, Lady Augusta Ada Lovelace Byron est considérée comme le premier programmeur de l'Humanité. Elle s'éteignit le 27 novembre 1852 d'un cancer, laissant derrière elle trois enfants, et fut inhumée aux cotés de son poète de père qu'elle ne connut jamais.

Ada 83

Durant les années 1970, le Département de la Défense des États-Unis d'Amérique s'inquiéta de la prolifération des langages de programmation utilisés pour ses besoins : plusieurs centaines furent recensés, certains complètement propriétaires et fermés, d'autres tout simplement dépassés. Un groupe fut chargé d'établir une liste de spécifications qu'un langage de programmation devrait satisfaire pour être utilisé dans le cadre de l'armée des États-Unis : aucun langage existant alors ne remplissait toutes les contraintes. Aussi un concours fut-il organisé en 1977 pour la création d'un nouveau langage.

Plusieurs propositions furent soumises, désignées par des couleurs. En 1979, la « proposition Verte », présentée par Jean Ichbiah (un français) de Honeywell Bull (société française), est retenue par le département de la défense. Le langage est nommé Ada en hommage à Lady Ada Lovelace Byron et son manuel de référence est approuvé le 10 décembre 1980, anniversaire de la naissance de Lady Ada. Le langage devint un standard ANSI en 1983, référencé ANSI/MIL-STD 1815 (MIL pour *military*, et 1815 pour l'année de naissance de Lady Ada) [2], puis un standard ISO en 1987, référencé ISO-8652:1987. Cette version du langage est communément désignée par « Ada83 ». Les caractéristiques de cette version sont, brièvement :

- Un typage fort, et même très strict : il est par exemple impossible d'affecter un entier à une variable réelle sans passer par une conversion explicite ;
- Contrôle à l'exécution : de nombreux tests sont effectués à l'exécution, pour détecter par exemple des dépassements de capacité pour les types entiers ou des conversions de types invalides ;
- Support des exceptions ;
- Support de la généricité, pour utiliser des types inconnus : il s'agit de l'équivalent des *templates* du langage C++ ;
- Support du parallélisme, pour l'exécution simultanée de plusieurs tâches d'un même programme ; on parle plutôt aujourd'hui de *multi-threading*, mais tandis que la plupart des langages nécessitent l'utilisation de bibliothèques externes pour le réaliser, Ada possède dans sa définition même les structures et mots-clefs nécessaires ;
- Facilités pour la création de bibliothèques d'outils (paquetages), afin d'atteindre une bonne réutilisation du code ;
- Une syntaxe claire, inspirée du langage Pascal.

Ces caractéristiques font de Ada un langage particulièrement robuste, utilisé dans des domaines critiques comme l'aérospatial ou l'armement de pointe. Dans la pratique, on peut dire qu'un programme en Ada est généralement plus difficile à écrire que son équivalent dans un autre langage plus répandu, par contre le résultat est presque toujours considérablement plus fiable. Autre aspect intéressant, ne peut se dénommer « compilateur Ada » qu'un compilateur respectant précisément la norme définie. On évite ainsi de nombreux maux de têtes engendrés par des compilateurs plus ou moins compatibles, qui respectent plus

ou moins les standards, comme c'est trop souvent le cas pour des langages comme C/C++ ou même Java.

Ada 95

Le langage connut une évolution majeure en 1995, avec l'acceptation en février 1995 du standard ISO-8652:1995 [3]. En plus d'un dépoussiérage, cette nouvelle version introduit dans le langage la programmation orientée objet, avec héritage simple et polymorphisme, ainsi que quelques annexes décrivant les possibilités offertes pour les systèmes temps réel, les architectures distribuées ou la programmation système. C'est sur cette version que nous allons nous appuyer dans ce qui suit. Incidemment, Ada95 est le premier langage de programmation orienté objet faisant l'objet d'une normalisation. La norme C++, par exemple, ne sera publiée que trois ans plus tard.

GNAT et Ada 2005

Pour faciliter l'évolution du standard et le développement du langage Ada, en 1992, l'armée de l'air des États-Unis (US Air Force) finança le développement d'un compilateur Ada en source ouvert (*open source*) : le compilateur GNAT (pour GNU Ada Translator), basé sur GCC, et dont le développement est assuré par la société Ada Core Technologie [4] [5] [10]. Depuis 2001, le compilateur GNAT est intégré à GCC.

Précisons qu'il existe actuellement trois versions de GNAT :

- GNAT Pro, compilateur commercial activement maintenu ;
- La version publique de GNAT, non maintenue mais de qualité industrielle ;
- La version de la Free Software Foundation, intégrée à GCC et suivant l'évolution de ce dernier.

Je m'appuierai sur la version de la Free Software Foundation, plus précisément celle intégrée à GCC 3.4 : les versions antérieures de GCC ne sont en effet pas considérées comme finalisées, du moins du point de vue de Ada. Pour l'essentiel, vous pouvez également utiliser la version publique, numérotée 3.15p, mais sachez que cette version n'est plus maintenue. Si vous utilisez une distribution Debian, la version publique est contenue dans le package `gnat`, la version FSF dans le package `gnat-3.3` (pour GCC 3.3) ou `gnat-3.4` (pour GCC 3.4).

Enfin, l'année 2005 devrait voir l'apparition d'une révision du langage Ada.

« Personne n'utilise Ada ! »

Voilà un commentaire que l'on entend régulièrement. S'il est vrai que son usage n'est pas très répandu chez les éditeurs de logiciels « classiques », rappelons que sa création a été l'initiative de l'armée des États-Unis d'Amérique : il est donc très utilisé par cette institution. Le langage Ada est utilisé par nombre d'entreprises et d'institutions, dans des situations où la fiabilité est la principale contrainte. Les avions Airbus A340 ou Boeing 777 utilisent largement Ada. Ainsi que le TGV français ou la ligne 14 du métro parisien (ligne automatique, sans conducteur). On le trouve également dans certaines institutions bancaires. La sonde Huygens, qui vient tout juste de renvoyer d'extraordinaires informations du sol de Titan, lune de Jupiter, embarque du code Ada. Le compilateur GNAT lui-même est écrit en Ada ! Pour plus de détails, consultez la page référencée en [8]. Moralité, « personne » représente quand même pas mal de monde...

Premier programme

Sacrifions à la tradition. Voici le « Bonjour, Monde ! » en Ada.

```
1 with Text_IO ;
2 use Text_IO ;
3 procedure bonjour is
4 begin
5   put_line("Bonjour, Monde !") ;
6 end bonjour ;
```

Stockez ce programme dans un fichier nommé `bonjour.adb`. Attention, le nom du fichier a son importance ! Pour le compiler, utiliser l'outil `gnatmake` fourni avec Gnat, puis exécutez-le :

```
$ gnatmake bonjour.adb
gnatgcc -c bonjour.adb
gnatbind -x bonjour.ali
gnatlink bonjour.ali
$ ./bonjour
Bonjour, Monde !
$
```

Examinons dans le détail ce modèle de trivialité. La première ligne signale que l'on va utiliser le contenu du paquetage `Text_IO`. Dans le verbiage Ada, un paquetage est ce que l'on appelle une bibliothèque en C/C++ ou un module en Python. `Text_IO` contient tout un ensemble de fonctions liées aux entrées-sorties de textes : la première ligne serait donc `#include <stdio.h>` en C ou `#include <iostream>` en C++.

Un paquetage ainsi référencé définit une sorte d'espace de nommage qui lui est propre. Python utilise une technique similaire pour ses modules et on peut rapprocher cela des *namespaces* du C++. La clause `use` utilisée en deuxième ligne permet de rendre le contenu du paquetage directement visible, sans qu'il soit besoin d'utiliser un préfixe. Réécrite en Python, cette ligne deviendrait `from Text_IO import *` ou encore `using namespace Text_IO` en C++. Arrêtons-nous un instant, pour signaler un aspect très important de la syntaxe de Ada. Contrairement à la plupart des langages répandus aujourd'hui, Ada n'est pas sensible à la casse des caractères – aucune différence n'est faite entre majuscules et minuscules. Ainsi, la première ligne aurait tout aussi bien pu être écrite `WitH teXT_iO`. Cela n'aurait fait aucune différence. Au début, c'est assez déroutant, surtout si on ne pratique pas couramment le Pascal ou le Basic. Par ailleurs, l'indentation est libre en Ada. Notre petit programme aurait pu être écrit sur une seule ligne. Il y a toutefois une limite, la longueur de la ligne : le standard Ada précise qu'un compilateur doit accepter des lignes et des identificateurs (noms de variables ou fonctions) jusqu'à au minimum 200 caractères (section 2.2, paragraphe 14), mais n'impose pas de maximum. Par conséquent, pour une portabilité maximale, ne dépassez pas 200 caractères par ligne. Continuons. La troisième ligne identifie ce que nous appellerons notre procédure principale. En C et C++, il s'agit d'une fonction toujours nommée *main* d'un type bien défini. En Ada, il s'agit d'une procédure d'un nom quelconque, ne prenant aucun argument. Petite contrainte, liée au compilateur Gnat : le fichier qui contient la procédure principale doit être nommé du nom de cette procédure, avec l'extension adb. Ici, notre procédure s'appelle bonjour, donc le fichier contenant notre programme doit être nommé `bonjour.adb`.

Pour délimiter les blocs, Ada n'utilise ni les accolades du C/C++/Perl, ni l'indentation de Python. On retrouve le vieux couple `begin/end`, qui existe également en Pascal. Notez que le nom de la procédure est rappelé après le `end` final, ligne 6 : ce n'est pas absolument nécessaire, mais facilite

la relecture. Par contre, si vous changez le nom de la procédure en ligne 3, changez-le également en ligne 6 : si les deux noms ne correspondent pas, vous aurez une erreur de compilation. Finalement, la seule instruction qui fait réellement quelque chose est en ligne 5. `put_line()` est une procédure du paquetage `Text_IO`, qui a pour seule fonction d'afficher la chaîne de caractères donnée en paramètre suivie d'un saut de ligne. Dans d'autres langages, cela pourrait être :

- En C : `printf("Bonjour, Monde !\n")` ;
- En C++ : `std::cout << "Bonjour, Monde !" << std::endl` ;
- En Python : `print 'Bonjour, Monde !'`

Si nous n'avions pas utilisé la clause `use` en ligne 2, l'instruction deviendrait :

```
5   Text_IO.put_line("Bonjour, Monde !") ;
```

Une écriture familière pour ceux qui pratiquent Python. Voilà pour la tradition, passons à autre chose.

Types fondamentaux et variables

Le standard impose l'existence des types suivants :

- `boolean`, type booléen prenant les deux valeurs `true` et `false` ;
- `character`, pour les caractères standards ISO 8859-1 – en gros, les caractères ASCII plus quelques caractères accentués, correspondant aux caractères latin-1 (caractères codés sur 8 bits) ;
- `wide_character`, pour les caractères du code ISO 10646, aussi appelé UCS, correspondant à l'Unicode (caractères codés sur 16 bits) ;
- `integer`, type entier signé défini au minimum dans l'intervalle de $-2^{15}+1$ à $2^{15}-1$, mais qui peut être plus large ;
- `float`, type réel en virgule flottante supportant au minimum six chiffres significatifs (mais peut-être plus) ;
- `string`, pour les chaînes de caractères « normales », en fait un tableau de `characters` ;
- `wide_string`, pour les chaînes de caractères UCS, en fait un tableau de `wide_characters`.

Toutefois, selon la plate-forme sur laquelle opère le compilateur et notamment le processeur sur lequel est censé tourner le programme, d'autres types peuvent être fournis. Les deux donnés ici sont très répandus :

- `long_integer`, entiers longs signés, au minimum dans l'intervalle de $-2^{31}+1$ à $2^{31}-1$;
- `long_float`, type réel en virgule flottante supportant au minimum 11 chiffres significatifs.

Il est intéressant et important de noter que le standard ne définit véritablement que des contraintes minimales. Par exemple, pour une plate-forme donnée, le standard n'interdit pas le type `integer` de couvrir l'intervalle autorisé par une représentation sur 128 bits. Par ailleurs, ne sont donnés ici que les types les plus fondamentaux et élémentaires (à l'exception de `string` et `wide_string`, qui sont des tableaux) : d'autres types existent, comme les nombres décimaux à virgule fixe, que nous rencontrerons par la suite. Enfin, curieusement ces types ne sont pas des mots-clefs du langage : ils sont définis dans un paquetage nommé `Standard`. Inutile de requérir l'utilisation de `Standard` avec la clause `with`, comme nous l'avons fait pour le paquetage `Text_IO` : il est toujours disponible et son contenu directement accessible.

La déclaration d'une variable, car en Ada les variables doivent être déclarées avant d'être utilisées, se fait ainsi :

```
mon_entier: integer ;
```

On est donc à l'inverse du C/C++ : ici, on donne d'abord le nom de la variable, puis son type. Pour initialiser la variable avec une valeur :

```
mon_entier: integer := 1 ;
```

Et enfin, si cette variable doit être une constante :

```
mon_entier: constant integer := 1 ;
```

Vous l'aurez compris, en Ada l'opérateur d'affectation est repris du Pascal : il s'agit du symbole composé `:=`.

Ada propose diverses possibilités pour l'écriture des entiers :

```
e1: integer := 1_000_000 ;
e2: integer := 16#12AB# ;
e3: integer := 7#456# ;
```

La première écriture montre l'utilisation du caractère de soulignement pour séparer les groupes de chiffres dans un nombre : cela permet de faciliter la lecture, mais cela n'a rien d'obligatoire. La deuxième montre comment donner un nombre en base 16 (hexadécimale) : il suffit d'encadrer le nombre par des dièses `#`, et de faire précéder le tout de la base voulue - 16 pour l'hexadécimal. Voyez la troisième ligne, où un nombre est donné en base 7. Naturellement, si vous donnez un nombre incohérent avec la base demandée (par exemple, un chiffre plus grand que 6 en base 7), vous serez gratifié d'une erreur de compilation. Pour ce qui est des nombres à virgule flottante, Ada utilise la notation usuelle. Les caractères sont quant à eux donnés comme en C, entre apostrophes. Je n'en dirai pas plus pour l'instant, les chaînes de caractères méritant un traitement spécial.

Les attributs

Chaque type en Ada possède un certain nombre de caractéristiques ou fonctions intrinsèques, appelées attributs. Ce ne sont pas des fonctions en tant que telles. On ne peut pas vraiment les rapprocher de données ou méthodes membres d'une classe. Un attribut est désigné en faisant précéder son nom d'une apostrophe, elle-même précédée d'un nom de type (ou d'un objet, selon les cas). Avant de trop m'embrouiller dans une définition difficile à exprimer, voyons quelques exemples.

- `integer'first` va donner un entier égal à la plus petite valeur que peut prendre le type `integer` ;
- `float'last` va donner un nombre réel égal à la plus grande valeur que peut prendre le type `float` ;
- `integer'size` donne le nombre de bits (et non d'octets) occupés par une variable de type `integer`.

Les attributs `first`, `last` et `size` sont applicables à tous les types élémentaires que nous avons rencontrés. Si ceux-ci ressemblent à de simples variables, il en est d'autres qui prennent la forme d'appels de fonctions :

- `integer'min(a, b)` retourne le plus petit des deux entiers a et b ;
- `float'max(x, y)` retourne le plus grand des deux réels x et y ;
- `integer'value("123")` retourne un entier résultant de la conversion de la chaîne de caractères ; cela fonctionne également avec la notation spéciale pour choisir une base, par exemple

`integer'value("16#FF#")` retourne l'entier 255 ;

`float'image(3.14)` est la réciproque du précédent et retourne une chaîne de caractères représentant la valeur donnée en paramètre. Nous verrons plus tard les possibilités pour afficher autre chose que des chaînes de caractères. Mais pour l'heure, l'attribut précédent nous permet d'afficher simplement des valeurs numériques :

```
 1 with Text_IO ;
 2 use Text_IO ;
 3 procedure attributs is
 4   i_min: integer := integer'first ;
 5   i_max: integer := integer'last ;
 6   f_min: float := float'first ;
 7   f_max: float := float'last ;
 8 begin
 9   put_line("i_min = " & integer'image(i_min)) ;
10   put_line("i_max = " & integer'image(i_max)) ;
11   put_line("f_min = " & float'image(f_min)) ;
12   put_line("f_max = " & float'image(f_max)) ;
13 end attributs ;
```

Le résultat de ce programme, qui peut varier selon votre architecture, est le suivant :

```
$ gnatmake attributs.adb
gnatgcc -c attributs.adb
gnatbind -x attributs.ali
gnatlink attributs.ali
$ ./attributs
i_min = -2147483648
i_max =  2147483647
f_min = -3.40282E+38
f_max =  3.40282E+38
```

L'attribut `image` ne permet pas de mise en forme précise, comme le nombre de chiffres significatifs. Mais encore une fois, nous verrons cela plus tard.

Sous-types et types dérivés

Il est assez courant de définir des types personnalisés à partir des types de base, même pour les types élémentaires. En C/C++, cela se fait usuellement à l'aide de l'instruction `typedef`, qui ne fait guère plus que définir un synonyme (dans ce cas très limité). En Ada, par contre, de tels types personnalisés sont à la base de la robustesse des programmes : ils font bien plus que simplement définir un nouveau nom. Supposons que dans un programme, nous ayons à manipuler des dimensions, certaines exprimées en millimètre, d'autres en pouce (unité de mesure obsolète fort appréciée de nos amis anglo-saxons).

Il semble évident qu'additionner sans précaution des valeurs en millimètre avec des valeurs en pouce conduirait à des résultats aberrants. Voyons trois programmes minimalistes réalisant cette opération, le premier en C++, les deux suivants en Ada (*voir tableau Programmes*).

Le programme C++ compile sans aucun problème et affichera la valeur 3, dont on peut bien se demander quelle est son unité. Le premier programme Ada, qui utilise les sous-types, compilera également sans rien de particulier et affichera également la valeur 3. Par contre, le troisième programme ne compilera tout simplement pas :

```
$ gnatmake ada_types_derives.adb
gnatgcc -c ada_types_derives.adb
ada_types_derives.adb:9:25: invalid operand types for operator "+"
ada_types_derives.adb:9:25: left operand has type "d_mm" defined at line 4
ada_types_derives.adb:9:25: right operand has type "d_in" defined at line 5
gnatmake: "ada_types_derives.adb" compilation error
```

Programmes

Programme 1

```
#include <iostream>
typedef float d_mm ;
typedef float d_in ;
int main (int argc, char* argv[])
{ d_mm a = 1.0 ;
  d_in b = 2.0 ;
  std::cout << a+b << "\n" ;
  return 0 ;
}
```

Programme 2

```
with Text_IO ;
use Text_IO ;
procedure ada_sous_types is
  subtype d_mm is float ;
  subtype d_in is float ;
a: d_mm := 1.0 ;
b: d_in := 2.0 ;
begin
  put_line(float'image(a+b)) ;
end ada_sous_types ;
```

Programme 3

```
with Text_IO ;
use Text_IO ;
procedure ada_types_derives is
  type d_mm is new float ;
  type d_in is new float ;
a: d_mm := 1.0 ;
b: d_in := 2.0 ;
begin
  put_line(float'image(a+b)) ;
end ada_types_derives ;
```

Si dans le premier programme Ada, `d_mm` et `d_in` sont des sous-types (*subtype*) du type `float`, dans le deuxième, ce sont bel et bien deux types différents, construits à partir d'un même type ! Dans le premier cas, ils restent des `float` et peuvent donc être combinés. Dans le deuxième, ils ont chacun leur identité propre. Une telle « dureté » de typage est inconnue du C++ et, en fait, de la plupart des langages de programmation. Cela peut paraître une contrainte supplémentaire et désagréable, mais la technique des *types* dérivés permet d'éviter de nombreuses erreurs de logique dans les calculs (celles-ci sont détectées par le compilateur) et confère donc une grande robustesse aux programmes Ada qui en font usage.

Contrôles à l'exécution

Toutes les erreurs ne peuvent être détectées à la compilation. Mais contrairement à d'autres langages qui laissent passer pas mal de choses, Ada permet de détecter de nombreuses sources de problèmes et de prévoir une réaction adaptée.

Comme exemple, considérez le petit programme suivant :

```
1 procedure test is
2 i: integer ;
3 f: float := 1.0E+35 ;
4 begin
5   i := integer(f) ;
6 end ;
```

Rien de bien terrible : une valeur en virgule flottante est explicitement convertie en un entier, à l'aide d'un *transtypage*, ligne 5. Syntaxiquement, tout est correct... sauf que la valeur de la variable `f` est trop grande pour être stockée dans la variable `i`, qui va alors prendre une valeur probablement incohérente. Mais cela, le compilateur ne peut pas le détecter. Avec un langage comme le C, il n'existe aucun moyen de se prémunir de ce genre d'erreurs. Ada permet de le faire, en activant le contrôle de débordement à la compilation au moyen du paramètre `-gnato` passé à `gnatmake` :

```
$ gnatmake -gnato test.adb
gnatgcc -c -gnato test.adb
gnatbind -x test.ali
gnatlink test.ali
gnatlink: warning: executable name "test" may conflict with
shell command
$ ./test
raised CONSTRAINT_ERROR : test.adb:5 overflow check failed
```

Remarquez l'avertissement qui signale que le nom du programme est le même que celui d'une commande de l'interpréteur de commande. Mais plus intéressante est l'exécution de notre programme. Au moment de l'affectation, le système de contrôle détecte un dépassement de capacité et provoque alors une exception. Ici, aucun traitement n'a été prévu dans cette situation (nous verrons plus tard la gestion des exceptions), aussi le programme se termine-t-il immédiatement. Mais dans une situation réelle, nous pourrions intercepter l'exception et réagir en conséquence. Naturellement, l'activation de ces contrôles à l'exécution génère des exécutables plus volumineux et plus lents. Mais ils sont précieux en phase de test et, dans une situation critique, il n'est pas recommandé de les désactiver.

Ariane 5 Vol 501

Si vous pensez que ce genre de technique est bien lourde, bien inutile et contraignante, permettez-moi d'évoquer le cas du vol 501 d'Ariane 5, le premier vol de test de la fusée Ariane 5, le 4 juin 1996. Ce premier vol, annoncé en fanfare, se conclut après quelques secondes par la destruction de la fusée devenue incontrôlable. Après enquête, il fut découvert que le responsable était une petite portion de code, justement écrite en Ada, reproduite ici (d'après [7]) :

```
1 -- ...
2 declare
3   vertical_veloc_sensor: float;
4   horizontal_veloc_sensor: float;
5   vertical_veloc_bias: integer;
6   horizontal_veloc_bias: integer;
7 --  ...
 8 begin
 9   declare
10     pragma suppress(numeric_error, horizontal_veloc_bias);
11   begin
12     sensor_get(vertical_veloc_sensor);
13     sensor_get(horizontal_veloc_sensor);
14     vertical_veloc_bias := integer(vertical_veloc_sensor);
15     horizontal_veloc_bias := integer(horizontal_veloc_
sensor);
16 --    ...
17   exception
18     when numeric_error => calculate_vertical_veloc();
19     when others => use_irs1();
20   end;
21 end irs2;
```

Pour information, les lignes qui commencent par deux tirets (1, 7 et 16) sont des commentaires : le double tiret est le symbole utilisé par Ada pour commencer un commentaire, qui se poursuit jusqu'à la fin de la ligne.

Sans entrer dans les détails, remarquez que la variable `horizontal_veloc_sensor` est de type réel, tandis que la variable `horizontal_veloc_bias` est de type entier. Ligne 15, la valeur de la première est placée dans la seconde, par l'intermédiaire d'un transtypage. Ce code a été récupéré d'Ariane 4, et certainement insuffisamment testé dans le cadre d'Ariane 5 : la poussée produite par Ariane 5 est très supérieure à celle d'Ariane 4.

En fait, durant le vol d'essai cette portion de code s'est trouvée dans la même situation que mon petit exemple de cinq lignes : la valeur réelle est devenue trop grande pour pouvoir être stockée dans la variable entière. Pourtant, cette portion de code prévoit un traitement des exceptions, lignes 17 à 20, justement destinée à gérer ce type de problème. Seulement, une directive du langage a été utilisée pour supprimer explicitement le contrôle des erreurs numériques pour la variable `horizontal_veloc_bias` : c'est le sens de la ligne 10. On peut s'interroger sur les motivations de l'écriture de cette ligne, probablement des considérations de vitesse d'exécution. Toujours est-il qu'au moment de l'exécution de la ligne 15, tout se passe comme si le programme n'avait pas été compilé avec le paramètre `-gnato` évoqué plus haut : un débordement de capacité survient, mais il est superbement ignoré.

D'après les rapports d'enquête, ce dépassement est survenu 36.7 secondes après le décollage. Un peu plus de deux secondes plus tard, la fusée était détruite. Coût estimé : environ deux milliards de francs de l'époque, soit environ 300

millions d'euros.Ce que l'on peut retenir de cette expérience, entre autres, c'est que bien que le langage offre toutes les facilités pour prévenir et éviter la catastrophe, la désactivation forcée d'une sécurité précise, associée à des tests insuffisants, a suffi à transformer un engin spatial exceptionnel en poussière de métal.

Conclusion

Voilà pour cette rapide présentation du langage Ada. J'espère qu'elle vous aura intéressé et vous donnera l'envie de poursuivre dans l'apprentissage de ce langage. Vous aurez remarqué que cet article est parsemé de comparaisons entre Ada et d'autres langages et peut-être n'aurez-vous pas apprécié certains commentaires.

Alors, abordons brièvement le délicat sujet du choix d'un langage de programmation. Il est probablement impossible de dire qu'un langage est meilleur qu'un autre dans toutes les situations. Chaque langage a été défini dans un contexte particulier, avec une certaine idée préconçue de son utilisation, qui diffère parfois de la réalité d'utilisation du langage dans la pratique. Forces et faiblesses d'un langage donné trouvent leur origine dans les objectifs considérés comme prioritaires de ses concepteurs.

Sans doute le langage Ada est-il plus contraignant qu'un autre. Si vous voulez écrire rapidement un programme prototype d'une idée fulgurante, Python est bien indiqué. Si vous dispose d'un peu plus de temps, et que vous voulez avoir assez vite un programme très rapide à l'exécution, alors C ou C++ sont probablement de bons choix. Si le programme doit être le plus solide possible, tolérant aux erreurs, et impliquer des sommes importantes ou des vies humaines, alors Ada est sûrement la voie à emprunter. Ce ne sont là que quelques aspects qui motivent le choix d'un langage plutôt que d'un autre.

Il en existe de nombreux autres qui peuvent faire pencher la balance. Il me semble que l'important est de connaître plusieurs langages, pour pouvoir effectuer le meilleur choix ou le choix le moins mauvais, quand la question se pose. C'est dans cet esprit que je souhaite vous présenter Ada. Et pour cela, nous réaliserons, au fil du temps, un petit programme. Et pour suivre ce qui va peut-être devenir une autre tradition dans ces pages, nous nous attaquerons une fois de plus aux sempiternelles Tours de Hanoï. Hé si !

Références

- [1] Lady Augusta Ada Lovelace Byron : http://en.wikipedia.org/wiki/Ada_Lovelace
- [2] Ada 83 (ANSI/MIL-STD 1815) : http://archive.adaic.com/standards/83lrm/html/Welcome.html
- [3] Ada 95 (ISO-8652:1995) : http://www.adaic.org/standards/95lrm/html/RM-TTL.html
- [4] Ada Core Technologies : http://www.gnat.com/
- [5] ACT Europe : http://www.act-europe.fr/
- [6] AdaPower : http://www.adapower.com/
- [7] Code Ariane 5, vol 501 : http://www-aix.gsi.de/~giese/swr/ariane5.html
- [8] Who's using Ada : http://www.seas.gwu.edu/~mfeldman/ada-project-summary.html
- [9] AdaWorld : http://adaworld.com/
- [10] Version publique de GNAT et autres outils : http://libre.act-europe.fr/

Par :: Yves Bailly :: http://www.kafka-fr.net ::

Le langage Ada : sous-programmes

À moins d'être trivial ou codé vite-fait mal-fait, un programme n'est jamais une suite linéaire d'instructions : certaines sont regroupées pour faciliter leur réutilisation. En C/C++, Python et de nombreux autres langages, on parle simplement de « fonctions ».

Le terme sous-programme, ancien en programmation, a une signification plus générale. Il a été repris par le langage Ada, qui divise les sous-programmes en deux catégories : d'une part les fonctions, d'autres part les procédures. Une distinction qui paraît bien étrange aujourd'hui...

Le terme de procédure lui-même ne date pas d'hier : Algol60, langage créé à la fin des années cinquante et duquel sont issus entre autres le C et le Pascal au début des années soixante-dix, l'utilisait déjà pour désigner un sous-programme. Mais que signifie la distinction entre fonctions et procédures ?

N'oublions pas que l'informatique en général, et la programmation en particulier, tirent leurs origines dans les mathématiques. La différence entre les deux concepts peut être résumée ainsi :

- Une procédure **fait** quelque chose ;
- Une fonction **vaut** quelque chose.

Pour faire le parallèle avec le langage C, une procédure (Ada) est une fonction (C) qui ne retourne rien (ou `void`), tandis qu'une fonction (Ada) est une fonction (C) qui retourne une valeur. Cette subtilité peut paraître de pure forme, la distinction purement sémantique. Mais c'est là justement son importance : plus que beaucoup d'autres, le langage Ada s'attache à la sémantique, au sens de ce que le programmeur veut exprimer. D'une certaine manière, on pourrait dire que Ada est un langage expressif, où des facilités sont données au programmeur pour traduire ce qu'il veut faire, par opposition à la plupart des autres langages que l'on pourrait qualifier d'effectifs, où le programmeur dit comment faire ce qu'il veut. C'est probablement ce qui est le plus déroutant quand on est expérimenté dans un ou plusieurs langages « classiques » et que l'on découvre Ada. Nous sommes habitués à exprimer le comment, alors que Ada nous incite à plutôt exprimer le quoi. Et il est souvent plus difficile (du moins en programmation) d'exprimer ce que l'on a en tête, que d'expliquer comment l'obtenir. Nous allons voir que cette subtile distinction entre fonctions et procédures n'est pas sans conséquences sur l'écriture du code en Ada.

Les procédures

Commençons par une procédure simple, ne prenant pas de paramètre. Il pourrait suffire de reprendre l'exemple « Bonjour, Monde ! » du mois dernier : il n'est composé que d'une unique procédure, laquelle contient le code exécuté au lancement du programme (l'équivalent de la fonction `main()` du C). Reprenons ce programme, en créant une autre procédure invoquée par cette procédure principale :

```
1 with Text_IO ;
2 use Text_IO ;
3 procedure bonjour is
4   procedure affiche is
5   begin
6     put_line("Bonjour") ;
7   end ;
8 begin
9   affiche ;
10 end bonjour ;
```

La nouvelle procédure est définie lignes 4 à 7, à l'intérieur de la procédure principale `bonjour`. Cela peut surprendre, mais en fait c'est parfaitement normal : la procédure principale est le programme (elle constitue ce que l'on appelle une unité de compilation), et dans notre cas il n'y a pas d'autre endroit pour définir variables et sous-programmes (mais rassurez-vous, nous verrons bientôt comment organiser les choses en unités séparées). En fait, en Ada, tout sous-programme peut contenir une ou plusieurs définitions d'autres « sous-sous-programmes ».

La définition de la procédure `affiche` ressemble à s'y méprendre à celle de `bonjour`. Remarquez toutefois que le nom n'est pas répété ligne 7, à la fin de la définition. Cette répétition n'est pas obligatoire, simplement vivement recommandée. Remarquez également l'absence de parenthèses, aussi bien dans la définition (ligne 4) que dans l'invocation (ligne 9). Comme il n'y a pas de paramètre, elles sont inutiles – et même interdites : écrire `affiche();` ligne 9, comme on le ferait en C, est une erreur de syntaxe.

Passage de paramètres

Voyons cet exemple contenant trois procédures (en plus de la procédure principale) :

```
1 with Text_IO ;
2 use Text_IO ;
3 procedure Double is
4   procedure Double(i: in Integer) is
5   begin
6     Put(Integer'Image(2*i)) ;
7   end ;
8   procedure Double2(ii: in out Integer) is
9   begin
10    ii := 2*ii ;
11    Put(Integer'Image(ii)) ;
12  end ;
13  procedure Double(i: Integer ; r: out Integer) is
14  begin
15    r := 2*i ;
16    Put(Integer'Image(r)) ;
17  end ;
```

```
18 entier: Integer := 1 ;
19 resultat: Integer := 0 ;
20 begin
21   Double(entier) ;
22   Double2(entier) ;
23   Double(entier, resultat) ;
24   Put_Line(Integer'Image(resultat)) ;
25 end Double ;
```

Ce programme contient trois procédures nommées `Double` : ceci pour montrer que Ada permet la surcharge des noms de sous-programmes. Rappelons, s'il est besoin, qu'il n'est pas fait de différence entre majuscules et minuscules, sauf bien sûr pour le contenu des chaînes de caractères (dont nous reparlerons plus tard en détail). Les langages comme Pascal ou C/C++ distinguent différentes façons de passer les paramètres, soit par valeur, soit par adresse ou par référence. Cela détermine si le sous-programme reçoit une copie de la zone mémoire associée à une variable (passage par valeur) ou s'il reçoit un moyen d'accéder à cette zone mémoire (passage par adresse ou par référence). Cette distinction sur la manière d'obtenir une variable (ou une valeur) transférée à un sous-programme existe en Ada, mais avec une approche différente. On distingue plutôt trois modes de passage :

- Le mode `in`, le mode par défaut, où le paramètre peut être lu et utilisé dans le sous-programme, mais pas modifié ;
- Le mode `in out`, où le paramètre peut être lu et modifié ; cela correspond à un passage par référence ;
- Le mode `out`, plus inhabituel, où le paramètre peut être lu et modifié, mais sa valeur est indéterminée à l'entrée dans la procédure.

Certains commencent peut-être à ouvrir de grands yeux : le mode `in` ressemble à un passage par valeur, n'est-ce pas ? Mais alors, que se passe-t-il si on souhaite donner au sous-programme, en mode « lecture seule », par exemple un gros tableau contenant des millions d'éléments ? En Pascal ou C, passer un tel objet par valeur impliquerait sa duplication sur la pile, ce qui peut être très long, voire provoquer un débordement de pile et un plantage du programme. Le code machine généré pour les paramètres en mode `in` dépend en fait du type du paramètre. Pour les types simples, les types fondamentaux, il s'agit effectivement d'un passage par valeur. Pour les types structurés, comme les tableaux, cela sera (presque) toujours un passage par référence. Le choix du mode de passage interne est dévolu au compilateur. Le programmeur n'a alors pas à se demander de quelle manière passer son paramètre, mais les limites d'accès qui lui sont imposées. Prenons la première procédure, lignes 4 à 7. La déclaration des paramètres ressemble à une déclaration de variable (dont deux exemples sont donnés lignes 18 et 19), en intercalant le mode de passage. Ici le paramètre est passé en mode `in` : sa valeur est donc utilisable par la procédure, mais en aucun cas modifiable. Essayez d'insérer une ligne comme `i:=2;` dans cette procédure : vous obtiendrez une erreur de compilation. Pour mémoire, l'écriture `Integer'Image()`, que nous avons rencontrée le mois dernier, permet de convertir un entier (de type `Integer`) en sa représentation sous forme de chaîne de caractères, au moyen de l'attribut `Image` appliqué au type `Integer`.

La deuxième procédure, lignes 8 à 12, montre un passage en mode `in out`. Cette fois le paramètre peut être modifié, ce qui est fait ligne 10. Dans ce cas, comme lors d'un passage par référence en C++, la procédure ne peut être appelée qu'en lui donnant une variable, alors que la première procédure pouvait être invoquée en donnant une valeur littérale. À l'issu de l'exécution de `Double2()`, la valeur de la variable qui lui aura été donnée sera modifiée. Enfin, la dernière procédure, lignes 13 à 17, montre l'utilisation du mode `out`... et l'absence de mode pour le premier paramètre. En l'absence de mode, c'est le mode `in` qui est appliqué, donc le paramètre `i` est passé en mode `in`. Concernant le paramètre `r`, la grande différence avec le mode `in out` est que la valeur de `r` est indéterminée quand on rentre dans la procédure, même si une valeur a été affectée préalablement à la variable associée. C'est ce que précise le standard du langage. Pour un compilateur donné, le paramètre peut être la variable associée passée par référence, auquel cas la valeur du paramètre sera celle de la variable au moment de l'appel. Un autre compilateur est libre d'adopter une autre stratégie, comme un passage par valeur, suivi d'une affectation à la terminaison de la procédure. Faire l'hypothèse du mode effectif de passage et en tenir compte conduit à un programme généralement non portable sur un autre compilateur. La signification du mode `out` en Ada95 est un peu différente de celle en Ada83 : dans ce dernier, il était explicitement interdit de lire la valeur du paramètre au sein du code de la procédure. La ligne 16 est alors une erreur de syntaxe, car pour afficher la valeur de `r`, il faut lire la valeur de `r`... Dans la pratique, il est préférable de traiter de cette manière les paramètres en mode `out` : ne pas tenter de lire leur valeur, tant qu'ils n'ont pas subi une affectation. Essayez de dupliquer la ligne 16 juste avant la ligne 15 : le compilateur devrait vous gratifier d'un avertissement, car vous tentez d'utiliser la valeur de `r` avant d'avoir affecté une valeur à `r` et ce malgré l'initialisation faite ligne 19.

Saurez-vous trouver l'affichage résultat de l'exécution du programme précédent ? Le voici :

```
$ ./double
 2 2 4 4
```

La procédure `Put()`, fournie par `Text_IO`, affiche une chaîne de caractères sans ajouter de saut à la ligne, contrairement à `Put_Line()`.

Les fonctions

La déclaration d'une fonction est un peu différente de celle d'une procédure. Voyons sur cet exemple, stocké dans `double2.adb` :

```
 1 with Text_IO ;
 2 use Text_IO ;
 3 procedure Double2 is
 4   function Double(i: in Integer) return Integer is
 5   begin
 6     return 2*i ;
 7   end ;
 8   function Double(i: in Integer) return String is
 9   begin
10     return Integer'Image(2*i) ;
11   end ;
12 begin
13   Put_Line(Integer'Image(Double(3))) ;
14   Put_Line(Double(4)) ;
15 end Double2 ;
```

Le mot clef `function` remplace le mot clef `procedure` et cette fois on indique le type retourné par la fonction au moyen du mot clef `return`. Ce même mot clef doit d'ailleurs être présent (une ou plusieurs fois) dans le corps de la fonction et faire partie du flux d'instructions : si le compilateur détecte qu'il est possible d'arriver à la fin des instructions de la fonction sans avoir rencontré `return`, alors une erreur de

compilation est affichée – tandis que ce n'est qu'un avertissement en C++.

Les programmeurs C++ seront peut-être surpris de l'exemple donné : nous avons en effet deux fonctions, de même nom, et prenant les mêmes paramètres... Elles ne se distinguent que par le type de retour. En effet, en Ada le type de retour fait partie de la signature de la fonction, ce qui n'est pas le cas en C++.

Il existe une autre particularité propre aux fonctions. Les paramètres ne peuvent pas être en mode `out` ou `in out`. Ceci provient des inspirations mathématiques du langage : une fonction au sens mathématique du terme ne peut que retourner un résultat, en aucun cas modifier les valeurs qui lui sont éventuellement données à traiter. Aussi les paramètres d'une fonction Ada ne peuvent qu'être en mode `in`, le mode par défaut – la présence du `in` dans l'exemple plus haut est en fait superflue. Toutefois, il est parfois bien pratique et même nécessaire de pouvoir malgré tout modifier un paramètre passé à une fonction. Nous verrons plus tard un quatrième mode de passage, le mode `access`, qui permet cela.

Les deux fonctions définies sont invoquées lignes 13 et 14 : c'est le contexte qui permet de déterminer laquelle effectivement appeler. Ligne 13, l'attribut `Image` appliqué au type `Integer` (ce que l'on pourrait aisément assimiler à un appel de fonction) ne peut accepter qu'un paramètre de type `Integer` : c'est donc la fonction `Double()` qui retourne un `Integer` qui doit être invoquée. De la même manière, la procédure `Put_Line()` (issue du paquetage `Text_IO`) n'accepte qu'une chaîne de caractères de type `String` : ligne 14, on n'a donc d'autre choix que d'invoquer la deuxième fonction.

Une telle souplesse n'est permise que grâce à l'aspect strictement typé de Ada : il n'y a en fait pas de conversion implicite entre types, comme cela est si courant en C/C++. Nous reviendrons plus précisément sur les notions de types et sous-types en Ada, mais l'exemple précédent pourrait être encore poussé plus loin en introduisant un type dérivé :

```
1 with Text_IO ;
2 use Text_IO ;
3 procedure Double3 is
4   type Entier is new Integer ;
5   function Double(i: in Integer) return Integer is
6   begin
7     return 2*i ;
8   end ;
9   function Double(i: in Integer) return Entier is
10  begin
11    return 2*Entier(i) ;
12  end ;
13 begin
14   Put_Line(Integer'Image(Double(3))) ;
15   Put_Line(Entier'Image(Double(4))) ;
16 end Double3 ;
```

Le type `Entier` déclaré ligne 4 possède toutes les caractéristiques du type prédéfini `Integer` et très certainement sa représentation interne binaire est exactement identique. Pourtant, il s'agit bien d'un type différent ! Ce programme compile et s'exécute sans problème, moyennant un transtypage explicite ligne 11. En effet, le type de `i` étant `Integer`, le type de l'expression `2*i` est également `Integer`. Or, la fonction doit retourner un `Entier`. Le typage strict de Ada, répétons-le, interdit la conversion implicite d'un `Integer` vers un `Entier`. Donc si nous avions inscrit simplement `return 2*i;` ligne 11, le compilateur signalerait une erreur. Deux solutions s'offrent alors : soit convertir le paramètre `i` en `Entier` avant de faire le calcul, comme ici, soit faire le calcul en `Integer` et convertir le tout en `Entier`, en écrivant `return Entier(2*i);`.

Dernier mot concernant les fonctions, la valeur retournée ne peut pas être ignorée, comme en C, Python et d'autres langages. Une instruction comme :

```
Double(3) ;
```

sera refusée par le compilateur : elle est assimilée à un appel de procédure, or il n'existe pas de procédure nommée `Double()` et prenant un paramètre entier. Incidemment, cela signifie que l'on peut déclarer une fonction et une procédure ayant le même nom et prenant les mêmes paramètres - encore une fois, le contexte permet de les distinguer.

Valeurs par défaut et paramètres nommés

Comme de nombreux langages, Ada permet de donner une valeur par défaut à certains paramètres d'une procédure (ou d'une fonction), toutefois seulement pour les paramètres en mode `in`. Mais en plus, il est possible de nommer explicitement les paramètres au moment de l'invocation, comme cela se fait également en Python.

Voyons cela sur un exemple :

```
1 with Text_IO ;
2 use Text_IO ;
3 procedure default is
4   procedure proc(a: in Integer ;
5                  b: in Integer := 22 ;
6                  c: in Integer ;
7                  d: in Integer := 44) is
8   begin
9     Put_Line("a = " & Integer'Image(a) &
10              ", b = " & Integer'Image(b) &
11              ", c = " & Integer'Image(c) &
12              ", d = " & Integer'Image(d)) ;
13  end ;
14 begin
15  proc(1, 2, 3, 4) ;
16  proc(d=>4, c=>3, b=>2, a=>1) ;
17  proc(1, c=>3) ;
18 end default ;
```

La déclaration de la procédure `proc`, qui s'étale des lignes 4 à 7 pour plus de lisibilité, prend quatre paramètres dont deux ont une valeur par défaut. Elle se contente d'afficher les valeurs reçues – remarquez à cette occasion l'opérateur `&`, qui permet de réaliser la concaténation de chaînes de caractères : l'équivalent du `+` entre chaînes en Python ou du `.` en Perl.

La première invocation, ligne 15, ne présente rien de particulier : la position de chaque paramètre réel détermine à quel paramètre formel il correspondra. La deuxième, par contre, montre comment il est possible de nommer les paramètres : ceux-ci sont donnés en ordre inverse, mais le résultat est exactement identique à la première. Pour nommer un paramètre, il suffit de donner le nom de celui-ci, puis le symbole `=>`, puis la valeur voulue. Lorsque l'on invoque un sous-programme recevant de nombreux paramètres, cette écriture permet d'améliorer considérablement la lisibilité du code.

Rassurez-vous, il n'y a pas de confusion possible entre noms de paramètres et noms de variables : ce qui est à gauche de `=>` est forcément un nom de paramètre, ce qui est à droite forcément une valeur ou une variable. Ainsi rien n'interdit de définir, par exemple, une variable nommée `b` dans le programme précédent et de la passer à la procédure avec une écriture comme `proc(a=>1, b=>b, c=>3, d=>4);`. Peut-être le membre `b=>b` vous paraît-il étrange, mais le compilateur s'y retrouve sans ambiguïté. Par contre, l'utilisation du *nommage* des paramètres impose une certaine précision : si vous donnez un nom qui n'existe pas dans la liste des paramètres du sous-programme, le compilateur renverra une erreur. Enfin, la dernière invocation ligne 17 montre que les paramètres `b` et `d` peuvent être omis

(puisqu'ils ont une valeur par défaut) et qu'il est possible de mélanger paramètres positionnés et paramètres nommés. En fait, du fait de la forme de la déclaration de la procédure, le nommage du paramètre `c` est pratiquement obligatoire si `b` est absent. En effet, une écriture comme `proc(1,2);` serait incorrecte : par le positionnement, la première valeur sera attribuée au paramètre `a`, la deuxième au paramètre `b`... et de fait, le paramètre `c` n'aura pas de valeur, alors qu'il en requiert une.

À noter qu'à partir du point où vous utilisez le nommage de paramètres, les paramètres suivants doivent obligatoirement être nommés, l'attribution des valeurs par simple positionnement n'étant plus possible. Voici ce qu'affiche ce programme à l'exécution :

```
$ ./default
a = 1, b = 2, c = 3, d = 4
a = 1, b = 2, c = 3, d = 4
a = 1, b = 22, c = 3, d = 44
```

Déclaration et récursivité

Comme tout langage de haut niveau, Ada supporte la récursivité : un sous-programme peut s'invoquer lui-même, deux ou plusieurs sous-programmes peuvent s'inter-invoquer. Mais pour qu'un sous-programme puisse en invoquer un autre, encore faut-il que le deuxième ait été déclaré au moment où apparaît le code du premier. La récursivité croisée n'est alors possible que par l'usage de la déclaration anticipée des sous-programmes. Par exemple :

```
 1 procedure recurse is
 2    procedure p1 ;
 3    procedure p2 ;
 4    procedure p1 is
 5    begin
 6       p2 ;
 7    end ;
 8    procedure p2 is
 9    begin
10       p1 ;
11       p2 ;
12    end ;
13 begin
14    p1 ;
15 end ;
```

Deux procédures, `p1` et `p2`, sont déclarées lignes 2 et 3. Au moment où arrive le code de `p1`, celle-ci peut invoquer `p2` (ligne 6), puisqu'elle a déjà été déclarée. D'un autre coté, la déclaration de `p1` n'est pas ici absolument nécessaire pour qu'elle soit invoquée dans `p2` (ligne 10) : la définition de `p1`, lignes 4 à 7, vaut déclaration.

Les plus observateurs auront remarqué que ce programme présente deux problèmes majeurs : d'une part, `p1` et `p2` vont s'invoquer l'une l'autre sans fin, ce qui en soit suffirait à provoquer un débordement de pile et donc un plantage du programme ; de plus, `p2` s'invoque elle-même également sans fin, avec les mêmes conséquences. Le compilateur Gnat a la gentillesse de nous prévenir de la possible récursivité infinie sur `p2` :

```
$ gnatmake recurse.adb
gcc-3.4 -c recurse.adb
recurse.adb:11:07: warning: possible infinite recursion
recurse.adb:11:07: warning: Storage_Error may be raised at
run time
gnatbind -x recurse.ali
gnatlink recurse.ali
```

Remarquez l'avertissement sur la ligne 11. Par contre, la récursivité croisée infinie ne sera malheureusement pas détectée.

Si vous exécutez ainsi ce programme, cela résultera en une erreur de segmentation (`Segmentation fault`), donc un plantage brutal. Nous verrons les exceptions plus tard, mais sachez qu'il est possible de prévenir ce genre de soucis en demandant un contrôle à l'exécution sur l'encombrement de la pile, au moyen du paramètre `-fstack-check` passé à `gnatmake` :

```
$ gnatmake -fstack-check recurse.adb
gcc-3.4 -c -fstack-check recurse.adb
recurse.adb:11:07: warning: possible infinite recursion
recurse.adb:11:07: warning: Storage_Error may be raised at
run time
gnatbind -x recurse.ali
gnatlink recurse.ali
$ ./recurse
raised STORAGE_ERROR : stack overflow detected
```

Le programme ne se termine plus brutalement lors du débordement de pile. Au lieu de cela, une exception est déclenchée (comme annoncé dans l'avertissement, d'ailleurs), ce qui laisse la possibilité de réagir dans le programme.

Comme aucun traitement n'a été prévu pour cela dans notre code, le programme se termine tout de même. Le prix de cette sécurité étant, naturellement, un code un peu plus gros et un peu plus lent. Exercice : essayez d'intercepter ce genre d'erreur en C++.

Renommer un sous-programme

Pour terminer, voyons une possibilité offerte par Ada pour renommer un sous-programme :

```
 1 with Text_IO ;
 2 use Text_IO ;
 3 procedure Renomme is
 4    procedure Une_Procedure_Avec_Un_Nom_Impossible is
 5    begin
 6       Put_Line("Hello !");
 7    end ;
 8    procedure Proc
 9       renames Une_Procedure_Avec_Un_Nom_Impossible ;
10
11    function Une_Fonction_Qui_N_Est_Pas_Mieux (i: in Integer)
12       return Integer is
13    begin
14       Put_Line("i = " & Integer'Image(i)) ;
15       return 0 ;
16    end ;
17    function Fonct(i: in Integer) return Integer
18       renames Une_Fonction_Qui_N_Est_Pas_Mieux ;
19
20 a: Integer ;
21 begin
22    Proc ;
23    a := Fonct(1) ;
24 end Renomme ;
```

Les lignes 4 à 7 définissent une procédure, dont le nom est un peu pénible à taper au clavier. Les lignes 11 à 16 définissent une fonction, dont le nom n'est guère plus sympathique.

Il est possible d'éviter des crampes aux doigts en renommant ces deux sous-programmes. Le renommage revient en fait à déclarer un nouveau sous-programme, en indiquant que le nom est un synonyme d'un autre sous-programme préalablement déclaré.

Voyez les lignes 8-9 d'une part et 17-18 d'autre part : les premières définissent un synonyme pour la procédure, les secondes un synonyme pour la fonction. Ces définitions prennent la forme de déclarations usuelles, avec toutefois le mot clef `renames` suivi du nom du sous-programme ainsi renommé.

Si l'intérêt de la technique ne paraît pas évident sur cet exemple, nous verrons qu'elle peut s'avérer fort appréciable quand nous aborderons les paquetages Ada, c'est-à-dire en gros les bibliothèques.

Conclusion

Voilà pour cette présentation des sous-programmes en Ada. Étonnamment, nous avons vu que le typage strict de Ada, s'il impose certaines contraintes fortes, offre également une certaine souplesse dans la déclaration de sous-programmes et permet aux compilateurs de détecter certains problèmes.

Le mois prochain, nous continuerons la découverte du langage avec les structures de contrôles que sont les boucles et autres alternatives.

Par :: Yves Bailly :: http://www.kafka-fr.net ::

Structures de contrôle en Ada

L'introduction de cet article pourrait reprendre le début de celle de l'article du mois dernier : à moins d'être trivial, un programme n'est jamais parfaitement linéaire. Nous allons découvrir aujourd'hui comment contrôler le flux d'instructions, comment choisir entre différentes possibilités ou répéter une portion de code.

Petite remarque avant d'aborder le sujet du jour. Vous trouverez, dispersées dans le texte, des références comme [AARM 5.8-4.b] ou [AARM J.5].

Ce sont des références au Manuel de Référence Ada Annoté ou *Annotated Ada Reference Manual*, pour vous permettre de retrouver la section du standard Ada (ISO/IEC 8652:1995-E) en relation avec le sujet traité.

Le premier nombre est le numéro de chapitre (ou la lettre d'une annexe, dans le deuxième exemple), le deuxième le numéro de section, éventuellement suivi par le numéro de clause (paragraphe). Utilisez ces informations si vous voulez plus de détails, mais cela n'a rien d'obligatoire !

Ce qui figure dans ces articles est normalement suffisant pour nos besoins. Vous trouverez un exemplaire du AARM sur le CD accompagnant le magazine ou par le lien donné en [1].

Le bloc en Ada

Grossièrement, on peut dire qu'un bloc en Ada est une suite d'instructions entre un `begin` et un `end` correspondant, introduite par certains mots-clefs – par exemple, `procedure` ou `function` que nous avons vus le mois dernier.

Si des variables locales doivent être utilisées dans les instructions du bloc, elles doivent être déclarées entre le mot-clef introductif et le `begin` qui suit : il n'est pas possible de déclarer une variable entre deux instructions comme en C++.

Mais il est parfois souhaitable de ne déclarer une variable qu'à un certain moment, dans une suite d'instructions et non dès le début d'un sous-programme. Il existe pour cela le mot-clef `declare` [AARM 5.6], par exemple :

```
 1 procedure Ma_Proc is
 2    v1: Integer ;
 3    procedure Autre_Proc is
 4       v2: Integer ;
 5    begin
 6       -- blablabla...
 7       null ;
 8    end ;
 9    v3: Integer ;
10 begin
11    -- du code...
12    null ;
13    declare
14       v4: Float ;
15       procedure Sous_Proc is
16          v5: Float ;
17       begin
18          -- des instructions...
19          null ;
20       end ;
21    begin
22       -- encore du code...
23       null ;
24    end ;
25    -- et pour finir...
26    null ;
27 end ;
```

Ceci est du code Ada parfaitement valide, il peut être compilé. Quelques remarques :

- La procédure `Autre_Proc`, déclarée ligne 3, est encadrée par la déclaration des variables `v1` et `v3`. Tout naturellement, `Autre_Proc` peut utiliser `v1`, mais pas `v3`.
- Tout aussi naturellement, `v2` n'existe qu'à l'intérieur de `Autre_Proc`, y faire référence dans le code de `Ma_Proc` (de la ligne 10 à la ligne 27) serait une erreur.
- Ces lignes, justement, contiennent deux blocs imbriqués : le premier commence ligne 13 et s'étend jusque ligne 24, le second va de la ligne 15 à ligne 19 (il s'agit en fait d'une procédure). Le premier apparaît effectivement au milieu d'autres instructions (lignes 12 et 26). Là encore, les règles de visibilité usuelles s'appliquent : `v4` n'existe qu'entre les lignes 14 et 24, `v5` n'existe qu'entre les lignes 16 et 20. `v1`, `Autre_Proc` et `v3` sont par contre utilisables.

Remarquez qu'il est parfaitement possible de déclarer un sous-programme dans un autre, voire dans un bloc au sein d'un sous-programme qui se trouve dans un sous-programme contenu dans un bloc...

Il n'y a pas de limite théorique à la « profondeur d'imbrication » des blocs. Nous reviendrons plus tard sur les blocs, mais passons maintenant aux outils qui permettent d'agir sur le flux d'instructions du programme.

Goto

Le presque inévitable `goto`, certains disent « l'infâme `goto` », est disponible en Ada. Inutile de s'appesantir sur son utilité, s'il est bien ou mal de s'en servir.

`goto` existe au moins formellement depuis les origines de l'algorithmique avec le mathématicien perse Abu Abdullah Muhammad bin Musa al-Khwarizmi, au

IXème siècle [2], ce qui ne nous rajeunit pas. Alors voici un exemple d'utilisation, avec cinq erreurs, pour montrer qu'on ne peut quand même pas faire n'importe quoi :

```
 1 procedure P is
 2    procedure PP is
 3    begin
 4       <<LABEL_1>>
 5       goto LABEL_1 ;
 6 --      goto LABEL_2 ;  -- erreur !
 7 --      goto LABEL_3 ;  -- erreur !
 8 --      goto LABEL_4 ;  -- erreur !
 9    end ;
10 begin
11    <<LABEL_2>>
12    declare
13       v: Integer ;
14    begin
15       <<LABEL_3>>
16 --      goto LABEL_1 ;  -- erreur !
17       goto LABEL_2 ;
18       goto LABEL_3 ;
19       goto LABEL_4 ;
20    end ;
21 --   goto LABEL_1 ;     -- erreur !
22    goto LABEL_2 ;
23 --   goto LABEL_3 ;     -- erreur !
24    goto LABEL_4 ;
25    <<LABEL_4>>
26    null ;
27 --   goto LABEL_3 ;     -- erreur !
28 end ;
```

Les noms des labels, destination des branchements par `goto`, sont donnés entre double chevrons (ligne 4, par exemple). Ici les noms sont en majuscules, mais c'est par pure convention : vous êtes libres de les nommer comme bon vous semble.

Les erreurs illustrent la règle générale qui gouverne les sauts par `goto` : il est interdit de sortir d'un bloc pour entrer dans un autre qui ne contient pas le bloc de départ [AARM 5.8-6]. En dehors de cette règle, toutes les horreurs sont permises.

L'instruction `null` en ligne 26 est imposée par la syntaxe : un label doit forcément être suivi par une instruction.

Branchement conditionnel

Voyons maintenant quelque chose de plus familier, le branchement conditionnel [AARM 5.3] :

```
 1 procedure P is
 2    i: Integer ;
 3 begin
 4    i := 8 ; -- ou autre chose...
 5    if i = 1
 6    then
 7       -- si i vaut 1, alors ...
 8       null ;
 9    elsif i > 2
10    then
11       -- si i est supérieur à  2, alors...
12       null ;
13    else
14       -- dans tous les autres cas
15       null ;
16    end if ;
17 end ;
```

Les parenthèses autour de la condition qui suit le `if` ne sont pas nécessaires. Par contre, le mot-clef `then` (alors) est indispensable.

Remarquez qu'il est possible d'effectuer plusieurs tests imbriqués, enchaînés en fait, au moyen de `elsif` (Python utilise `elif` pour le même but). Les instructions suivants le `else` final ne sont exécutées que si tous les tests précédents ont échoués. Enfin, l'ensemble de la structure se termine par un `end if`.

Ici aussi, les `nulls` sont imposés par la syntaxe. Par ailleurs, le type du résultat de l'expression qui figure dans les lignes `if` ou `elsif` doit être d'un type booléen, c'est-à-dire d'un type issu de Boolean (ou de Boolean lui-même). À noter qu'en Ada, une affectation avec `:=` n'est pas une expression : c'est une instruction et elle n'a pas de valeur. Donc, si vous écrivez par maladresse quelque chose comme `if i := 1`, vous serez gratifié d'une rouspétance du compilateur. La célèbre confusion entre `=` et `==` qui survient si souvent en C/C++ est ici tout simplement impossible. Voici l'occasion de dresser une liste des opérateurs booléens et de comparaison en Ada, par ordre de précédence croissante (certains des opérateurs suivants sont également utilisés dans d'autres contextes, que nous verrons par la suite) :

Listes des opérateurs Booléens

OPÉRATEUR	OPÉRATION	TYPE DES OPÉRANDES	TYPE DU RÉSULTAT
`and`, `or`, `xor`	ET, OU, OU exclusif logiques	Boolean tableau de Boolean	Boolean tableau de Boolean
`and then`, `or else`	voir plus bas !	Boolean	Boolean
`= /=`	égalité, différence	Quelconque	Boolean
`<`, `<=`, `>`, `>=`	inférieur, inférieur ou égal, supérieur, supérieur ou égal	Scalaire (numérique) tableau unidimensionnel	Boolean
`in`, `not in`	appartenance à un intervalle ou a un type	scalaire – intervalle quelconque - nom d'un type	Boolean
`not`	Négation	Boolean tableau de Boolean	Boolean tableau de Boolean

Sont donnés ici les opérateurs qui produisent un résultat sous forme de tableau, mais comme nous ne verrons les tableaux que le mois prochain, n'en tenez pas plus compte (c'est juste par souci de complétude). La deuxième ligne appelle probablement quelques commentaires. Considérons deux expressions, X et Y, à valeur booléenne. Ce peuvent être de simples variables ou bien quelque chose de beaucoup plus compliqué contenant des appels de fonctions – n'importe quoi qui vaut une valeur booléenne. En C/C++, l'équivalent de X and Y est X && Y et dans ce cas le terme Y n'est évalué que si le terme X est vrai. Pas en Ada : dans l'expression X and Y, les deux opérandes sont toujours évalués dans un ordre arbitraire. En fait, ceci est vrai pour tous les opérateurs... sauf les deux de la deuxième ligne.

`X and then Y` donne le même résultat que `X and Y`, à ceci prêt que X est toujours évalué avant Y, ce dernier n'étant évalué que si X est vrai – on retrouve là un comportement équivalent à l'opérateur `&&` du C/C++. De la même manière, `X or else Y` donne le même résultat que `X or Y`, sauf que X est évalué en premier et que Y n'est évalué que si X est faux – comme le `||` du C/C++. [AARM 4.5.1-7]

Si le mode de fonctionnement des opérateurs `and`, `or` et consorts vous étonne, sachez que ces opérateurs sont en réalité de simples raccourcis pour des fonctions de la forme suivante :

```
function "and"(Left, Right : Boolean) return Boolean ;
function "or" (Left, Right : Boolean) return Boolean ;
function "xor"(Left, Right : Boolean) return Boolean ;
```

Et ainsi de suite. Or, dans un appel de fonction, les expressions pour chacun des paramètres sont évaluées dans un ordre

quelconque – la plus à droite peut-être avant la plus à gauche. Il est donc tout à fait logique que les opérateurs correspondants suivent la même « incertitude ».

Incidemment, puisque nous avons là des fonctions (bien qu'avec un nom un peu particulier, une chaîne de caractères), cela signifie que nous pouvons les surdéfinir pour nos propres types et nos propres besoins. Il n'est par contre pas permis de définir de nouveaux opérateurs, donc une déclaration comme :

```
function "machin" (a, b: Un_Type) return Un_Autre_Type ;
```

est interdite.

De nombreux autres langages de programmation font souvent l'économie du mot `then` et parfois même de la terminaison par `end if`.

Ils sont obligatoires en Ada. Si cela vous paraît une lourdeur inutile, rappelons que Ada a été conçu pour la réalisation de programmes complexes, avec une très forte exigence de fiabilité et susceptibles de maintenance et modifications sur plusieurs années par de nombreuses personnes différentes.

En sacrifiant la légèreté à la clarté, les concepteurs de Ada ont estimé que ces objectifs seraient plus facilement satisfaits.

Le choix multiple

Il s'agit ici de déterminer les instructions à exécuter selon la valeur d'une expression parmi une série de valeurs possibles.

On peut réaliser cela en empilant des `if..then..elsif..else..end if`, mais la structure `case` [AARM 5.4] est précisément dédiée à une telle situation :

```
 1 with Text_IO ; use Text_IO ;
 2 procedure P is
 3    i: Integer := 7 ;
 4 begin
 5    case i is
 6       when 1 =>
 7          Put_Line("C'est 1 !") ;
 8       when 2..5 =>
 9          Put_Line("Entre 2 et 5 !") ;
10       when 6 | 8 | 10 =>
11          Put_Line("6, 8, ou 10 !") ;
12       when 9 | 11..15 | 7 =>
13          Put_Line("Heu, compliqué...") ;
14       when others =>
15          Put_Line("Cas général.") ;
16    end case ;
17 end
```

L'exemple précédent, disons-le immédiatement, ne montre pas toute la richesse possible de cette structure – mais il donne une idée que j'espère assez alléchante. Il n'aura échappé à personne qu'il s'agit là de l'équivalent de la structure `switch...case` du C/C++ ou de Java (entre autres), avec tout de même quelques nuances de taille. Pour commencer, la structure s'écrit plutôt `case..when`.

Ensuite, remarquez l'absence d'équivalent au `break` du C : contrairement à ce langage (ou à C++ ou à Java...), chacun des cas du choix multiple s'arrête là où commence le suivant.

Par ailleurs, si les valeurs déterminantes du choix doivent être statiques (c'est-à-dire connues à la compilation), il est possible de regrouper plusieurs valeurs en une seule expression : la ligne 8 définit un intervalle de valeurs, la ligne 10 une liste, la ligne 12 combine les deux procédés. Ainsi :

- La ligne 9 ne sera exécutée que si i est compris entre 2 et 5 inclus ;
- La ligne 11 ne sera exécutée que si i vaut 6, 8 ou 10 ;
- La ligne 12 ne sera exécutée que si i est compris entre 11 et 15 inclus ou bien vaut 7 ou bien vaut 9.

La dernière clause ligne 14, `when others`, n'est exécutée que si i ne rentre dans aucun des cas précédents. Cette clause est optionnelle, mais si elle est présente elle doit forcément être la dernière. Enfin, la structure se termine par un `end case`.

Ici une seule instruction est donnée pour chaque cas. Rien ne vous empêche d'en donner plusieurs. Si vous avez besoin de déclarer une variable, il faudra avoir recours au `declare..begin..end` que nous avons vu au début de cet article.

Pour information, un `goto` ne permet pas de « sauter » d'un choix à un autre.

Les boucles générales

Comme dans la plupart des langages, Ada propose différentes manières d'écrire les boucles. Voici la même boucle sous trois formes différentes :

```
1 procedure P is
2    i: Integer := 0 ;
3 begin
4    i := 0 ;
5    loop
6       -- des instructions...
7       i := i + 1 ;
8       if i > 5
9       then
10         exit ;
11      end if ;
12   end loop ;
13   --
14   i := 0 ;
15   loop
16      -- instructions...
17      i := i + 1 ;
18      exit when i > 5 ;
19   end loop ;
20   --
21   i := 0 ;
22   while not (i > 5)
23   loop
24      -- du code...
25      i := i + 1 ;
26   end loop ;
27 end ;
```

Une boucle est essentiellement définie par des instructions comprises entre les mots `loop` et `end loop` [AARM 5.5]. La différence réside en fait dans la façon de noter la condition de sortie de la boucle.

Dans le premier exemple, lignes 5 à 12, on fait simplement appel au mot-clef `exit` [AARM 5.7] au sein d'une condition : celui-ci permet de sortir de la boucle la plus proche qui le contient.

Mais comme cette forme est très commune, elle peut être abrégée comme montré dans le deuxième exemple, ligne 18 : l'effet est rigoureusement le même, mais avec moins de caractères à taper au clavier.

À noter que dans ces deux exemples, si le flux de code ne tombe jamais sur une instruction `exit`, la boucle tournera jusqu'à la fin des temps.

Enfin, le dernier exemple (lignes 22 à 26) fait intervenir le mot-clef `while`.

Cette boucle est identique aux précédentes, à ceci près que la condition est évaluée avant l'exécution des instructions dans le corps de la boucle.

La boucle for

Les boucles précédentes pouvaient être qualifiées de non bornées : elles ne s'arrêtent que lorsqu'une certaine condition est remplie, c'est-à-dire potentiellement jamais. Ada propose une dernière forme de boucle, faisant intervenir un compteur [AARM 5.5-9] :

```
1 procedure P is
2    type Entier is new Integer ;
3 begin
4    for i in 1..5
5    loop
6       -- du code...
7       null ;
8    end loop ;
9    for i in Entier range 1..5
```

```
10    loop
11       -- des instructions...
12       null ;
13    end loop ;
14 end ;
```

Contrairement à l'instruction du même nom en C/C++, l'instruction `for` n'est pas un simple synonyme pour une boucle fondée sur un `while`.

D'abord, le compteur (i dans les deux exemples) n'est pas n'importe quelle variable : remarquez qu'il n'est pas déclaré avant la première boucle.

En fait, l'en-tête introduit par `for` vaut implicitement déclaration de la variable qui suit. Cette variable n'existe que pour la durée de la boucle : c'est pourquoi nous pouvons réutiliser le nom i ligne 9, il s'agit d'une toute autre variable.

Ensuite, ce compteur doit obligatoirement être d'un type discret, c'est-à-dire en gros équivalent à un entier. Le type du compteur est défini par le type des bornes de l'intervalle qu'il devra parcourir, lequel intervalle est défini après le mot-clef `in` : ainsi, la ligne 4 définit un compteur nommé i, implicitement de type `Integer`, qui va parcourir successivement les valeurs de 1 à 5 incluses.

La ligne 9 définit un compteur nommé i de type Entier (revoyez éventuellement le premier article de cette série sur la définition d'un type dérivé), qui va parcourir la même plage de valeurs.

Les bornes de l'intervalle ne sont pas nécessairement des valeurs littérales : 1 et 5 pourraient parfaitement être remplacées par des variables, pourvu qu'elles soient de même type. La plage de valeurs ainsi définie peut être nulle, par exemple 2..0 : dans ce cas, les instructions dans la boucle ne sont pas exécutées.

Par ailleurs, les bornes de l'intervalle ne sont évaluées qu'une seule fois, au moment où le `for` est rencontré : cela signifie que si l'une des deux bornes (ou les deux) est une variable, modifier la valeur de cette variable à l'intérieur de la boucle n'aura aucune influence sur le déroulement de celle-ci (contrairement à ce qu'il est possible de faire en C/C++, par exemple). Enfin, si le `in` est suivi du mot-clef `reverse`, les valeurs de l'intervalle sont parcourues en ordre inverse (de la fin vers le début).

Pour finir, au sein de la boucle le compteur est considéré comme une constante : il est interdit de modifier sa valeur par une affectation.

Toute tentative dans ce but sera rejetée par le compilateur. Il est par contre toujours possible d'utiliser l'une des formes du mot-clef `exit` si on souhaite sortir prématurément de la boucle.

On pourra regretter une petite limitation, à savoir que les valeurs de l'intervalle sont forcément parcourues par incréments de 1 : la boucle `for` ne permet pas, par exemple, de n'envisager qu'une valeur sur deux. Pour cela, il faut se rabattre sur la forme générale basée sur `while`.

Un nom pour un bloc

Nous en arrivons à une possibilité très intéressante offerte par Ada : celle de nommer un bloc. Voici un premier exemple, où un même nom de variable a bêtement été utilisé dans trois blocs imbriqués :

```
 1 with Text_IO ; use Text_IO ;
 2 procedure P is
 3    i: Integer ;
 4 begin
 5    i := 1 ;
 6    declare
 7       i: Integer ;
 8       procedure PP is
 9          i: Integer := 3 ;
10       begin
11          Put(Integer'Image(i)&" ") ;
12       end ;
13    begin
14       i := 2 ;
15       PP ;
16       Put(Integer'Image(i)&" ") ;
17    end ;
18    Put_Line(Integer'Image(i)) ;
19 end ;
```

Ce n'est pas très malin, mais parfois la vie mouvementée d'un programme compliqué abouti à ce genre de chose. Dans ces situations, chaque nouvelle déclaration de i « masque » la déclaration précédente : le i de la ligne 11 est celui déclaré ligne 9, celui des lignes 14 et 16 est celui déclaré ligne 7, celui de la ligne 18 est déclaré ligne 5.

Il n'y a aucune ambiguïté. Mais comment faire alors, si on veut utiliser dans PP le i déclaré ligne 5 ou celui ligne 7 ? Utiliser dans le bloc entre les lignes 6 et 17 le i déclaré ligne 5 ?

Il est possible de préfixer une variable avec le nom du bloc dans lequel elle est définie, en utilisant la notation pointée. Dans le cas d'une procédure ou d'une fonction, son nom est celui du bloc : par exemple, si on écrivait `P.i` ligne 11, on ferait référence au i déclaré ligne 3. Il est par contre évidemment aberrant d'écrire `PP.i` ligne 14, car la variable i déclarée ligne 9 n'existe pas à cet endroit : ce qui est déclaré dans un bloc est invisible en dehors de celui-ci.

Voyons maintenant le cas du i déclaré ligne 7. Nous ne sommes pas dans une procédure : le bloc en question est anonyme.

Pour lui donner un nom, il suffit de faire précéder le `declare` ligne 6 par un identifiant suivi des deux-points. Voici le même programme, le bloc en question étant nommé :

```
 1 with Text_IO ; use Text_IO ;
 2 procedure P is
 3    i: Integer ;
 4 begin
 5    i := 1 ;
 6    Nom_Du_Bloc:
 7    declare
 8       i: Integer ;
 9       procedure PP is
10          i: Integer := 3 ;
11       begin
12          Put(Integer'Image(Nom_Du_Bloc.i)&" ") ;
13       end ;
14    begin
15       i := 2 ;
16       PP ;
17       Put(Integer'Image(i)&" ") ;
18    end Nom_Du_Bloc ;
19    Put_Line(Integer'Image(i)) ;
20 end ;
```

Remarquez la ligne 6, qui donne un nom au bloc. Dans ce cas, il est obligatoire de répéter ce nom après le `end` qui termine le bloc, comme cela est fait ligne 18 : si vous oubliez cette répétition, le compilateur vous rappellera à l'ordre. Dès lors, on peut faire référence aux variables de ce bloc explicitement, par exemple ligne 12.

Mieux que l'accès aux variables locales, cette manière de nommer les blocs permet de réaliser des branchements très intéressants, qui seraient assez pénibles à reproduire dans de nombreux autres langages.

Ceci grâce à la possibilité de nommer les structures de contrôle, notamment les boucles que nous avons vues précédemment.

Voyez cet exemple :

```
1 with Text_IO ; use Text_IO ;
2 procedure P is
3 begin
4    put_line("Avant Boucle_1") ;
5    Boucle_1:
```

```
 6   for i in 11..19
 7   loop
 8     put_line("   Avant Boucle_2") ;
 9     Boucle_2:
10     for j in 21..29
11     loop
12       put_line("      Avant Boucle_3") ;
13       Boucle_3:
14       for k in 31..39
15       loop
16         if i > 12
17         then
18           exit Boucle_1 ;
19         end if ;
20         exit Boucle_2 when j > 22 ;
21         exit when k > 32 ;
22         put_line("         "&
23           Integer'Image(i)&" "&
24           Integer'Image(j)&" "&
25           Integer'Image(k)) ;
26       end loop Boucle_3 ;
27       put_line("      Après Boucle_3") ;
28     end loop Boucle_2 ;
29     put_line("   Après Boucle_2") ;
30   end loop Boucle_1 ;
31   put_line("Après Boucle_1") ;
32 end ;
```

Nous avons là trois boucles `for` imbriquées (mais ceci s'applique également aux autres types de boucles), chacune programmée pour faire 9 tours, donc un total théorique de 729 exécutions du code le plus intérieur (lignes 16 à 25).

Chacune des boucles est nommée (lignes 5, 9 et 13), les noms étant répétés au moment de terminer le bloc de la boucle (lignes 26, 28 et 30).

Ici aussi cette répétition est obligatoire, imposée par le langage [AARM 5.5-5].

Ce programme n'effectuera pas 729 tours de boucles : les lignes 16 à 21 provoquent des sorties prématurées de chacune des boucles.

Remarquez comment nous sortons de la boucle la plus extérieure à partir du code exécuté dans la boucle la plus intérieure : l'instruction `exit` ligne 18 est tout simplement suivie du nom du bloc duquel nous voulons sortir.

De même pour la ligne 20, qui utilise la forme abrégée de `exit`.

En l'absence de nom, la ligne 21 provoquera implicitement la sortie de la boucle la plus proche, c'est-à-dire la troisième (dont le nom ne sert en fait à rien).

Saurez-vous deviner l'affichage de ce programme ?

Le voici :

```
Avant Boucle_1
   Avant Boucle_2
      Avant Boucle_3
          11  21  31
          11  21  32
      Après Boucle_3
      Avant Boucle_3
          11  22  31
          11  22  32
      Après Boucle_3
      Avant Boucle_3
   Après Boucle_2
   Avant Boucle_2
      Avant Boucle_3
          12  21  31
          12  21  32
      Après Boucle_3
      Avant Boucle_3
          12  22  31
          12  22  32
      Après Boucle_3
      Avant Boucle_3
   Après Boucle_2
   Avant Boucle_2
      Avant Boucle_3
Après Boucle_1
```

Remarquez les deux dernières lignes qui montrent bien que l'on entre dans la troisième boucle (la plus intérieure), pour sortir directement après la boucle la plus extérieure du fait des lignes 16 à 19.

Des langages conçus bien après Ada, comme Java ou Perl, ont retenu cette technique, sans aller pourtant jusqu'à nommer des blocs de nature similaire à `declare`. On la trouve également sous une forme amoindrie, par exemple dans PHP ou le *shell* Bash, où il est possible de donner un nombre de niveaux d'imbrications à « dépiler » à la suite de l'équivalent de l'instruction `exit`. Les langages les plus utilisés que sont le C et le C++ n'ont par contre rien de comparable. Au-delà des facilités et perspectives qu'ouvre le nommage des blocs, son utilisation permet bien souvent de clarifier un code autrement difficile à suivre et facilite grandement la maintenance : ajouter ou retirer un niveau d'imbrication est moins problématique si chaque niveau est clairement nommé et identifié. En plus d'apporter une grande souplesse et une grande rigueur, nommer les blocs conduit à écrire du code « auto-documenté ».

Conclusion

Voilà pour cette présentation des structures de contrôle fondamentales du langage Ada. Il en existe de plus sophistiquées, si l'on considère la gestion des exceptions et le parallélisme comme des structures de contrôle – nous les explorerons plus tard. Le mois prochain, nous aborderons les structures de données que sont les tableaux et les enregistrements.

Références

[1] *Annotated Ada Reference Manual (AARM)*, http://adaic.org/standards/ada95.html

[2] Les algorithmes, http://en.wikipedia.org/wiki/Algorithm

Par :: Yves Bailly :: http://www.kafka-fr.net ::

Le langage Ada 4 : utilisation des tableaux

Après avoir examiné les structures de contrôle le mois dernier, nous allons maintenant commencer à découvrir ce qui fait la véritable force du langage Ada : ses structures de données fondamentales, aussi simples que souples. Mais avant cela, je voudrais vous toucher un mot de la prochaine version du langage.

Ada 2005... ou 2006 ? Il semble en effet que le comité de standardisation estime que la prochaine norme Ada ne pourra être validée que début 2006.

La prochaine version du langage sera donc probablement désignée par « Ada2006 » (mais le nom « officiel » du langage demeure simplement Ada).

Voici un très bref aperçu de ce qui va changer [1], du moins ce qui me paraît le plus remarquable :

- Support pour les standards Unicode 2.0 et 4.0 jusque dans le code source (une constante écrite littéralement « π » a même été définie) ;
- Le principe des interfaces, introduit notamment par Java, permettra une forme limitée d'héritage multiple ;
- La bibliothèque standard du langage proposera des types conteneurs (listes, tableaux associatifs...), des fonctions et procédure de manipulation des fichiers et répertoires, ainsi que des outils mathématiques comme les vecteurs et les matrices ;
- Diverses améliorations pour les applications et systèmes multitâches et critiques.

Une bonne partie de tout cela est déjà disponible dans la dernière version du compilateur GNAT fournie avec la collection GCC 4.0, sortie tout récemment.

Si la lecture d'un standard ne vous effraie pas trop, une ébauche de ce que sera le prochain standard Ada [2] est disponible.

En attendant sa finalisation, nous continuerons à nous concentrer sur la norme Ada 95.

Types énumérés

Avant d'attaquer les tableaux, voyons les possibilités pour définir un type énuméré – similaire aux `enum` du C/C++.

Il s'agit de définir un type en donnant la liste exhaustive des valeurs qui le composent, sous la forme d'identifiants. Par exemple :

```
type Rgb is (Rouge, Vert, Bleu) ;
```

Ceci définit simplement un type nommé `Rgb`, dont les valeurs sont `Rouge`, `Vert` et `Bleu`. Ces valeurs sont implicitement ordonnées : Rouge < Vert est vrai, mais Vert > Bleu est faux. En fait, ce type `Rgb` est un type discret ressemblant fortement à un entier. Pour ce genre de types (tous les types énumérés et tous les types entiers), un ensemble d'attributs fonctionnels sont disponibles. Ils sont résumés dans le tableau ci-dessous, agrémenté d'un exemple.

Essayez de deviner ce qu'affiche ce petit programme...

Voici :

```
$ gnatmake enum.adb
$ ./enum
ROUGE
1
BLEU
ROUGE
BLEU
VERT
VERT
ROUGE
BLEU
```

:: Tab. 1 :: Attributs pour les types discrets ::

```
1 with Text_IO ; use Text_IO ;
2 procedure Enum is
3    type Rgb is (Rouge, Vert, Bleu) ;
4 begin
5    Put_Line(Rgb'Image(Rouge)) ;
6    Put_Line(Integer'Image(Rgb'Pos(Vert))) ;
7    Put_Line(Rgb'Image(Rgb'Val(2))) ;
8    Put_Line(Rgb'Image(Rgb'First)) ;
9    Put_Line(Rgb'Image(Rgb'Last)) ;
10   Put_Line(Rgb'Image(Rgb'Succ(Rouge))) ;
11   Put_Line(Rgb'Image(Rgb'Pred(Bleu))) ;
12   Put_Line(Rgb'Image(Rgb'Min(Rouge, Bleu))) ;
13   Put_Line(Rgb'Image(Rgb'Max(Rouge, Bleu))) ;
14 end Enum ;
```

`Image(a)`	Donne une chaîne de caractères représentant la valeur.
`Pos(a)`	Indice de la valeur dans la liste des valeurs du type.
`Val(i)`	Valeur du type d'indice `i` dans la liste des valeurs.
`First`	Première (plus petite) valeur du type.
`Last`	Dernière (plus grande) valeur du type.
`Succ(a)`	Valeur suivante de la valeur `a`.
`Pred(a)`	Valeur précédente de la valeur `a`.
`Min(a, b)`	La plus petite des deux valeurs données.
`Max(a, b)`	La plus grande des deux valeurs données.

N'est-ce pas merveilleux ? Les valeurs sont affichées, sans qu'il soit besoin de recourir à une fonction de correspondance laborieuse pour obtenir des chaînes. Du tableau précédent, on peut affirmer que, quelle que soit une valeur `X` d'un type discret `T`, on a toujours :

```
T'Val(T'Pos(X)) = X
T'Succ(X) = T'Val(T'Pos(X)+1)
T'Pred(X) = T'Val(T'Pos(X)-1)
```

Les positions des valeurs sont indicées à partir de 0. Naturellement, si on cherche à prendre la valeur précédente de la première valeur ou la valeur suivante de la dernière, on risque des ennuis, plus précisément la levée d'une exception nommée `Constraint_Error`. Par ailleurs, il n'est pas possible d'associer explicitement une valeur numérique à une valeur symbolique, comme cela se fait couramment en C/C++.

Nous allons maintenant voir que ces types énumérés peuvent servir à autre chose que simplement servir de réceptacles à des valeurs symboliques.

Les tableaux à une dimension

En Ada, il est d'usage de déclarer un type de tableau avant de s'en servir. Un tel type est déclaré par exemple ainsi :

```
type TTableau is array (Integer range 1..3) of Integer ;
```

Cette ligne déclare un type nommé `TTableau`, qui est un tableau (`array`) dont les indices sont du type `Integer`, compris entre 1 et 3 et dont les éléments sont de type `Integer`. Relisez bien la phrase précédente : la représentation des tableaux en Ada est assez différente que ce que l'on trouve dans la plupart des autres langages de programmation.

Première remarque : l'indice du tableau est typé. En C/C++, Python ou de nombreux autres langages, l'indice est implicitement un type entier « universel », sur la nature exacte duquel on ne s'attarde pas. Ici notre indice possède bien un type. Deuxième remarque : la définition du tableau précise l'intervalle de validité de l'indice (la partie `range 1..3`). Cela signifie que l'indice de la première valeur du tableau **n'est pas forcément 0**, contrairement à la plupart des langages. Ce premier indice peut être n'importe quelle valeur valide pour le type de l'indice – il en va naturellement de même pour la borne supérieure. Le type tableau `TTableau` est donc destiné à contenir 3 entiers, lesquels seront référencés par des indices compris entre 1 et 3 inclus. Un tableau de même taille aurait été obtenu en écrivant `range -3..-1`, sauf que les indices auraient tous été négatifs. Dans la pratique, la base d'indice préférée en Ada est 1 plutôt que 0. Notez que si la première valeur de l'intervalle est strictement inférieure à la deuxième (par exemple, `range 3..1`), le type correspondant ne pourra pas contenir la moindre valeur, bien qu'une telle déclaration soit syntaxiquement correcte. On a alors un type « vide ».

Voici deux autres manières de définir un type tableau, les deux types ayant exactement la même taille :

```
type Entier is new Integer range 1..3 ;
type TPetitTab is array (Entier)of Integer ;
type Rgb is (Rouge, Vert, Bleu) ;
type TRgbTab is array (Rgb) of Integer ;
```

La première ligne crée un nouveau type d'entiers `(Entier)`, dont les valeurs sont limitées entre 1 et 3. La deuxième ligne déclare un type tableau `TPetitTab`, mais sans donner explicitement l'intervalle des indices : celui-ci est tout simplement déduit du type de l'indice. La répétition du `range 1..3` est donc superflue. Ce type de tableaux contient trois entiers.

On retrouve en troisième ligne le type énuméré `Rgb` introduit plus haut. Voyez comment on l'utilise pour déclarer un type de tableau `TRgbTab` dont les indices sont de type `Rgb` : cela définit un tableau contenant trois entiers (car le type `Rgb` ne comporte que trois valeurs), les indices des éléments du tableau étant compris entre les valeurs `Rouge` et `Bleu`. Rien ne nous empêche de limiter l'intervalle des indices, même pour un type énuméré, par exemple :

```
type CoulComp is (Rouge, Vert, Bleu, Teinte, Saturation, Valeur) ;
type TRgbTab is array (CoulComp range Rouge..Bleu) of Integer ;
type THsvTab is array (Teinte..Valeur) of Integer ;
```

Les deux types ont exactement la même taille (ils contiennent tous deux trois entiers) et des indices de même type, mais les bornes valides pour les indices ne sont pas les mêmes. Remarquez la deuxième définition, qui fait l'économie du type de l'indice : cela n'est autorisé que parce qu'il n'y a aucune ambiguïté possible. Les identifiants `Teinte` et `Valeur` étant de type `CoulComp`, le type de l'indice du type tableau `THsvTab` est alors forcément de type `CoulComp`.

Utilisation des tableaux

Maintenant que nous avons déclaré plein de types de tableaux, déclarons quelques variables et utilisons-les dans un petit programme :

```
 1 with Text_IO ; use Text_IO ;
 2 procedure Tableaux1 is
 3    type TTableau is array (Integer range 1..3) of Integer ;
 4    type Entier is new Integer range 1..3 ;
 5    type TPetitTab is array (Entier) of Integer ;
 6    type Rgb is (Rouge, Vert, Bleu) ;
 7    type TTab is array (Rgb) of Integer ;
 8    type CoulComp is (Rouge, Vert, Bleu, Teinte, Saturation, Valeur) ;
 9    type TRgbTab is array (CoulComp range Rouge..Bleu) of Integer ;
10    type THsvTab is array (Teinte..Valeur) of Integer ;
11 t1: TTableau ;
12 t2: TPetitTab ;
13 t3: TRgbTab ;
14 t4: THsvTab ;
15 t5: TTab ;
16 begin
17    for i in 1..3
18    loop
19       t1(i) := 2*i ;
20    end loop ;
21    for i in t2'First..t2'Last
22    loop
23       t2(i) := 3*Integer(i) ;
24    end loop ;
25    for i in t3'Range
26    loop
27       t3(i) := 0 ;
28    end loop ;
29    for i in THsvTab'Range
30    loop
31       t4(i) := 0 ;
32    end loop ;
33    for i in Rgb
34    loop
35       t5(i) := 0 ;
36    end loop ;
37 end ;
```

Chacune des boucles remplit simplement le contenu de l'une des variables déclarées lignes 11 à 15. Remarquez que l'indice pour accéder à un élément est donné entre parenthèses, et non entre crochets.

Si la première boucle ne présente rien de particulier, examinons les en-têtes des suivantes. Ligne 21, on fait appel aux attributs `First` et `Last`, que nous avons rencontrés dans la première partie. Appliqués à un type tableau, ces attributs donnent respectivement le premier et le dernier indice du tableau – donc ici, respectivement 1 et 3.

Il est ainsi possible de parcourir un tableau sans s'encombrer l'esprit de l'intervalle d'indices valides. Remarquez le transtypage explicite ligne 23 : il est nécessaire, car les éléments de `t2` sont de type `Integer`, tandis que ses indices sont de type `Entier`.

Cela implique que la variable de boucle `i` est elle-même de type `Entier`, donc le résultat de l'opération `3*i` serait également de type `Entier`. Le typage strict de Ada nous interdit d'affecter une valeur de type `Entier` à une variable de type `Integer` : la conversion ne peut donc être évitée. Cela tient au fait que `Entier` est un type dérivé de `Integer`, un nouveau type ; si la ligne 4 avait plutôt déclaré un sous-type, avec :

```
4    subtype Entier is Integer range 1..3 ;
```

...alors le transtypage n'aurait pas été nécessaire.

La ligne 25 fait appel à l'attribut `Range`, qui est en fait un raccourci pour `t3'First..t3'Last`. Il est appliqué à la variable, mais il peut également être appliqué au type tableau lui-même, comme cela est fait ligne 29 (les attributs `First` et `Last` sont également applicables au type). Enfin, ligne 33 on donne simplement le type de l'indice (`Rgb`), duquel sera déduit l'intervalle que devra parcourir la variable de boucle `i`. Dernier mot, l'attribut `Length` donne le nombre d'éléments contenus dans un tableau sous la forme d'un entier (par exemple, `t1'Length` ou `TTab'Length` valent tous deux `3`).

Jusque-là, mis à part les contraintes liées au typage fort, rien de véritablement extraordinaire.

Agrégats et affectations

Mais voici quelques possibilités fort intéressantes, qui rappellent un peu ce que l'on trouve en langage Python (sauf que Python a été conçu bien après Ada). Tout d'abord, il peut être pertinent d'initialiser le contenu du tableau dès la déclaration :

```
type TTab is array (1..3) of Float ;
t: TTab := (1.0, 2.0, 3.0) ;
```

Le type `TTab` est un type tableau contenant trois valeurs en virgule flottante, indicées par des entiers entre 1 et 3. La deuxième ligne déclare une variable `t` de ce type, et l'affectation permet tout simplement de donner le contenu du tableau. La liste entre parenthèses à droite de l'affectation est un agrégat. Cette notation va bien quand on n'a que quelques valeurs, mais qu'en est-il pour un tableau de grande taille ? On fait appel au mot-clef `others`, que nous avons déjà rencontré le mois dernier dans le cadre de l'instruction `case` :

```
type TTab is array (1..100_000) of Float ;
t: TTab := (others => 0.0) ;
```

Cette fois `TTab` contient 100000 valeurs (remarquez l'utilisation du caractère de soulignement dans l'écriture du nombre, qui n'a rien d'obligatoire mais permet de le rendre plus lisible). L'agrégat pourrait se lire comme « pour tous les autres indices, affecter la valeur 0.0 » : à la suite de l'affectation, toutes les valeurs du tableau prennent donc la valeur 0.0.

Les agrégats prennent tout leur intérêt lorsqu'ils sont utilisés en dehors de la déclaration du tableau. De plus, ils peuvent prendre des formes sensiblement plus sophistiquées. Par exemple, en reprenant notre grand tableau, supposons que l'on souhaite initialiser toutes les valeurs à `0.0`, sauf les valeurs d'indices `7` et `77` qui doivent prendre la valeur `1.0`, les indices `101` et `1010` qui doivent prendre la valeur `2.0`, les indices entre `2000` et `3000` qui doivent prendre la valeur `3.0`... tout cela peut s'exprimer en une seule instruction :

```
t := (7|77 => 1.0, 101|1010 => 2.0, 2000..3000 => 3.0, others => 0.0) ;
```

On retrouve en fait là une notation similaire à celle disponible dans l'instruction `case` (voir l'article du mois dernier). Encore plus fort : on peut limiter la portion du tableau affectée par l'affectation, en donnant un intervalle à gauche du signe `:=` :

```
t(123...456) := (others => 0.0) ;
```

Ceci a pour effet de mettre à `0.0` les valeurs aux indices compris entre `123` et `456` inclus, sans toucher aux autres valeurs du tableau. L'agrégat donné à droite peut naturellement prendre une forme aussi complexe que vous voulez, l'important étant qu'il représente exactement autant de valeurs que la « tranche » spécifiée à gauche.

Opérateurs pour tableaux

Tout un ensemble d'opérateurs sont définis pour les tableaux à une seule dimension. Supposons que nous ayons les déclarations suivantes :

```
type Tab2Int is array (Integer range 1..2) of Integer ;
type Tab4Int is array (Integer range 1..4) of Integer ;
type Tab2Bool is array (Integer range 1..2) of Boolean ;
t1: Tab2Int ;
t2: Tab2Int ;
tb1: Tab2Bool := (True, False) ;
tb2: Tab2Bool := (True, True) ;
tb3: Tab2Bool ;
t4: Tab4Int ;
```

Alors le tableau suivant résume les différentes opérations possibles (Tab. 2)

Pour les habitués des langages C/C++, si ce n'était pas encore clair, remarquez bien qu'un tableau en Ada n'a rien à voir avec un pointeur.

C'est un type à part entière, qui ne se contente pas de contenir des données mais transporte avec lui pas mal d'informations « administratives ».

Les opérateurs précédents, notamment l'affectation et l'égalité, en sont la meilleure preuve.

Les tableaux anonymes

Jusqu'ici nous avons toujours déclaré un type de tableaux avant d'en créer une variable.

Ce n'est pas réellement obligatoire, il est possible de déclarer ainsi une variable représentant un tableau :

```
t1: array (1..10) of Integer ;
t2: array (1..10) of Integer := (others => 2) ;
```

Les deux variables `t1` et `t2` semblent parfaitement équivalentes... pourtant elles ne sont pas de même type et c'est là l'inconvénient principal de cette méthode.

Cela implique qu'il est impossible de réaliser des affectations comme `t1 := t2`, des comparaisons, etc. Aussi l'utilisation des tableaux anonymes est-elle vraiment déconseillée.

De plus, ils ne peuvent pas apparaître comme type de paramètre passé à un sous-programme.

Tableaux presque dynamiques

Peut-être certains sont-ils un peu inquiets à la suite de la remarque concernant la différence fondamentale entre un tableau Ada et un tableau C/C++.

Comment faire pour déclarer un tableau dont la taille n'est pas connue à l'avance ou, encore pire, Ada ne possède peut-être pas de pointeurs ?

Que tout le monde se rassure, Ada possède bien des pointeurs – nous en parlerons plus tard.

Regardez maintenant le petit programme suivant, qui montre que les pointeurs ne sont pas forcément nécessaires pour déclarer des tableaux de taille inconnue à la compilation :

:: Tab. 2 :: Opérateurs pour tableaux ::

Opérateur	Description	Exemple

:=	Affectation entre tableaux ou agrégats de même type.	`t1 := (3, 4) ;`
		`t2 := t1 ;`
		`-- t2 = (3, 4)`

=, /=	Égalité, inégalité : retourne un booléen résultat d'une comparaison élément par élément. Les tableaux doivent être de même taille.	`t1 = t2 ;  -- vrai`
		`t1(1) := 0 ;`
		`t1 = t2 ;  -- faux`

not, and, or, xor	Opérations booléennes sur des tableaux de mêmes tailles contenant des booléens. Le résultat est un tableau de booléens, dont les bornes sont celles de l'opérande de gauche. Pratique pour manipuler globalement des tableaux représentant des champs de bits.	`tb3 := not tb1 ;`
		`-- tb3 = (False, True)`
		`tb3 := tb1 xor tb2 ;`
		`-- tb3 = (False, True)`

&	Concaténation de tableaux ou d'agrégats, en général pour affecter le résultat à un tableau plus grand. Notez que si vous donnez des tranches de tableaux (comme dans le deuxième exemple), les types de tableau doivent être identiques.	`t4 := (5, 6) & (7, 8) ;`
		`-- t4 = (5, 6, 7, 8)`
		`t4 := t4(3..4) & t4(1..2) ;`
		`-- t4 = (7, 8, 5, 6)`

<, <=, >, >=	Opérations de comparaison sur des tableaux contenant un type discret (entier, énuméré). L'ordre lexicographique est utilisé, c'est-à-dire celui utilisé dans un dictionnaire. Les tableaux n'ont pas forcément la même taille, mais ils doivent être de mêmes types.	`t1 < t2 ; -- vrai`
		`t4 < t4(1..2) ; -- faux`
		`t4 < t4(2..3) ; -- vrai`

```
1 with Text_IO ; use Text_IO ;
2 procedure Tableaux3 is
3    procedure Carres (n: in Integer) is
4       type TTableau is array (1..n) of Integer ;
5       tab: TTableau ;
6    begin
7       for i in tab'Range
8       loop
9          tab(i) := i*i ;
10      end loop ;
11      for i in tab'Range
12      loop
13         Put(Integer'Image(tab(i))) ;
14         if i < tab'Last
15         then
16            Put(", ") ;
17         else
18            New_Line ;
19         end if ;
20      end loop ;
21   end ;
22 ch: String(1..5) := (others => ' ') ;
23 ch_l: Integer ;
24 nb: Integer ;
25 begin
26    Put_Line("Taille : ") ;
27    Get_Line(ch, ch_l) ;
28    nb := Integer'Value(ch) ;
29    Carres(nb) ;
30 end Tableaux3 ;
```

Intéressons-nous d'abord à la partie principale, entre les lignes 22 et 30. `ch` est une variable chaîne pouvant contenir 5 caractères (si l'écriture vous laisse supposer que le type `String` est en fait un type tableau contenant des caractères, vous avez parfaitement raison).

Cette chaîne récupère l'entrée de l'utilisateur, obtenue ligne 27 par la procédure `Get_Line` fournie par `Text_IO` ; le deuxième paramètre `ch_l` est simplement la longueur de la chaîne lue (nous reviendrons dans un prochain article sur les fonctions d'entrées-sorties). Puis cette chaîne est convertie en entier stocké dans `nb` (ligne 28), grâce à l'attribut `Value` du type `Integer`.

Cet attribut permet de convertir une chaîne en la valeur correspondante pour un type scalaire, c'est-à-dire tous les types numériques et les types énumérés. Par exemple, `Rgb'Value("rOugE")` donnera la valeur `Rouge` du type énuméré `Rgb` présenté au début de cet article – remarquez qu'il n'est fait aucun cas de la casse des lettres, Ada ne faisant lui-même pas la différence entre majuscules et minuscules.

Enfin on invoque une procédure `Carres`, qui va construire un tableau des `nb` premiers carrés puis afficher le contenu de ce tableau. C'est cette procédure qui nous intéresse.

Clairement la taille du tableau à créer n'est pas connue à l'avance. Cela ne pose aucun problème à Ada : la ligne 4 déclare un type tableau `TTableau`, en donnant tout simplement le paramètre `n` reçu comme borne supérieure (le type de l'indice est implicitement entier).

En fait, les deux bornes ne sont pas nécessairement des constantes : on pourrait parfaitement avoir une définition de type comme

```
type T is array (2*n+1..n**3-2) of Integer ;
```

...ce qui peut résulter en un type vide, si `n` vaut `0` (l'opérateur `**` est l'élévation à la puissance : `n**3 = n*n*n`).

Le reste de la procédure ne devrait pas présenter de difficultés particulières. Une variable du type fraîchement créé est déclarée ligne 5, une première boucle remplit le tableau (lignes 7 à 10), une deuxième affiche son contenu (lignes 11 à 20).

La procédure `New_Line` invoquée ligne 18 vient de `Text_IO` et permet simplement de passer à la ligne (l'équivalent d'un `printf("\n")`). Voici un exemple d'utilisation :

```
$ gnatmake tableaux3.adb && ./tableaux3
Taille :
5
1, 4, 9, 16, 25
```

Cela fonctionne donc bien !

Tableaux non contraints

La souplesse d'Ada concernant les tableaux ne s'arrête pas là. Tous les types déclarés jusqu'à maintenant étaient contraints, c'est-à-dire que dès la déclaration du type, les bornes sont connues, fut-ce à l'exécution.

Mais ce n'est pas toujours satisfaisant. Imaginez le problème : écrire une procédure capable d'afficher le contenu d'un tableau d'entiers, quelles que soient les bornes de ce tableau.

Avec ce que nous savons pour l'instant, c'est tout simplement impossible. Alors pour nous sauver, voici les tableaux non contraints, dont la déclaration ressemble à ceci :

```
type Tableau is array (Integer range <>) of Integer ;
```

On donne toujours le type de l'indice et le type des éléments, mais l'intervalle de l'indice a été remplacé par le symbole <> (qui se lit « boîte », ou « *box* » en anglais). Ceci déclare un type de tableau dont les bornes ne sont pas limitées a priori (sauf par les limites du type de l'indice, bien sûr).

Pour déclarer une variable de ce type, il est nécessaire de préciser l'intervalle voulu par exemple ainsi :

```
t1: Tableau(1..10) := (others => 1) ;
-- t1 = (1, 1, 1, 1, 1, 1, 1, 1, 1, 1)
t2: Tableau(-5..0) := (-5 => 2, -3 => 3, others => 1) ;
-- t2 = (2, 1, 3, 1, 1, 1)
```

Remarquez l'agrégat étrange utilisé pour initialiser le deuxième tableau. Ce qui est intéressant là-dedans, c'est que bien que `t1` et `t2` soient de tailles différentes, ils sont de même type. On peut dès lors s'amuser à certaines manipulations autrement impossibles, par exemple :

```
t2(-4..-2) := t1(7..9) ;
-- t2 = (2, 1, 1, 1, 1, 1)
t1 < t2 ; -- vrai
```

Mais revenons à notre procédure. Voici quel pourrait être son code :

```
procedure AffTab(t: in Tableau) is
begin
   for i in t'Range
   loop
      Put(Integer'Image(t(i))) ;
      if i < t'Last
      then
         Put(",") ;
      else
         New_Line ;
      end if ;
   end loop ;
end ;
```

Aucune hypothèse n'est faite sur les bornes ou la taille du tableau. Même un tableau vide (dont la borne inférieure est plus grande que la borne supérieure) convient : simplement rien ne sera affiché.

On a ainsi gagné une certaine abstraction, une certaine généricité, sans faire appel au mécanisme du même nom. Ada possède en effet dès sa conception (en 1979 !) la notion de généricité, que l'on retrouve en C++ par les classes et fonctions `template`.

Mais c'est une autre histoire que nous aborderons plus tard.

Les tableaux multidimensionnels

Il existe essentiellement deux techniques pour obtenir plusieurs dimensions : soit créer des tableaux de tableaux, soit donner plusieurs indices à un tableau.

Les deux approches présentent divers avantages et divers inconvénients chacune. Préférer l'une ou l'autre dépend de ce que l'on veut faire du tableau.

Le programme suivant déclare un tableau de tableaux :

```
 1 with Text_IO ; use Text_IO ;
 2 procedure Tab_Multi_1 is
 3    type Tab4Int is array (1..4) of Integer ;
 4    type Tab3Tab4Int is array (1..3) of Tab4Int ;
 5    procedure Aff(t: in Tab3Tab4Int) is
 6    begin
 7       Put("(") ;
 8       for l in t'Range
 9       loop
10          if l > t'First
11          then
12             Put(" ") ;
13          end if ;
14          Put("(") ;
15          for c in t(l)'Range
16          loop
17             Put(Integer'Image(t(l)(c))) ;
18             if c < t(l)'Last
19             then
20                Put(",") ;
21             end if ;
22          end loop ;
23          Put(")") ;
24          if l < t'Last
25          then
26             Put_Line(",") ;
27          end if ;
28       end loop ;
29       Put_Line(")") ;
30    end ;
31 t1: Tab3Tab4Int := (others => (others => 0)) ;
32 t2: Tab3Tab4Int := ((11, 12, 13, 14),
33                    (21, 22, 23, 24),
34                    (31, 32, 33, 34)) ;
35 begin
36    t1(2)(2..3) := t2(3)(1..2) ;
37    Aff(t1) ;
38    -- t1 = (( 0,  0,  0,  0),
39    --       ( 0, 31, 32,  0),
40    --       ( 0,  0,  0,  0))
41    t1(1..2) := t2(1..2) ;
42    Aff(t1) ;
43    -- t1 = ((11, 12, 13, 14),
44    --       (21, 22, 23, 24),
45    --       ( 0,  0,  0,  0))
46    t1(1..2) := t1(2) & t1(1) ;
47    Aff(t1) ;
48    -- t1 = ((21, 22, 23, 24),
49    --       (11, 12, 13, 14),
50    --       ( 0,  0,  0,  0))
51 end Tab_Multi_1 ;
```

Il n'y a en fait pas grand-chose de particulier. Les bornes données ne doivent pas nécessairement être connues à la compilation. La plupart des facilités des tableaux sont disponibles, comme les agrégats ou les affectations.

Mais il y a un inconvénient notable. Les éléments d'un tableau doivent nécessairement être contraints.

Une déclaration comme celle-ci est autorisée :

```
type T1 is array (1..10) of Float ;
type T2 is array (Integer range <>) of T1 ;
```

Le type `T1` est contraint, mais pas le type `T2` : on a donc un tableau non contraint de tableaux contraints. Mais si on change la déclaration de `T1` pour un tableau non contraint, la déclaration de `T2` ne sera pas acceptée, qu'elle soit contrainte ou non :

```
type T1 is array (Integer range <>) of Float ; -- T1 non contraint
type T2 is array (1..2) of T1 ;                -- erreur !
type T2 is array (Integer range <>) of T1 ;    -- erreur !
```

Si on a besoin de la souplesse d'un type non contraint sur plusieurs dimensions, on fait appel aux tableaux multidimensionnels. Le programme suivant montre comment les utiliser, ainsi que le transtypage entre types tableaux :

```
 1 with Text_IO ; use Text_IO ;
 2 procedure Tab_Multi_2 is
 3    type Tab3x4Int is array (1..3, 1..4) of Integer ;
 4    type TabMulti is array (Integer range <>, Integerrange
<>) of Integer ;
 5    procedure Aff(t: in TabMulti) is
 6    begin
 7       Put("(") ;
 8       for l in t'Range(1)
 9       loop
10          if l > t'First(1)
11          then
12             Put(" ") ;
13          end if ;
14          Put("(") ;
15          for c in t'Range(2)
16          loop
17             Put(Integer'Image(t(l, c))) ;
18             if c < t'Last(2)
19             then
20                Put(",") ;
21             end if ;
22          end loop ;
23          Put(")") ;
24          if l < t'Last(1)
25          then
26             Put_Line(",") ;
27          end if ;
28       end loop ;
29       Put_Line(")") ;
30    end ;
31 t1: Tab3x4Int ;
32 t2: TabMulti(7..10, -5..-4) := ((11, 12),
33                                (21, 22),
34                                (31, 32),
35                                (41, 42)) ;
36 t3: TabMulti := ((11, 12, 13, 14),
37                 (21, 22, 23, 24),
38                 (31, 32, 33, 34)) ;
39 begin
40    Aff(t2) ;
41    Aff(t3) ;
42    t1 := Tab3x4Int(t3) ;
43    t1(2, 3) := 0 ;
44    t3 := TabMulti(t1) ;
45    Aff(t3) ;
46 end Tab_Multi_2 ;
```

Voyez comment les éléments du tableau sont référencés différemment que dans l'exemple précédent. Quand on écrit `t(i)(j)` pour un tableau de tableaux, on écrit `t(i,j)` pour un tableau à deux dimensions.

Par contre, les agrégats sont tout à fait semblables. Remarquez par ailleurs la déclaration de la variable `t3`, lignes 36 à 38 : elle est d'un type non contraint, pourtant on ne donne pas les bornes (comme cela a été fait pour `t2`, ligne 32).

Les bornes sont en fait déduites de l'agrégat utilisé pour l'initialisation. Utilisez les attributs pour connaître les valeurs de ces bornes.

Les attributs, justement, se voient agrémentés d'un paramètre (voyez le code de la procédure `Aff`, aux lignes 8, 10, 15, 18 et 24). Il indique quelle est la dimension interrogée.

Ainsi `t'Range(1)` donne l'intervalle pour le premier indice, `t'First(2)` donne la borne inférieure du deuxième indice, et ainsi de suite. Encore une fois, voyez la souplesse d'Ada, qui nous permet d'écrire une procédure d'affichage relativement générique sans connaître les bornes des tableaux.

Enfin, les lignes 42 et 44 réalisent deux conversions de type tableaux. Bien qu'ils soient de mêmes dimensions, on ne pourrait pas réaliser d'affectations entre `t1` et `t3` simplement avec `t1 := t3` : ils sont de types différents. Mais parce qu'ils sont de mêmes dimensions, il est possible de les convertir l'un dans l'autre.

Par contre le transtypage entre `t1` et `t2` n'est pas possible : les dimensions sont incompatibles. Bien qu'une affectation comme `t1 := Tab3x4Int(t2)` soit acceptée par le compilateur (mais seulement parce que `t2` est d'un type non contraint), à l'exécution une exception `Constraint_Error` sera levée.

Dernier mot, il n'y a pas de limite théorique au nombre de dimensions, seulement celles imposées par le matériel et le compilateur.

Conclusion

Voilà pour cette présentation des tableaux en Ada. S'ils n'ont pas toute la puissance de ceux que l'on peut trouver dans les langages interprétés comme Python ou Ruby, ils sont considérablement plus souples et pratiques à manipuler que dans les langages comme C/C++ ou Java.

Le mois prochain, nous découvrirons les enregistrements (ou structures), ce qui sera l'occasion de commencer le programme des Tours de Hanoï qui nous accompagnera durant quelques articles.

Références

[1] Ada 2005 sur Wikibooks : http://en.wikibooks.org/wiki/Programming:Ada:2005

[2] Ébauche du prochain standard : http://www.adaic.com/standards/rm-amend/html/AA-TTL.html

Le langage Ada – 5
Les enregistrements

Yves Bailly

EN DEUX MOTS **Les tableaux du mois dernier ne permettent de stocker que des informations d'un même type. Comme beaucoup d'autres langages, Ada offre un moyen de placer en une seule entité des données de types différents. Là où le langage C parle de structures ou Python de classes, Ada a repris la terminologie du Pascal et parle plutôt d'enregistrements.**

L'enregistrement est la structure de donnée fondamentale en Ada pour représenter les idées du programmeur. Doté d'une grande souplesse, on imagine mal comment un programme non trivial pourrait s'en passer, à moins de tolérer une effroyable complexité dans l'écriture du code. Pour l'essentiel, ce que nous allons voir ici n'est pas propre à Ada. On retrouve nombre des aspects dans nombre de langages – mais quelques possibilités sont uniques à Ada.

Déclaration

Supposons que nous souhaitions réaliser un programme pour jouer aux Tours de Hanoï (toute ressemblance avec d'autres articles n'aurait rien de fortuit). Il nous faut une structure pour représenter une tour, qui contiendra essentiellement un tableau de disques et le nombre de disques actuellement empilés sur la tour. Ce qui pourrait ressembler à ceci :

```
1 type Disque is new Integer range 0..10 ;
2 type Étage is new Integer range 0..10 ;
3 type Pile_Disques is array (Étage) of Disque ;
4
5 type Tour is
6 record
7    dernier_étage: Étage := 0 ;
8    pile: Pile_Disques := (others=>0) ;
9 end record ;
```

Les trois premières lignes définissent simplement quelques types qui nous seront utiles par la suite, la pile de disque étant représentée par un simple tableau (ligne 3). Le véritable objet de notre attention aujourd'hui se situe entre les lignes 5 et 9. Nous définissons ici un type nommé `Tour`, qui est un enregistrement (`record`) contenant deux champs nommés `dernier_étage` (de type `Étage`) et `pile` (de type `PileDisques`).

Petite remarque en passant : vous aurez peut-être remarqué les accents présents dans ce code source, ce qui peut paraître incongru. Je me suis permis cette fantaisie pour montrer une possibilité du compilateur GNAT, qui est justement d'accepter ce genre de caractères dans les identifiant.

En fait, cela fera bientôt partie de la norme Ada officielle : les compilateurs de la prochaine version du langage devraient être capables de traiter un code source contenant des noms de variables composés de caractères grecs et turcs (ce qui ne manquerait pas de piquant), ou autres, naturellement. Il est pour l'instant nécessaire de donner l'option `-gnatif` au compilateur, par exemple :

```
$ gnatmake -gnatif hanoi_decl.adb
```

Mais revenons à notre structure. Cela ressemble fort à une déclaration `struct` du langage C, avec toutefois cette différence notable qu'il est possible de donner une valeur par défaut pour les champs, ce qui est vivement recommandé.

On reproduit ainsi un comportement comparable à celui du constructeur d'une classe C++ ou Java, quoiqu'en plus limité.

Naturellement, rien ne nous empêche d'utiliser ce type pour composer d'autres types, par exemple :

```
1 type Num_Tour is new Integer range 1..3 ;
   2 type Rangée_Tours is array (Num_Tour) of Tour ;
   3 type Num_Mouvements is new Integer range 0..Integer'Last ;
   4 type Jeu_Hanoï is
   5 record
   6    tours: Rangée_Tours ;
   7    nb_mouvements: Num_Mouvements := 0 ;
   8 end record ;
```

La ligne 2 déclare un type tableau `Rangée_Tours` dont les éléments sont de type `Tour`, ce tableau étant lui-même utilisé dans un enregistrement `Jeu_Hanoï` contenant différents paramètres du jeu. Remarquez que nous n'avons pas donné de valeur initiale pour le premier champ.

Non pas que cela ne serait pas possible (nous allons bientôt voir comment), simplement ce n'est pas strictement nécessaire car les éléments du tableau sont des enregistrements eux-mêmes déjà initialisés. On pourrait objecter à cela que le jeu risque de se retrouver ainsi dans un état incohérent, sans aucun disque sur aucune tour...

Utilisation

La déclaration d'une variable d'un type enregistrement se fait comme n'importe quelle autre :

```
jeu: Jeu_Hanoï;
```

De même que pour les tableaux, on peut donner un agrégat pour initialiser les composants de l'enregistrement, sauf qu'au lieu de donner l'indice de l'élément on donne son nom.

Dans notre cas, nous avons un enregistrement qui contient un tableau d'enregistrements contenant un tableau. Une initialisation complète pourrait ressembler à ceci :

```
jeu: Jeu_Hanoï := (
   tours => (
      1 => (
         dernier_étage => 10,
         pile => (0, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1)),
      others => (
         dernier_étage => 0,
         pile => (others => 0))),
   nb_mouvements => 0) ;
```

L'accès aux éléments d'un enregistrement se fait en utilisant la notation pointée usuelle. Voici par exemple une procédure qui affiche le contenu d'une instance de **Jeu_Hanoï** :

```
 1 procedure Afficher(jeu: in Jeu_Hanoï) is
 2 begin
 3  Loop_Tours:
 4  for ind_tour in Num_Tour
 5  loop
 6    Put("Tour " & Num_Tour'Image(ind_tour) & " : ") ;
 7    Loop_Etages:
 8    for ind_etage in 1..Étage'Last
 9    loop
10      Put(Disque'Image(jeu.tours(ind_tour).pile(ind_etage))) ;
11    end loop Loop_Etages ;
12    New_Line ;
13  end loop Loop_Tours ;
14 end Afficher ;
```

Remarquez que rien dans cette portion de code ne suggère que nous ayons un maximum de 10 disques, selon la définition des types `Étage` et `Tour`. La ligne 10 montre l'accès aux différents éléments : `jeu.tours` est le tableau des tours, `jeu.tours(ind_tour)` désigne l'enregistrement de type `Tour` à l'indice `ind_tour` dans ce tableau, etc.

Les enregistrements supportent de plus les opérations élémentaires que sont l'affectation (`:=`) et la comparaison (`=` et `/=`). Ces opérations sont appliquées récursivement sur chacun des éléments de l'enregistrement. Par exemple, si la longue valeur d'initialisation de la déclaration plus haut vous rebute, voici une procédure qui permet d'initialiser une variable de type `Jeu_Hanoï`, en fonction du nombre initial de disques que l'on veut sur la première tour :

```
 1 procedure Init(jeu:     out   Jeu_Hanoï ;
 2                nb_init: in    Étage) is
 3    jeu_tmp: Jeu_Hanoï ;
 4 begin
 5    jeu_tmp.tours(1).dernier_étage := nb_init ;
 6    for ind_etage in 1..nb_init
 7    loop
 8       jeu_tmp.tours(1).pile(ind_etage) :=
 9          Disque(Étage'Last-(ind_etage-1)) ;
10    end loop ;
11    jeu := jeu_tmp ;
12 end Init ;
```

Le passage par une variable temporaire nous assure que tous les éléments auront au moins une valeur par défaut. Le traitement se limite donc aux éléments à modifier selon le paramètre `nb_init`. Le résultat de l'initialisation est renvoyé dans le paramètre en sortie `jeu` grâce à l'affectation de la ligne 11.

Remarquez le transtypage ligne 9 : il est nécessaire, car le membre `pile` contient des éléments de type `Disque`, tandis que `ind_etage` et `Étage'Last` sont de type `Étage`. Et encore une fois, nous n'avons pas utilisé explicitement la limite de 10 du nombre de disques : elle est tout simplement récupérée par l'attribut `Last` du type `Étage`.

Représentation

Une différence fondamentale entre des langages comme le C et Ada est que ce dernier n'impose rien quant à la représentation en mémoire des structures de données déclarées dans le programme. Le compilateur a toute latitude pour stocker tout cela en mémoire comme bon lui semble, en fonction des options qui lui sont données pour minimiser l'encombrement mémoire ou optimiser la vitesse d'exécution. Par exemple, un enregistrement ne contenant que deux entiers d'un type borné entre 1 et 10 pourrait parfaitement n'occuper qu'un seul octet, ce qui est binairement suffisant ; un tableau de six booléens (type `Boolean`) pourrait n'occuper que six bits, c'est-à-dire moins d'un octet ; au contraire, notre enregistrement de deux entiers pourrait s'étaler sur 16 octets, si on demande la meilleure performance possible et que le matériel sous-jacent est plus efficace pour accéder aux données tous les 8 octets... Ne soyez pas étonnés de cela, Ada a été conçu pour pouvoir être utilisé pour des logiciels embarqués (comme les sondes spatiales) où les contraintes et limites matérielles sont parfois très fortes.

Une différence fondamentale entre des langages comme le C et Ada est que ce dernier n'impose rien quant à la représentation en mémoire des structures de données déclarées dans le programme.

Mais parfois ces « aléas » dans la représentation mémoire ne sont pas souhaitables : on peut vouloir contrôler très précisément comment les données sont agencées en mémoire. Ada propose différents moyens pour cela, par le biais d'attributs et de directives `pragma`. Cela peut s'avérer fort utile si le programme doit manipuler directement un périphérique par ses ports d'entrées/sorties.

Voyons un petit programme d'exemple :

```
 1 with Text_IO ; use Text_IO ;
 2 procedure test_bits is
 3    type Petit is new Integer range 1..5 ;
 4 --   for Petit'Size use 5 ;
 5    type Enreg is
 6    record
 7       a: Petit ;
 8       b: Boolean ;
 9       c: Petit ;
10       d: Petit ;
11       e: Boolean ;
12    end record ;
13 --   pragma Pack(Enreg) ;
14    type Tab1 is array (Petit) of Petit ;
15 --   for Tab1'Component_Size use 10 ;
16    type Tab2 is array (Petit) of Enreg ;
```

```
17 --   pragma Pack(Tab2) ;
18 e: Enreg ;
19 begin
20    Put_Line("Petit'Size = " &
21       Integer'Image(Petit'Size)) ;
22    Put_Line("Enreg'Size = " &
23       Integer'Image(Enreg'Size)) ;
24    Put_Line("Enreg.a'First_Bit = " &
25       Integer'Image(e.a'First_Bit)) ;
26    Put_Line("Enreg.b'First_Bit = " &
27       Integer'Image(e.b'First_Bit)) ;
28    Put_Line("Enreg.c'First_Bit = " &
29       Integer'Image(e.c'First_Bit)) ;
30    Put_Line("Enreg.d'First_Bit = " &
31       Integer'Image(e.d'First_Bit)) ;
32    Put_Line("Enreg.e'First_Bit = " &
33       Integer'Image(e.e'First_Bit)) ;
34    Put_Line("Enreg.a'Position = " &
35       Integer'Image(e.a'Position)) ;
36    Put_Line("Enreg.b'Position = " &
37       Integer'Image(e.b'Position)) ;
38    Put_Line("Enreg.c'Position = " &
39       Integer'Image(e.c'Position)) ;
40    Put_Line("Enreg.d'Position = " &
41       Integer'Image(e.d'Position)) ;
42    Put_Line("Enreg.e'Position = " &
43       Integer'Image(e.e'Position)) ;
44    Put_Line("Tab1'Size = " &
45       Integer'Image(Tab1'Size)) ;
46    Put_Line("Tab1'Component_Size = " &
47       Integer'Image(Tab1'Component_Size)) ;
48    Put_Line("Tab2'Size = " & Integer'
49       Image(Tab2'Size)) ;
50    Put_Line("Tab2'Component_Size = " &
51       Integer'Image(Tab2'Component_Size)) ;
52 end test_bits ;
```

Répétons-le : tout n'est que souplesse en Ada.

Les lignes commentées (4, 13, 15 et 17) contiennent des instructions précisant la représentation mémoire à adopter pour différents objets. Le programme affiche ensuite simplement des informations sur la taille et la position de ces objets. Aspect essentiel à bien garder en mémoire : les tailles et positions sont données en bits et non en octets.

Les attributs impliqués ici sont :

- `Size` donne la taille (en bits, donc – voir [AARM 13.3-40,45]) de l'objet lorsqu'il est interrogé (par exemple ligne 21) ou permet de la spécifier (par exemple ligne 4) ;
- `Component_Size` ne concerne que les types tableaux et permet d'obtenir (ligne 47) ou de définir (ligne 15) la taille occupée par chaque élément dans le tableau ;à noter qu'il est permis à l'implémentation [AARM 13.3-73] d'adapter la valeur donnée ;
- `First_Bit` [AARM 13.5.2] est appliqué à un élément d'un enregistrement et donne le décalage du premier bit de cet élément par rapport au début de l'octet mémoire qui le contient (en nombre de bits) ; l'attribut `Last_Bit` donne le décalage du dernier bit ; ces deux attributs ne peuvent pas être définis (enfin, pas directement) ;
- Enfin, `Position` donne la position d'un élément dans un enregistrement, cette fois exprimée en octets ; cet attribut ne peut pas être défini.

Tout cela nous donne déjà pas mal de souplesse. Mais ce n'est pas tout. Le langage Ada fait un usage assez intensif des directives `pragma`, dont le rôle est essentiellement de donner des informations ou des conseils au compilateur. La première que nous rencontrons est la directive `pragma Pack` (lignes 13 et 17) : elle indique au compilateur qu'il doit tenter de minimiser l'encombrement mémoire des instances du type auquel elle est appliquée, qui doit être un type composé (tableau ou enregistrement). C'est une recommandation forte, qui doit être suivie même au prix de la vitesse d'exécution.

Pour fixer les idées, exécutez le programme précédent en jouant sur les lignes en commentaires.

La troisième version du programme, si elle devait créer beaucoup d'instances du type `Enreg` dans un tableau, occuperait beaucoup moins de mémoire (presque neuf fois moins) que la première version. Par contre, elle serait sensiblement plus lente, car il est nécessaire d'ajouter des instructions de calcul et manipulations binaires pour accéder aux éléments. Un test rapide montre que le parcours d'un tableau de 10000 éléments est presque quatre fois plus lent dans la première version (plus de huit fois si on compile avec l'optimisation -O2).

Il existe encore d'autres possibilités pour affiner la représentation mémoire : consultez la norme pour plus de détails.

Erratum

Je profite de ce chapitre sur les représentations des données pour m'excuser d'une erreur qui s'est glissée dans l'article du moins dernier, au sujet des types énumérés, précisément ce membre de phrase : « il n'est pas possible d'associer explicitement une valeur numérique à une valeur symbolique ». C'est tout simplement faux : la section 13.4 du AARM expose justement une syntaxe permettant d'effectuer une telle association. Par exemple :

```
type TT is (AA, BB, CC) ;
for TT use (AA=>25, BB=>3213, CC=>6598) ;
```

Les symboles AA, BB et CC du type TT seront représentés en mémoire par les entiers 25, 3213 et 6598.

Paramétrisation

Répétons-le : tout n'est que souplesse en Ada. Il est ainsi possible de paramétriser un enregistrement, selon un ou plusieurs critères. Le cas le plus simple consiste à utiliser une valeur pour dimensionner un tableau non contraint. Prenons un exemple :

```
1 procedure Record_Param is
2    type TTab is array (Integer range <>) of Integer ;
3    type TRec(taille: Integer) is
4    record
5       t: TTab(1..taille) := (others => 0) ;
6    end record ;
7 nb: Integer := 10 ;
8 r1: TRec(nb) ;
```

Effets d'instructions de représentation

TOUTES LES LIGNES DÉSACTIVÉES	LIGNES 4 ET 17 ACTIVÉES	TOUTES LES LIGNES ACTIVÉES
Petit'Size = 3 Enreg'Size = 136 Enreg.a'First_Bit = 0 Enreg.b'First_Bit = 0 Enreg.c'First_Bit = 0 Enreg.d'First_Bit = 0 Enreg.e'First_Bit = 0 Enreg.a'Position = 0 Enreg.b'Position = 4 Enreg.c'Position = 8 Enreg.d'Position = 12 Enreg.e'Position = 16 Tab1'Size = 160 Tab1'Component_Size = 32 Tab2'Size = 800 Tab2'Component_Size = 160	Petit'Size = 5 Enreg'Size = 40 Enreg.a'First_Bit = 0 Enreg.b'First_Bit = 0 Enreg.c'First_Bit = 0 Enreg.d'First_Bit = 0 Enreg.e'First_Bit = 0 Enreg.a'Position = 0 Enreg.b'Position = 1 Enreg.c'Position = 2 Enreg.d'Position = 3 Enreg.e'Position = 4 Tab1'Size = 40 Tab1'Component_Size = 8 Tab2'Size = 200 Tab2'Component_Size = 40	Petit'Size = 5 Enreg'Size = 17 Enreg.a'First_Bit = 0 Enreg.b'First_Bit = 5 Enreg.c'First_Bit = 6 Enreg.d'First_Bit = 3 Enreg.e'First_Bit = 0 Enreg.a'Position = 0 Enreg.b'Position = 0 Enreg.c'Position = 0 Enreg.d'Position = 1 Enreg.e'Position = 2 Tab1'Size = 50 Tab1'Component_Size = 10 Tab2'Size = 88 Tab2'Component_Size = 17
Remarquez comme la taille de Petit est petite, seulement 3 bits : le compilateur a déterminé que c'était suffisant pour contenir les valeurs de ce type. Dans l'enregistrement, tous les First_Bit sont à zéro : les composants commencent tous à une limite d'octet. La taille des éléments du type Tab1 nous montre qu'en réalité Petit occupe 4 octets en mémoire.	La ligne 4 impose une taille pour Petit : celle-ci influe naturellement sur le reste. En particulier, l'enregistrement est plus petit : le fait de donner une taille à Petit réduit son encombrement, bien que cette taille soit plus grande que la taille minimum déterminée précédemment. Mais il y a encore beaucoup de vide dans tout cela...	Cette fois l'encombrement de l'enregistrement est véritablement minimisé : il n'occupe plus que 17 bits, soit à peine plus que deux octets. Il y a encore un peu de perte dans Tab2 (3 bits), parce que le compilateur refuse de créer des tableaux d'une taille non multiple de 8 étant donné le matériel sous-jacent (en l'occurrence, un Intel Pentium 4). L'augmentation de la taille de Tab1 vient du fait qu'on a imposé une taille « trop » grande à ses éléments (ligne 15).

```
 9 r2: TRec(nb) ;
10 r3: TRec(2*nb) ;
11 begin
12    r1 := r2 ;
13    r1 := r3 ;
14 end Record_Param ;
```

Nous déclarons ligne 2 un type tableau non contraint TRec. Il est alors normalement interdit d'utiliser ce type comme élément d'un autre tableau ou d'un autre enregistrement, à moins de le contraindre. Les lignes 3 à 6 définissent un type enregistrement TRec, qui contient justement un membre de ce type tableau. Celui-ci est contraint grâce à un paramètre donné à TRec, nommé taille. Dans le cadre d'un enregistrement, on appelle un tel paramètre un discriminant. Celui-ci fait partie intégrante de TRec, sa valeur étant obtenue lors de l'instanciation du type (lignes 8 à 10). Il joue alors le rôle d'un membre, que l'on peut consulter comme n'importe quel autre (par exemple avec r1.taille), mais dont on ne peut modifier la valeur : une instruction comme r1.taille := 5 est interdite. Remarquez que la valeur du discriminant n'est pas nécessairement statique, elle peut être issue d'une variable ou d'un calcul.

Les affectations entre instances des lignes 12 et 13 sont syntaxiquement valides. Toutefois, si la première ne pose aucun problème, la seconde provoquera la levée d'une exception (nommément Constraint_Error) : les affectations entre enregistrements contenant des discriminants ne sont en effet possibles que si les valeurs des discriminants sont égales. Les discriminants peuvent également être utilisés pour adapter l'apparence d'un type enregistrement. Imaginons un instant que nous voulions réaliser une petite application pour stocker notre vaste collection de livres et de films. Ces deux types d'information ont au moins une chose en commun : le titre. Par contre, on peut vouloir stocker le nombre de pages pour un livre, mais la durée pour un film. Il nous faudrait alors définir deux types différents pour les représenter... à moins d'utiliser un discriminant :

```
 1 type Type_Info is (Livre, Film) ;
 2 type Info(nature: Type_Info) is
 3 record
 4    titre: String(1..200) := (others=>' ') ;
 5    case nature is
 6       when Livre =>
 7          nb_pages: Integer ;
 8          isbn: String(1..13) ;
 9       when Film =>
10          durée: Float ;
11    end case ;
12 end record ;
```

La syntaxe utilisée est très similaire à celle du choix multiple. Selon la valeur du discriminant nature, l'enregistrement Info présentera

différents visages. Si cette valeur est le symbole `Livre`, alors `Info` contiendra (en plus du titre) les éléments `nb_pages` et `isbn`. Si cette valeur est `Film`, alors `Info` contiendra `durée`. Ce procédé est à rapprocher des types `union` du C/C++. Remarquez que les différents visages d'un enregistrement ainsi « discriminé » ne doivent pas nécessairement contenir des types compatibles ou un même nombre de champs.

La prochaine norme Ada apportera la notion d'« interfaces », inspirée par le langage Java, réalisant ainsi une forme limitée d'héritage multiple.

L'utilisation d'un tel type se fait ainsi :

```
1 info1: Info(Livre) ;
2 info2: Info(Film) ;
3 -- .....
4 info1.nb_pages := 200 ;
5 info2.durée := 1.5 ;
6 info2.nb_pages := 1 ;
```

La valeur du discriminant est donnée à la déclaration, puis on accède normalement aux éléments. Remarquez la dernière ligne : elle est syntaxiquement correcte, mais comme `info2` a été déclaré comme étant un `Film`, il ne montre pas de champs nommé `nb_pages`. À l'exécution, l'exception `Contraint_Error` sera levée. Il est également possible de donner une valeur initiale lors de la déclaration d'une variable, ce qui est en fait le seul cas où l'affectation au discriminant est autorisée :

```
1 info3: Info := (
2    nature => Livre,
3    titre => "Titre" & Chaine_Vide(1..195),
4    nb_pages => 100,
5    isbn => "1-2345-6789-A") ;
```

La valeur du discriminant est donnée dans la liste d'initialisation (bien qu'il n'eût pas été incorrect de la répéter dans le nom du type en ligne 1). Cette fois, si on donne une valeur à un élément qui n'existe pas selon le discriminant donné, on obtient une erreur de compilation. De même, si la valeur du discriminant n'est pas donnée avec le nom du type, la liste d'initialisation est obligatoire. Il est ainsi interdit de déclarer une variable simplement avec `info4: Info;`, sauf dans le cas d'un paramètre de sous-programme, par exemple :

```
1 procedure Afficher(i: Info) is
2 begin
3    Put_Line("Titre = " & i.titre) ;
4    case i.nature is
5       when Livre =>
6          Put_Line("ISBN = " & i.isbn) ;
7       when Film =>
8          Put_Line("Durée = " & Float'Image(i.durée)) ;
9    end case ;
10 end Afficher ;
```

Dernier mot, rien n'interdit de déclarer plusieurs discriminants de types différents pour un type enregistrement, de donner une valeur par défaut ou d'utiliser le mécanisme des paramètres nommés lors de la déclaration d'une variable. La syntaxe des discriminants d'enregistrement est en fait assez proche de celle des déclarations et appels de sous-programmes.

Extension

Dernier aspect des enregistrements que nous verrons aujourd'hui, en manière d'anticipation sur de prochains articles : les enregistrements marqués (`tagged`) et l'extension de leur contenu. Prenons un exemple classique, un enregistrement décrivant un cercle :

```
type Cercle is tagged record
   x: Float ;
   y: Float ;
   rayon: Float ;
end record ;
```

Un enregistrement classique, mais remarquez tout de même la présence du mot `tagged` dans la définition. Maintenant nous voudrions une ellipse, décrite par un centre, un petit rayon et un grand rayon. Il serait naturellement possible de recréer un type pour cela. Mais il serait peut-être plus intéressant de conserver le lien de parenté qui existe entre un cercle et une ellipse... c'est-à-dire, de construire notre ellipse comme une extension de notre cercle. Comme ceci :

```
type Ellipse is new Cercle with
record
   grand_rayon: Float ;
end record ;
```

Ceci définit un type `Ellipse` comme étant un nouveau `Cercle` contenant en plus un élément nommé `grand_rayon`. Selon cette représentation, une `Ellipse` « est aussi » un `Cercle` (ce qui est discutable sur le plan mathématique, mais c'est une autre histoire). Dit autrement, le type `Cercle` est l'ancêtre du type `Ellipse` ou encore le type `Ellipse` dérive du type `Cercle`... Certains auront probablement reconnu là un vocabulaire habituellement utilisé pour parler de programmation objet. Bonne nouvelle, c'est précisément de cela qu'il s'agit : ce mécanisme, introduit par la norme Ada95 (donc absent de la norme Ada83), est destiné à apporter à Ada les fonctionnalités d'un langage orienté objet. Ce que nous venons de voir n'est rien de plus que de l'héritage. Ada95 ne supporte que l'héritage simple. La prochaine norme Ada apportera la notion d'« interfaces », inspirée par le langage Java, réalisant ainsi une forme limitée d'héritage multiple. Rassurez-vous, les aspects objet de Ada feront l'objet d'un article spécifique, voire de plusieurs.

Conclusion

Voilà en ce qui concerne les enregistrements en Ada. Comme le laisse supposer la dernière section, nous les retrouverons bientôt – en fait, nous les utiliserons sans cesse au fil de nos découvertes. La prochaine fois, nous aborderons le mécanisme des paquetages, par lequel Ada permet les compilations séparées et la définition et la diffusion de composants réutilisables – autrement dit, nous découvrirons les bibliothèques en Ada.

Yves Bailly,

Le langage Ada 95 – 6
Les paquetages

Yves Bailly

EN DEUX MOTS **Jusqu'ici, nos exemples contenaient en un seul fichier toutes les instructions nécessaires à l'exécution du programme. Pour la création de logiciels complexes de grande envergure, cette manière de faire n'est évidemment pas idéale, sans parler de l'absence totale de modularité, de compilation séparée et de possibilité de réutilisation. Comme tous les langages de haut niveau, Ada propose un mécanisme pour parvenir à ces fins : les paquetages.**

Le mécanisme de paquetages est à la base de l'encapsulation des données en Ada, ainsi que de la compilation séparée des composants d'un programme – de la même façon qu'en C/C++, un logiciel peut être décomposé en bibliothèques compilables séparément et offrant chacune une interface de programmation (API) pour accéder à ses services.

On peut voir un paquetage comme une petite bibliothèque ; on parle toutefois plutôt d'unité de compilation (en anglais *compilation unit*). Incidemment, les programmes que nous avons écrits jusqu'ici, composés d'un unique fichier ne contenant qu'une seule procédure (qui peut elle-même contenir d'autres sous-programmes) sont également désignés par les termes d'« unité de compilation ».

Contrairement à d'autre langages, Ada impose une distinction extrêmement stricte entre la partie déclarative d'un paquetage, où sont annoncés les types et sous-programmes disponibles, et la partie implémentation, qui contient les instructions des sous-programmes annoncés dans la partie déclarative (ainsi que d'autres types et sous-programmes, si besoin est).

Dans le verbiage Ada, la première partie est appelée la « spécification d'un paquetage », tandis que la deuxième est le « corps du paquetage ».

La spécification et le corps

Prenons un exemple élémentaire. Nous allons créer un paquetage qui va contenir deux fonctions, chacune effectuant l'addition de deux entiers donnés en paramètres, mais la première retournant le résultat sous forme d'un entier tandis que l'autre retourne le résultat sous la forme d'un réel. La spécification d'un tel paquetage pourrait ressembler à ceci, dans un fichier `addition.ads` (notez le `s` pour spécification de l'extension) :

```
 1 package Addition is
 2
 3    function Ajoute(int_1: in Integer ;
 4                    int_2: in Integer)
 5       return Integer ;
 6    -- --------------------------------
 7    function Ajoute(int_1: in Integer ;
 8                    int_2: in Integer)
 9       return Float ;
10
11 end Addition ;
```

Remarquez qu'une paire de déclarations équivalentes en C/C++ serait invalide : les deux fonctions ne se distinguent que par leur type de retour, ce qui est parfaitement correct en Ada.

La spécification d'un paquetage est introduite par le mot-clef `package`, suivi du nom du paquetage, suivi de `is`. Puis apparaissent les déclarations de types et de sous-programmes. Enfin la spécification se termine par la répétition du nom du paquetage précédée du mot-clef `end`.

Rien de bien extraordinaire, tout cela ressemble fort à ce que l'on trouve dans un fichier d'en-tête C/C++, à ceci près qu'il est impossible et interdit d'introduire du code (des instructions) au sein de la spécification.

Voyons maintenant le corps de notre paquetage, dans un fichier `addition.adb` (notez le `b` pour *body*, corps, de l'extension) :

```
 1 package body Addition is
 2
 3    function Ajoute(int_1: in Integer ;
 4                    int_2: in Integer)
 5       return Integer is
 6    begin
 7       return int_1 + int_2 ;
 8    end Ajoute ;
 9    -- --------------------------------
10    function Ajoute(int_1: in Integer ;
11                    int_2: in Integer)
12       return Float is
13    begin
14       return Float(int_1 + int_2) ;
15    end Ajoute ;
16
17 end Addition ;
```

Le corps est signalé par la présence du mot-clef `body` entre le mot `package` et le nom du paquetage, en première ligne. Ensuite, on reprend la spécification, en donnant les instructions de chacun des sous-programmes annoncés. Rien n'interdit de déclarer ici de nouveaux types ou de nouveaux sous-

programmes. Toutefois, ceux-ci ne seront pas accessibles par les utilisateurs du paquetage : ils ne seront visibles qu'au sein de ce corps, pas ailleurs. Enfin, on termine comme précédemment, par le mot-clef `end` suivi du nom du paquetage.

Maintenant que nous avons un paquetage, voyons comment l'utiliser.

Utilisation

Réalisons un petit programme de test, dans un fichier `test.adb` :

```
 1 with Text_IO ; use Text_IO ;
 2 with Addition ;
 3 procedure Test is
 4    entier: Integer ;
 5    reel  : Float ;
 6 begin
 7    entier := Addition.Ajoute(1, 2) ;
 8    Put_Line("entier = " & Integer'Image(entier)) ;
 9    reel   := Addition.Ajoute(3, 4) ;
10    Put_Line("reel   = " & Float'Image(reel)) ;
11 end Test ;
```

La première ligne est maintenant familière, elle nous permet d'utiliser la procédure `Put_Line()` pour afficher du texte à l'écran. La deuxième signale que nous souhaitons utiliser un paquetage nommé `Addition`, au moyen d'une clause `with`.

Les lignes 7 et 9 font justement appel aux deux fonctions que nous avons précédemment définies. Ada appliquant un typage strict, il n'y a pas de conversion implicite entre types : c'est ainsi que le type de la variable dans laquelle placer le résultat de chacune des fonctions permet de déterminer laquelle invoquer, sur la base du type de retour.

On utilise ici la notation pointée, que l'on retrouve dans de nombreux autres langages (comme Python) : on fait précéder le nom du sous-programme par un préfixe rappelant dans quel paquetage il se trouve. Cela permet de lever les ambiguïtés, si deux paquetages différents définissent deux sous-programmes parfaitement identiques dans leur déclaration. On peut toutefois éviter ce préfixe, en utilisant une clause `use` à la suite de la clause `with` : c'est ce qui est fait en première ligne pour le paquetage `Text_IO`.

Pour compiler ce petit programme à l'aide du compilateur GNAT, il suffit d'utiliser l'utilitaire gnatmake :

```
$ gnatmake test
gnatgcc -c test.adb
gnatgcc -c addition.adb
gnatbind -x test.ali
gnatlink test.ali
gnatlink: warning: executable name "test" may conflict with shell
command
```

La commande `gnatmake test` va automatiquement chercher un fichier nommé `test.adb` et le compiler, produisant l'habituel fichier `object test.o`, ainsi qu'un fichier des types déclarés privés.

Lors de cette compilation, les clauses `with` vont être interprétées pour déterminer quels paquetages sont nécessaires au fonctionnement du programme. La première ligne va provoquer la recherche d'un fichier `text_io.ads`, la deuxième d'un fichier `addition.ads`.

Le premier sera évidemment trouvé dans un emplacement standard prédéfini propre au compilateur, typiquement `/usr/lib/gcc-lib/i486-linux/2.8.1/adainclude` dans le cas du compilateur GNAT 3.15p ; le deuxième sera recherché dans le répertoire courant, ou dans tout répertoire indiqué par une option `-I` (la même que celle utilisée pour indiquer les fichiers d'en-tête d'un programme C/C++).

Les fichiers `*.ads` trouvés sont lus et interprétés à leur tour. Dans certains cas, par exemple lorsque la spécification d'un paquetage ne contient que des déclarations de types mais pas de sous-programme, cela suffit au bonheur du compilateur.

Mais dans la plupart des situations, une spécification implique un corps quelque part. gnatmake recherche alors une paire de fichiers d'extension `.o` et `.ali` ayant le nom du paquetage concerné – dans notre exemple, `addition.o` et `addition.alious_Paq`. Le contenu de celui-ci est .adb (ici, `addition.adb`) est recherché et compilé, produisant la paire voulue.

Le processus se poursuit récursivement, si un paquetage fait lui-même référence à un ou plusieurs autres paquetages.

À noter que gnatmake recherche et signale les éventuelles interdépendances qui constituent une erreur de compilation.

L'extension .ali signifie *Ada Library Information*. Ces fichiers contiennent de nombreuses informations diverses qui seront utilisées entre autres lors de l'édition des liens, ainsi que la version du compilateur, les options utilisées, des informations de références croisées, etc.

En plus d'être utilisés par gnatmake, ces fichiers sont forts utiles pour des programmes d'analyse de code afin d'examiner la structure d'un logiciel.

Enfin l'édition des liens est réalisée, pour générer un fichier exécutable. Dans notre cas, ce fichier est `test` : remarquez l'avertissement de gnatmake (en réalité, de gnatlink, le programme réalisant l'édition des liens) signalant que le nom de notre exécutable est également celui d'une commande du shell courant.

Par la suite, une nouvelle invocation de gnatmake ne provoquera la recompilation que de ce qui est nécessaire. Si vous modifiez le contenu du fichier `test.adb`, lui seul sera recompilé.

De même, si vous modifiez le contenu de `addition.adb`. Par contre, si vous modifiez le contenu de `addition.ads`, c'est-à-dire de la spécification du paquetage `Addition`, tout sera recompilé.

En effet, le corps d'un paquetage dépend de sa spécification, ainsi que toute unité de compilation (paquetage ou programme principal) référençant cette spécification par une clause `with`.

Voici l'affichage de notre petit programme :

```
$ ./test
entier =  3
reel   =  7.00000E+00
```

Ce qui est bien ce que nous attendions.

Partie privée d'un paquetage

Une spécification, telle que nous l'avons écrite plus haut, décrit la partie publique d'un paquetage, c'est-à-dire ce qui est visible et utilisable depuis l'extérieur du paquetage, par d'autres unités de compilation.

Mais souvent, on souhaite restreindre l'accès à certains éléments du paquetage, notamment aux types.

Reprenons l'exemple consacré aux Tours de Hanoï de l'article précédent. La spécification d'un paquetage qui s'appellerait « `Hanoï` » pourrait ressembler à ceci :

```
 1 package Hanoi is
 2
 3    type Disque is new Integer range 0..10 ;
 4    type Étage is new Integer range 0..10 ;
 5    type Pile_Disques is array (Étage) of Disque ;
 6
 7    type Tour is
 8    record
 9       dernier_étage: Étage := 0 ;
10       pile:          Pile_Disques := (others=>0) ;
11    end record ;
12
13    type Num_Tour is new Integer range 1..3 ;
14    type Rangée_Tours is array (Num_Tour) of Tour ;
15    type Num_Mouvements is new Integer range 0..Integer'Last ;
16    type Jeu_Hanoï is
17    record
18       tours: Rangée_Tours ;
19       nb_mouvements: Num_Mouvements := 0 ;
20    end record ;
21
22    procedure Init(jeu     : out   Jeu_Hanoï ;
23                   nb_init: in    Étage) ;
24
25    procedure Afficher(jeu: in Jeu_Hanoï) ;
26
27 end Hanoi ;
```

Je vous laisse le soin de réaliser le corps de ce paquetage. Un programme de test pourrait être :

```
1 with Hanoi ;
2 procedure Test is
3    jeu: Hanoi.Jeu_Hanoï ;
4 begin
5    Hanoi.Init(jeu, 5) ;
6    Hanoi.Afficher(jeu) ;
7 end Test ;
```

Ce n'est toutefois pas satisfaisant. Les types `Tours` et `Jeu_Hanoï` étant déclarés et définis dans la partie publique et visible du paquetage, un utilisateur pourrait accéder directement aux données contenues dans ces types.

Par exemple, il serait parfaitement possible de réaliser une affectation comme `jeu.tours(1).pile(2) := 3`, ce qui pourrait avoir des conséquences catastrophiques sur le programme : les données ne sont plus cohérentes.

Il est donc nécessaire de protéger les données pour en contrôler l'accès. Autrement dit, nous avons besoin d'encapsulation.

Cela peut se faire au niveau du paquetage au moyen d'une section privée. La spécification du paquetage devient alors (à quelques omissions près) :

```
 1 package Hanoi is
 2
 3 -- types Disque, Étage et Pile_Disques...
 4
 5    type Tour is private ;
 6
 7 -- types Num_Tour, Rangée_Tours et Num_Mouvements...
 8
 9    type Jeu_Hanoï is private ;
10
11 -- les deux procédures, comme précédemment...
12
13 private
14
15    type Tour is
16    record
17       dernier_étage: Étage := 0 ;
18       pile:          Pile_Disques := (others=>0) ;
19    end record ;
20
21    type Jeu_Hanoï is
22    record
23       tours: Rangée_Tours ;
24       nb_mouvements: Num_Mouvements := 0 ;
25    end record ;
26
27 end Hanoi ;
```

Rien ne change dans le corps du paquetage ou le programme de test. Mais désormais, le contenu des types Tour et Jeu_Hanoï ne sont plus visibles, donc accessibles, depuis l'extérieur du paquetage.

Seul le corps peut y avoir accès (ce qui est heureux). Remarquez comment les types sont déclarés, lignes 5 et 9 : ils sont simplement qualifiés de private (privés). Leur description complète n'apparaîtra que dans une section particulière de la spécification, introduite justement par le mot-clef private, ligne 13.

Cette partie privée peut contenir ce que bon vous semble : des définitions de types (qui n'auront pas forcément été annoncés dans la partie publique), des déclarations de sous-programmes, des variables globales (beurk !), etc. Tous ces éléments ne seront pleinement utilisables que dans le corps du paquetage.

Pourquoi alors ne pas se contenter de les déclarer directement dans le corps, du moins pour ceux qui ne sont pas destinés à être visibles de l'extérieur ? Les raisons d'un tel choix apparaîtront bientôt.

Naturellement, il est nécessaire de prévoir des sous-programmes permettant de manipuler les variables des types déclarés privés. Les deux procédures Init() et Afficher() auront les possibilités offertes par l'un des aspects les plus puissants.

Paquetages imbriqués

Pour des besoins d'organisation, il est parfois intéressant de déclarer un paquetage dans un autre. Voici un exemple de spécification :

```
 1 package Paq is
 2    procedure P1 ;
 3    -- ----------------
 4    package Sous_Paq is
 5       procedure SP1 ;
 6    private
 7       procedure SP2 ;
 8    end Sous_Paq ;
 9    -- ----------------
10 private
11    procedure P2 ;
12 end Paq ;
```

Le paquetage Paq définit deux procédures, dont l'une est en section privée, ainsi qu'un paquetage imbriqué nommé Sous_Paq. Ce « sous-paquetage » contient lui-même deux procédures, dont l'une également en section privée.

Depuis l'extérieur de ce paquetage, seules sont accessibles les entités publiquement visibles – ici, les procédures P1 et SP1. On pourrait les utiliser ainsi :

```
1 with Paq ;
2 procedure Test is
3 begin
4    Paq.P1 ;
5    Paq.Sous_Paq.SP1 ;
6 end Test ;
```

Le fait de déclarer l'accès au paquetage Paq (ligne 1) donne automatiquement l'accès au paquetage Sous_Paq.

Son contenu est alors référencé en faisant apparaître le chemin d'accès à ses entités au moyen de la notation pointée (ligne 5).

Mais voyons le corps de ce paquetage et les intéressantes possibilités d'accès qu'on y trouve :

```
 1 with Text_IO ; use Text_IO ;
 2 package body Paq is
 3    procedure P1 is
 4    begin
 5       Put_Line("Paq.P1") ;
 6       P2 ;
 7       Sous_Paq.SP1 ;
 8    end P1 ;
 9    -- ----------------
10    package body Sous_Paq is
11       procedure SP1 is
12       begin
13          Put_Line("Paq.Sous_Paq.SP1") ;
14          P2 ;
15          SP2 ;
16       end SP1 ;
17       procedure SP2 is
18       begin
19          Put_Line("Paq.Sous_Paq.SP2") ;
20          P2 ;
21       end SP2 ;
22    end Sous_Paq ;
23    -- ----------------
24    procedure P2 is
25    begin
26       Put_Line("Paq.P2") ;
27    end P2 ;
28 end Paq ;
```

Dans le corps d'un paquetage, les mentions private disparaissent : tout est public... ou presque.

La procédure P1(), par exemple, a naturellement accès aux entités privées du paquetage dont elle est issue (comme la procédure P2(), invoquée ligne 6), ainsi qu'aux entités publiques du paquetage imbriqué Sous_Paq... mais pas aux entités privées de ce dernier.

Les entités privées de Sous_Paq ne sont accessibles que depuis Sous_Paq. Par contre, comme le montre les corps des procédures SP1() et SP2(), Sous_Paq a accès aux entités privées du paquetage qui le contient : voyez les invocations de la procédure privée P2() depuis SP1() et SP2(), lignes 14 et 20.

D'une manière générale, un paquetage imbriqué a accès aux entités privées (et publiques, bien sûr) du paquetage qui le contient.

Si Sous_Paq contenait lui-même un troisième paquetage Sous_Sous_Paq, celui-ci aurait accès aux entités privées de Sous_Paq et de Paq.

La création de paquetages imbriqués permet de hiérarchiser clairement les idées mises en œuvre dans un programme. Mais il y a encore mieux...

Paquetages enfant

Les paquetages imbriqués présentent quelques inconvénients. Les fichiers ont tendance à devenir fort longs. Ils ne favorisent pas vraiment la compilation séparée des différents éléments du programme, au contraire.

Aussi est-il souvent préférable d'utiliser la technique des paquetages enfant. Reprenons l'exemple précédent, cette fois-ci organisé de la façon suivante :

Paquetage Paq

paq.ads

```
1 package Paq is
2    procedure P1 ;
3 private
4    procedure P2 ;
5 end Paq ;
```

paq.adb

```
1 with Text_IO ; use Text_IO ;
2 with Paq.Sous_Paq ;
3 package body Paq is
4    procedure P1 is
5    begin
6       Put_Line("Paq.P1") ;
7       P2 ;
8       Sous_Paq.SP1 ;
9    end P1 ;
10   procedure P2 is
11   begin
12      Put_Line("Paq.P2") ;
13   end P2 ;
14 end Paq ;
```

Paquetage Paq.Sous_Paq

paq-sous_paq.ads

```
1 package Paq.Sous_Paq is
2    procedure SP1 ;
3 private
4    procedure SP2 ;
5 end Paq.Sous_Paq ;
```

paq-sous_paq.adb

```
1 with Text_IO ; use Text_IO ;
2 package body Paq.Sous_Paq is
3    procedure SP1 is
4    begin
5       Put_Line("Paq.Sous_Paq.SP1") ;
6       P2 ;
7       SP2 ;
8    end SP1 ;
9    procedure SP2 is
10   begin
11      Put_Line("Paq.Sous_Paq.SP2") ;
12      P2 ;
13   end SP2 ;
14 end Paq.Sous_Paq ;
```

Exemple de paquetages enfant

Les noms des fichiers ne doivent rien au hasard. Pour que gnatmake fonctionne dans les meilleures conditions, il y a trois règles simples à respecter :

- Les fichiers contenant des spécifications ont l'extension .ads, ceux contenant des corps, l'extension .adb.
- Le nom des fichiers est le nom du paquetage, en minuscules.
- Dans le cas d'un paquetage enfant, les noms des fichiers sont composés du nom du paquetage précédé du nom du paquetage parent, séparé par un tiret.

Le nom complet du paquetage Sous_Paq est Paq.Sous_Paq, car il est enfant de Paq.

Remplacez le point par un tiret, passez tout en minuscules, et vous obtenez le nom de base des fichiers qui concernent Sous_Paq : paq-sous_paq.ads et paq-sous_paq.adb.

Incidemment, de la dernière règle découle le fait qu'un nom de paquetage ne puisse pas contenir un tiret.

Remarquez que bien que se trouvant dans des fichiers séparés, Sous_Paq a comme précédemment accès aux entités privées de Paq (lignes 6 et 12 dans paq-sous_paq.adb).

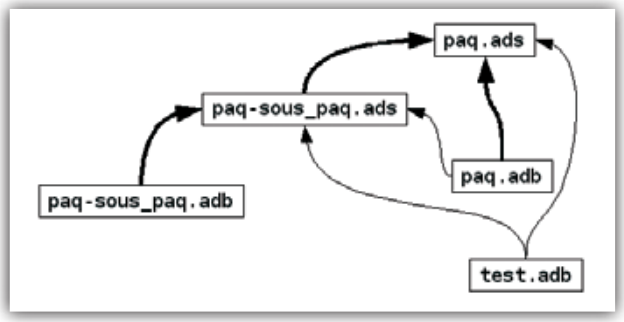
De plus, le corps d'un paquetage parent peut utiliser la spécification d'un de ses paquetages enfant (ligne 2 dans paq.adb).

Par contre, la spécification d'un parent ne peut utiliser la spécification d'un enfant : cela créerait une interdépendance. En effet, implicitement l'enfant dépend du parent.

Le programme de test se voit simplement ajouter une ligne :

```
1 with Paq ;
2 with Paq.Sous_Paq ;
3 procedure Test is
4 begin
5    Paq.P1 ;
6    Paq.Sous_Paq.SP1 ;
7 end Test ;
```

Tout cela est finalement assez naturel. On peut établir un graphe des dépendances de compilation entre les différents fichiers, les flèches se lisant « dépend de » :



Les flèches épaisses signalent les dépendances implicites, impliquées par les relations de parenté d'une part, et les relations entre corps et spécifications d'autre part.

On peut déduire de ce schéma qu'une modification dans paq.ads provoquera une recompilation totale (car tout en dépend, directement ou non). Par contre, une modification dans paq.adb ne provoquera que la recompilation de paq.adb, rien d'autre. Ce qui est tout de même une économie notable. Pour revenir sur l'encapsulation, un paquetage enfant peut être qualifié de privé : il suffit pour cela de placer le mot-clef private devant l'en-tête de la spécification. Par exemple, pour

que `Sous_Paq` devienne un enfant privé de `Paq`, sa première ligne deviendrait :

```
1 private package Paq.Sous_Paq is
```

Le contenu de `Sous_Paq` devient alors inaccessible à toute unité de compilation (programme principal ou autre paquetage), à moins que celle-ci ne soit elle-même un enfant privé de `Paq`.

Quelques paquetages standards

Comme pour la plupart des langages de programmation, le compilateur Ada vient, accompagné d'un certain nombre d'outils généraux, pour accomplir des tâches communes. Sans entrer dans trop de détails, voici quelques paquetages normalement présents avec tout compilateur Ada.

▶Le paquetage Standard contient les définitions des types fondamentaux, ainsi que les opérations disponibles. Cela peut surprendre, mais les types de base que sont `Boolean`, `Integer` ou `Float` ne sont pas définis au sein même du compilateur, mais dans ce paquetage. Il est inutile de l'inclure avec un `with` Standard, il est toujours implicitement disponible.

▶`Ada.Characters.Handling`, paquetage enfant de `Characters` lui-même enfant de `Ada`, offre des facilités pour l'utilisation des caractères comme des fonctions de classification (pour savoir si un caractère est une lettre, un chiffre...) ou de conversion depuis ou vers les caractères Unicode.

▶`Ada.Strings` concerne les chaînes de caractères et se décompose en plusieurs enfants, comme `Ada.Strings.Unbounded` pour manipuler des chaînes à longueur variable ou `Ada.Strings.Wide_Unbounded` pour des chaînes Unicode.

▶`Ada.Text_IO` (dont le nom peut se réduire à `Text_IO`) et `Ada.Wide_Text_IO` fournissent fonctions et procédures pour l'entrée-sortie de caractères, le second destiné à l'Unicode. Nous avons par exemple déjà fréquemment utilisé la procédure `Put_Line`.

▶Le paquetage `System` contient des définitions dépendantes de l'implémentation et de la plate-forme. Par exemple, `System.Max_Int` contient la valeur du plus grand entier positif signé disponible. `System.Address` est un type représentant l'adresse d'une unité de mémoire, chacune de ces unités étant composée de `System.Storage_Unit` bits. Par exemple, sur une plate-forme « classique » comme un PC, `System.Storage_Unit` vaut 8 (un octet) et `System.Address` n'est rien d'autre qu'un pointeur non typé.

Il en existe de nombreux autres que nous verrons progressivement selon nos besoins.

Conclusion

Voilà pour cette petite présentation des paquetages en Ada. Ils sont un outil essentiel pour la bonne organisation d'un logiciel et l'encapsulation de ses données. La prochaine fois, nous nous pencherons sur la généricité, technique d'une grande puissance, offerte par le langage Ada dès sa création en 1979, bien avant les *templates* du langage C++.

Yves Bailly,

Le langage Ada 95 – 7 : La généricité

Yves Bailly

EN DEUX MOTS **Assez naturelle dans les langages interprétés comme Perl ou Python (quoique l'on parle plutôt de typage dynamique), la généricité est considérée comme une fonctionnalité de très haut niveau dans les langages compilés, apportant un outil extrêmement puissant pour réaliser l'abstraction des données et ainsi facilitant grandement la réutilisation du code laborieusement écrit. Voyons comment Ada se positionne.**

Le principe de la généricité, si vous ne le connaissez pas déjà, est en fait assez simple. L'idée est de pouvoir écrire du code, d'implémenter un algorithme, sans connaître précisément le type des données mises en jeu dans les traitements. C'est une manière de s'affranchir (partiellement) du typage statique habituellement rencontré.

Par exemple, un algorithme de tri reste identique, qu'il s'agisse de trier des entiers, des réels ou des pommes ; il suffit de disposer d'un moyen pour comparer deux objets. On est alors plus intéressé par une opération disponible sur le type que par le type lui-même, qui est finalement sans grand intérêt. Mais cette idée simple, comme nous allons le voir, peut donner lieu à des développements assez sophistiqués.

[1] Document STEELMAN : http://en.wikisource.org/wiki/Steelman_language_requirements

Petit historique

La généricité faisait partie des exigences du cahier des charges émis par le Département de la Défense des États-Unis d'Amérique en 1977 (document « STEELMAN » [1]), alors demandeur d'un nouveau langage de programmation pour remplacer les nombreux langages qui étaient utilisés à ce moment.

Ayant remporté le concours, cette facilité figure donc au nombre des nombreuses fonctionnalités du langage Ada dès cette époque – en fait, Ada est le premier langage compilé à proposer pleinement la généricité.

Celle-ci apparaîtra plus tard dans des langages orientés objet comme Eiffel (1985) ou C++, ce dernier popularisant la notion au moyen des *templates*. Les langages plus récents comme Java ou C# ne proposeront une forme de généricité que bien plus tard.

On peut donc dire que Ada, dans ce domaine comme dans bien d'autres, a été un précurseur.

Sous-programme générique

Considérons dans un premier temps un exemple d'un classicisme à faire peur : nous souhaitons créer une procédure permettant d'échanger la valeur de deux variables. Son code sera trivial, mais imaginez qu'il s'agit de quelque chose de plus compliqué. L'approche consistant à surdéfinir la procédure, et donc dupliquer le code, pour chaque type concerné n'est pas vraiment efficace. La généricité permet de simplifier grandement la tâche.

Commençons par un programme simple, placé dans un fichier `swap_1.adb` :

```
 1 with Text_IO ;
 2 use  Text_IO ;
 3 procedure Swap_1 is
 4    generic
 5       type Un_Type is private ;
 6    procedure Generic_Swap(v1: in out Un_Type ;
 7                           v2: in out Un_Type) ;
 8
 9    procedure Generic_Swap(v1: in out Un_Type ;
10                           v2: in out Un_Type) is
11       temp: Un_Type := v1 ;
12    begin
13       v1 := v2 ;
14       v2 := temp ;
15    end Generic_Swap ;
16
17    procedure Integer_Swap is new Generic_Swap(Integer) ;
18    procedure Float_Swap is new Generic_Swap(Un_Type => Float) ;
19
20 i1: Integer := 1 ;
21 i2: Integer := 2 ;
22 f1: Float := 11.0 ;
23 f2: Float := 22.0 ;
24
25 begin
26    Put_Line("i1 = " & Integer'Image(i1) &
27       ", i2 = " & Integer'Image(i2)) ;
28    Integer_Swap(i1, i2) ;
29    Put_Line("i1 = " & Integer'Image(i1) &
30       ", i2 = " & Integer'Image(i2)) ;
31
32    Put_Line("f1 = " & Float'Image(f1) &
33       ", f2 = " & Float'Image(f2)) ;
34    Float_Swap(f1, f2) ;
35    Put_Line("f1 = " & Float'Image(f1) &
36       ", f2 = " & Float'Image(f2)) ;
37 end Swap_1 ;
```

Notre procédure générique est déclarée lignes 4 à 7. Cette déclaration (sans code) est nécessaire pour tout sous-programme générique. Elle est introduite par le mot-clef `generic`, suivi des paramètres génériques formels. Ici un seul paramètre est utilisé : il s'agit d'un type de données (`type`),

connu dans la procédure sous le nom `Un_Type`, dont le contenu est inconnu (`is private`). Ce peut être aussi bien un type fondamental qu'un enregistrement ou un type tableau. Le préfixe `Generic_` dans le nom de la procédure n'a rien d'obligatoire, ce n'est qu'une convention de *nommage*.

Ensuite vient le code proprement dit de la procédure, lignes 9 à 15. Le type générique est utilisé naturellement, la seule hypothèse qui est faite étant qu'il est possible de réaliser des affectations entre objets de ce type.

Il est tout simplement interdit de réaliser toute autre hypothèse sur le type : par exemple, tenter d'additionner les deux paramètres v1 et v2 résulterait en une erreur de compilation, même si le type réel permet une telle opération. C'est là une différence fondamentale avec le mécanisme des templates du C++. Voici une déclaration à peu près équivalente en C++ :

```
1 template <class Un_Type>
2 void generic_swap(Un_Type& v1, Un_Type& v2)
3 {
4    Un_Type temp = v1 + v2 ;
5    v1 = v2 ;
6    v2 = temp ;
7 }
```

L'addition en ligne 4 ne poserait aucun problème, tant que le type réel donné pour `Un_Type` accepte cette opération pour chacune des instanciations rencontrées. L'erreur de compilation ne surviendra que dans le cas contraire (par exemple, en tentant d'invoquer `generic_swap()` avec des pointeurs en paramètres).

Ada est beaucoup plus restrictif : même si toutes les instanciations permettent l'addition, celle-ci sera refusée par le compilateur, car elle sous-entend une hypothèse sur le type générique `Un_Type` qui est interdite par la qualification `is private`. À noter une autre différence fondamentale avec le C++, ou du moins les compilateurs les plus répandus : en Ada, les sous-programmes génériques sont effectivement compilés à part, et non pas seulement au moment de l'instanciation. L'obligation qui existe avec la plupart des compilateurs C++ de faire apparaître dans une même unité de compilation le code générique et le code qui l'utilise n'existe pas en Ada, comme nous le verrons plus loin avec les paquetages génériques.

Voyons justement comment effectuer l'instanciation de notre procédure générique. Cela revient en fait à créer une nouvelle procédure, en indiquant le type réel souhaité : deux exemples sont donnés en lignes 17 et 18. Cette instanciation doit être explicite : contrairement au C++, il est impossible d'invoquer `Generic_Swap()` directement. La ligne 18 montre qu'il est possible de nommer le paramètre générique lorsqu'on lui affecte sa « valeur », de la même manière que les paramètres d'un sous-programme peuvent être nommés. Cela n'a rien d'obligatoire, mais clarifie grandement le code et facilite sa relecture. Un principe clef dans la conception du langage Ada était que le code est écrit une fois, mais relu de nombreuses fois.

La suite du programme ne fait qu'illustrer la mise en œuvre de la procédure générique instanciée. Les deux instanciations sont utilisées naturellement, comme s'il s'agissait de procédures sans rien de particulier. Voici la sortie de ce programme :

```
$ gnatmake swap_1 && ./swap_1
gnatgcc -c swap_1.adb
gnatbind -x swap_1.ali
gnatlink swap_1.ali
i1 =  1, i2 =  2
i1 =  2, i2 =  1
f1 =  1.10000E+01, f2 =  2.20000E+01
f1 =  2.20000E+01, f2 =  1.10000E+01
```

L'instanciation explicite peut paraître lourde, mais encore une fois elle permet de clarifier le code. Elle est de plus nécessaire, étant donné qu'en Ada il n'existe tout simplement pas de conversions implicites d'un type vers un autre. Par ailleurs, nous verrons plus loin comment limiter ou renforcer les restrictions imposées aux paramètres génériques.

Poursuivons. Notre procédure fonctionne. Elle est manifestement d'une utilité très générale : plutôt que de la placer ainsi directement dans un programme, il serait préférable de l'empaqueter dans un paquetage. Il suffit de placer la déclaration dans une spécification et l'implémentation dans un corps. Voici ce que deviendrait notre programme si la procédure était placée dans un paquetage `Swap` (dans les fichiers `swap.ads` et `swap.adb`) :

Paquetage générique

Un sous-programme générique, c'est bien. Un ensemble de sous-programmes regroupés dans un paquetage générique, c'est mieux. Imaginons par exemple une série de sous-programmes qui manipuleraient des tableaux, comme rechercher un élément, trier les éléments... Quels seraient les paramètres (types) génériques ?

- Clairement, le type d'un élément de tableau ;
- Très probablement, le type tableau lui-même, de préférence non contraint (bornes indéfinies) ;
- Moins évident peut-être, mais nécessaire, le type de l'indice du tableau : rappelez-vous qu'en Ada, les indices d'un tableau peuvent être de n'importe quel type discret (types entiers ou énumérés), ces indices ne commençant pas nécessairement à zéro...

Cela nous fait trois paramètres. Il serait fastidieux de les répéter pour chaque sous-programme : la création d'un paquetage générique s'impose d'elle-même. Notre paquetage va fournir quatre sous-programmes :

Procédure générique dans un paquetage

swap.ads

```
1 package Swap is
2    generic
3       type Un_Type is private ;
4    procedure Generic_Swap(v1: in out Un_Type ;
5                           v2: in out Un_Type) ;
6 end Swap ;$
```

swap.adb

```
1 package body Swap is
2    procedure Generic_Swap(v1: in out Un_Type ;
3                           v2: in out Un_Type) is
4       temp: Un_Type := v1 ;
5    begin
6       v1 := v2 ;
7       v2 := temp ;
8    end Generic_Swap ;
9 end Swap ;
```

swap_2.adb

```
 1 with Text_IO ;
 2 use  Text_IO ;
 3 with Swap ;
 4 procedure Swap_2 is
 5
 6    type Tableau is array(1..10) of Float ;
 7
 8    procedure Integer_Swap is new Swap.Generic_Swap(Integer) ;
 9    procedure Tableau_Swap is new Swap.Generic_Swap(Un_Type => Tableau) ;
10
11 i1: Integer := 1 ;
12 i2: Integer := 2 ;
13 t1: Tableau := (others => 11.0) ;
14 t2: Tableau := (others => 22.0) ;
15
16 begin
17
18    Put_Line("i1 = " & Integer'Image(i1) &
19       ", i2 = " & Integer'Image(i2)) ;
20    Integer_Swap(i1, i2) ;
21    Put_Line("i1 = " & Integer'Image(i1) &
22       ", i2 = " & Integer'Image(i2)) ;
23
24    Put_Line("t1(3) = " & Float'Image(t1(3)) &
25       ", t2(3) = " & Float'Image(t2(3))) ;
26    Tableau_Swap(t1, t2) ;
27    Put_Line("t1(3) = " & Float'Image(t1(3)) &
28       ", t2(3) = " & Float'Image(t2(3))) ;
29
30 end Swap_2 ;
```

Juste pour l'exemple, la deuxième instanciation permet d'échanger les valeurs contenues dans deux tableaux dont le type est déclaré ligne 6. Le paquetage Swap est intégré ligne 3, le nom de la procédure générique étant simplement préfixé (lignes 8 et 9). L'écriture est finalement assez naturelle : déclarez le sous-programme générique dans une spécification d'un paquetage, implémentez-le dans le corps du paquetage, puis instanciez et utilisez ce sous-programme dans un programme principal ou un autre paquetage.

- Une procédure pour échanger deux valeurs à deux indices différents d'un tableau ;
- Une procédure pour trier les éléments du tableau, indépendante de la fonction de comparaison ;
- Une fonction pour vérifier si un tableau est trié ou non, également indépendante de la fonction de comparaison ;
- Enfin, une procédure pour afficher le contenu d'un tableau.

Définition

Voici quelle pourrait être une spécification d'un tel paquetage :

```
 1 generic
 2    type Type_Elements is private ;
 3    type Type_Indice is (<>) ;
 4    type Type_Tableau is
 5       array (Type_Indice range <>) of Type_Elements ;
 6 package Generic_Tableaux is
 7
 8    procedure Echanger(tab  : in out Type_Tableau ;
 9                       ind_1: in     Type_Indice ;
10                       ind_2: in     Type_Indice) ;
11
12    generic
13       with function "<" (e1: in Type_Elements ;
14                          e2: in Type_Elements)
15          return Boolean ;
16    procedure Generic_Trier(tab: in out Type_Tableau) ;
```

```
17
18    generic
19       with function "<" (e1: in Type_Elements ;
20                          e2: in Type_Elements)
21          return Boolean ;
22    function Generic_Est_Trie(tab: in Type_Tableau)
23       return Boolean ;
24
25    generic
26       with function Element_To_String(elem: in Type_Elements)
27          return String ;
28    procedure Generic_Put_Line(tab: in Type_Tableau) ;
29
30 end Generic_Tableaux ;
```

Quelques commentaires. D'abord sur les paramètres génériques. Le premier d'entre eux, `Type_Elements`, représente le type des éléments du tableau : on retrouve la même notation (`is private`) que dans la section précédente, qui nous indique qu'aucune hypothèse n'est faite sur le type, en dehors de l'affectation standard prédéfinie.

Le deuxième, `Type_Indice`, représente le type utilisé pour les indices du type tableau que nous allons vouloir manipuler. En lieu et place de `private`, on trouve le symbole (`<>`) (que l'on peut lire « boîte », ou « *box* » an anglais). Celui-ci indique que le type en question doit être un type « discret », c'est-à-dire un type entier ou un type énuméré. On restreint dès lors fortement les types pouvant être employés pour ce paramètre, ce qui améliore l'intégrité sémantique du programme. En contrepartie, il est possible de faire certaines hypothèses sur les objets de type `Type_Indice`, en particulier qu'ils sont ordonnés, qu'il existe un premier et un dernier, qu'ils possèdent tous un prédécesseur et un successeur, qu'il est possible de les afficher (d'obtenir une chaîne de caractères représentant la valeur de chacun d'eux).

Enfin, le paramètre `Type_Tableau` décrit précisément le type tableau attendu : il s'agit d'un tableau non contraint (`range <>`) dont les éléments sont de type `Type_Elements`, indexés par des valeurs de type `Type_Indice`. Au sein du corps du paquetage, nous pourrons donc utiliser toutes les facilités offertes par Ada pour manipuler les tableaux, ce dont nous n'allons pas nous priver.

Jetons un œil aux sous-programmes proposés. Le premier, `Echanger()`, ne présente rien de particulier : on y retrouve simplement les types d'indice et de tableaux pour réaliser l'échange de deux valeurs. Le deuxième, `Generic_Trier()`, est plus intéressant : c'est une procédure de tri, elle-même générique – un peu comme une méthode template au sein d'une classe `template` en C++. Mais le paramètre générique n'est pas un type : c'est un sous-programme, en l'occurrence une fonction, qui sera utilisée pour comparer les éléments entre eux.

Ce paramètre générique est indispensable : tel qu'a été déclaré le paramètre générique correspondant au type des éléments du tableau, encore une fois, on ne peut faire aucune hypothèse quant aux fonctionnalités du type (en-dehors de l'affectation prédéfinie). En particulier, il n'existe pas de relation d'ordre implicite. C'est pourquoi il est nécessaire d'en « passer » une à la procédure de tri. Le choix a été fait de nommer cette fonction de comparaison `"<"`, afin de pouvoir comparer deux objets `a` et `b` de type `Type_Elements` à l'aide de la simple écriture `a < b`. On aurait tout aussi bien pu l'appeler (par exemple) `Comparer()`, mais le code de la procédure de tri s'en serait trouvé inutilement alourdi. Une solution alternative eût été d'ajouter un paramètre à la procédure de tri, qui aurait été un pointeur sur une fonction de comparaison – mais nous n'avons pas encore étudié les pointeurs (ou ce qui en tient lieu) en Ada...

La procédure suivante, `Generic_Est_Trie()`, a pour vocation de vérifier si le contenu d'un tableau est trié ou non. À nouveau, nous devons passer une fonction de comparaison. Enfin, la dernière procédure, `Generic_Put_Line()`, affiche le contenu d'un tableau. Cette fois le paramètre générique attendu est une fonction effectuant la conversion d'un objet de type `Type_Elements` en une chaîne de caractères.

Le corps du paquetage ne présente pas grand-chose de particulier. Examinons tout de même le code de la procédure de tri (basée sur l'algorithme récursif QuickSort) où apparaît une ou deux petites subtilités :

```
 1 with Text_IO ;
 2 use  Text_IO ;
 3 package body Generic_Tableaux is
   .....
17
18    procedure Generic_Trier(tab: in out Type_Tableau) is
19       i_inf: Type_Indice := tab'First ;
20       i_sup: Type_Indice := tab'Last ;
21       i_mid: Type_Indice :=
22          Type_Indice'Val(
23             (Type_Indice'Pos(i_inf)+Type_Indice'Pos(i_sup))/2) ;
24       pivot: Type_Elements ;
25    begin
26       if tab'Length < 2
27       then
28          return ;
29       elsif tab'Length = 2
30       then
31          if tab(i_sup) < tab(i_inf)
32          then
33             Echanger(tab, i_sup, i_inf) ;
34          end if ;
35       else
36          if tab(i_mid) < tab(i_inf)
37          then
38             Echanger(tab, i_mid, i_inf) ;
39          end if;
40          if tab(i_sup) < tab(i_mid)
41          then
42             Echanger(tab, i_sup, i_mid) ;
43             if tab(i_mid) < tab(i_inf)
44             then
45                Echanger(tab, i_mid, i_inf) ;
46             end if;
```

```
47          end if;
48          if tab'Length > 3
49          then
50             -- cas général
51             i_inf := Type_Indice'Succ(i_inf) ;
52             i_sup := Type_Indice'Pred(i_sup) ;
53             pivot := tab(i_mid) ;
54             DECOUPAGE:
55             loop
56                ELEM_INF:
57                loop
58                   exit DECOUPAGE when i_sup <= i_inf ;
59                   exit ELEM_INF when pivot < tab(i_inf) ;
60                   i_inf := Type_Indice'Succ(i_inf) ;
61                end loop ELEM_INF ;
62                ELEM_SUP:
63                loop
64                   exit DECOUPAGE when i_sup <= i_inf ;
65                   exit ELEM_SUP when tab(i_sup) < pivot ;
66                   i_sup := Type_Indice'Pred(i_sup) ;
67                end loop ELEM_SUP ;
68                Echanger(tab, i_inf, i_sup) ;
69                i_inf := Type_Indice'Succ(i_inf) ;
70                i_sup := Type_Indice'Pred(i_sup) ;
71             end loop DECOUPAGE ;
72             Generic_Trier(tab(tab'First..i_inf)) ;
73             Generic_Trier(tab(i_sup..tab'Last)) ;
74          end if ;
75       end if ;
76    end Generic_Trier ;
77
   ......
110 end Generic_Tableaux ;
```

Pour mémoire, l'algorithme récursif QuickSort repose sur le découpage en deux du tableau de départ en une partie inférieure et une partie supérieure, contenant les éléments respectivement plus petits et plus grands qu'une valeur pivot, chacune des deux parties étant triée à son tour. Ici la valeur pivot est extraite du tableau lui-même, en l'occurrence la valeur de l'élément situé au milieu (c'est parfaitement arbitraire et probablement pas optimal). Nous avons donc besoin au minimum de trois indices : un pour parcourir le tableau du début vers la fin (`i_inf`, ligne 19), un pour le parcourir en sens inverse depuis la fin (`i_sup`, ligne 20) et enfin un troisième correspondant à l'indice médian – le milieu du tableau, `i_mid` en ligne 21.

Il va donc nous falloir manipuler des indices, ce qui n'est pas une surprise. Ceux-ci sont représentés par le type générique `Type_Indice`, déclaré comme étant d'un type discret... sans plus de contrainte. Conséquence : si i est de type `Type_Indice`, l'écriture `i+1` n'est pas forcément légitime – donc elle ne l'est pas du tout. En fait, toute opération arithmétique est interdite. Comment faire alors ?

En faisant appel aux attributs. Nous avons vu dans le quatrième article de cette série, consacré aux tableaux, les attributs qui pouvaient être appliqués aux types discrets. Notre paramètre type générique `Type_Indice` étant un type discret (par déclaration), on peut parfaitement lui appliquer ceux-ci. Ainsi, pour incrémenter notre indice inférieur, plutôt que d'écrire `i_inf := i_inf + 1` nous écrirons `i_inf := Type_Indice'Succ(i_inf)`, l'attribut `'Succ` donnant le successeur d'une valeur d'un type discret (ligne 51, par exemple). C'est là la première subtilité dans le code de cette procédure.

La deuxième réside dans la récursivité, mise en œuvre lignes 72 et 73. Si vous avez déjà codé l'algorithme QuickSort, peut-être avez-vous été étonné de ne pas voir en paramètre à la procédure les bornes du sous-tableau à trier. C'est en fait inutile en Ada : les types tableau sont des types « riches », ils contiennent l'information concernant les bornes inférieures et supérieures des indices. Mais regardez tout de même ces lignes 72 et 73 : on invoque la procédure récursivement, en lui passant une tranche du tableau initial.

On pourrait s'attendre à ce que la procédure reçoive une copie de cette tranche, et donc n'agisse pas sur le tableau lui-même. Il n'en est rien : les tranches ainsi données sont en fait des références au tableau de départ (enfin, à une partie). Cela ne fonctionne que parce que le mode de passage du paramètre tableau est `in out`, qui équivaut à un passage par référence (avec `&`) en C++.

Instanciation

Maintenant que nous avons notre paquetage générique, voyons comment l'utiliser dans un programme de test, que voici :

```
 1 with Text_IO ;
 2 use  Text_IO ;
 3 with Generic_Tableaux ;
 4 procedure Test_Tableaux is
 5
 6    type Tab_Integer is array (Positive range <>) of Integer ;
 7
 8    package Integer_Tableaux is
 9       new Generic_Tableaux(Type_Elements => Integer,
10                            Type_Indice   => Positive,
11                            Type_Tableau  => Tab_Integer) ;
12
13    procedure Trier is
14       new Integer_Tableaux.Generic_Trier("<" => "<") ;
15
16    function Est_Trie_1 is
17       new Integer_Tableaux.Generic_Est_Trie("<" => "<=") ;
18    function Est_Trie_2 is
19       new Integer_Tableaux.Generic_Est_Trie("<" => "<") ;
20
21    procedure Put_Line is
22       new Integer_Tableaux.Generic_Put_Line(Integer'Image) ;
23
24 tab_1: Tab_Integer(1..10) := (4, 8, 5, 2, 0, 5, 9, 6, 1, 7) ;
25
26    type Jours is (Lundi, Mardi, Mercredi,
27                   Jeudi, Vendredi, Samedi,
28                   Dimanche) ;
29    type Mois is (Janvier,   Février, Mars,     Avril,
30                  Mai,       Juin,    Juillet,  Août,
31                  Septembre, Octobre, Novembre, Décembre) ;
32    type Tab_Repos is array (Mois range <>) of Jours ;
```

```
33
34    package Jours_Tableaux is
35       new Generic_Tableaux(Type_Elements => Jours,
36                            Type_Indice   => Mois,
37                            Type_Tableau  => Tab_Repos) ;
38
39    procedure Trier is
40       new Jours_Tableaux.Generic_Trier("<" => "<") ;
41
42    function Est_Trie_1 is
43       new Jours_Tableaux.Generic_Est_Trie("<" => "<=") ;
44    function Est_Trie_2 is
45       new Jours_Tableaux.Generic_Est_Trie("<" => "<") ;
46
47    procedure Put_Line is
48       new Jours_Tableaux.Generic_Put_Line(Jours'Image) ;
49
50 tab_2: Tab_Repos(Mois) := (Lundi, Vendredi, Mercredi, Mardi,
51                            Jeudi, Lundi, Dimanche, Samedi,
52                            Mercredi, Jeudi, Mardi, Samedi) ;
53
54 begin
55
56    Put("tab_1 = ") ; Put_Line(tab_1) ;
57    Put_Line("Est trié (1) ? -> " & Boolean'Image(Est_Trie_1(tab_1))) ;
58    Put_Line("Est trié (2) ? -> " & Boolean'Image(Est_Trie_2(tab_1))) ;
59    Trier(tab_1) ;
60    Put("tab_1 = ") ; Put_Line(tab_1) ;
61    Put_Line("Est trié (1) ? -> " & Boolean'Image(Est_Trie_1(tab_1))) ;
62    Put_Line("Est trié (2) ? -> " & Boolean'Image(Est_Trie_2(tab_1))) ;
63    Put_Line("------------") ;
64    Put("tab_2 = ") ; Put_Line(tab_2) ;
65    Put_Line("Est trié (1) ? -> " & Boolean'Image(Est_Trie_1(tab_2))) ;
66    Put_Line("Est trié (2) ? -> " & Boolean'Image(Est_Trie_2(tab_2))) ;
67    Trier(tab_2) ;
68    Put("tab_2 = ") ; Put_Line(tab_2) ;
69    Put_Line("Est trié (1) ? -> " & Boolean'Image(Est_Trie_1(tab_2))) ;
70    Put_Line("Est trié (2) ? -> " & Boolean'Image(Est_Trie_2(tab_2))) ;
71
72 end Test_Tableaux ;
```

Il est naturellement nécessaire d'indiquer que nous allons utiliser le paquetage (ligne 3). Le programme définit ensuite un premier type tableau, des entiers indexés par des entiers positifs (ligne 6). L'instanciation du paquetage générique se fait de façon similaire à l'instanciation d'un sous-programme générique, sauf que plutôt que de créer un nouveau sous-programme, on crée un nouveau paquetage, lignes 8 à 11. Le nommage des paramètres est utilisé pour clarifier le code. Cette instruction revient véritablement à créer un nouveau paquetage, exactement comme si on avait intégré une spécification et un corps dans notre programme (ce qui est par ailleurs parfaitement autorisé) pour un paquetage qui serait ici nommé `Integer_Tableaux`.

Nous n'en avons pas terminé avec les instanciations. Notre nouveau paquetage contient trois sous-programmes eux-mêmes génériques : il nous faut les instancier à leur tour pour pouvoir les utiliser. On commence par la procédure de tri, lignes 13 et 14. Le paramètre générique attendu doit être une fonction de comparaison : on lui donne la fonction prédéfinie pour le type des éléments des tableaux. Comme ce sont des entiers, cette fonction prédéfinie a pour nom `"<"`... ce qui donne l'écriture un peu étrange `"<" => "<"`. Le premier `"<"` est le nom donné au paramètre générique formel, le deuxième est le nom de la fonction donnée. N'ayez crainte, le compilateur s'y retrouve fort bien et avec un peu de pratique l'écriture devient naturelle.

Ensuite, on instancie à deux reprises la procédure de vérification du tri, une première fois avec une comparaison stricte, une deuxième avec une comparaison large. La première renverra toujours « faux » (`False`) pour un tableau contenant deux fois la même valeur, qu'il ait été trié ou non. Enfin vient le tour de la procédure d'affichage : on donne comme fonction de conversion en chaîne de caractères tout simplement l'attribut `'Image` du type des éléments, qui existe puisque ce sont des entiers. Cet attribut n'est pas utilisable dans le corps du paquetage générique : le paramètre `Type_Elements` étant déclaré `is private`, l'attribut n'est pas visible à ce niveau. Il est par contre visible au moment de l'instanciation.

Continuons. Après cette première instanciation « évidente », voyons une instanciation utilisant non pas des entiers, mais des types énumérés que nous aurons créés nous-mêmes. Ces types sont déclarés lignes 26 à 31, le type tableau étant défini ligne 32. L'instanciation du paquetage générique puis des sous-programmes génériques qu'il contient se fait exactement de la même manière que précédemment, comme si les types utilisés étaient des types entiers.

Le corps du programme lui-même se contente de faire quelques invocations. Remarquez qu'il est possible de donner des noms identiques à deux instanciations différentes d'une même procédure générique : cela n'est autorisé, naturellement, que si les types des paramètres de cette procédure permettent sans ambiguïté de résoudre la surcharge.

Voici l'affichage du programme :

```
$ gnatmake -gnatif test_tableaux && ./test_tableaux
gnatgcc -c -gnatif test_tableaux.adb
gnatgcc -c -gnatif generic_tableaux.adb
gnatbind -x test_tableaux.ali
gnatlink test_tableaux.ali
tab_1 = [ 1 =>  4,  2 =>  8,  3 =>  5,  4 =>  2,  5 =>  0,  6 =>  5,
7 =>  9,  8 =>  6,  9 =>  1,  10 =>  7]
Est trié (1) ? -> FALSE
Est trié (2) ? -> FALSE
tab_1 = [ 1 =>  0,  2 =>  1,  3 =>  2,  4 =>  4,  5 =>  5,  6 =>  5,
7 =>  6,  8 =>  7,  9 =>  8,  10 =>  9]
Est trié (1) ? -> TRUE
Est trié (2) ? -> FALSE
------------
tab_2 = [JANVIER => LUNDI, FéVRIER => VENDREDI, MARS => MERCREDI,
AVRIL => MARDI, MAI => JEUDI, JUIN => LUNDI, JUILLET => DIMANCHE,
AOûT => SAMEDI, SEPTEMBRE => MERCREDI, OCTOBRE => JEUDI, NOVEMBRE
```

```
=> MARDI, DéCEMBRE => SAMEDI]
Est trié (1) ? -> FALSE
Est trié (2) ? -> FALSE
tab_2 = [JANVIER => LUNDI, FéVRIER => LUNDI, MARS => MARDI, AVRIL =>
MARDI, MAI => MERCREDI, JUIN => MERCREDI, JUILLET => JEUDI, AOûT =>
JEUDI, SEPTEMBRE => VENDREDI, OCTOBRE => SAMEDI, NOVEMBRE => SAMEDI,
DéCEMBRE => DIMANCHE]
Est trié (1) ? -> TRUE
Est trié (2) ? -> FALSE
```

Il semble bien que notre paquetage générique fonctionne.

Aparté : question de performances

Le langage Ada possède une réputation de lenteur, c'est-à-dire que si on considère la vitesse d'exécution de deux codes comparables, l'un en C++, l'autre en Ada, le premier sera toujours beaucoup plus rapide que le second.

Vérifions cela avec notre petit paquetage. Deux programmes, l'un en Ada et l'autre en C++, vont chacun créer un tableau de dix millions d'entiers aléatoires, puis trier ce tableau. Le tri sera effectué en Ada par la procédure `Generic_Trier()` présentée plus haut et par la fonction générique standard `std::sort()` en C++. Pour les tests, deux versions des compilateurs ont été utilisées pour chacun des langages :

[2] Compilateur GNAT GPL 2005 : http://libre.adacore.com/

- Pour le programme en Ada, le compilateur GNAT a été utilisé :
  - la version 3.15p, généralement celle fournie avec les distributions Linux, qui s'appuie sur GCC 2.8.1 ;
  - la version GPL 2005 apparue en août dernier, qui s'appuie sur GCC 3.4.3 [2].
- Pour le programme C++, c'est directement g++ qui a été utilisé :
  - d'abord la version 3.3.5 ;
  - puis la version 3.4.3.

Ce ne sont pas les dernières versions disponibles, mais les versions généralement considérées comme étant les plus stables disponibles. Par ailleurs, les temps d'exécution du programme Ada ont été mesurés avec ou sans le paramètre `-gnatp` passé au compilateur, qui a pour effet de désactiver la plupart des tests et contrôles effectués durant l'exécution, comme ceux sur les indices de tableau pour détecter les débordements.

Chaque test est effectué à trois reprises, la valeur retenue étant la moyenne. Voici les résultats, les temps sont exprimés en secondes :

Options	Vitesses de tri en Ada et C++				
	Ada (GNAT)			C++ (g++)	
	options	*3.15p*	*GPL 2005*	*3.3.5*	*3.4.3*
Aucune	aucune	8.90	6.97	2.96	2.96
	`-gnatp`	4.70	5.13		
`-02`	aucune	4.70	3.61	1.82	1.78
	`-gnatp`	3.64	3.11		
`-03`	aucune	4.73	3.32	1.79	1.78
	`-gnatp`	3.36	2.67		

Les deux valeurs les plus comparables sont celles donnant les temps pour GNAT GPL 2005 avec `-gnatp` d'une part, et pour g++ 3.4.3 d'autre part.

D'abord les deux s'appuient sur la même version de GCC, ensuite les niveaux de « solidité » des deux codes sont comparables (le C++ n'effectue aucune vérification en cas de sortie de tableau).

On constate un avantage pour C++ d'un peu moins d'une seconde, soit un gain d'environ 33% par rapport à Ada. Cela peut paraître important, mais cette valeur est à relativiser :

- Même avec `-gnatp`, Ada (compilé par GNAT) effectue toujours quelques vérifications d'intégrité durant l'exécution.
- L'algorithme de tri est un pur QuickSort implémenté par mes soins : il est donc loin d'être optimal, alors que celui fourni avec la librairie C++ standard qui accompagne g++ est fortement optimisé.

L'un dans l'autre et toutes choses étant égales par ailleurs, on peut estimer que dans ce petit comparatif Ada s'en sort fort bien.

La réputation de lenteur d'un programme Ada est donc sans doute à reconsidérer (du moins dans ce contexte très limité) et en tout cas à mettre en balance avec la robustesse qu'apporte le langage.

Si on considère une optimisation « normale », avec `-02` mais sans `-gnatp`, l'écart de performance est plus important mais il faut garder à l'esprit qu'un indice qui déborde du tableau provoquera probablement un plantage du programme en C++, tandis que l'erreur pourra être interceptée par une exception et dûment traitée en Ada.

Natures des paramètres génériques

Les différents exemples précédents ont montré que les paramètres d'un paquetage ou d'un sous-programme générique pouvaient être de différentes natures.

Il est en effet possible de préciser assez finement ce que l'on attend d'un paramètre générique. Différentes notations, dont voici quelques-unes, sont disponibles :

Quelques modèles de paramètres génériques	
Notation	**Description**
`Nom: Type`	Le paramètre générique est en fait une valeur d'un type quelconque, tout comme on indiquerait un paramètre à un sous-programme. Au sein du corps du paquetage ou sous-programme générique, `Nom` joue le rôle d'une constante de type `Type`.
`type T is private` `type T is limited private`	La première forme que nous avons vue. Elle indique simplement que `T` désigne un type, sur lequel on ne dispose de rien de plus que l'affectation standard et l'égalité standard. Si on ajoute le mot-clef `limited`, alors même l'affectation et l'égalité ne font plus partie des opérations disponibles pour le type `T`. Le type réel passé en paramètre doit être un type défini.
`type T(<>) is private`	Le type `T` est cette fois indéfini, comme un type tableau non contraint (par exemple, le type chaîne de caractères `String`) ou un type enregistrement possédant un discriminant.
`type T is (<>)`	Le type `T` est un type discret, c'est-à-dire soit un type énuméré, soit un type entier. Mais même si le type réel passé en paramètre est un type entier, les opérations arithmétiques usuelles ne sont pas disponibles dans le corps du paquetage ou du sous-programme générique.
`type T is range <>`	Le type `T` est un type numérique entier signé. Les opérations arithmétiques usuelles sont disponibles.
`type T is digits <>`	Le type `T` est un type numérique en virgule flottante. Les opérations arithmétiques usuelles sont disponibles.
`with function F ... return ...` `with procedure P ...`	Le paramètre générique n'est cette fois pas un type, mais plutôt un sous-programme. Celui-ci doit être déclaré normalement, avec ses paramètres et leurs types, et sa valeur de retour dans le cas d'une fonction.
`with package P is new Q(...)`	Peut-être la forme la plus difficile à appréhender de paramètre générique : ce n'est ni une valeur, ni un type, ni un sous-programme, mais un paquetage complet. L'écriture indique que le paramètre qui sera connu sous le nom de `P` doit être un paquetage construit à partir d'un paquetage `Q`, lui-même étant un paquetage générique.
Ce tableau n'est qu'un résumé, il existe d'autres formes et subtilités. Mais cela devrait déjà vous permettre de faire pas mal de choses.	

Conclusion

Voilà pour ce survol de la généricité en Ada. Si vous connaissez le C++, vous pouvez constater que la généricité ainsi mise en œuvre est à la fois plus robuste (car plus restrictive) et plus souple que le mécanisme des templates. Tout est fait pour éviter les erreurs et les incohérences dans la programmation. Il est remarquable qu'Ada possède une telle fonctionnalité si avancée dès sa conception.

La prochaine fois, nous reviendrons plus en détail sur les chaînes de caractères et les possibilités d'entrées/sorties offertes par Ada.

Yves Bailly,

Le langage Ada 95 – 8 : Chaînes et fichiers

Yves Bailly

EN DEUX MOTS **Comme tout langage de haut niveau, Ada propose un ensemble relativement complet pour la manipulation des chaînes de caractères et la lecture ou l'écriture de fichiers. Nous allons voir quelques-unes des possibilités, une description exhaustive remplirait le magazine entier.**

Types chaînes de base

Les chaînes de caractères de base d'Ada sont représentées par le type prédéfini `String`, qui est foncièrement un tableau non contraint de caractères du type prédéfini `Character`, correspondant au jeu de caractères Latin-1.

Ces deux types sont grossièrement les équivalents des types `std::string` et `char` du C++. Ada95 a introduit les types `Wide_Character` et `Wide_String` pour représenter les caractères et chaînes Unicode sur 16bits (norme ISO/IEC 10646:2003), les équivalents des types `wchar_t` et `std::wstring` du C++. Enfin, le prochain standard Ada (2006) apportera les types `Wide_Wide_Charater` et `Wide_Wide_String` pour les caractères et chaînes Unicode sur 32 bits.

Chacun des types chaînes sont des tableaux non contraints, déclarés ainsi :

```
-- dès les origines de Ada
subtype Positive is Integer range 1 .. Integer'Last;
type String is array(Positive range <>) of Character;
-- depuis Ada95
type Wide_String is array(Positive range <>) of Wide_Character;
-- À partir de Ada2006
type Wide_Wide_String is array(Positive range <>) of Wide_Wide_Character;
```

Le sous-type `Positive`, qui correspond aux entiers strictement positifs, est très communément utilisé pour les indices de tableaux. En particulier, on voit ici que les caractères des chaînes sont indexés à partir de 1, et non de 0 comme c'est usuellement le cas dans la plupart des langages plus « courants ».

Le paquetage standard `Ada.Characters.Handling` contient tout un ensemble de sous-programmes pour effectuer des conversions entre les types de caractères et de chaînes de base, ainsi que pour obtenir diverses informations (notamment la classification d'un caractère).

En Ada 2006, ce paquetage sera « allégé » et les sous-programmes de conversion se retrouveront améliorés dans `Ada.Characters.Conversions`.

Les fonctions de classification dans `Ada.Characters.Handling` sont les suivantes, chacune prenant en unique paramètre un `Character` et retournant un booléen (type `Boolean`) :

Classification des caractères	
Fonction	**Description**
`Is_Control`	Teste si le caractère est un caractère de contrôle, c'est-à-dire dont le code est compris entre 0 et 32 ou 127 et 159.
`Is_Graphic`	Teste si le caractère est graphique (affichable) ; ceux dont le code est compris entre 32 et 126, et 160 et 255.
`Is_Letter`	Teste si le caractère est une lettre : de `'A'` à `'Z'`, de `'a'` à `'z'` ou de code 192 à 214, 216 à 246 ou 248 à 255.
`Is_Lower`	Teste s'il s'agit d'une lettre minuscule : de `'a'` à `'z'` ou de code 223 à 246 ou 248 à 255.
`Is_Upper`	Teste s'il s'agit d'une lettre majuscule : de `'A'` à `'Z'` ou de code 192 à 214 ou 216 à 222.
`Is_Basic`	Teste s'il s'agit d'une lettre de base, c'est-à-dire en gros une lettre sans accent, plus les lettres `'Æ'`, `'æ'`, `'Ð'`, `'ð'`, `'Þ'`, `'þ'`, et `'ß'`.
`Is_Digit` `Is_Decimal_Digit`	Teste s'il s'agit d'un chiffre décimal (les deux fonctions sont synonymes), de `'0'` à `'9'`.
`Is_Hexadecimal_Digit`	Teste s'il s'agit d'un chiffre hexadécimal, c'est-à-dire soit d'un décimal soit une lettre entre `'a'` et `'f'` ou `'A'` et `'F'`.
`Is_Alphanumeric`	Teste s'il s'agit soit d'une lettre, soit d'un chiffre décimal.
`Is_Special`	Teste si le caractère est un caractère graphique non alphanumérique.

Fichiers binaires

Pour ce qui est des fichiers binaires, Ada distingue deux grandes familles : les fichiers à accès séquentiel et les fichiers à accès direct. Ces deux familles ont comme point commun d'être homogènes : les fichiers sont des suites d'éléments d'un même

type. Pas question, par exemple, de manipuler par l'une de ces deux familles un fichier contenant à la fois des entiers et des réels désordonnés (les flux décrits plus loin apporteront la souplesse voulue). Toutefois, le type de base pourra être un type composé, comme un enregistrement (`record`).

La différence réside dans la possibilité (ou non) de se déplacer dans le fichier. Pour un fichier à accès séquentiel, les éléments sont lus (ou écrits) forcément l'un après l'autre, dans l'ordre, du début à la fin. Il est impossible de « reculer » dans l'accès au fichier. Pour accéder à un élément se trouvant avant la position courante, la seule solution est de parcourir de nouveau le fichier depuis le début. Après chaque opération de lecture ou d'écriture, le pointeur de fichier (la position courante) est avancé sur l'élément suivant. On ne peut pas non plus « sauter » des éléments, tous doivent être lus. Le fichier est ainsi vu comme une séquence linéaire.

Un fichier à accès direct se comporte de la même façon, mais il est en plus possible de se positionner librement dans le fichier. Celui-ci est donc davantage vu comme un ensemble d'éléments occupant des positions consécutives.

Ces deux familles sont décrites par deux paquetages génériques : `Ada.Sequential_IO` et `Ada.Direct_IO`. Le paramètre générique est le type des éléments contenus dans le fichier.

À titre d'exemple, voici un petit programme qui remplit un fichier avec quelques entiers premiers, en utilisant la méthode du crible d'Ératosthène (inefficace, mais ce n'est pas la question), puis relit le fichier pour afficher son contenu :

```
 1 with Text_IO ;
 2 use  Text_IO ;
 3 with Ada.Sequential_IO ;
 4 procedure Crible_1 is
 5    -- instanciation de Ada.Sequential_IO
 6    package Int_Seq_IO is
 7       new Ada.Sequential_IO(Element_Type=>Integer) ;
 8    -- type tableau d'entiers
 9    type Int_Array is array(1..50) of Integer ;
10 -- variables
11 entiers       : Int_Array ;
12 fichier_sortie: Int_Seq_IO.File_Type ;
13 fichier_entree: Int_Seq_IO.File_Type ;
14 premier       : Integer ;
15 sub_index     : Integer ;
16 --
17 begin
18
19    Int_Seq_IO.Create(File => fichier_sortie,
20                      Mode => Int_Seq_IO.Out_File,
21                      Name => "./crible_1.bin") ;
22    -- préparation du tableau
23    for i in entiers'Range
24    loop
25       entiers(i) := i+1 ;
26    end loop ;
27    -- crible et écriture
28    for index in entiers'Range
29    loop
30       if entiers(index) /= 0
31       then
32          premier := entiers(index) ;
33          Int_Seq_IO.Write(fichier_sortie, premier) ;
34          sub_index := index ;
35          while sub_index <= entiers'Last
36          loop
37             entiers(sub_index) := 0 ;
38             sub_index := sub_index + premier ;
39          end loop ;
40       end if ;
41    end loop ;
42    Int_Seq_IO.Close(fichier_sortie) ;
43    -- relecture
44    Int_Seq_IO.Open(File => fichier_entree,
45                    Mode => Int_Seq_IO.In_File,
46                    Name => "./crible_1.bin") ;
47    while not Int_Seq_IO.End_Of_File(fichier_entree)
48    loop
49       Int_Seq_IO.Read(fichier_entree, premier) ;
50       Put(Integer'Image(premier)) ;
51    end loop ;
52    Int_Seq_IO.Close(fichier_entree) ;
53    New_Line ;
54 end Crible_1 ;
```

Rappelez-vous, le paquetage `Ada.Sequential_IO` est générique : il est donc nécessaire de l'instancier pour l'utiliser, ce qui est fait lignes 6 et 7. Il aurait été parfaitement légitime de faire suivre cette instanciation d'une instruction `use Int_Seq_IO`, pour éviter d'avoir à préfixer systématiquement les types et sous-programmes utilisés.

Un fichier est représenté par une instance du type `File_Type`, déclarée dans le paquetage. On peut l'assimiler au type `FILE` du C.

Le fichier est créé lignes 19 à 21. Le paramètre `Mode` donne le type d'accès au fichier : `In_File` pour une lecture seule, `Out_File` pour une écriture seule avec remise à zéro du fichier s'il existe déjà, `Append_File` pour une écriture seule en ajout à un fichier éventuellement déjà existant. Possibilité intéressante, le paramètre `Name` peut accepter une chaîne vide (`""`) : dans ce cas un fichier temporaire est créé, qui sera détruit à la fin du programme.

On trouve ensuite des opérations classiques : l'écriture avec `Write` (ligne 33), la lecture avec `Read` (ligne 49), la fermeture avec `Close` (lignes 42 et 52), le test de fin de fichier avec la fonction `End_Of_File` (ligne 47). À noter que dans cet exemple, il aurait été possible de « réutiliser » la même variable de type fichier `File_Type` pour l'écriture et la lecture, les deux opérations étant bien séparées.

Compilez et exécutez ce programme :

```
$ gnatmake crible_1.adb && ./crible_1
gnatgcc -c crible_1.adb
gnatbind -x crible_1.ali
gnatlink crible_1.ali
 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47
```

Reprenons ce même exemple, mais cette fois les entiers de départ seront inscrits dans un fichier à accès direct plutôt que dans un tableau mémoire :

```
 1 with Text_IO ;
 2 use  Text_IO ;
 3 with Ada.Sequential_IO ;
 4 with Ada.Direct_IO ;
 5 procedure Crible_2 is
 6    -- infos sur un nombre premier
 7    type Infos_Premier is
 8    record
 9       premier: Integer := 0 ;
10       rang   : Integer := 0 ;
11       rapport: Float   := 0.0 ;
12    end record ;
13    -- instanciations
14    package Premiers_Seq_IO is
15       new Ada.Sequential_IO(Element_Type => Infos_Premier) ;
16    use Premiers_Seq_IO ;
17    package Int_Dir_IO is
18       new Ada.Direct_IO(Element_Type => Integer) ;
19    use Int_Dir_Io ;
20 -- variables
21 fichier_entiers : Int_Dir_IO.File_Type ;
22 fichier_premiers: Premiers_Seq_IO.File_Type ;
23 entier          : Integer ;
24 premier        : Infos_Premier ;
25 sub_index      : Integer ;
26 --
27 begin
28    -- préparation
29    Create(File => fichier_entiers,
30           Mode => InOut_File,
31           Name => "") ;
32    Put_Line("Fichier temporaire : " & Name(fichier_entiers)) ;
33    for entier in 1..50
34    loop
35       Write(fichier_entiers, entier+1) ;
36    end loop ;
37    Reset(fichier_entiers) ;
38
39    Create(File => fichier_premiers,
40           Mode => Out_File,
41           Name => "./crible_2.bin") ;
42    -- crible et écriture
43    for index in 1..50
44    loop
45       Set_Index(fichier_entiers, Int_Dir_IO.Count(index)) ;
46       Read(fichier_entiers, entier) ;
47       if entier /= 0
48       then
49          premier.premier := entier ;
50          premier.rang := premier.rang + 1 ;
51          premier.rapport := Float(entier) / Float(premier.rang) ;
52          Write(fichier_premiers, premier) ;
53          sub_index := index ;
54          while sub_index <= 50
55          loop
56             Set_Index(fichier_entiers, Int_Dir_IO.Count(sub_index)) ;
57             Write(fichier_entiers, 0) ;
58             sub_index := sub_index + premier.premier ;
59          end loop ;
60       end if ;
61    end loop ;
62    Close(fichier_premiers) ;
63    Close(fichier_entiers) ;
64    -- relecture
65    Open(File => fichier_premiers,
66         Mode => In_File,
67         Name => "./crible_2.bin") ;
68    while not End_Of_File(fichier_premiers)
69    loop
70       Read(fichier_premiers, premier) ;
71       Put(" (" & Integer'Image(premier.premier) & ", " &
72           Float'Image(premier.rapport) & ")") ;
73    end loop ;
74    Close(fichier_premiers) ;
75    New_Line ;
76 end Crible_2 ;
```

Cette fois, on utilise les clauses `use` pour tenir compte de notre fatigue (lignes 16 et 19). Notez toutefois qu'il est par moment nécessaire d'utiliser une notation « complète », pour lever d'éventuelles ambiguïtés. Par exemple, chacun des paquetages `Ada.Sequential_IO` et `Ada.Direct_IO` définissant un type nommé `File_Type`, le préfixe est nécessaire pour déterminer le type auquel on fait référence (lignes 21 et 22). Par ailleurs, on stocke plus d'informations sur un nombre premier : son rang et le rapport entre sa valeur et son rang. Nous allons donc manipuler un fichier (séquentiel) de structures, non plus de simples entiers.

La différence essentielle réside dans l'utilisation d'un fichier à accès direct, ce qui nous permet d'utiliser la procédure `Set_Index()` (lignes 45 et 56) pour nous positionner sur n'importe quel élément du fichier. Celle-ci prend en premier paramètre l'objet fichier, en deuxième la position voulue, les éléments du fichier étant numérotés à partir de 1. Remarquez également que le fichier initial contenant les entiers est créé sans lui donner de nom (ligne 31) : cela va provoquer la création d'un fichier temporaire, donc le nom est affiché pour information.

La compilation et l'exécution de cette nouvelle version donne ceci :

```
$ gnatmake crible_2.adb && ./crible_2
gnatgcc -c crible_2.adb
gnatbind -x crible_2.ali
gnatlink crible_2.ali
Fichier temporaire : /tmp/gnat-lhYWnh
 ( 2,  2.00000E+00) ( 3,  1.50000E+00) ( 5,  1.66667E+00) ( 7,
1.75000E+00)
 ( 11,  2.20000E+00) ( 13,  2.16667E+00) ( 17,  2.42857E+00) ( 19,
2.37500E+00)
 ( 23,  2.55556E+00) ( 29,  2.90000E+00) ( 31,  2.81818E+00) ( 37,
3.08333E+00)
 ( 41,  3.15385E+00) ( 43,  3.07143E+00) ( 47,  3.13333E+00)
```

Vous pourrez vérifier que le fichier temporaire a bien été effacé à l'issu de l'exécution.

Fichiers textes

L'ensemble des procédures et fonctions permettant de manipuler des fichiers textes ou plus généralement les entrées/sorties de textes, se trouvent dans le paquetage `Ada.Text_IO` (le plus souvent abrégé en simplement `Text_IO` pour des raisons historiques) et ses sous-paquetages, certains étant imbriqués.

Ceci vaut pour les chaînes de caractères « classiques » constituées d'octets. Pour les chaînes Unicode à caractères sur 16 bits, c'est le paquetage `Ada.Wide_Text_IO` qu'il faut utiliser. Enfin, pour l'Unicode 32 bits, le prochain standard Ada 2006 prévoit le paquetage `Ada.Wide_Wide_Text_IO`. Ce qui suit se concentre sur `Text_IO`, mais ce qui est dit est valable pour les chaînes Unicode.

Le paquetage `Text_IO` définit un type `File_Type` pour représenter un fichier, tout comme les deux paquetages génériques évoqués plus haut. Il fournit également les mêmes sous-programmes pour créer, ouvrir ou fermer un fichier. On ne retrouve toutefois pas de procédures `Read()` ou `Write()`, mais des remplaçants adaptés au cas particulier des fichiers textes.

Entrées/sorties standards

Nous avons déjà fréquemment utilisé les procédures `Put()` et `Put_Line()` pour afficher du texte à l'écran. Elles agissent sur la sortie standard, l'équivalent du `stdout` en C. L'instance de `File_Type` correspondante peut être obtenue à l'aide de la fonction `Standard_Output()`. Il est possible de la modifier à l'aide de la procédure `Set_Output()` ; la fonction `Current_Output()` retourne le fichier actuellement utilisé pour la sortie standard.

Remplacez `Output` par `Input` dans les noms précédents et vous obtenez les informations concernant l'entrée standard (l'équivalent du `stdin` du C). Remplacez avec `Error` et on parle alors de la sortie d'erreur standard (le `stderr`).

Les `Put()` et `Put_Line()` que nous avons utilisées ne prenaient qu'un seul argument nommé `Item`, la chaîne à afficher. Elles existent également avec un argument supplémentaire nommé `File` (en première position) destiné à recevoir une instance de `File_Type`. Pour les entrées, on dispose des procédures `Get()` et `Get_Line()`, qui fonctionnent de manière similaire sauf que le paramètre `Item` est une chaîne passée en mode `out`.

Il existe également toute une panoplie de procédures pour n'extraire qu'un seul caractère, éventuellement sans le consommer. On trouve également de nombreuses procédures pour spécifier les hauteurs et largeurs de pages, positionner le point d'insertion à une ligne ou une colonne donnée... Cela peut paraître un peu étrange, mais n'oublions pas qu'Ada est né à une époque où les terminaux graphiques n'étaient pas vraiment légions et où les périphériques de sortie de choix étaient encore des imprimantes où tous les caractères occupaient la même largeur.

Pour les nombres entiers

Les entrées/sorties de nombres entiers sous forme de textes sont définies dans le paquetage générique et imbriqué `Ada.Text_IO.Integer_IO`. Le paramètre générique est le type entier particulier que l'on souhaite manipuler : il est donc nécessaire d'instancier ce paquetage avant de pouvoir utiliser ses services, par exemple :

```
1 with Ada.Text_IO ;
2 -- ...
3 type Mon_Entier is new Integer range 4..11 ;
4 package Mon_Entier_IO is
5    new Ada.Text_IO.Integer_IO(Num => Mon_Entier) ;
6 -- ...
7 Mon_Entier_IO.Put_Line(9) ;
```

De cette manière, un contrôle strict est effectué lorsque l'on tente de lire ou d'écrire un nombre. Cela peut paraître tiré par les cheveux, voire franchement pénible, mais cela participe de la robustesse offerte par Ada. Ceci dit, il existe un paquetage non générique nommé `Ada.Integer_Text_IO` qui est l'équivalent d'une instanciation de `Ada.Text_IO.Integer_IO` pour le type prédéfini `Integer`. Des paquetages non génériques existent également pour les types `Long_Integer` et `Long_Long_Integer`, selon les implémentations.

Un certain contrôle peut être effectué sur la mise en forme des nombres affichés. Deux variables dans `Ada.Text_IO.Integer_IO` sont disponibles :

- `Default_Width`, de type `Field` (un sous-type de `Integer`), donne la largeur de base pour les nombres ;
- `Default_Base`, de type `Number_Base` (encore un sous-type de `Integer`), est la base de numération à utiliser, entre 2 et 16 inclus.

Modifier ces valeurs a une incidence sur toutes les opérations suivantes d'affichage. Il est généralement préférable de passer ces mises en forme aux procédures `Put()` adaptées. Voici par exemple un petit programme qui attend un nombre saisi au clavier, puis affiche ses représentations binaires, octales et hexadécimales :

```
 1 with Ada.Integer_Text_IO ;
 2 use  Ada.Integer_Text_IO ;
 3 with Ada.Text_IO ;
 4 use  Ada.Text_IO ;
 5 procedure Num_Convert is
 6    nb: Integer ;
 7 begin
 8    loop
 9       Put("Un nombre : ") ;
10       Get(nb) ;
11       exit when nb = 0 ;
12       Put("   Binaire :") ;
13       Put(Item => nb, Width => 20, Base => 2) ;
14       New_Line ;
15       Put("   Octal :") ;
16       Put(Item => nb, Width => 22, Base => 8) ;
17       New_Line ;
18       Put("   Hexadécimal :") ;
19       Put(Item => nb, Width => 16, Base => 16) ;
20       New_Line ;
21    end loop ;
22 end Num_Convert ;
```

Comme il est tard, la version non générique des entrées/sorties d'entiers a été utilisée. Le `Get()` en ligne 10 vient de `Ada.Integer_Text_IO`, ainsi que les `Put()` des lignes 13, 16 et 19 (il n'existe malheureusement pas de `Put_Line()` pour les nombres).

Un exemple d'exécution :

```
Un nombre : 2005
   Binaire :      2#11111010101#
   Octal :              8#3725#
   Hexadécimal :        16#7D5#
Un nombre : 1991
   Binaire :      2#11111000111#
   Octal :              8#3707#
   Hexadécimal :        16#7C7#
Un nombre : 0
```

Difficile de faire plus simple. Remarquez tout de même comment Ada indique la base de numération : elle préfixe le nombre dont la représentation est encadrée de dièses. C'est la même syntaxe que celle utilisée dans un code source.

Le cas des nombres réels

Naturellement, les nombres décimaux ne sont pas oubliés. Pour eux existe le paquetage générique et imbriqué `Ada.Text_IO.Float_IO`, le paramètre générique étant naturellement le type réel concerné. Comme pour les entiers, il existe une version non générique `Ada.Float_Text_IO` pour le type de base `Float`.

Les possibilités de mise en forme sont un peu plus poussées, un nombre réel étant plus compliqué qu'un entier. Il est composé d'au moins deux parties, de part et d'autre du point décimal, auxquelles s'ajoute éventuellement un exposant. On trouve donc trois variables dans le paquetage de type `Field` pour paramétrer la globalité des sorties :

- `Default_Fore`, qui vaut 2 par défaut, est le nombre de caractères (avec le signe) à gauche du point décimal ;
- `Default_Aft` est le nombre de caractères à droite du point ;
- `Default_Exp` est le nombre de caractères pour l'exposant (si besoin est avec le signe), 3 par défaut.

Là encore, les procédures `Put()` acceptent des paramètres évitant de modifier globalement les sorties. Par exemple, considérons le programme :

```
 1 with Ada.Text_IO ;
 2 use  Ada.Text_IO ;
 3 procedure Sortie_Float is
 4    package Long_Float_IO is
 5       new Ada.Text_IO.Float_IO(Long_Float) ;
 6    use Long_Float_IO ;
 7    f1: Long_Float := 1995.2005 ;
 8    f2: Long_Float := f1**12 ;
 9 begin
10    Put("f1 = ") ;
11    Put(Item => f1) ; New_Line ;
12    Put("f2 = ") ;
13    Put(Item => f2) ; New_Line ;
14    Put("f1 = ") ;
15    Put(Item => f1, Fore => 5, Aft => 2, Exp => 0) ; New_Line ;
16    Put("f2 = ") ;
17    Put(Item => f2, Fore => 5, Aft => 2, Exp => 0) ; New_Line ;
18    Put("f1 = ") ;
19    Put(Item => f1, Fore => 3, Aft => 4, Exp => 4) ; New_Line ;
20    Put("f2 = ") ;
21    Put(Item => f2, Fore => 3, Aft => 4, Exp => 4) ; New_Line ;
22 end Sortie_Float ;
```

Cette fois, une instanciation a été utilisée (en fait parce qu'il n'existe pas de paquetage `Ada.Long_Float_IO`). Pour mémoire, l'opérateur `**` utilisé ligne 8 est l'élévation à la puissance. `f2` vaut donc $f1^{12}$.

L'affichage :

```
f1 =  1.99520050000000E+03
f2 =  3.97959191276602E+39
f1 =  1995.20
f2 = 3979591912766016980000000000000000000000.00
f1 =   1.9952E+003
f2 =   3.9796E+039
```

Les deux premières lignes montrent l'affichage par défaut des deux valeurs. Remarquez l'effet obtenu en forçant le nombre de caractères de l'exposant à zéro (lignes 15 et 17). De même, le fait de limiter le nombre de décimales provoque un arrondi, visible sur le dernier chiffre du troisième affichage de `f2` : un 5 suivi d'un 9 est devenu un 6.

Flux

Les flux en Ada permettent une approche plus souple de l'accès aux fichiers, en cela qu'ils combinent les aspects séquentiels et accès directs (lorsque la « source » ou la « destination » du flux autorise le positionnement libre) et permettent la lecture ou l'écriture d'éléments de types différents. Ils sont ainsi typiquement utilisés pour les transmissions par réseau. Les flux généraux sont décrits dans le paquetage `Streams.Stream_IO`, tandis que les flux de texte se trouvent dans les paquetages `Text_IO.Text_Streams` et `Wide_Text_IO.Text_Streams`. Nous allons examiner ici les flux généraux.

On trouve dans le paquetage `Streams.Stream_IO` des éléments comparables à ceux que nous avons déjà utilisés : un type pour représenter un fichier (`File_Type`), des procédures de création, d'ouverture et de fermeture (`Create`, `Open` et `Close`), des procédures de lecture et d'écriture (`Read` et `Write`), etc. Toutefois ces sous-programmes opèrent sur le type `Stream_Element_Array`, lequel est un tableau non contraint indexé à partir de 0 d'éléments de type `Stream_Element`. La nature exacte de ce type dépend de l'implémentation, c'est-à-dire en gros de la plate-forme. En général, il correspond à un simple octet (caractère sur 8 bits), mais ce n'est pas forcément le cas. Le fichier est alors vu comme une suite de `Stream_Elements`.

Ce qui nous intéresse ici est le type `Stream_Access` et la fonction `Stream()` qui permet d'en obtenir une valeur à partir d'une instance de `File_Type` (donc à partir d'un fichier ouvert). Ce type et cette fonction vont nous permettre d'utiliser le fichier comme un flux de données hétérogènes.

Assez curieusement, les flux sont mis en œuvre au moyen d'attributs sur les types : les attributs 'Read (lecture) et 'Write (écriture), d'une part, les attributs 'Input (entrée) et 'Output (sortie) d'autre part. Voyons la première paire, en reprenant une fois de plus notre programme de nombres premiers :

```
 1 with Text_IO ;
 2 use  Text_IO ;
 3 with Ada.Streams.Stream_IO ;
 4 use  Ada.Streams.Stream_IO ;
 5 procedure Crible_Flux_1 is
 6    type Int_Array is array(1..50) of Integer ;
 7 -- variables
 8 entiers         : Int_Array ;
 9 fichier_premiers: Ada.Streams.Stream_IO.File_Type ;
10 flux_premiers   : Stream_Access ;
11 premier         : Integer ;
12 sub_index       : Integer ;
13 nb_premiers     : Integer := 0 ;
14 rapport         : Float ;
15 --
16 begin
17    -- préparation
18    for i in entiers'Range
19    loop
20       entiers(i) := i+1 ;
21    end loop ;
22    -- crible et écriture
23    Create(File => fichier_premiers,
24           Mode => Out_File,
25           Name => "./crible_flux_1.bin") ;
26    flux_premiers := Stream(fichier_premiers) ;
27    for index in 1..50
28    loop
29       if entiers(index) /= 0
30       then
31          premier := entiers(index) ;
32          nb_premiers := nb_premiers + 1 ;
33          rapport := Float(premier)/Float(nb_premiers) ;
34          Integer'Write(flux_premiers, premier) ;
35          Float'Write(flux_premiers, rapport) ;
36          sub_index := index ;
37          while sub_index <= entiers'Last
38          loop
39             entiers(sub_index) := 0 ;
40             sub_index := sub_index + premier ;
41          end loop ;
42       end if ;
43    end loop ;
44    -- relecture
45    Reset(fichier_premiers, In_File) ;
46    while not End_Of_File(fichier_premiers)
47    loop
48       Integer'Read(flux_premiers, premier) ;
49       Float'Read(flux_premiers, rapport) ;
50       Put(" (" & Integer'Image(premier) & ", " &
51           Float'Image(rapport) & ")") ;
52    end loop ;
53    Close(fichier_premiers) ;
54    New_Line ;
55 end Crible_Flux_1 ;
```

Cette version est proche de celle exposée pour les fichiers à accès séquentiel. Toutefois, dans le fichier, chaque nombre premier est suivi du rapport entre sa valeur et son rang, comme dans l'exemple des fichiers à accès direct.

Ces deux informations ne sont pas rassemblées en une structure unique : on a donc bien des données hétérogènes inscrites dans le fichier.

Le fichier est créé « normalement », lignes 23 à 25. Mais cette fois, on récupère une référence sur le flux associé, ligne 26. Cette référence va être utilisée pour l'écriture des données à l'aide de l'attribut 'Write des types Integer et Float, lignes 34 et 35. Ces deux lignes ont pour effet d'inscrire dans le fichier successivement un entier suivi d'un nombre à virgule flottante.

Plutôt que de fermer le fichier avant de le relire, ce programme utilise la procédure Reset() (ligne 45) : elle permet de repositionner le pointeur de fichier au début de celui-ci, et éventuellement de changer de mode d'accès – comme ici, où l'on passe d'un fichier ouvert en écriture (Out_File) à un fichier ouvert en lecture (In_File). Les données du fichier sont ensuite lues à l'aide de l'attribut 'Read des types Integer et Float, de façon parfaitement symétrique à l'écriture (lignes 48 et 49).

Mais on peut faire encore mieux. Les attributs 'Read et 'Write n'écrivent dans le fichier que la valeur de la donnée. Dans certains cas, ce peut être insuffisant, par exemple si on souhaite écrire le contenu d'un tableau dans le fichier. Si le type de base est un tableau non contraint, il est probablement nécessaire d'inscrire également les valeurs des bornes. Ce qui n'est pas forcément pratique.

Modifions quelques lignes du programme précédent pour inscrire les valeurs sous forme de chaînes de caractères. Rappelez-vous que les chaînes sont des tableaux non contraints. Les lignes 34 et 35 deviennent alors :

```
34    String'Output(flux_premiers, Integer'Image(premier)) ;
35    String'Output(flux_premiers, Float'Image(rapport)) ;
```

On utilise cette fois l'attribut 'Output, appliqué au type String. Cet attribut inscrit, en plus de la valeur de la donnée, les informations « administratives » que sont les bornes du tableau. Si on avait utilisé 'Write, seule la suite d'octets de la chaîne aurait été écrite. Remarquez qu'on ne s'occupe absolument pas de la longueur des chaînes, d'ailleurs dans notre exemple certaines sont plus courtes que d'autres.

La lecture du fichier doit également être modifiée :

```
46    while not End_Of_File(fichier_premiers)
47    loop
48       declare
```

```
49          chaine_premier: String :=
50             String'Input(flux_premiers) ;
51          chaine_rapport: String :=
52             String'Input(flux_premiers) ;
53       begin
54          Put(" (" & chaine_premier & ", " &
55              chaine_rapport & ")") ;
56       end ;
57    end loop ;
```

L'attribut miroir de `'Output` est `'Input`, sauf que celui-ci se comporte comme une fonction, contrairement à `'Read` qui s'apparente plus à une procédure. Nous savons que nous allons lire des chaînes, mais nous ne connaissons pas à l'avance leur longueur : il est donc impossible de déclarer à l'avance une variable « vide » de type `String`. La seule façon de déclarer une variable d'un type tableau non contraint sans donner de contrainte est de fournir une valeur d'initialisation : celle-ci est obtenue de l'attribut `'Input`.

Notez bien que tout ce qui est dit jusqu'ici est valable pour des types composés, comme des tableaux à plusieurs dimensions ou des enregistrements possédant éventuellement des discriminants.

Enfin et pour terminer, signalons qu'Ada nous permet de personnaliser la lecture et l'écriture d'un type de données dans un flux. Ceci peut s'appliquer à n'importe quel type que nous aurions défini. Par exemple, reprenons notre premier programme de flux mais, plutôt que d'écrire la représentation binaire des valeurs réelles, nous voulons écrire une représentation sous forme de chaîne. On pourrait utiliser la technique du programme précédent, mais pour une raison qui nous échappe les attributs `'Read` et `'Write` ont notre préférence.

Pour cela, on va commencer par définir un nouveau type réel, nommé `Mon_Float`, puis nous allons surdéfinir les attributs `'Read` et `'Write` de ce type :

```
 1 with Text_IO ;
 2 use  Text_IO ;
 3 with Ada.Streams.Stream_IO ;
 4 use  Ada.Streams ;
 5 use  Ada.Streams.Stream_IO ;
 6 procedure Crible_Flux_3 is
 7    -- nouveau type réel
 8    type Mon_Float is new Float ;
 9    -- procédure d'écriture
10    procedure Float_Write(Stream : access Root_Stream_Type'Class;
11                          Item   : in Mon_Float) ;
12    for Mon_Float'Write use Float_Write ;
13    -- procédure de lecture
14    procedure Float_Read(Stream : access Root_Stream_Type'Class;
15                         Item   : out Mon_Float) ;
16    for Mon_Float'Read use Float_Read ;
17    -- implémentations
18    procedure Float_Write(Stream : access Root_Stream_Type'Class;
19                          Item   : in Mon_Float) is
20    begin
21       String'Output(Stream, Mon_Float'Image(Item)) ;
22    end Float_Write ;
23    procedure Float_Read(Stream : access Root_Stream_Type'Class;
24                         Item   : out Mon_Float) is
25       chaine: String := String'Input(Stream) ;
26    begin
27       Item := Mon_Float'Value(chaine) ;
28    end Float_Read ;
...
```

La suite du programme est inchangée, à ceci près qu'il faut remplacer toutes les occurrences de `Float` par `Mon_Float`. Ce dernier est déclaré ligne 8. La procédure à utiliser pour l'écriture est déclarée lignes 10 et 11 et implémentée lignes 18 à 22. La séparation entre déclaration et implémentation est obligatoire pour utiliser cette technique.

Le nom donné à cette procédure, `Float_Write`, est parfaitement arbitraire : vous pouvez l'appeler `Schtroumpfette` si vous voulez. L'important se trouve ligne 12 : elle indique que pour (`for`) l'attribut `'Write` appliqué au type `Mon_Float`, il faut utiliser (`use`) la procédure `Float_Write`.

Ainsi, lorsque dans la suite du programme nous rencontrerons une instruction comme `Mon_Float'Write(flux, valeur)`, c'est notre procédure qui sera invoquée, et non l'écriture prédéfinie.

On procède de manière similaire pour la lecture.

Pour information, le type `Root_Stream_Type` est déclaré dans le paquetage `Ada.Streams`. Son mode de passage (`access`) a trait à la notion de « pointeurs » en Ada et correspond grosso modo à un passage par référence.

L'attribut `'Class` qui lui est appliqué concerne les fonctionnalités de programmation orientée objet. En fait, ce qui est mis en œuvre ici n'est ni plus ni moins que du polymorphisme. Mais c'est une autre histoire que je vous conterai une prochaine fois...

Conclusion

On pourrait encore dire énormément de choses sur les chaînes de caractères et les fichiers en Ada.

Nous n'avons par exemple pas évoqué des paquetages comme `Ada.Strings.Maps`, qui permet de manipuler des ensembles de caractères et d'effectuer des traductions entre eux, ou comme `Ada.Strings.Unbounded`, qui définit un type de chaînes de caractères pouvant s'étendre ou se réduire au besoin (plus proche du `std::string` du C++ que le type de base `String`). À vous d'explorer !

Le mois prochain, nous aborderons la notion de « pointeurs » et la gestion de la mémoire en Ada. Encore une fois, nous découvrirons une approche originale de ces outils pourtant si familiers.

Yves Bailly,

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

Le langage Ada – 10 : Les exceptions

Yves Bailly

EN DEUX MOTS **Le mécanisme des exceptions, qui permet de réagir à une situation anormale et inattendue, a souvent été évoqué dans les articles précédents. Il est temps de voir les possibilités offertes par Ada pour intercepter et déclencher des exceptions, assurant ainsi un bon déroulement du programme même en cas d'erreur critique.**

Actualités

Un petit mot avant d'aborder notre sujet. Les lecteurs des articles précédents auront sans doute remarqué le changement survenu dans le titre : cette série d'articles n'est plus intitulée « Ada 95 », mais « Le langage Ada ». Nous sommes en effet maintenant en 2006, année durant laquelle devrait normalement être validé le nouveau standard du langage Ada [1]. Celui-ci ayant été conçu pour l'essentiel durant l'année 2005, on parle généralement de la version « Ada 2005 » (ou Ada05) du langage. Les dernières versions du compilateur de référence utilisé, le compilateur Gnat qui fait partie de la suite GCC, progresse régulièrement dans le support de ce nouveau standard. Aussi ai-je décidé de sauter le pas et de concentrer cette présentation du langage Ada sur cette dernière mouture du langage.

À noter également le remarquable développement du WikiBook [2] consacré au langage Ada, sous forme de didacticiels (ou tutoriels) progressifs sur plus de 200 pages. Il s'agit en réalité du premier ensemble de textes introduisant spécifiquement les nouveautés du standard Ada 2005. Enfin, une récente enquête [3] a montré que le marché autour du langage Ada peut être estimé à quelque chose comme 5.6 milliards de dollars, Europe et Amérique du Nord combinées, pour environ 322 millions de ligne de code. Les industries de la défense et de l'aérospatiale sont largement représentées, ce qui est traditionnel, mais le champ d'application du langage est beaucoup plus vaste, de la recherche génétique aux machines à voter en passant par le monde de la finance. Voilà sans doute de quoi modérer le dédain de ceux qui pensent que « personne n'utilise Ada... ». Mais revenons à nos exceptionnels moutons.

Interception

Commençons par générer un petit programme qui va lever une paire d'exceptions. Par exemple, sortir des limites d'un tableau et tenter de lire un fichier inexistant. Le voici :

```
 1 with Text_IO ;
 2 use  Text_IO ;
 3 procedure Interception_1 is
 4    type TTab is array(1..10) of Integer ;
 5    t: TTab ;
 6    f: File_Type ;
 7 begin
 8    t(0) := 1 ;
 9    Open(File => f,
10         Mode => In_File,
11         Name => "n_importe_quoi") ;
12    Put_Line("Fin du programme.") ;
13 end Interception_1 ;
```

Rien de bien méchant. À noter que la compilation du programme est assortie d'un avertissement :

```
$ gnatmake -gnat05 interception_1.adb
gcc -c interception_1.adb
interception_1.adb:8:06: warning: value not in range of subtype of
"Standard.Integer" defined at line 4
interception_1.adb:8:06: warning: "Constraint_Error" will be
raised at run time
gnatbind -x interception_1.ali
gnatlink interception_1.ali
```

Le paramètre `-gnat05` est destiné à activer le support du standard Ada 2005 par le compilateur Gnat, lequel a le bon goût de nous avertir que la ligne 8, bien que syntaxiquement correcte, va lever une exception (avec même l'indication de la position dans la ligne). Ce que l'on peut vérifier à l'exécution :

```
$ ./interception_1
raised CONSTRAINT_ERROR : interception_1.adb:8 range check failed
```

On obtient bien l'exception annoncée, `Contraint_Error`, lors de l'exécution de la ligne 8. Modifions dans un premier temps le programme de façon à intercepter toutes les exceptions. La syntaxe utilisée en Ada rappelle un peu celle du C++ ou de Python, les instructions à « protéger » étant inclues dans un bloc, les traitements d'exceptions étant donnés ensuite comme dans un choix multiple (`case...when`) :

```
 7 begin
 8    begin
 9       t(0) := 1 ;
10       Open(File => f,
11            Mode => In_File,
12            Name => "n_importe_quoi") ;
13    exception
14       when others =>
15          Put_Line("Oups, une erreur !") ;
16    end ;
17    Put_Line("Fin du programme.") ;
18 end Interception_2 ;
```

Les exceptions interceptées seront donc celles levées durant l'exécution des lignes 9 à 12. Les traitements sont introduits

par le mot-clef exception et terminés par la fin (end) du bloc, entre les lignes 13 et 16. La clause when others marque le début d'un traitement dédié à toutes les exceptions, le mot others étant à rapprocher de celui utilisé dans un bloc case. Avec cette modeste modification, notre programme ne plante plus :

```
$ ./interception_2
Oups, une erreur !
Fin du programme.
```

Seulement, c'est un peu grossier comme traitement d'exception. Il serait préférable d'avoir un traitement spécialisé par type d'exception, ainsi que de récupérer le message associé. Modifions à nouveau le programme, en utilisant le paquetage Ada.Exceptions, qui contient quelques outils pratiques liés aux exceptions :

```
 1 with Text_IO ;
 2 use  Text_IO ;
 3 with Ada.Exceptions ;
 4 procedure Interception_3 is
 5    type TTab is array(1..10) of Integer ;
 6    t: TTab ;
 7    f: File_Type ;
 8 begin
 9    begin
10       t(0) := 1 ;
11       Open(File => f,
12            Mode => In_File,
13            Name => "n_importe_quoi") ;
14    exception
15       when Ex: Constraint_Error =>
16          Put_Line("Erreur de contrainte : " &
17             Ada.Exceptions.Exception_Message(Ex)) ;
18       when Ex: others =>
19          Put_Line("Oups, une erreur : " &
20             Ada.Exceptions.Exception_Name(Ex) & ": " &
21             Ada.Exceptions.Exception_Message(Ex)) ;
22    end ;
23    Put_Line("Fin du programme.") ;
24 end Interception_3 ;
```

Le traitement spécialisé est introduit ligne 15. Le petit morceau « Ex: », facultatif, permet de donner un nom à l'exception reçue et donc d'en faire éventuellement quelque chose. Ce nom peut être ce que bon vous semble, en particulier quelque chose de plus explicite que Ex ; de plus vous pouvez choisir un nom différent pour chaque traitement, ici Ex apparaît deux fois par pur manque d'imagination. Derrière se trouve le nom de l'exception attendue, ou others pour un traitement général (comme ligne 18). Le message de l'exception est obtenu sous la forme d'une chaîne de caractères retournée par la fonction Exception_Message() du paquetage Ada.Exceptions : on peut ainsi l'afficher, ce qui aide toujours à la résolution du problème. L'exécution donne ceci :

```
$ ./interception_3
Erreur de contrainte : interception_3.adb:10 range check failed
Fin du programme.
```

C'est bien le traitement spécialisé qui a été exécuté. Le traitement général est toujours présent, lignes 18 à 21, affichant cette fois le nom de l'exception (obtenu par la fonction Exception_Name() de Ada.Exceptions) avec son message. Ce traitement général doit toujours apparaître après tous les traitements spécifiques.

Si on corrige l'erreur ligne 10, par exemple en changeant le 0 en 1, l'exécution donne :

```
$ ./interception_4
Oups, une erreur : ADA.IO_EXCEPTIONS.NAME_ERROR: s-fileio.adb:935
Fin du programme.
```

On tombe dans le traitement général, avec un message moins explicite. Cette exception est déclenchée par la tentative d'ouverture d'un fichier inexistant, lignes 11-13. Notez qu'elle n'apparaissait pas auparavant : le bloc d'instructions entre les lignes 9 et 13 est en effet interrompu dès qu'une exception survient. Donc quand l'instruction incorrecte ligne 10 est exécutée (avant qu'elle ne soit corrigée), les instructions suivantes sont tout simplement sautées. Ce qui est un comportement tout à fait normal. Notez le nom de l'exception : cela ressemble à ce que pourrait être le nom d'un objet contenu dans un paquetage IO_Exceptions, lui-même contenu dans le paquetage Ada... Voyons comment sont déclarées et déclenchées les exceptions.

Déclaration et déclenchement

La déclaration d'une exception se fait par une syntaxe horriblement complexe :

```
Nom_Exception: exception ;
```

C'est simple, mais cela ne permet pas de définir une hiérarchie d'exceptions, comme on peut le faire en C++. C'est là un désagrément avec lequel il va falloir vivre. Le déclenchement d'une exception se fait au moyen du mot-clef raise. Par exemple :

```
 1 with Text_IO ;
 2 use  Text_IO ;
 3 with Ada.Exceptions ;
 4 use  Ada.Exceptions ;
 5 procedure Decl_1 is
 6    Pas_Paire: exception ;
 7    procedure Test_Paire(i: in Integer) is
 8    begin
 9       if (i mod 2) /= 0
10       then
11          raise Pas_Paire
12             with Integer'Image(i) & " n'est pas paire" ;
13 --          Raise_Exception(Pas_Paire'Identity,
14 --             Integer'Image(i) & " n'est pas paire") ;
15       end if ;
16       Put_Line(Integer'Image(i) & " est paire") ;
17    end Test_Paire ;
18 begin
19    begin
20       Test_Paire(2) ;
21       Test_Paire(3) ;
22    exception
23       when Ex: Pas_Paire =>
24          Put_Line(Exception_Message(Ex)) ;
25    end ;
26    Put_Line("Fin du programme.") ;
27 end Decl_1 ;
```

Une exception nommée Pas_Paire est déclarée ligne 6. La procédure Test_Paire()

Quelques exceptions prédéfinies — Tableau 1

Paquetage	Exception	Description
Standard	Constraint_Error	Sans doute la plus fréquente, cette exception est levée dès l'utilisation d'une valeur en dehors de l'intervalle de définition de son type. Elle est également levée si vous tentez d'accéder au contenu d'un pointeur nul.
	Storage_Error	Levée si vous tentez de réserver plus de mémoire qu'il n'en est disponible.
	Program_Error	Assez rare, cette exception survient lorsque le programme se trouve dans un état incohérent, par exemple si un sous-programme dans un paquetage est invoqué alors que ce paquetage n'a pas encore été élaboré (c'est-à-dire, initialisé).
Ada.Strings	Index_Error	Liée aux opérations sur les chaînes de caractères, cette exception survient si un indice donné à un sous-programme (comme la recherche d'une sous-chaîne) est invalide.
Ada.Direct_IO Ada.Directories Ada.IO_Exceptions Ada.Sequential_IO Ada.Streams.Stream_IO Ada.Text_IO	Name_Error	Exception déclenchée si vous tentez d'accéder à un fichier inexistant ou inaccessible.
	End_Error	Déclenchée si vous tentez de lire au-delà de la fin d'un fichier.
	Data_Error	Déclenchée si, lors de la lecture d'un fichier, les caractères reçus ne correspondent pas au type attendu.

vérifie si l'entier reçu en paramètre est pair ou non ; s'il ne l'est pas, l'exception Pas_Paire est levée (ligne 11). Celle-ci est interceptée ligne 23. Résultat :

```
$ ./decl_1
 2 est paire
 3 n'est pas paire
Fin du programme.
```

La syntaxe utilisée ici pour l'instruction raise est la nouvelle forme introduite par le standard Ada 2005. Auparavant, une exception était levée simplement par :

```
raise Nom_Exception ;
```

Cette forme est toujours disponible. Il est maintenant possible d'y adjoindre un membre « with "chaîne" » afin d'associer un message à l'exception, qui pourra être récupérée par Exception_Message() (ligne 24). En Ada95, pour obtenir le même effet, il fallait avoir recours à la procédure Raise_Exception() du paquetage Ada.Exceptions : un exemple est donné par les deux lignes commentées 13 et 14.

Types et attribut

Toute exception possède un identifiant, dont le type est Exception_Id déclaré dans Ada.Exceptions. Cet identifiant peut être obtenu à partir du nom de l'exception avec l'attribut 'Identity, comme cela est fait ligne 13 dans l'exemple précédent. Réciproquement, le nom de l'exception peut être retrouvé à partir de l'identifiant à l'aide de Exception_Name(), qui retourne une chaîne de caractères. Wide_Exception_Name() et Wide_Wide_Exception_Name() font de même, mais retournent respectivement une chaîne Unicode 16 et une chaîne Unicode 32. Nous avons vu qu'il était possible de donner un « nom » à une exception interceptée, par exemple le nom Ex ligne 23 du programme précédent. Ce nom est en fait celui d'une variable de type Exception_Occurrence (déclarée dans Ada.Exceptions) : ce n'est pas le nom de l'exception, mais un nom donné à une instance de celle-ci. Divers sous-programmes prennent un tel objet en paramètre, comme Exception_Message() (déjà rencontré), Exception_Identity() (pour obtenir l'identifiant) ou Exception_Name().

Exceptions prédéfinies

Le standard du langage impose l'existence de quelques exceptions, que vous êtes assuré de retrouver sur n'importe quelle implémentation. Celles-ci sont déclarées dans divers paquetages, selon l'emploi qui en est fait. À noter qu'un même nom d'exception peut apparaître dans différents paquetages : il s'agit alors naturellement de différentes exceptions, chacune possédant son propre identifiant. Voici quelques-unes des exceptions les plus courantes (consultez la section Q-4 du standard Ada pour une liste exhaustive Tabl. 1). Pour mémoire, le paquetage Standard n'a pas besoin d'être préfixé : il est toujours supposé visible et utilisé par défaut (comme si une clause use Standard était systématiquement insérée au début de tout programme ou paquetage).

Conclusion

Voilà pour cette petite présentation des exceptions en langage Ada. Pour plus de détails, voyez la section 11 du manuel de référence Ada [4], plus particulièrement la sous-section 11.4.1 consacrée au paquetage Ada.Exceptions. La prochaine fois, nous aborderons les aspects « orienté objet » du langage Ada, ce qui nous occupera quelques temps.

LIENS

- [1] Ada 2005 : http://adaic.org/standards/ada05.html
- [2] Ada WikiBook : http://en.wikibooks.org/wiki/Ada_Programming
- [3] Enquête sur le marché du langage Ada : http://adaic.org/news/survey-results.html
- [4] Le manuel de référence Ada 2005 : http://www.adaic.com/standards/05rm/html/RM-TTL.html

Yves Bailly,

Le langage Ada : l'orientation objet

Yves Bailly

EN DEUX MOTS **Pratiquement tous les langages modernes suivent plus ou moins le paradigme objet, qu'ils soient purement objet (comme Smalltalk ou Ruby) ou autorisent la programmation procédurale (comme C++, Python, Perl...). Dès sa conception en 1979, le langage Ada possède des fondements objets sans pour autant pouvoir être qualifié de langage orienté objet. La norme Ada95 a introduit dans le langage des techniques et principes spécifiquement objets, faisant de ce langage le premier langage orienté objet à être standardisé.**

Avec l'évolution des techniques depuis 1995, il semble naturel que la norme Ada 2005 apporte une refonte importante de l'infrastructure objet du langage Ada. Tout en conservant un aspect procédural fort, nous allons voir comment Ada s'est vu enrichi des techniques objets les plus modernes, de la dérivation au polymorphisme en passant par la notion d'interface empruntée au langage Java.

Des types taggés

Une structure en Ada, type de donnée introduit par le mot-clef `record`, est très similaire en mémoire à une structure en langage C. Elle contient les données indiquées, rien de plus. Imaginons un instant un enregistrement représentant une forme géométrique générique, ne contenant que sa position :

```
type Geom is record
   pos_x: Float ;
   pos_y: Float ;
end record ;
```

Toute variable de ce type occupera très exactement `2*Float'Size` bits en mémoire (c'est-à-dire 64 bits, ou 8 octets, sur architecture 80x86 32bits).

Les habitués de la littérature consacrée à la programmation objet me voient sans doute venir : et si maintenant nous voulions par exemple un cercle ? La position désigne le centre, mais nous avons également besoin du rayon.

Le problème avec notre type `Geom` est qu'il est « figé », pas moyen de lui ajouter des informations. De plus, sa taille étant exactement égale à la somme des tailles de ses composants, on voit mal comment mettre en place un mécanisme comme les méthodes virtuelles du C++...

Depuis Ada 95, ces contraintes sont levées par l'introduction de fonctionnalités orientées objet. Modifions légèrement la déclaration de notre type :

```
type Geom is tagged record
   pos_x: Float ;
   pos_y: Float ;
end record ;
```

L'ajout du discret mot-clef `tagged` change complètement la façon dont notre type est représenté en mémoire et ce que l'on peut en faire. Si vous demandez la valeur de `Geom'Size`, vous n'obtiendrez plus 64 (pour mémoire, l'attribut `'Size` donne la taille d'un objet en bits), mais 96, soit 32 bits de plus, ce qui correspond à la taille d'un pointeur (sur architectures 32bits).

Dès lors, notre type est marqué, ce n'est plus un simple enregistrement, mais un type objet. Plein de choses deviennent alors possibles.

Opérations primitives

Contrairement aux langages objets comme C++, Python, Java et consorts, le langage Ada ne fait pas figurer les déclarations des méthodes d'un type objet au sein de la structure : elles sont simplement déclarées immédiatement après la déclaration du type, dans la même spécification de paquetage.

Plutôt que le terme « méthodes », le verbiage Ada utilise les termes « opérations primitives » pour désigner l'ensemble des sous-programmes (procédures et fonctions) qui agissent directement sur un type taggé. Pour être considéré comme une opération primitive d'un type donné, un sous-programme doit répondre à certaines conditions :

- être déclaré juste après le type, dans la même spécification de paquetage ;
- pour une procédure, prendre un paramètre de ce type, ou un pointeur (`access`) sur ce type ;
- pour une fonction, soit prendre un paramètre de ce type ou un pointeur sur ce type, soit retourner une instance du type ou un pointeur sur une instance du type.

D'autres conditions ou possibilités existent, nous les verrons plus loin.

Ajoutons donc quelques opérations primitives à notre type `Geom`, par exemple une fonction retournant l'aire de l'objet et une procédure permettant de le déplacer, dans un fichier `geom_pkg.ads` :

```
1 package Geom_Pkg is
2
3    type Geom is tagged record
```

```
 4       pos_x: Float := 0.0 ;
 5       pos_y: Float := 0.0 ;
 6    end record ;
 7    function Aire(g: in Geom) return Float ;
 8    procedure Déplacer(g : in out Geom ;
 9                       dx: in Float ;
10                       dy: in Float) ;
11 end Geom_Pkg ;
```

Tout cela ne semble rien avoir de bien particulier, si ce n'est le mot-clef `tagged` ligne 3. Le corps de ce paquetage (dans `geom_pkg.adb`) est trivial :

```
 1 package body Geom_Pkg is
 2    function Aire(g: in Geom) return Float is
 3    begin
 4       return 0.0 ;
 5    end Aire ;
 6    procedure Déplacer(g : in out Geom ;
 7                       dx: in Float ;
 8                       dy: in Float) is
 9    begin
10       g.pos_x := g.pos_x + dx ;
11       g.pos_y := g.pos_y + dy ;
12    end Déplacer ;
13 end Geom_Pkg ;
```

Voyons maintenant un petit programme d'exemple `geom_1.adb` utilisant notre paquetage :

```
 1 with Text_IO ; use Text_IO ;
 2 with Geom_Pkg ;
 3 procedure Geom_1 is
 4    g: Geom_Pkg.Geom ;
 5 begin
 6    Put_Line(Float'Image(Geom_Pkg.Aire(g))) ;
 7    Put_Line(Float'Image(g.Aire)) ;
 8    Geom_Pkg.Déplacer(g, 1.0, 0.0) ;
 9    Put_Line(Float'Image(g.pos_x)) ;
10    g.Déplacer(1.0, 0.0) ;
11    Put_Line(Float'Image(g.pos_x)) ;
12 end Geom_1 ;
```

Pour compiler ce programme, utilisez la commande :

```
$ gnatmake -Wall -gnat05 -gnatif geom_1
```

L'option `-gnatif` est nécessaire pour pouvoir utiliser des caractères non-ASCII dans les identificateurs, comme c'est le cas de la procédure `Déplacer`.

La ligne 6 affiche le résultat de la fonction `Aire`, appliquée à la variable `g` déclarée ligne 4. La ligne suivante (7) fait exactement la même chose, mais en utilisant une nouvelle possibilité apportée par Ada 2005 : la notation pointée, si commune dans l'immense majorité des autres langages objets. Celle-ci n'est utilisable que si le sous-programme invoqué est une opération primitive du type de la variable sur laquelle on l'applique. Les lignes 6 et 7 sont ainsi parfaitement synonymes.

Remarquez que la fonction `Aire` ne prenant qu'un seul paramètre (de type `Geom`, ce qui en fait une opération primitive sur ce type, étant donné qu'elle est déclarée juste après celui-ci), celui-ci lui est implicitement passé dans la notation pointée. Tout se passe alors comme si elle ne prenait pas de paramètre : les parenthèses ne doivent donc pas être utilisées, comme pour tout sous-programme ne prenant pas de paramètre.

La situation est tout à fait similaire à celle d'une méthode dans une classe C++ qui ne prendrait pas de paramètre, ou d'une méthode d'une classe Python dont le seul paramètre serait `self`.

De la même façon, les lignes 8 et 10 sont parfaitement équivalentes : la première utilise la notation traditionnelle en Ada, tandis que la seconde utilise la notation pointée. Remarquez que le paramètre de type `Geom` « disparaît » : il est implicitement passé à la procédure, un peu comme un paramètre qui serait nommé `this` pour une méthode d'une classe C++.

Pour information, la notation pointée n'est supportée par le compilateur GNAT que dans les versions récentes de la suite GCC. Si votre compilateur date un peu, il est possible qu'il renvoie une erreur si vous l'utilisez.

À titre de comparaison, voyons comment serait déclaré notre type `Geom` de façon équivalente en C++ et en Python :

Trois langages pour un type

Ada

```
type Geom is tagged record
   pos_x: Float := 0.0 ;
   pos_y: Float := 0.0 ;
end record ;
function Aire(g: in Geom) return Float ;
procedure Déplacer(g : in out Geom ;
                   dx: in Float ;
                   dy: in Float) ;
```

C++

```
struct Geom
{
   Geom():
      pos_x(0.0),
      pos_y(0.0)
   { }
   float Aire() const ;
   void Deplacer(float dx,
                 float dy) ;
   float pos_x ;
   float pos_y ;
} ;
```

Python

```
class Geom:
   def __init__(self):
      self.pos_x = 0.0
      self.pos_y = 0.0
   def Aire(self):
      return 0.0
   def Deplacer(self, dx, dy):
      self.pos_x += dx
      self.pos_y += dy
```

Il est intéressant de constater que ce langage réputé si verbeux qu'est Ada nécessite moins de lignes que les deux autres pour cette déclaration.

Opérations nulles

Il n'est pas rare en conception orientée objet de prévoir une opération sur un type de base qui ne fait rien. Cette opération est destinée à être surchargée dans les types dérivés (que nous verrons un peu plus loin), mais elle présente un sens même sur le type de base. Par exemple, une telle opération pourrait être une mise à l'échelle (homothétie), qui n'est pas en elle-même absurde sur un objet géométrique se résumant à un point. Simplement, elle ne fait rien.

On pourrait déclarer cette opération primitive ainsi :

```
procedure Homothétie(g : in out Geom ;
                     sx: in Float ;
                     sy: in Float) ;
```

Son implémentation ne contiendrait alors pas d'instruction, seulement le mot-clef `null` imposé par le langage :

```
procedure Homothétie(g : in out Geom ;
                     sx: in Float ;
                     sy: in Float) is
begin
   null ;
end Homothétie ;
```

Ce qui fait bien beaucoup de code pour si peu de chose. Mais Ada prévoit une syntaxe particulière pour ce genre de situations. Notre opération peut être avantageusement qualifiée par `is null` :

```
procedure Homothétie(g : in out Geom ;
                     sx: in Float ;
                     sy: in Float) is null ;
```

L'ajout du qualificatif `is null` stipule que cette opération ne fait rien, qu'aucun code ne lui est associé. Procéder de la sorte présente plusieurs avantages :

- L'utilisateur du paquetage est informé du fait que l'invocation de cette procédure n'aura aucun effet.
- Le compilateur peut utiliser cette information pour améliorer l'optimisation du code exécutable généré.
- Enfin, si vous décidez subitement de donner un corps à cette procédure dans le corps du paquetage, le compilateur vous signalera qu'il y a un problème : cela peut permettre d'identifier des erreurs pouvant survenir lors d'une reconception un peu hâtive.

Le dernier point est probablement le plus important, dans la perspective de la maintenance et de l'évolution du programme. La conception du langage Ada est dirigée par le constat simple (parmi d'autres) que le code est écrit une seule fois, mais relu et modifié de nombreuses fois. Le qualificatif `is null` est donc un outil destiné à améliorer la qualité de l'évolution des programmes.

Naturellement, une fonction ne peut pas recevoir ce qualificatif : toute fonction doit retourner une valeur, ce qu'il est impossible de faire sans au moins une instruction, aussi triviale soit-elle.

Opérations abstraites

Il arrive également en conception objet qu'une opération n'ait pas de sens sur un type de base, qu'elle doive donc être définie explicitement dans les types dérivés, mais qu'elle s'impose pour la cohérence générale. C'est une opération abstraite ou encore une opération virtuelle. Reconsidérons un instant l'homothétie évoquée précédemment. On peut la concevoir de deux façons : soit une mise à l'échelle à partir de la position de l'objet, soit à partir de l'origine (le point de coordonnées (0,0)). La première pourrait fort bien entrer dans la catégorie des opérations abstraites, selon une conception un peu plus stricte de notre type `Geom`.

Une opération abstraite est déclarée par le qualificatif `is abstract` :

```
procedure Homothétie_Position(g : in out Geom ;
                              sx: in Float ;
                              sy: in Float) is abstract ;
```

Définir une opération abstraite sur un type en fait de facto un type abstrait, qui doit alors impérativement être déclaré ainsi :

```
type Geom is abstract tagged record
   pos_x: Float := 0.0 ;
   pos_y: Float := 0.0 ;
end record ;
```

Le mot-clef `abstract` a été ajouté devant `tagged` : le type `Geom` devient abstrait, donc il possède (au moins) une opération abstraite. Non seulement aucun code n'est associé à une telle opération (comme pour une opération nulle), mais en plus son invocation n'a pas de sens : elle est donc interdite.

S'il est impossible d'invoquer une opération définie sur un type, alors créer une instance de ce type n'a pas de sens non plus : il est donc impossible de créer une variable d'un type abstrait. Cela implique que notre programme de test ne compilera plus, une erreur sera signalée sur la ligne 4.

On se trouve ici dans la même situation que celle des classes abstraites en C++, qui contiennent (au moins) une méthode virtuelle pure. De tels types ne sont pas utilisables directement. Par contre, ils prennent tout leur sens lorsque l'on manipule des pointeurs sur ces types, par le mécanisme du polymorphisme associé à la dérivation.

Types dérivés

Ils ont été évoqués depuis le début de cet article, les voici enfin : les types dérivés. Un type dérivé est construit à partir d'un type dit « ancêtre », auquel on ajoute des données et des opérations. Ce principe est également désigné par le terme d'« héritage ».

Créons par exemple un type représentant une ellipse à partir de notre type Geom, dans un paquetage Ellipses :

```
 1 with Geom_Pkg ;
 2 package Ellipses is
 3    type Ellipse is new Geom_Pkg.Geom with
 4    record
 5       rayon_x: Float := 1.0 ;
 6       rayon_y: Float := 1.0 ;
 7    end record ;
 8    overriding
 9    function Aire(e: in Ellipse) return Float ;
10    overriding
11    procedure Homothétie_Position(e : in out Ellipse ;
12                                  sx: in Float ;
13                                  sy: in Float) ;
14    overriding
15    procedure Homothétie_Origine(e : in out Ellipse ;
16                                 sx: in Float ;
17                                 sy: in Float) ;
18    not overriding
19    procedure Put(e: in Ellipse) ;
20 end Ellipses ;
```

Comme nous allons étendre le type Geom défini dans le paquetage Geom_Pkg, il semble naturel de commencer par indiquer son utilisation (ligne 1). La déclaration du nouveau type Ellipse commence ligne 3. La partie is new Geom_Pkg.Geom indique que Ellipse dérive du type Geom_Pkg.Geom. La partie with record..end record donne le contenu de l'extension du type, c'est-à-dire les données que nous lui ajoutons. Si vous ne voulez ajouter aucune donnée supplémentaire, il suffit d'écrire with null record.

Suivent les opérations définies sur ce type. On retrouve la fonction donnant l'aire, les deux types d'homothétie évoquées plus haut, ainsi qu'une nouvelle opération Put() dont la vocation est simplement d'afficher une chaîne représentant l'ellipse (sa position et ses rayons). Deux remarques.

D'abord, l'opération Homothétie_Position() n'est plus abstraite : nous allons lui donner un corps. Dès lors, le type Ellipse ne présentant plus aucune opération abstraite, il devient un type concret : nous pourrons déclarer des variables de ce type.

Ensuite, les déclarations des opérations sont précédées du mot-clef overriding, lui-même précédé de not dans le cas de Put(). Ces qualificatifs ne sont pas obligatoires, mais il est vivement recommandé de les utiliser. Leur signfication est simple.

overriding signale une opération surdéfinie par rapport à celle du type ancêtre. Ici, les deux homothéties existaient pour le type ancêtre Geom. Dérivant de celui-ci, le type Ellipse hérite de ces opérations primitives. Mais elles doivent parfois être adaptées pour tenir compte des spécificités du type dérivé. C'est exactement ce que nous faisons ici. Au contraire, l'opération Put() est nouvelle pour le type Ellipse, elle n'existait pas dans le type ancêtre : elle ne surdéfinie donc aucune opération, d'où la présence de la négation not devant overriding.

Encore une fois, ces informations ne sont pas obligatoires, le compilateur s'y retrouve très bien sans elles. Alors, pourquoi s'embêter à les écrire ? Par prévention, en prévision des modifications futures du code.

Imaginons que pour une raison ou une autre, vous décidiez de supprimer l'opération Homothétie_Position() au niveau du type Geom. Elle est abstraite, aussi cela ne semble-t-il pas avoir beaucoup de conséquences. Sans la présence de overriding dans la surdéfinition au niveau de Ellipse, pas de problème, le paquetage Ellipses compile toujours. Mais ce changement malgré tout assez important peu avoir des conséquences pour d'autres utilisateurs du type Geom. Si overriding a été inscrit, le compilateur vous signalera immédiatement une erreur : en effet, au niveau de Ellipse l'opération Homothétie_Position() ne surdéfinit plus rien, ce qui peut être un avertissement d'un problème de conception. C'est une sécurité contre des manipulations aux conséquences parfois difficilement prévisibles dans un grand programme.

Dans l'autre sens, supposons que vous décidiez d'ajouter une opération Put() au niveau de Geom, sous la forme d'une procédure ne prenant qu'un seul paramètre (de type Geom). Dès lors, l'opération éponyme de Ellipse devient une surdéfinition de celle-ci. Ce n'est pas très grave dans notre exemple, mais dans un programme plus vaste cela peut conduire à des incohérences et des bogues difficiles à identifier. Grâce à la présence de not overriding, là encore le compilateur signalera une erreur, vous invitant à bien vérifier ce que vous faites. Encore une sécurité, destinée à assurer la bonne qualité du programme au fil du temps.

Voici maintenant le corps du paquetage Ellipses :

```
 1 with Text_IO ; use Text_IO ;
 2 package body Ellipses is
 3    overriding
 4    function Aire(e: in Ellipse) return Float is
 5    begin
 6       return 3.14159*e.rayon_x*e.rayon_y ;
 7    end Aire ;
 8    overriding
 9    procedure Homothétie_Position(e : in out Ellipse ;
10                                  sx: in Float ;
11                                  sy: in Float) is
12    begin
13       e.rayon_x := e.rayon_x * sx ;
14       e.rayon_y := e.rayon_y * sy ;
15    end Homothétie_Position ;
16    overriding
17    procedure Homothétie_Origine(e : in out Ellipse ;
18                                 sx: in Float ;
19                                 sy: in Float) is
20    begin
21 --       Geom_Pkg.Homothétie_Origine(Geom_Pkg.Geom(e), sx, sy) ;
22       Geom_Pkg.Geom(e).Homothétie_Origine(sx, sy) ;
23       Homothétie_Position(e, sx, sy) ;
24    end Homothétie_Origine ;
25    not overriding
```

```
26    procedure Put(e: in Ellipse) is
27    begin
28       Put("Ellipse[(" & Float'Image(e.pos_x) & ", " & Float'Image(e.pos_y) &
29          "), " & Float'Image(e.rayon_x) & "x" & Float'Image(e.rayon_y) & "]") ;
30    end Put ;
31 end Ellipses ;
```

On retrouve les `(not) overriding`, scrupuleusement reproduits. Regardez le corps de `Homothétie_Origine()`, lignes 16 à 18. La première est commentée, car elle est exactement équivalente à la seconde, qui utilise la notation pointée. Le but est d'invoquer l'opération ancêtre, celle définie sur le type `Geom` du paquetage `Geom_Pkg`.

Pour cela, on effectue un transtypage du paramètre `e` : l'écriture `Geom_Pkg.Geom(e)` « transforme » `e` pour qu'il présente l'aspect du type `Geom`. Cela n'est naturellement possible que parce que le type `Ellipse` dérive de `Geom` : dans le cas contraire, la conversion serait refusée par le compilateur. Cela n'a rien à voir avec la conversion de type du langage C, qui permet de transformer des carottes en lapins sans que le compilateur n'y trouve rien à redire. On pourrait plutôt l'assimiler au `static_cast<>()` du C++.

Voici un petit programme d'exemple :

```
 1 with Text_IO ; use Text_IO ;
 2 with Geom_Pkg ;
 3 with Ellipses ;
 4 procedure Geom_2 is
 5    e: Ellipses.Ellipse := (pos_x => 0.1,
 6                            pos_y => 0.2,
 7                            rayon_x => 1.0,
 8                            rayon_y => 2.0)  ;
 9 begin
10    e.Put ; New_Line ;
11    e.Homothétie_Position(2.0, 3.0) ;
12    e.Put ; New_Line ;
13    e.Homothétie_Origine(2.0, 3.0) ;
14    e.Put ; New_Line ;
15 end Geom_2 ;
```

Rappelons que le type `Geom_Pkg.Geom` étant abstrait, nous ne pouvons pas créer d'instance de ce type. Par contre le type `Ellipse` est concret : une instance en est déclarée ligne 5, avec un agrégat d'initialisation. Remarquez que les données de `Geom` y figurent, comme s'il s'agissait de données de `Ellipse` – ce qui est plus ou moins le cas.

Voici l'affichage de ce programme :

```
$ ./geom_2
Ellipse[( 1.00000E-01,  2.00000E-01),  1.00000E+00x 2.00000E+00]
Ellipse[( 1.00000E-01,  2.00000E-01),  2.00000E+00x 6.00000E+00]
Ellipse[( 2.00000E-01,  6.00000E-01),  4.00000E+00x 1.80000E+01]
```

Tout cela est finalement assez simple, surtout si vous avez déjà pratiqué la programmation objet. Mais voyons maintenant une petite subtilité, qui aura par la suite de grandes conséquences.

La classe, le type et l'objet

Imaginons une hiérarchie de types, issus de `Geom`, un peu plus fournie. De `Geom` on tire `Triangle`, de `Ellipse` on tire `Ellipse_Inclinée` et `Rectangle`, duquel on obtient `Rectangle_Incliné`. On a donc six types taggés différents. Maintenant, interrogeons-nous un instant sur ce qui fait un type. D'un point de vue théorique, un type est défini par l'ensemble de ses valeurs et l'ensemble des opérations disponibles sur ces valeurs. Cela peut sembler assez évident quand on parle d'un type entier comme `Integer`, mais cela vaut également pour des types taggés comme `Geom` ou `Ellipse`. En Ada, à chaque type taggé `T` est associé une classe désignée par `T'Class`. La classe d'un type `T` est un type indéfini (un peu comme un tableau non contraint) ne possédant pas d'opérations primitives propres, et dont les valeurs sont l'union des valeurs du type `T` et des valeurs des types dérivés de `T`. Voyons cela sur un exemple. Le diagramme suivant est une modélisation UML de la hiérarchie de types évoquée plus haut. Les cadres noirs représentent la « portée » des classes de chaque type dans cette hiérarchie (Fig. 1).

Considérons maintenant un petit programme, contenant une procédure affichant simplement l'aire de l'objet géométrique passé en paramètre :

```
 1 with Text_IO ; use Text_IO ;
 2 with Ellipses ; use Ellipses ;
 3 with Rectangles ; use Rectangles ;
 4 procedure Geom_3 is
 5    procedure Aff_Aire(ell: in Ellipse) is
 6    begin
 7       Put_Line("Aire :" & Float'Image(ell.Aire)) ;
 8    end Aff_Aire ;
 9    e: Ellipse ;
10    r: Rectangle ;
11 begin
12    Aff_Aire(e) ;
13    Aff_Aire(r) ;
14 end Geom_3 ;
```

Tout cela paraît fort bien, surtout du point de vue d'un programmeur C++. Les lignes 12 et 13 semblent naturelles : après tout, un `Rectangle` « est » une `Ellipse`, n'est-ce pas ? Pas en Ada. Si vous l'aviez oublié, Ada est un langage au typage strict. En l'état, ce programme ne compile pas : la ligne 13 est une erreur, car on tente de passer une variable de type `Rectangle` à une procédure qui attend un paramètre de type `Ellipse`. Donc cela ne va pas.

C'est là qu'intervient la notion de « classe ». Modifiez la ligne 5 ainsi :

```
5    procedure Aff_Aire(ell: in Ellipse'Class) is
```

Maintenant la procédure `Aff_Aire()` attend un paramètre de type `Ellipse'Class`, c'est-à-dire une valeur dont le type est `Ellipse` ou l'un des types dérivés (directement ou non) de `Ellipse`. La variable `r` est de type `Rectangle`, qui dérive de `Ellipse`, donc elle peut être passée à la procédure : la ligne 13 n'est plus une erreur. Voici le résultat de l'exécution :

```
$ ./geom_3
Aire : 3.14159E+00
Aire : 4.00000E+00
```

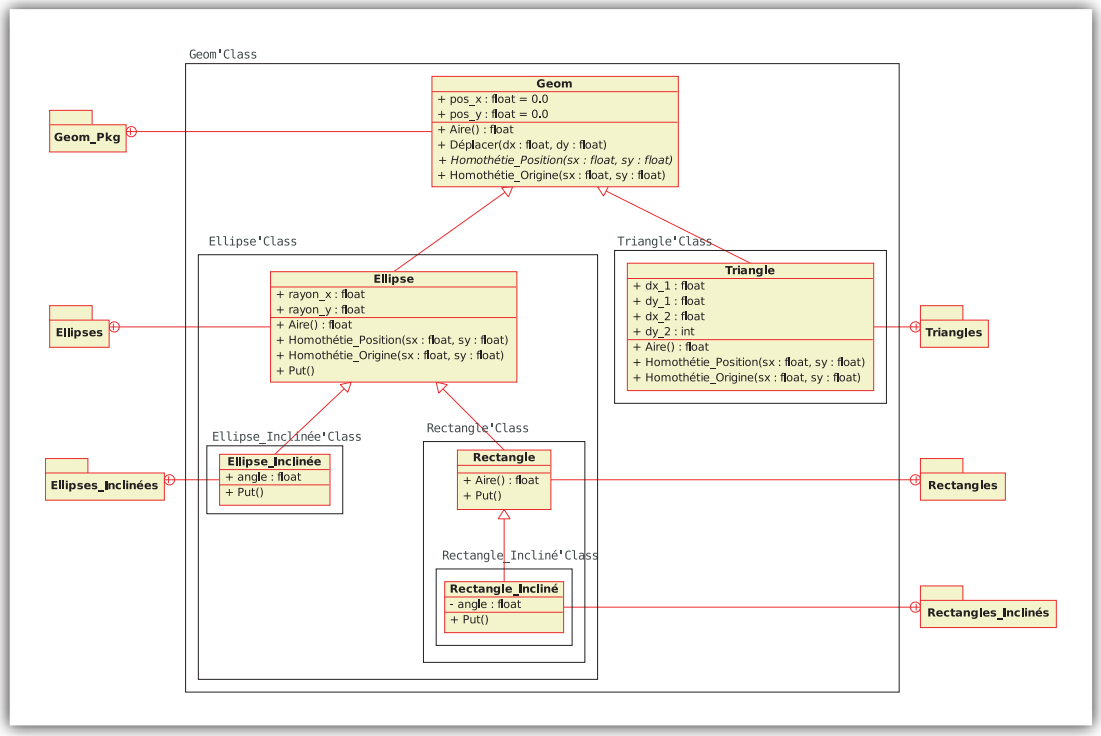


Fig. 1

Remarquez que c'est la « bonne » fonction `Aire()` qui est invoquée : celle du type `Ellipse` lorsque est passée une variable de type `Ellipse`, celle (surdéfinie) du type `Rectangle` lorsque est passée une variable de type `Rectangle`. Nous avons là un bel exemple de polymorphisme.

Le langage Ada fait donc une distinction entre un type objet et la classe de ce type, deux notions qui sont généralement confondues dans la plupart des autres langages objets. Reprenez la comparaison de la déclaration du type `Geom` entre les langages Ada, C++ et Python.

En réalité, ces déclarations ne sont pas tout à fait équivalentes. En C++ et Python, on définit à la fois un type (nommé `Geom`) et une classe de types (nommée `Geom`). En Ada, on ne définit que un type nommé `Geom`, la classe de types associée étant nommée `Geom'Class`.

Considérez les déclarations suivantes :

```
1 ell       : Ellipse ;
2 rect      : Rectangle ;
3 ell_class : Ellipse'Class := rect ;
4 rect_class: Rectangle'Class := ell ;
```

Les deux premières sont « normales ». La troisième déclare une variable de type `Ellipse'Class`, c'est-à-dire de la classe du type `Ellipse`. Il a été signalé plus haut qu'un type classe est indéfini, comme un tableau non contraint : une variable d'un tel type doit donc impérativement être initialisée pour être effectivement définie.

La ligne 3 est correcte, car on affecte à `ell_class` une valeur de type `Rectangle`, laquelle fait bien partie de l'ensemble des valeurs de `Ellipse'Class`, étant donné que `Rectangle` dérive de `Ellipse`.

Si on passait la variable `ell_class` à la procédure `Aff_Aire()` de l'exemple précédent, l'opération `Aire()` invoquée serait celle du type réel de `ell_class`, donc celle de `Rectangle`.

Par contre, la quatrième ligne est invalide. Une valeur de type `Ellipse` ne fait pas partie de l'ensemble des valeurs du type `Rectangle'Class`.

Le polymorphisme n'opère que vers le haut (la racine) de la hiérarchie de types, pas vers le bas. La ligne 4 sera donc refusée par le compilateur.

Cette distinction entre type et classe est fondamentale en Ada. Elle est à la base du mécanisme du polymorphisme tel que proposé par Ada, mécanisme que l'on désigne plus souvent sous le terme de *dispatching*.

Interfaces

Information qui va décevoir les développeurs C++ ou Python : Ada ne supporte pas l'héritage multiple. Information qui va ravir les développeurs Java : Ada supporte la notion d'« interface », depuis le standard Ada 2005.

La version annotée de celui-ci fait même explicitement référence à Java comme inspiration de cette fonctionnalité.

On désigne par interface un « pseudo-type » ne contenant aucune donnée, ne pouvant pas être instancié et n'offrant que des opérations abstraites. Il s'agit donc d'un type purement abstrait.

Ce type peut être utilisé lors de la dérivation d'un type de base, éventuellement avec d'autres interfaces. Cela permet de garantir qu'un type donné supporte au minimum un certain ensemble d'opérations.

Considérons le cas de l'opération `Aire()` de notre petite hiérarchie d'objets géométriques. Chaque type (ou presque) possède son propre principe pour calculer la surface d'une instance.

Par ailleurs, cette notion d'aire n'est pas limitée à des formes en deux dimensions, elle a également un sens pour des formes en trois dimensions – mais pas toutes... On se trouve là typiquement dans la situation où la définition d'une interface devient pertinente.

Reconsidérons donc notre hiérarchie ainsi :

- à la racine se trouve un type `Geom`, pour lequel l'aire n'a pas de sens ;
- de ce type, sont dérivés les types `Ellipse` et `Triangle`, pour lesquels l'aire a un sens ;
- de `Ellipse`, on dérive `Rectangle` ; l'aire a un sens mais est calculée différemment.

Il semble alors logique de créer une interface `Surface`, fournissant une opération `Aire()`. Une représentation UML (partielle) ressemblerait à la figure 2.

Clairement, les trois types `Ellipse`, `Triangle` et `Rectangle` dérivent de deux choses : cela ressemble à de l'héritage multiple, mais cela n'en est pas, car un type ne peut dériver que d'un seul autre type.

Par contre, il peut également « dériver » de plusieurs interfaces, une interface pouvant elle-même dériver de plusieurs interfaces.

Déclarons tout cela dans un paquetage Ada (dans la réalité, chaque type aurait son propre paquetage) :

```
 1 package Geometrie is
 2
 3    type Geom is abstract tagged null record ;
 4
 5    type Surface is interface ;
 6    function Aire(s: in Surface) return Float is abstract ;
 7
 8    type Triangle is new Geom and Surface with null record ;
 9    overriding
10    function Aire(t: in Triangle) return Float ;
11
12    type Ellipse is new Geom and Surface with null record ;
13    overriding
14    function Aire(e: in Ellipse) return Float ;
15
16    type Rectangle is new Ellipse with null record ;
17    overriding
18    function Aire(r: in Rectangle) return Float ;
19
20    procedure Aff_Aire(s: in Surface'Class) ;
21
22 end Geometrie ;
```

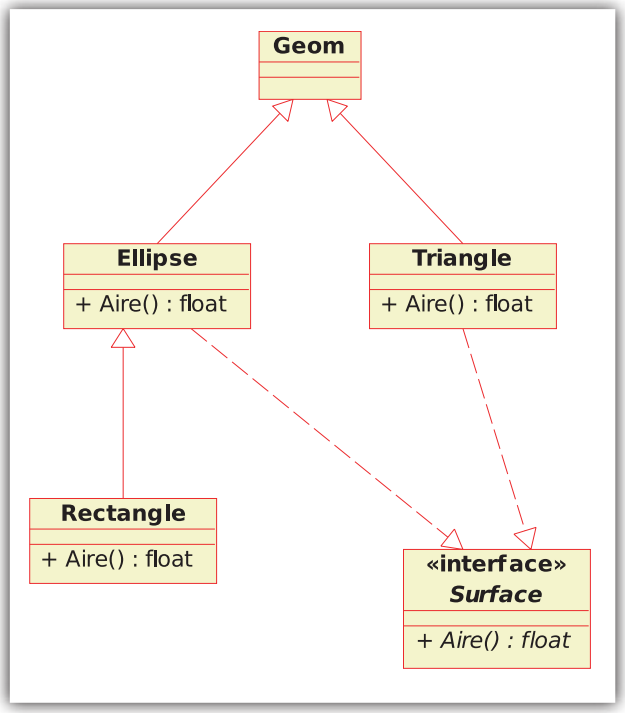


Fig. 2

Pour simplifier, aucun type ne contient de donnée : c'est pourquoi ils sont tous déclarés avec `with null record`, écriture raccourcie pour spécifier un contenu vide.

L'interface `Surface` est déclarée ligne 5, son opération `Aire()` ligne 6. Celle-ci est qualifiée `abstract` : une interface étant toujours un type abstrait, ses opérations sont forcément abstraites. Remarquez qu'on ne lui donne aucun contenu. Ce n'est pas par choix, c'est obligatoire : par définition, une interface ne contient aucune donnée.

Les types `Triangle` et `Ellipse`, qui héritent à la fois du type abstrait `Geom` et de l'interface `Surface`, surdéfinissent l'opération `Aire()`. S'ils ne le faisaient pas, ils demeureraient des types abstraits (car contenant au moins une opération abstraite) et ne pourraient donc pas être instanciés.

Le type `Rectangle` dérive de `Ellipse`. Par conséquent, il possède également l'opération `Aire()`. Elle n'est ici surdéfinie que pour tenir compte de la spécificité du type. Ce n'est pas syntaxiquement obligatoire. `Rectangle` hérite indirectement de `Surface`.

Enfin, on retrouve la procédure `Aff_Aire()` pour afficher l'aire d'une surface. Son paramètre peut être n'importe quel type dérivant (directement ou non) de l'interface `Surface` :

on garantit ainsi que ce paramètre possède au moins les opérations de Surface, sans faire aucune hypothèse sur d'autres opérations ou d'éventuelles données. On pourrait donc passer à Aff_Aire() un paramètre d'un type complètement différent, issu d'un autre paquetage, du moment qu'il dérive de Surface.

Voyons le corps de ce paquetage, volontairement simplifié :

```
 1 with Text_IO ; use Text_IO ;
 2 package body Geometrie is
 3
 4    overriding
 5    function Aire(t: in Triangle) return Float is
 6    begin
 7       return 1.0 ;
 8    end Aire ;
 9
10    overriding
11    function Aire(e: in Ellipse) return Float is
12    begin
13       return 2.0 ;
14    end Aire ;
15
16    overriding
17    function Aire(r: in Rectangle) return Float is
18    begin
19       return 3.0 ;
20    end Aire ;
21
22    procedure Aff_Aire(s: in Surface'Class) is
23    begin
24       Put_Line("Aire =" & Float'Image(s.Aire)) ;
25    end Aff_Aire ;
26
27 end Geometrie ;
```

Et un petit programme d'exemple :

```
 1 with Geometrie ;
 2 procedure Geom_4 is
 3    t: Geometrie.Triangle ;
 4    e: Geometrie.Ellipse ;
 5    r: Geometrie.Rectangle ;
 6 begin
 7    Geometrie.Aff_Aire(t) ;
 8    Geometrie.Aff_Aire(e) ;
 9    Geometrie.Aff_Aire(r) ;
10 end Geom_4 ;
```

Une instance de chaque type est créée, puis on demande l'affichage de l'aire pour chacune, ce qui donne :

```
$ ./geom_4
Aire = 1.00000E+00
Aire = 2.00000E+00
Aire = 3.00000E+00
```

C'est bien la « bonne » opération Aire() qui est invoquée pour chaque instance. Le polymorphisme a encore frappé.

Imaginons un instant un type Segment dérivant de Geom. On peut estimer que la notion d'« aire » n'a pas de sens pour un segment (cela est discutable d'un point de vue mathématique, mais passons là-dessus). Logiquement, Segment n'hériterait pas de l'interface Surface.

Tenter alors de passer une instance de Segment à Aff_Aire() résulterait immédiatement en une erreur de compilation.

Conclusion

Vous pouvez constater que le langage Ada n'a pas grand-chose à envier à d'autres langages concernant les fonctionnalités orientées objet.

Le mois prochain, nous verrons quelques applications de ce principe en relation avec l'encapsulation ultime que constituent les types limités et les types contrôlés.

Yves Bailly

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,

Le langage Ada : liaison avec d'autres langages

Yves Bailly

EN DEUX MOTS **L'une des exigences du langage Ada, lors de sa conception, était de pouvoir s'interfacer avec les autres principaux langages existants à l'époque. De ce point de vue, Ada est peut-être plus ouvert que d'autres langages, permettant la réutilisation de bibliothèques existantes assez facilement.**

Le standard Ada prévoit, dans son Annexe B, diverses facilités pour utiliser d'autres langages, nommément les langages C/C++, Fortran et Cobol. Ce dernier a connu son heure de gloire, dominant largement l'industrie informatique pendant de nombreuses années, mais il est aujourd'hui, disons, « un peu » passé de mode. Aussi ne le verrons-nous pas.

Par contre, le langage Fortran est toujours très utilisé dans les domaines gros consommateurs de calculs, qu'ils soient scientifiques ou industriels. Quant au langage C, il est probablement encore le plus répandu et le plus courant pour interfacer des systèmes.

Fortran

Précisons dès maintenant que votre serviteur est bien loin d'être un expert en langage Fortran. Toutefois, ces maigres connaissances permettent tout de même d'écrire une procédure (ou sous-routine, *subroutine* en anglais) permettant de résoudre le problème des Tours de Hanoï (avouez que cela vous manquait), dans un fichier `f_hanoi.f` que voici :

```
 1 subroutine Hanoi(nb, depuis, par, vers)
 2    integer, intent(in) :: nb
 3    integer, intent(in) :: depuis
 4    integer, intent(in) :: par
 5    integer, intent(in) :: vers
 6    if (nb == 1) then
 7       call Deplacer(depuis, vers)
 8    else
 9       call Hanoi(nb-1, depuis, vers, par)
10       call Hanoi(1, depuis, par, vers)
11       call Hanoi(nb-1, par, depuis, vers)
12    end if
13 end subroutine Hanoi
```

On retrouve l'algorithme classique. Remarquez l'invocation ligne 7, d'une procédure `Deplacer()` qui apparemment n'existe pas : elle sera en fait fournie par un paquetage Ada, lequel va lui-même importer la procédure `Hanoi()`. Voici la spécification du paquetage, dans `a_hanoi.ads` :

```
 1 with Interfaces.Fortran ; use Interfaces.Fortran ;
 2 package A_Hanoi is
 3
 4    procedure Hanoi(nb     : in Fortran_Integer ;
 5                    depuis : in Fortran_Integer ;
 6                    par    : in Fortran_Integer ;
 7                    vers   : in Fortran_Integer) ;
 8    pragma Import(Fortran, Hanoi, "hanoi_") ;
 9
10    procedure Deplacer(depuis: in Fortran_Integer ;
11                       vers  : in Fortran_Integer) ;
12    pragma Export(Fortran, Deplacer, "deplacer_") ;
13
14 end A_Hanoi ;
```

On importe, pour commencer, le paquetage `Interfaces.Fortran` (ligne 1), qui définit quelques types et quelques sous-programmes adaptés au Fortran. Ici, nous n'allons utiliser que le type `Fortran_Integer`, un type Ada entier compatible avec le type Fortran `integer`. Vient la déclaration de la procédure `Hanoi()`. Sa spécification est similaire à celle du programme Fortran. Seulement, elle n'aura pas d'implémentation : la ligne 8 est une directive `pragma` qui signale que cette procédure `Hanoi()` (deuxième paramètre) sera importée (`Import`) selon la convention d'appel `Fortran` (premier paramètre) depuis le nom `"hanoi_"` (troisième paramètre). Par défaut, le compilateur Fortran ajoute systématiquement un caractère de soulignement aux noms des symboles avant qu'ils ne soient traités par l'éditeur de liens, d'où sa présence dans la chaîne de caractères. Nous trouvons ensuite la procédure `Deplacer()`, cette fois-ci une « vraie » procédure Ada qui aura un corps. Mais pour qu'elle puisse être utilisée par le code Fortran, on la fait suivre d'une directive `pragma` similaire à la précédente, sauf que, cette fois, il s'agit d'exporter (`Export`) un point d'entrée. Remarquez le troisième paramètre, un caractère de soulignement a été ajouté au nom.

Pour être complet, voici le corps du paquetage, dans `a_hanoi.adb` :

```
 1 with Ada.Text_IO ; use Ada.Text_IO ;
 2 package body A_Hanoi is
 3    procedure Deplacer(depuis: in Fortran_Integer ;
 4                       vers  : in Fortran_Integer) is
 5    begin
 6       Put_Line(Fortran_Integer'Image(depuis) &
 7                " --> " &
 8                Fortran_Integer'Image(vers)) ;
 9    end Deplacer ;
10 end A_Hanoi ;
```

Reste à créer un petit programme pour tester tout cela :

```
1 with A_Hanoi ;
2 procedure Hanoi is
3 begin
4    A_Hanoi.Hanoi(3, 1, 2, 3) ;
5 end Hanoi ;
```

Difficile de faire plus élémentaire : ce programme se contente d'appeler la procédure `Hanoi()` du paquetage `A_Hanoi`, le code de celle-ci étant en réalité obtenu dans le source Fortran. Pour compiler l'ensemble, deux étapes sont nécessaires : d'abord compiler le code Fortran, puis compiler et lier le programme Ada. C'est-à-dire :

```
$ gfortran --free-form -c f_hanoi.f
$ gnatmake hanoi -largs f_hanoi.o -lgfortran
gcc -c hanoi.adb
gcc -c a_hanoi.adb
gnatbind -x hanoi.ali
gnatlink hanoi.ali f_hanoi.o -lgfortran
$ ./hanoi
 1 -->  3
 1 -->  2
 3 -->  2
 1 -->  3
 2 -->  1
 2 -->  3
 1 -->  3
```

Les paramètres qui suivent `-largs` sur la ligne `gnatmake` sont passés directement à l'éditeur de liens (ici `gnatlink`). Il nous suffit donc de lier le fichier objet résultant de la première commande aux fichiers objets du programme Ada. L'exécution prouve que tout cela fonctionne.

C

Les facilités pour connecter Ada et C sont plus développées que pour Fortran. Elles sont divisées en trois paquetages :

- `Interfaces.C` contient différents types compatibles avec les types de base du C, comme `int`, `float` ou `char_array` (pour les tableaux de caractères) ;
- `Interfaces.C.Strings` fournit un ensemble d'outils dédiés aux chaînes de caractères de type `char*` ;
- Enfin, `Interfaces.C.Pointers` (générique) apporte des opérations usuelles pour la manipulation des pointeurs.

Reprenons l'exemple précédent, mais cette fois le programme principal sera implémenté en C. Il fera appel aux sous-programmes exportés par un paquetage Ada, lequel fera lui-même appel à une fonction C pour l'affichage. Voici la spécification du paquetage en question :

```
 1 with Interfaces.C;
 2 with Interfaces.C.Strings;
 3 package Hanoi_A is
 4
 5    package C renames Interfaces.C;
 6    package CS renames Interfaces.C.Strings;
 7
 8    function Chars_To_Int(c_str: in CS.chars_ptr) return C.int;
 9    pragma Export(C, Chars_To_Int, "Chars_To_Int");
10
11    procedure Afficher_Depl(c_str: in CS.chars_ptr);
12    pragma Import(C, Afficher_Depl, "Afficher_Depl");
13
14    procedure Hanoi(depuis: in C.int;
15                    via   : in C.int;
16                    vers  : in C.int;
17                    nb    : in C.int);
18    pragma Export(C, Hanoi, "Hanoi");
19
20 end Hanoi_A;
```

Les lignes 5 et 6 sont un moyen commode et usuel d'éviter de devoir saisir de grands noms de paquetages, comme `Interfaces.C.Strings` : elles ne créent pas de nouveau paquetage, simplement un synonyme pour deux paquetages existants. Ainsi, par exemple ligne 8, l'écriture `CS.chars_ptr` est équivalente à `Interfaces.C.Strings.chars_ptr`.

Ce type, justement, est destiné à être l'équivalent du type C `char*`, autrement dit un pointeur sur une chaîne de caractères. Une chaîne est elle-même représentée par le type tableau non contraint `char_array` du paquetage `Interfaces.C`, tableau dont les éléments sont d'un type `char` correspondant au type `char` du C. Le type `Interfaces.C.int` est quant à lui garanti correspondre au type C `int` sur la plate-forme considérée. D'autres types sont également proposés.

On retrouve dans cette spécification les directives `pragma Import` et `Export` que nous avons déjà rencontrées pour le Fortran, sauf que, cette fois, la convention d'appel à utiliser est celle du langage C. La chaîne donnée en dernier paramètre est, comme précédemment, le nom utilisé pour l'édition des liens. Cette chaîne est facultative, mais il est vivement recommandé de la donner systématiquement. En particulier, n'oubliez pas que le langage Ada est insensible à la casse, contrairement au langage C : par défaut, le nom exporté pour la fonction `Chars_To_Int()` serait donc `"chars_to_int"`, créant ainsi une ambiguïté dans l'écriture.

Voyons maintenant le corps du paquetage :

```
 1 with Ada.Text_IO; use Ada.Text_IO;
 2 package body Hanoi_A is
 3
 4    use C;
 5
 6    function Chars_To_Int(c_str: in CS.chars_ptr) return C.int is
 7       s: constant String := CS.Value(c_str);
 8    begin
 9       Put_Line("Chars_To_Int");
10       return C.int(Integer'Value(s));
11    end Chars_To_Int;
12
13    procedure Hanoi(depuis: in C.int;
14                    via   : in C.int;
15                    vers  : in C.int;
16                    nb    : in C.int) is
17    begin
18       if nb = 1
19       then
20          declare
21             s: constant String := C.int'Image(depuis) &
22                                   " -> " &
23                                   C.int'Image(vers);
24             c_s: CS.chars_ptr := CS.New_String(s);
25          begin
26             Afficher_Depl(c_s);
```

```
27            CS.Free(c_s);
28         end;
29      else
30         Hanoi(depuis, vers, via, nb-1);
31         Hanoi(depuis, via, vers, 1);
32         Hanoi(via, depuis, vers, nb-1);
33      end if;
34   end Hanoi;
35
36 end Hanoi_A;
```

Considérons un instant la fonction `Chars_To_Int()`, dont le rôle est de convertir une chaîne de caractère reçu du C en un entier, comme le ferait `atoi()`.

L'attribut `'Value()` permet de faire précisément cela (ligne 10), mais il attend en paramètre une chaîne Ada : nous devons donc convertir la chaîne C en chaîne Ada. Le paquetage `Interfaces.C.Strings` propose pour cela la fonction `Value()` (ligne 7), dont le paramètre est de type `chars_ptr`.

À l'inverse, la procédure `Hanoi()` va utiliser la fonction `Afficher_Depl()` obtenue du C pour effectuer l'affichage. Les lignes 21 à 23 construisent la chaîne à afficher. Mais c'est une chaîne Ada, qui doit donc être convertie en chaîne C. Là encore, `Interfaces.C.Strings` propose la fonction `New_String()` (ligne 24), qui retourne un `chars_ptr`. Celui-ci peut être transmis au code C.

Notez toutefois que cette fonction crée une nouvelle chaîne en mémoire par une allocation dynamique : lorsque vous n'en avez plus besoin, vous devez donc libérer la mémoire à l'aide de la procédure `Free()` (ligne 27).

Voici enfin le programme principal en C :

```
 1 #include <stdio.h>
 2 extern void adainit();
 3 extern void adafinal();
 4 extern int Chars_To_Int(const char* s);
 5 extern void Hanoi(int depuis, int via, int vers, int nombre);
 6 void Afficher_Depl(const char* s);
 7 int main(int argc, char* argv[])
 8 {
 9    adainit();
10    int nb = Chars_To_Int(argv[1]);
11    Hanoi(1, 2, 3, nb);
12    adafinal();
13    return 0;
14 }
15 void Afficher_Depl(const char* s)
16 {
17    printf("%s\n", s);
18 }
```

Les deux premières déclarations, lignes 2 et 3, sont nécessaires pour que l'initialisation et la terminaison de la partie Ada du programme se passent correctement. `adainit()` doit être appelée avant tout autre appel au code Ada, tandis que `adafinal()` effectue divers nettoyages lorsque celui-ci n'est plus utilisé.

Suivent les déclarations des deux sous-programmes importés depuis le paquetage Ada, le plus naturellement du monde.

Pour compiler, plusieurs étapes sont nécessaires :

```
$ gcc -c hanoi_c.c
$ gnatmake -c hanoi_a.adb
$ gnatbind -n hanoi_a.ali
$ gnatlink hanoi_a.ali hanoi_c.o -o hanoi
```

Le rôle de `gnatbind` est de vérifier la cohérence du programme et de déterminer l'ordre d'initialisation des différents paquetages. Le fichier `.ali` passé en paramètre est issu de la compilation par `gnatmake`. Enfin, l'édition des liens est effectuée par `gnatlink`.

Exécutez le programme en lui passant en paramètre le nombre de disques :

```
$ ./hanoi 3
 1 ->  3
 1 ->  2
 3 ->  2
 1 ->  3
 2 ->  1
 2 ->  3
 1 ->  3
```

Tout se passe donc bien. Le meilleur exemple de connexion entre C et Ada est sans doute la bibliothèque GtkAda [1], qui permet d'utiliser l'intégralité de la célèbre bibliothèque Gtk+ (à la base de Gimp et Gnome) en Ada. GtkAda avait déjà été évoqué dans un article de Simon Descarpentries paru dans le *Linux Magazine 66* de novembre 2004.

Maintenant commencent les difficultés...

C++

Malheureusement, la communication avec le langage C++ n'est pas normalisée. La raison en est principalement que la représentation binaire des symboles, c'est-à-dire l'encodage des noms (*name mangling*), n'est elle-même pas normalisée, contrairement aux langages C et Fortran. Ainsi, chaque compilateur fait plus ou moins ce qu'il veut dans ce domaine, selon la plate-forme.

Bien souvent, lorsqu'un pont est jeté entre une bibliothèque C++ et Ada, cela se fait en « convertissant » le C++ en C. Autrement dit, chaque méthode de classe est remplacée par une fonction qualifiée par `extern "C"`, de façon à se ramener dans la situation de la section précédente. Cette approche présente toutefois un revers de taille : dans l'opération, on perd la sémantique objet. Les relations de dérivation entre les types disparaissent. Il devient alors assez difficile de créer de nouveaux types tout en bénéficiant du polymorphisme.

Le compilateur Gnat offre tout un ensemble de directives `pragma` adaptées au C++. Ainsi un type correspondant à une classe peut être qualifié par `pragma Cpp_Class`, une fonction par `pragma Cpp_Constructor` dans le cas d'un constructeur... Toutefois, d'une part, leur mise en œuvre est assez pénible (il faut utiliser les noms encodés, qui peuvent ressembler

à des choses comme `_ZNK7QString3midEjj` pour la simple méthode `mid()` de la classe `QString` de la bibliothèque Qt), d'autre part, ces directives ne sont pas portables sur d'autres compilateurs.

On peut, malgré tout, se débrouiller un peu, tout en conservant une certaine portabilité. L'idée consiste à créer une sorte de double passerelle, les codes Ada et C++ collaborant au travers d'une interface C. Considérons un exemple simple, deux classes dans un fichier `cpp_classes.h` :

```
#ifndef __CPP_CLASSES_H__
#define __CPP_CLASSES_H__ 1
class Base
{  public:
      Base() ;
      virtual ~Base();
      void method() const ;
      virtual void virtual_method() const ;
} ; // class Base
class Derived: public Base
{  public:
      Derived() ;
      virtual ~Derived();
      virtual void virtual_method() const ;
} ; // class Derived
#endif // __CPP_CLASSES_H__
```

Rien de bien méchant. Seule particularité, la méthode `method()` invoque la méthode virtuelle `virtual_method()`, redéfinie dans la classe dérivée `Derived`. Notre objectif est d'utiliser ces classes dans un programme Ada, tout en conservant leur relation d'héritage et la possibilité d'en dériver de nouveaux types, toujours en Ada.

Pour commencer, la manière de manipuler la mémoire n'est pas forcément identique entre un programme C/C++ et un programme Ada. Principe de base : les instances des classes C++ devront être créées dans un contexte C++. Le recours à la mémoire dynamique est presque obligatoire, le coté Ada ne stockant qu'une adresse mémoire. On va donc se créer un type Ada de base pour intégrer tout cela, dans un paquetage `Cpp_Accessors` :

```
    1 with System;
    2 with Ada.Finalization;
    3 package Cpp_Accessors is
    4    type Cpp_Ptr is new System.Address;
    5    Null_Cpp_Ptr: constant Cpp_Ptr := Cpp_Ptr(System.
Null_Address);
    6    type Ada_Ptr is new System.Address;
    7    Null_Ada_Ptr: constant Ada_Ptr := Ada_Ptr(System.
Null_Address);
    8    type Cpp_Accessor is new Ada.Finalization.Controlled with
private;
    9    not overriding
   10    procedure Create(ca : access Cpp_Accessor;
   11                     cpp: in     Cpp_Ptr);
   12    overriding
   13    procedure Finalize(ca: in out Cpp_Accessor);
   14    not overriding
   15    function Get_Cpp_Ptr(ca: in Cpp_Accessor'Class)
   16       return Cpp_Ptr;
   17 private
   18    type Cpp_Accessor is new Ada.Finalization.Controlled with
   19    record
   20       cpp: Cpp_Ptr := Null_Cpp_Ptr;
   21    end record;
   22 end Cpp_Accessors;
```

On fera ici l'hypothèse que le type `Address` du paquetage standard `System` est équivalent à un pointeur C++ comme `void*`. C'est généralement le cas, mais il paraît qu'il existe des exceptions.

Les deux premières déclarations des types `Cpp_Ptr` et `Ada_Ptr`, représentant respectivement l'adresse d'un objet C++ et l'adresse d'un objet Ada, ne sont là que pour économiser quelques touches et faire en sorte que le compilateur détecte d'éventuelles affectations hasardeuses. Le type intéressant est `Cpp_Accessor` (ligne 8), un type contrôlé afin d'en maîtriser surtout la destruction. Ce type sera chargé de « transporter » l'adresse d'un objet C++ : tous les autres types Ada correspondants à des classes C++ que nous allons déclarer dériveront de `Cpp_Accessor`. La procédure `Finalize()`, invoquée lorsqu'un objet de ce type est détruit, aura pour rôle de détruire l'objet C++ associé.

À ce type, on fait correspondre une classe C++, chargée cette fois de transporter l'adresse d'un objet Ada :

```
    1 #ifndef __ADA_ACCESSOR_H__
    2 #define __ADA_ACCESSOR_H__ 1
    3 class Ada_Accessor
    4 { public:
    5       Ada_Accessor(void* ada_ptr);
    6       virtual ~Ada_Accessor();
    7       void* Get_Ada_Ptr() const;
    8    private:
    9       Ada_Accessor(const Ada_Accessor&);
   10       Ada_Accessor& operator=(const Ada_Accessor&);
   11       void* _ada_ptr;
   12 }; // class Ada_Accessor
   13 extern "C" void Delete_Cpp(Ada_Accessor* aa);
   14 #endif // __ADA_ACCESSOR_H__
```

Remarquez la présence d'un destructeur virtuel (ligne 6) : sa présence est extrêmement importante pour la suite. La fonction `Delete_Cpp()`, ligne 13, contient la simple instruction `delete aa` (après vérification de la non-nullité du pointeur). Qualifiée `extern "C"`, elle pourra donc être invoquée depuis Ada :

```
    1 package body Cpp_Accessors is
    2    procedure Delete_Cpp(w: in Cpp_Ptr) ;
    3    pragma Import(C, Delete_Cpp, "Delete_Cpp") ;
...
   12    overriding
   13    procedure Finalize(ca: in out Cpp_Accessor) is
   14    begin
   15       Delete_Cpp(Get_Cpp_Ptr(ca));
   16    end Finalize;
...
   25 end Cpp_Accessors;
```

Ainsi la destruction d'un objet Ada dérivant de `Cpp_Accessor` entraînera automatiquement la destruction de l'objet C++ associé. Passons maintenant aux types Ada miroirs de nos classes C++ :

```
 1 with Cpp_Accessors;
 2 use  Cpp_Accessors;
 3 package Ada_Classes is
 4    type Base is new Cpp_Accessor with null record;
 5    not overriding
 6    procedure Create(b: access Base);
 7    not overriding
 8    procedure Virtual_Method(b: in Base);
 9    not overriding
10    procedure Method(b: in Base'Class);
11    --
12    type Derived is new Base with null record;
13    overriding
14    procedure Create(b: access Derived);
15    overriding
16    procedure Virtual_Method(b: in Derived);
17 end Ada_Classes;
```

Ces types sont créés à l'image des classes C++, dont elles reprennent les noms, jusqu'aux méthodes. Le type `Base` dérive de `Cpp_Accessor` comme annoncé, le type `Derived` (ligne 12) dérive de `Base` - reflet de la situation en C++. Pour établir le lien entre les deux langages, nous allons devoir créer une paire de classes et quelques fonctions, déclarées dans l'en-tête `binding.h` :

```
 1 #ifndef __BINDING_H__
 2 #define __BINDING_H__ 1
 3 #include "cpp_classes.h"
 4 #include "ada_accessor.h"
 5 class Base_Accessor: public Base, public Ada_Accessor
 6 {
 7    public:
 8       Base_Accessor(void* ada_ptr);
 9       virtual void virtual_method() const;
10 }; // class Base_Accessor
11 extern "C"
12 {
13    Base_Accessor* Base_Base(void* ada_ptr);
14    void Base_Method(Base_Accessor* ba);
15    void Base_Virtual_Method(Base_Accessor* ba);
16    void Dispatch_Base_Virtual_Method(void* ada_ptr);
17 }
18 class Derived_Accessor: public Derived, public Ada_Accessor
19 {
20    public:
21       Derived_Accessor(void* ada_ptr);
22       virtual void virtual_method() const;
23 }; // class Derived_Accessor
24 extern "C"
25 {
26    Derived_Accessor* Derived_Derived(void* ada_ptr);
27    void Derived_Virtual_Method(Derived_Accessor* ba);
28    void Dispatch_Derived_Virtual_Method(void* ada_ptr);
29 }
30 #endif // __BINDING_H__
```

À partir de chacune des classes que nous voudrons utiliser en Ada, on dérive une nouvelle classe fondée également sur `Ada_Accessor`, transporteur d'adresses d'objets Ada vue plus haut (lignes 5 à 10 et 18 à 23). Ainsi, par exemple, `Base_Accessor` dérive simultanément de `Base` et `Ada_Accessor`. Ces nouvelles classes surdéfinissent les méthodes virtuelles pour lesquelles elles vont constituer une interface. Par ailleurs, des fonctions C sont prévues pour accéder aux méthodes des classes (lignes 13 à 15 et 26-27). Les fonctions dont le nom commence par `Dispatch_` (lignes 16 et 28) ne sont en fait que des points d'accès à des fonctions exportées en Ada. Maintenant accrochez-vous, nous allons examiner simultanément le corps du paquetage `Ada_Classes` et l'implémentation associée à l'en-tête précédent (Tab. 1).

Ouf ! Le reste est très similaire. Revenons un instant sur le mécanisme en place et tous ses appels indirects. Supposons l'existence d'une instance Ada du type `Base`. Appelons-la `b_ada`. À celle-ci correspond une instance dynamique de `Base_Accessor`, que nous appellerons `b_cpp` (bien qu'il n'existe pas de variable coté C++). La figure 1 montre le déroulement des appels lorsque le code Ada invoque l'opération `Method()` sur `b_ada` (Tab. 2).

Vérifions tout cela sur un petit exemple. Créons un paquetage `Ext` dérivant un nouveau type à partir de `Base` :

Dérivation d'un type Ada à partir d'un type C++

```
with Ada_Classes; use Ada_Classes;
package Exts is
   type Ext_Base is new Base with null record;
   overriding
   procedure Virtual_Method(e: Ext_Base);
...
end Exts;
```

```
with Ada.Text_IO; use Ada.Text_IO;
package body Exts is
   overriding
procedure Virtual_Method(e: Ext_Base) is
   begin
      Put_Line("(Ada) Ext_Base.Virtual_Method");
   end Virtual_Method;
...
end Exts;
```

Tableau 3

Remarquez qu'il n'y a plus trace de toute la lourde mécanique précédente : tout cela n'est que du pur Ada. Voici le programme d'exemple :

```
 1 with Ada.Text_IO; use Ada.Text_IO;
 2 with Ada_Classes; use Ada_Classes;
 3 with Exts; use Exts;
 4 procedure Test is
 5    p: access Base'Class;
 6 begin
 7    Put_Line("...Pointeur sur Base...");
 8    p := new Base;
 9    p.Create;
10    p.Method;
11    Put_Line("...Pointeur sur Ext_Base...");
12    p := new Ext_Base;
13    p.Create;
14    p.Method;
15 end Test;
```

Établissement de la liaison entre le code Ada et le code C++

ada_classes.adb

```
 1 package body Ada_Classes is
 2    function Base_Base(ap: in Ada_Ptr)
 3       return Cpp_Ptr;
 4    pragma Import(C,
 5                  Base_Base,
 6                  "Base_Base");
 7    procedure Base_Method(cpp: in Cpp_Ptr);
 8    pragma Import(C,
 9                  Base_Method,
10                  "Base_Method");
11    procedure Base_Virtual_Method(cpp: in Cpp_Ptr);
12    pragma Import(C,
13                  Base_Virtual_Method,
14                  "Base_Virtual_Method");
```

binding.cpp

```
 1 #include <iostream>
 2 #include "binding.h"
 3 extern "C"
 4 {
 5    Base_Accessor* Base_Base(void* ada_ptr)
 6    {
 7       return new Base_Accessor(ada_ptr);
 8    }
 9    void Base_Method(Base_Accessor* ba)
10    {
11       if ( ba != 0 )
12          ba->method();
13    }
14    void Base_Virtual_Method(Base_Accessor* ba)
15    {
16       if ( ba != 0 )
17          ba->Base::virtual_method();
18    }
19 }
```

Les fonctions C sont importées depuis le code C++ par le code Ada. La première a pour rôle de créer une instance de `Base_Accessor`, en stockant l'adresse d'un objet Ada correspondant. L'adresse retournée sera stockée du coté Ada.

Les deux autres fonctions ne sont guère que des relais vers les méthodes de la classe à utiliser. Remarquez tout de même le cas de la méthode virtuelle : ce n'est pas celle (surdéfinie) de `Base_Accessor` qui est invoquée, mais celle de sa classe ancêtre `Base`.

```
23    not overriding
24    procedure Create(b: access Base) is
25       cpp_base: constant Cpp_Ptr := Base_Base(Ada_Ptr(b.
all'Address));
26    begin
27       Create(Cpp_Accessor(b.all)'Access, cpp_base);
28    end Create;
```

```
20 Base_Accessor::Base_Accessor(void* ada_ptr):
21    Base(),
22    Ada_Accessor(ada_ptr)
23 {
24 }
```

La première opération primitive du type Ada `Base` est la création d'une instance. Celle-ci est obtenue par la fonction C `Base_Base()`, qui ne fait que créer une instance (dynamique) de `Base_Accessor`. Le pointeur sur l'objet Ada est obtenu à partir du paramètre de la procédure, à l'aide de l'attribut `'Address`, puis passé à la fonction qui le transmet au constructeur.

À l'issu de la ligne (Ada) 25, on a donc une instance de `Base_Accessor` contenant l'adresse d'un objet Ada. Le pointeur retourné par la fonction `Base_Base()` est ensuite communiqué à la procédure `Create()` du type Ada de base `Cpp_Accessor` : à l'issu de la ligne 25, l'instance Ada de `Base` contient donc l'adresse de l'instance C++ de `Base_Accessor`. Chacune des deux instances « connaît » donc son alter ego. Passons sur la procédure Ada `Method()`. Elle ne fait qu'invoquer la fonction C `Base_Method()` en lui transmettant l'adresse préalablement stockée.

```
15    procedure Dispatch_Base_Virtual_Method(b: access Base'Class);
16    pragma Export(C,
17                  Dispatch_Base_Virtual_Method,
18                  "Dispatch_Base_Virtual_Method");
19    procedure Dispatch_Base_Virtual_Method(b: access Base'Class) is
20    begin
21       b.Virtual_Method;
22    end Dispatch_Base_Virtual_Method;
```

```
25 void Base_Accessor::virtual_method() const
26 {
27    std::cout << "(Cpp) Base_Accessor[" << this
28          << "]::virtual_method()\n";
29    Dispatch_Base_Virtual_Method(Get_Ada_Ptr());
30 }
```

Voyons maintenant le cas de la méthode virtuelle. Du coté Ada, on trouve la déclaration et l'implémentation de la procédure `Dispatch_Base_Virtual_Method()`. Son paramètre est un type classe (au sens Ada du terme, revoyez éventuellement l'article consacré à la programmation orientée objet en Ada pour plus de détails) : l'invocation de `Virtual_Method()` ligne 21 est donc un appel polymorphe, le code effectivement exécuté dépendra du type réel du paramètre au moment de l'exécution.

Par ailleurs, le mode de passage `access` est compatible (dans notre cas) avec un pointeur C : vous l'aurez compris, cette procédure sera invoquée du coté C++ avec en paramètre un pointeur sur un objet Ada, précédemment communiqué à `Base_Base()`. Cette procédure, déclarée dans l'en-tête ligne 16, est utilisée par la méthode virtuelle surdéfinie dans `Base_Accessor`. Lorsque cette méthode est invoquée, par exemple par une autre méthode de `Base`, l'appel est transmis au code Ada qui effectue un appel polymorphe (*dispatching*) de l'opération primitive `Virtual_Method()`.

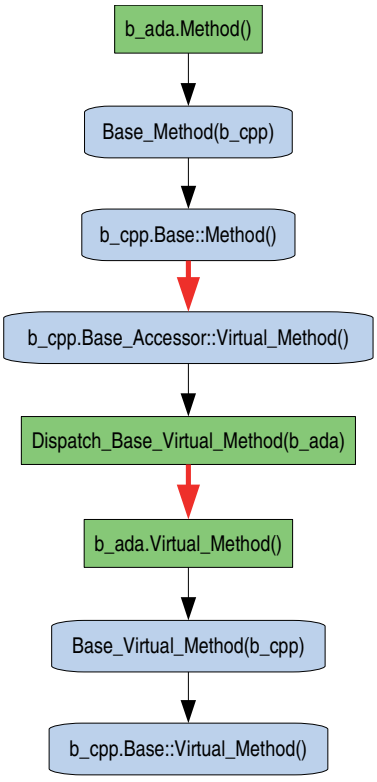
```
29    not overriding
30    procedure Virtual_Method(b: in Base) is
31    begin
32       Base_Virtual_Method(Get_Cpp_Ptr(b));
33    end Virtual_Method;
```

```
14    void Base_Virtual_Method(Base_Accessor* ba)
15    {
16       if ( ba != 0 )
17          ba->Base::virtual_method();
18    }
```

Cette opération primitive, dans son état « par défaut », invoque la fonction `Base_Virtual_Method()`, qui renvoit elle-même à la méthode virtuelle de la classe `Base`, non pas de la classe `Base_Accessor`, comme cela est indiqué explicitement ligne 17.

Tableau 1

Échanges polymorphes entre Ada et C++



Les portions de code en Ada sont sur fond vert, tandis que le code C++ est sur fond bleu. Les flèches rouges signalent les appels de fonction polymorphes.

Pour mémoire, la méthode `method()` de la classe (C++) `Base` invoque la méthode virtuelle `virtual_method()` de cette même classe. Il semble donc logique, voire rassurant, que dans notre exemple le fait d'appeler l'opération `Method()` sur le type Ada `Base` aboutisse finalement à la méthode `virtual_method()` de la classe C++ `Base`. Alors, pourquoi tout ce chemin ?

Rappelez-vous que nous cherchons à conserver la sémantique objet lors du passage en Ada. Cela signifie que si on dérive un type `Ext_Base` à partir de `Base`, et que l'on redéfinisse l'opération `Virtual_Method()`, il serait souhaitable que cela soit l'opération redéfinie qui soit appelée – et ce, que l'appel vienne du code Ada ou du code C++ initial. Le long cheminement mis en place a en quelque sorte pour effet de mettre en adéquation le polymorphisme C++ avec le polymorphisme Ada.

Dans la pratique, cela signifie que si on invoque l'opération `Method()` sur le type dérivé `Ext_Base()`, on aboutira bien dans l'opération `Virtual_Method()` surdéfinie, bien que l'on soit passé par le code C++.

On peut donc bien parler de « double passerelle » : les fonctions `extern "C"` permettent à Ada d'utiliser le code C++, tandis que le C++ s'appuie sur les procédures `Dispatch_*` pour la mise en œuvre du polymorphisme.

Tableau 2

La variable `p` est de type accès sur le type classe `Base` : c'est un peu comme si en C++ on déclarait un pointeur sur la classe `Base`. Cette variable peut donc être initialisée dynamiquement à n'importe quel type dérivant de `Base`.

Pour commencer, on lui affecte un objet de type `Base` (ligne 8), dûment créé, sur lequel on invoque l'opération (non virtuelle) `Method()`. On l'a vu, celle-ci doit normalement aboutir à la méthode virtuelle `virtual_method()` de la classe C++ `Base`.

Puis, on lui affecte un objet de type `Ext_Base`, dérivant de `Base` et redéfinissant la méthode virtuelle en Ada. Voici ce que cela donne :

```
...Pointeur sur Base...
(Cpp) Base[0x8070838]::virtual_method()
...Pointeur sur Ext_Base...
(Ada) Ext_Base.Virtual_Method
```

C'est bien la méthode redéfinie qui est invoquée, sans qu'il soit besoin d'ajouter de nouveau code C++.

Nous pouvons donc dériver de nouveaux types à partir des types importés du C++, tout en concernant le mécanisme du polymorphisme.

Compilation

Pour compiler un tel programme, le plus simple est de passer par un fichier décrivant le projet, une sorte de `Makefile` qui sera soumis à l'utilitaire `gprmake` fourni avec le compilateur Gnat. Voici un modèle, qui va compiler tous les fichiers C++ (d'extensions `.cpp`) et Ada du répertoire courant, puis assembler tous les fichiers objets obtenus lors de l'édition des liens :

```
 1 project Cpp is
 2    for Languages use ("C++", "Ada");
 3    for Main use ("test");
 4    package Naming is
 5       for Specification_Suffix ("C++") use ".h";
 6       for Implementation_Suffix ("C++") use ".cpp";
 7    end Naming;
 8    package Compiler is
 9       for Default_Switches("C++")
10          use("-Wall",
11              "-pedantic");
12       for Default_Switches("Ada")
13          use ("-Wall",
14               "-gnatwa",
15               "-gnatVa",
16               "-s");
17    end Compiler;
18    package Builder is
19    end Builder;
20 end Cpp;
```

Le nom du fichier doit être celui du projet (ici `Cpp`, ligne 1) et avoir l'extension `.gpr`. La compilation se fait simplement avec :

```
$ gprmake -P cpp.gpr
```

Ce qui fournira l'exécutable `test`. Consultez la documentation de Gnat pour plus de détails concernant le format de ces fichiers projets.

Perspectives

La technique évoquée plus haut ne résout pas tous les problèmes. Elle demande naturellement à être approfondie et renforcée, mais elle laisse entrevoir la possibilité de réaliser des passerelles en Ada vers de grandes bibliothèques C++, comme par exemple la bibliothèque graphique Qt4. Voici une bien modeste démonstration de ce qui n'est encore qu'un projet en devenir :

Et le code associé :

```
with Qt.Core.QStrings;
with Qt.Gui.QApplications;
with Qt.Gui.QWidgets.QLabels;
procedure Test_Qt is
   qapp: aliased Qt.Gui.QApplications.QApplication;
   ql  : aliased Qt.Gui.QWidgets.QLabels.QLabel;
   qs  : aliased Qt.Core.QStrings.QString;
begin
   Qt.Gui.QApplications.Create(qapp'Access);
   Qt.Core.QStrings.Create(
      qs'Access,
      "Un <i>binding</i> <b>Ada</b> pour <b>Qt4</b>");
   Qt.Gui.QWidgets.QLabels.Create(ql'Access, qs);
   Qt.Gui.QWidgets.Show(ql);
   Qt.Gui.QApplications.Exec;
end Test_Qt;
```

S'il existe des volontaires pour un coup de main, ce sera avec le plus grand plaisir !

Conclusion

Les possibilités offertes par l'ouverture du langage Ada sur d'autres langages, notamment le langage C, sont sans limites. Il est ainsi possible d'associer la puissance et la robustesse du langage Ada à la richesse et la diversité des bibliothèques existantes, sans devoir systématiquement réinventer la roue. La prochaine fois, nous nous pencherons sur les fonctionnalités offertes par le langage pour l'exécution de tâches en parallèle, ouvrant la voie à la réalisation de programmes multitâches sans avoir recours à des bibliothèques spécialisées.

Yves Bailly,

RÉFÉRENCES

- [1] GtkAda : https://libre2.adacore.com/GtkAda/main.html
- [2] Codes sources de l'article : http://www.kafka-fr.net/articles/ada/sources_13.tar.bz2

Programmation

Le langage Ada – 14 : tâches parallèles

14/07/2008

Posté par La rédaction | Signature : Yves Bailly
Tags : GLMF
0 Commentaire | Ajouter un commentaire

Retrouvez cet article dans : Linux Magazine 87

Après la course à la fréquence, il semble que la mode pour les ordinateurs personnels soit passée à la course au nombre de processeurs. Il devient dès lors très intéressant, pour les développeurs, de se pencher sur les possibilités offertes de mettre en œuvre un véritable parallélisme afin d'exploiter au mieux ces architectures. Comme dans bien d'autres domaines, le langage Ada était, dès sa conception, très en avance sur ce point par rapport aux autres langages étrangement plus populaires.

Nous allons voir dans cet article à quel point il est aisé de faire s'exécuter plusieurs portions de code en parallèle dans un programme Ada. Tandis que les langages plus répandus doivent avoir recours à des bibliothèques spécialisées, telles que pthread, boost-thread, etc., dès 1979 le langage Ada contient en lui-même tous les outils permettant l'exécution de tâches parallèles. Ainsi, le développeur n'a-t-il pas à se préoccuper de la mise en œuvre de ses tâches au niveau du système : ces aspects rebutants sont laissés à la charge du compilateur.

Un programme de fractales

Comme exemple de programme pouvant aisément être parallélisé, considérons le calcul d'une image fractale. Afin d'avoir un maximum de souplesse, ce programme sera décomposé en plusieurs paquetages, la formule de la fractale étant représentée par une interface abstraite. Voici la spécification du type Fractal :

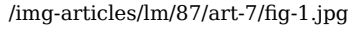
```
 1 with Ada.Numerics.Long_Complex_Types;
   2 use  Ada.Numerics.Long_Complex_Types;
   3 package Fractals is
   4    type Iterations is new Natural;
   5    type Fractal is interface;
   6    procedure Compute(fract   : in out Fractal;
   7                      from    : in     Complex;
   8                      bailout : in     Long_Float;
   9                      max_iter: in     Iterations;
  10                      nb_iters:    out Iterations)
  11       is abstract;
  12    function Is_Inside(fract  : in Fractal;
  13                       point  : in Complex;
  14                       bailout: in Long_Float)
  15       return Boolean
  16       is abstract;
  17 end Fractals;
```

Nous allons calculer des fractales " classiques ", c'est-à-dire dans l'ensemble des nombres complexes. La bibliothèque standard du langage Ada offre justement un type générique de nombres complexes, dans le paquetage Ada.Numerics.Generic_Complex_Types, le paramètre générique étant le type réel de base. La spécification précédente utilise une spécialisation de ce paquetage utilisant le type réel Long_Float (lignes 1 et 2), correspondant (en général) au type double du langage C.
Rappelons que le calcul d'une image fractale, au sens où nous l'entendons ici, consiste pour chaque pixel à évaluer une certaine formule de manière répétitive. Sont considérés dans l'ensemble fractale les pixels pour lesquels la formule n'aboutit pas à un éloignement à l'infini, c'est-à-dire ceux pour lesquels l'évaluation d'une certaine distance ne dépasse pas une certaine valeur (nommée " bailout "), après un certain nombre d'évaluations de la formule principale. Les autres pixels se voient attribués une couleur en fonction du nombre d'itérations nécessaires pour obtenir ce dépassement.
Aussi, commençons-nous par définir un type représentant les itérations, ligne 4, fondamentalement un entier positif ou nul. Ensuite, le type Fractal lui-même est déclaré, sous la forme d'une interface : chaque utilisateur pourra ainsi donner ses propres formules d'évaluation des pixels et des distances. La procédure Compute() effectue le calcul de la valeur fractale d'un point donné (paramètre from), soit jusqu'au dépassement de la distance de référence (paramètre bailout), soit pour un maximum de max_iters itérations de la formule.
Le nombre d'itérations effectuées est retourné dans le paramètre nb_iters. Enfin, la fonction Is_Inside() est chargée d'indiquer si un point se trouve dans l'ensemble fractale, c'est-à-dire d'évaluer la distance voulue et la comparer à la

valeur de référence ~~bailout~~. Ces deux sous-programmes sont déclarés abstraits (~~abstract~~), car ils sont des opérations primitives du type interface (donc abstrait) ~~Fractal~~.
Pour fixer les idées, définissons une dérivation du type Fractal permettant le calcul de la fractale classique de Mandelbrot, définie par la suite complexe suivante :

$$z_0 = c = \text{pixel}\ z_{n+1} = z_n^2 + c$$

Si la norme MathML vous pose des problèmes, voici cette même équation sous forme d'image :


/img-articles/lm/87/art-7/fig-1.jpg

Ce type ~~Mandelbrot~~ sera déclaré dans un paquetage ~~Mandelbrots~~ par :

```
6    type Mandelbrot is new Fractal with null record;
```

Naturellement, les sous-programmes ~~Compute()~~ et ~~Is_Inside()~~ devront être surdéfinis. Voici l'implémentation :

```
 1 package body Mandelbrots is
 2    procedure Compute(fract   : in out Mandelbrot;
 3                      from    : in     Complex;
 4                      bailout : in     Long_Float;
 5                      max_iter: in     Iterations;
 6                      nb_iters:    out Iterations) is
 7       iter_count: Iterations := 0;
 8       zn        : Complex := from;
 9    begin
10       while fract.Is_Inside(zn, bailout) and
11             (iter_count < max_iter)
12       loop
13          zn := zn*zn + from;
14          iter_count := iter_count + 1;
15       end loop;
16       nb_iters := iter_count;
17    end Compute;
18    function Is_Inside(fract  : in Mandelbrot;
19                       point  : in Complex;
20                       bailout: in Long_Float)
21       return Boolean is
22    begin
23       return (abs point) < bailout;
24    end Is_Inside;
25 end Mandelbrots;
```

L'image fractale elle-même sera représentée par un type ~~Fractal_Image~~, déclarée dans un paquetage éponyme :

```
 1 with Ada.Numerics.Long_Complex_Types;
 2 use  Ada.Numerics.Long_Complex_Types;
 3 with Fractals;
 4 use  Fractals;
 5 package Fractal_Images is
 6    type Fractal_Image(width : Positive;
 7                       height: Positive) is
 8       tagged private;
 9    procedure Set_Fractal(fi   : in out Fractal_Image;
10                          fract: access Fractal'Class);
11    procedure Set_Domain(fi    : in out Fractal_Image;
12                         center: in     Complex;
13                         radius: in     Long_Float);
14    procedure Set_Limits(fi       : in out Fractal_Image;
15                         bailout  : in     Long_Float;
16                         max_iters: in     Iterations);
17    procedure Compute_Pixel(fi: in out Fractal_Image;
18                            x : in     Positive;
19                            y : in     Positive);
20 private
21    type Iterations_Array is
22       array(Positive range <>,
23             Positive range <>) of Iterations;
24    type Fractal_Image(width : Positive;
25                       height: Positive) is
26    tagged record
27       fract    : access Fractal'Class;
28       min_z    : Complex;
29       max_z    : Complex;
30       step_z   : Complex;
31       bailout  : Long_Float := 4.0;
32       max_iters: Iterations := 720;
33       iters : Iterations_Array(1..width, 1..height);
34    end record;
35 end Fractal_Images;
```

Le type ~~Fractal_Image~~ est un type paramétré, les paramètres définissant la largeur et la hauteur de l'image (en pixels). Le domaine du plan complexe représenté par l'image est défini par un centre et un rayon, à la manière de l'excellent programme XaoS [2]. Les résultats des calculs pour chaque pixel, c'est-à-dire le nombre d'itérations effectuées de la formule fractale, sont stockés dans le tableau interne ~~iters~~ (ligne 33), de type ~~Iterations_Array~~ (lignes 21-23). Nous ne détaillerons pas l'implémentation, assez simple.
Enfin, voici le programme qui va effectivement calculer notre fractale :

```
 1 with Ada.Text_IO;
 2 use  Ada.Text_IO;
 3 with Ada.Numerics.Long_Complex_Types;
 4 use  Ada.Numerics.Long_Complex_Types;
 5 with Ada.Real_Time;
 6 use  Ada.Real_Time;
 7 with Fractal_Images;
 8 use  Fractal_Images;
 9 with Mandelbrots;
10 use  Mandelbrots;
11 procedure Fract is
12    fract_image: Fractal_Image(1024, 768);
13    mandel     : aliased Mandelbrot;
14    start      : Time;
15    stop       : Time;
16 begin
17    fract_image.Set_Fractal(mandel'Access);
18    fract_image.Set_Domain((-1.18764, -0.30278),
19                           0.00283869);
20    start := Clock;
21    for x in 1..fract_image.width
22    loop
23       for y in 1..fract_image.height
24       loop
25          fract_Image.Compute_Pixel(x, y);
26       end loop;
27    end loop;
28    stop := Clock;
29    Put_Line("Time:" &
30             Duration'Image(To_Duration(stop-start)));
31 end Fract;
```

Ce programme va donc calculer une portion de la fractale de Mandelbrot (ligne 17) dans une image de 1024x768 pixels (ligne 12), centrée au point de coordonnées (-1.18764, -0.30278) sur un rayon de 0.00283869 (lignes 18-19). Le temps de calcul est mesuré au moyen du type ~~Time~~ déclaré dans le paquetage standard ~~Ada.Real_Time~~ (lignes 5-6, 14-15, 20 et 28), puis affiché en secondes (lignes 29-30). Compilez ce programme au moyen de la simple commande :

```
 $ gnatmake fract
```

Si vous souhaitez appliquer certaines optimisations, faites-le de la même manière que pour gcc, par exemple :

```
$ gnatmake -O2 -march=pentium4 fract
```

Vous obtenez un exécutable nommé ~~fract~~. Voici quelques temps de calcul (en secondes), qui nous serviront de références pour la suite :

/img-articles/lm/87/art-7/t1.jpg

Les processeurs Celeron ne possèdent pratiquement rien pour faciliter l'exécution de tâches en parallèle, sans parler du cache interne de faible taille ; l'exécution de tâches parallèles ne devrait donc pas apporter beaucoup d'améliorations. La technologie HyperThreading [4] simule deux processeurs en un seul, offrant la possibilité de traiter beaucoup de données tout en réduisant l'impact des défauts de cache ; notre exemple ne traitant finalement que peu de données, on peut s'attendre à un gain limité en performance.
Enfin, le Pentium D est un processeur à double cœur, c'est-à-dire qu'en réalité deux processeurs sont contenus dans une même puce. Ce n'est donc plus de la simulation. Les temps de calculs devraient donc être sensiblement réduits en utilisant des tâches parallèles (un grand merci à Nicolas Boulay pour m'avoir permis d'abuser de son temps et de son matériel).

Fractale en parallèle

Passons donc aux choses sérieuses. Par nature, le calcul d'une image fractale telle que nous le faisons ici se prête bien au parallélisme : une première tâche peut s'occuper du calcul d'une première moitié de l'image, une seconde tâche de la

seconde moitié. Multipliez les tâches selon le nombre de processeurs dont vous disposez. Modifions notre programme fract ainsi (les premières clauses with et use sont inchangées) :

```
11 procedure Fract is
12    task type Fractal_Task is
13       entry Set_Interval(x_min: in Positive;
14                          x_max: in Positive);
15    end Fractal_Task;
16
17    fract_image: Fractal_Image(1024, 768);
```

Les lignes 12 à 15 déclarent un type tâche Fractal_Task, lequel contient un point d'entrée (entry) nommé Set_Interval(). La spécification d'un point d'entrée ressemble à celle d'une procédure, mais il ne s'agit pas d'un sous-programme.

Cela permet d'établir une communication synchrone avec une tâche en cours d'exécution, au moyen d'un rendez-vous. Ici, ce point d'entrée est destiné à définir un intervalle de l'image fractale à calculer par la tâche, d'où les deux paramètres x_min et x_max.

Remarquez que notre type est déclaré dans la procédure principale du programme : on pourrait également placer cette déclaration dans un paquetage dédié. Voici le code qui sera exécuté par les tâches de ce type :

```
19    task body Fractal_Task is
20       min_x: Positive;
21       max_x: Positive;
22    begin
23       accept Set_Interval(x_min: in Positive;
24                           x_max: in Positive)
25       do
26          min_x := x_min;
27          max_x := x_max;
28       end Set_Interval;
29       for x in min_x..max_x
30       loop
31          for y in 1..fract_image.height
32          loop
33             fract_image.Compute_Pixel(x, y);
34          end loop;
35       end loop;
36    end Fractal_Task;
```

Le corps du type tâche est introduit par task body, de façon similaire au corps d'un paquetage (ligne 19). Le code effectivement exécuté se trouve entre la paire begin/end des lignes 22 et 36. L'espace entre le task body et le begin peut contenir des déclarations de variables, de sous-programmes, etc. qui seront disponibles dans le code qui suit.

Nommons IT une instance de notre type tâche (nous verrons bientôt comment créer des instances de tâches). La première action de notre tâche est d'attendre que le point d'entrée Set_Interval() soit activé par une autre tâche, par la commande accept. Cela signifie que l'exécution de IT est bloquée à la ligne 23, jusqu'à ce que quelqu'un invoque le point d'entrée. Lorsque c'est le cas, l'appelant est lui-même bloqué durant l'exécution du code compris dans la paire do/end, lignes 25 à 28. Cela fait, IT et l'appelant poursuivent leur exécution normale. Dans le cas de IT, cela signifie calculer les valeurs d'itérations pour la zone spécifiée.

Remarquez que le corps de la tâche peut accéder à la variable fract_image, celle-ci ayant été déclarée auparavant, ligne 17. Si nous avions placé la déclaration de la tâche dans la spécification d'un paquetage, puis son corps dans le corps du paquetage, il aurait été nécessaire de communiquer une référence sur fract_image à la tâche.

Poursuivons notre programme :

```
38    mandel: aliased Mandelbrot;
39    start : Time;
40    stop  : Time;
41
42 begin
43    fract_image.Set_Fractal(mandel'Access);
44    fract_image.Set_Domain((-1.18764, -0.30278),
45                           0.00283869);
46    start := Clock;
47    declare
48       task_1: Fractal_Task;
49       task_2: Fractal_Task;
50    begin
51       task_1.Set_Interval(1,
52                           fract_image.width/2);
53       task_2.Set_Interval(fract_image.width/2+1,
54                           fract_image.width);
```

```
55       Put_Line("Computing...");
56    end;
57    stop := Clock;
58    Put_Line("Time:" &
59             Duration'Image(To_Duration(stop-start)));
60 end Fract;
```

Les variables globales sont inchangées, ainsi que l'initialisation de notre fractale. Le calcul est effectué dans le bloc compris entre les lignes 47 et 55.
Les lignes 48 et 49 déclarent tout simplement deux instances de notre type tâche ~~Fractal_Task~~, comme s'il s'agissait de variables classiques. Il est ainsi possible, par exemple, de déclarer un tableau de tâches. Ces deux tâches sont activées dès le début du corps du bloc, c'est-à-dire que l'exécution de leur corps commence dès le begin de la ligne 50. Cette exécution débutant par une instruction ~~accept~~, les deux tâches sont immédiatement bloquées. Leur exécution ne se poursuit qu'après l'invocation du point d'entrée ~~Set_Interval()~~, effectuée lignes 51 et 53. Le calcul simultané des deux moitiés de la fractale est alors entamé.
Si vous exécutez ce programme, vous constaterez que le message affiché ligne 55 apparaît presque immédiatement, puis que le programme semble bloqué avant l'affichage du temps de calcul. En fait, on peut voir notre programme principal comme une tâche principale, les deux tâches ~~task_1~~ et ~~task_2~~ étant des sous-tâches de celle-ci. Pour la tâche principale, l'appel à ~~Set_Interval()~~ ligne 51 est bloquant : elle ne poursuivra son exécution qu'après l'exécution de la petite portion de code du point d'entrée – qui est très rapide. L'appel ligne 53 est traité de la même façon. La suite du bloc consiste à afficher un message, puis à se terminer, ce qui implique la destruction des deux instances de notre type tâche.
L'attente se situe en fait dans cette terminaison : les instances des sous-tâches ne sont détruites à la sortie du bloc qu'une fois qu'elles ont terminé leur exécution, c'est-à-dire que l'exécution a atteint la fin de leur corps pour chacune. Le fait de déclarer ainsi les instances des tâches dans un bloc crée donc de fait une synchronisation entre la tâche principale et les deux sous-tâches.
Essayez de déclasser les déclarations de ~~task_1~~ et ~~task_2~~ avant la ligne 42 : vous verrez le premier message, puis le temps d'exécution s'afficher presque instantanément, puis le programme restera bloqué. L'attente aura en fait lieu sur la ligne 60, au moment où sont détruites les variables de la procédure principale, ainsi que les deux sous-tâches, qui ne sont pas encore terminées.
Pour fixer les idées, voici quelques temps de calcul de ce programme : (voir tableau 2)

/img-articles/lm/87/art-7/t2.jpg

Le Celeron donne un temps légèrement supérieur, ce qui n'est pas très étonnant : pour un système doté d'un unique processeur, l'exécution de tâches en parallèle peut s'avérer légèrement coûteuse du fait des changements de contextes. Toutefois, selon la charge du système et la taille des traitements à effectuer, deux tâches concurrentes peuvent s'avérer légèrement plus efficaces car le programme dans son ensemble peut se voir attribuer un peu plus de temps processeur. Par contre, l'HyperThreading du Pentium4 améliore le résultat d'environ 10%, bien qu'il ne s'agisse toujours que d'un unique processeur en simulant deux. Remarquez qu'un processeur double cœur, bien que légèrement moins rapide en fréquence pure, est plus efficace : la charge de calcul est mieux répartie, offrant un gain de plus de 20%.
Si vous vous demandez pourquoi le gain du double cœur n'avoisine pas les 50%, la raison en est que la zone choisie pour la fractale n'est pas symétrique : la moitié droite demande plus de calculs que la moitié gauche, ce que nous allons vérifier immédiatement (comme exercice simple, modifiez le code pour que la division soit horizontale et non plus verticale, puis constatez l'amélioration des performances).

Affichage : objets protégés

Calculer des fractales c'est bien, les voir c'est mieux. Nous allons pour cela utiliser la bibliothèque ~~Qt~~ version 4, au travers d'une interface nommée ~~" Qt4Ada "~~ Celle-ci, sur laquelle vous trouverez quelques détails supplémentaires plus loin, permet à un programme Ada d'accéder aux facilités de ~~Qt~~.
Notre nouveau programme va produire l'affichage suivant, une fois le calcul terminé :

/img-articles/lm/87/art-7/fig-2.jpg

À mesure de l'avancement des calculs, les pixels d'une image (de type ~~QPixmap~~) seront colorés selon le nombre d'itérations effectuées pour le point correspondant. À chaque fois qu'une colonne est terminée, l'affichage est rafraîchi, afin de visualiser la progression.
On pourrait donc imaginer simplement modifier notre programme ainsi :

```
    1 -- déclarations usuelles
...
   22 procedure Fract is
   23    task type Fractal_Task is
   24       entry Set_Interval(x_min: in Positive;
   25                          x_max: in Positive);
   26    end Fractal_Task;
   27    fract_image: Fractal_Image(1024, 768);
   28    pix        : QPixmap;
   29    label      : QLabel;
   30    black      : QColor;
   31    task body Fractal_Task is
   32       min_x  : Positive;
   33       max_x  : Positive;
   34       iter   : Iterations;
   35       color  : QColor;
   36       painter: QPainter;
   37       ok     : Boolean;
   38    begin
   39       painter.Setup;
   40       color.Setup;
   41       accept Set_Interval(x_min: in Positive;
   42                           x_max: in Positive)
   43       do
   44          min_x := x_min;
   45          max_x := x_max;
   46       end Set_Interval;
   47       for x in min_x..max_x
   48       loop
   49          for y in 1..fract_image.height
   50          loop
   51             fract_image.Compute_Pixel(x, y);
   52             iter := fract_image.Get_Pixel(x, y);
   53             if iter >= 720
   54             then
   55                color.Set_Hsv(0, 255, 0);
   56             else
   57                color.Set_Hsv(Integer(iter mod 360),
   58                              255,
   59                              255);
   60             end if;
   61             painter.Begin_Paint(pix, ok);
   62             painter.Set_Pen(color);
   63             painter.Draw_Point(x-1, y-1);
   64             painter.End_Paint(ok);
   65          end loop;
   66          label.Set_Pixmap(pix);
   67          label.Repaint;
   68       end loop;
   69    end Fractal_Task;
   70    mandel: aliased Mandelbrot;
   71    start : Time;
   72    stop  : Time;
   73    app   : aliased QApplication;
   74    ret   : Integer;
   75 begin
   76    app.Setup;
   77    black.Setup(0,0,0);
   78    pix.Setup(1024, 768);
   79    pix.Fill(black);
   80    label.Setup;
   81    label.Set_Pixmap(pix);
   82    label.Set_Fixed_Size(1024, 768);
   83    label.Show;
   84    Process_Events;
   85    -- la suite comme précédemment
...
  103    ret := Exec;
  104 end Fract;
```

Ligne 28 est déclarée une image de type ~~QPixmap~~, initialisée lignes 78-79. Le type ~~QLabel~~, dont une instance est déclarée ligne 29, permet d'afficher une telle image. Dans le corps de la tâche, chaque point de l'image est coloré selon le nombre d'itérations (lignes 51 à 64), l'affichage étant rafraîchi à la fin de chaque coordonnée x (lignes 66 et 67). Tout cela semble parfait.

Seulement, cela a toutes les chances de ne pas fonctionner. Du fait du multitâche, il n'existe aucune garantie quant à l'ordre dans lequel les instructions sont exécutées. En particulier, cela signifie que les deux tâches peuvent tenter de dessiner en même temps sur l'image, ou bien de l'afficher en même temps, ou pire encore, l'une peut tenter de l'afficher tandis que l'autre est en train de la modifier. Dans le meilleur des cas, vous n'aurez qu'une image partielle (avec des messages du type ~~X Error: BadGC~~ en provenance du serveur X), au pire, le programme plantera complètement.

Dans notre situation, l'affichage est une ressource critique qui supporte assez mal les accès simultanés. Une telle ressource doit être protégée, afin que les accès y soient dûment contrôlés et surtout sérialisés. Ce genre de principe se retrouve, par exemple, dans les systèmes de base de données ou les pilotes de matériels. Finalement, pour certaines opérations, on se retrouve à faire du mono-tâche.

On pourrait obtenir la protection recherchée en créant une nouvelle tâche destinée à l'affichage, qui prendrait la forme d'une boucle infinie dans laquelle la modification de l'image et le rafraîchissement de l'affichage seraient placés à la suite de points d'entrées. Mais cette façon de faire est lourde et laborieuse. Une bien meilleure solution est d'utiliser un objet protégé. Celui-ci est déclaré de la façon suivante:

```
27    protected type Pixmap is
28       procedure Setup;
29       entry Draw_Point(x   : in Integer;
30                        y   : in Integer;
31                        iter: in Iterations);
32       entry Update;
33    private
34       pix     : QPixmap;
35       label   : QLabel;
36       color   : QColor;
37       painter : QPainter;
38       ready   : Boolean := False;
39       drawing : Boolean := False;
40       updating: Boolean := False;
41    end Pixmap;
```

Un objet protégé, introduit par les mots-clefs ~~protected type~~, se compose d'une partie publique et d'une partie privée, comme un paquetage. La partie publique contient des fonctions, procédures et points d'entrées permettant d'accéder aux données de la partie privée. Ces trois types d'accès présentent les caractéristiques suivantes :

- les fonctions ne permettent qu'un accès en lecture seule aux données ; celles-ci n'étant pas modifiables au sein de la fonction, plusieurs tâches peuvent invoquer simultanément une fonction ;
- les procédures autorisent la modification des données ; afin de garantir l'intégrité de celles-ci, si plusieurs tâches invoquent une même procédure, elles sont placées dans une file d'attente et " servies " l'une après l'autre ;
- les points d'entrée (~~entry~~) sont similaires aux procédures, mais, en plus, elles ne peuvent être invoquées que lorsqu'une condition est vérifiée : elles sont gardées.

Les instances de ~~QLabel~~ et de ~~QPixmap~~ se trouvant dans la partie privée du type protégé ~~Pixmap~~ (lignes 34 et 35), elles ne sont pas accessibles depuis l'extérieur de ce type. Voyons l'implémentation (le corps) du type ~~Pixmap~~ :

```
43    protected body Pixmap is
44       procedure Setup is
45       begin
46          color.Setup(0,0,0);
47          pix.Setup(1024, 768);
48          pix.Fill(color);
49          label.Setup;
50          label.Set_Pixmap(pix);
51          label.Set_Fixed_Size(1024, 768);
52          painter.Setup;
53          label.Show;
54          ready := True;
55       end Setup;
```

La procédure ~~Setup()~~ ressemble à n'importe quelle autre procédure. Son rôle est d'initialiser l'image, l'affichage et les outils qui vont nous permettre de les modifier.

```
 56       entry Draw_Point(x   : in Integer;
57                         y   : in Integer;
58                         iter: in Iterations)
59          when ready and
60               (not drawing) and
61               (not updating) is
62          ok: Boolean;
63       begin
64          drawing := True;
65          if iter >= 720
66          then
67             color.Set_Hsv(0, 255, 0);
68          else
69             color.Set_Hsv(Integer(iter mod 360),
70                           255,
71                           255);
72          end if;
73          painter.Begin_Paint(pix, ok);
74          painter.Set_Pen(color);
75          painter.Draw_Point(x-1, y-1);
```

```
76          painter.End_Paint(ok);
77          drawing := False;
78       end Draw_Point;
```

Le point d'entrée ~~Draw_Point()~~ dessine dans l'image aux coordonnées données, dans une couleur dépendant du nombre d'itérations donné. Le gardien de ce point d'entrée figure lignes 59 à 61 : le code qui suit ne pourra être invoqué que si la variable ~~ready~~ vaut ~~True~~, et que la variable ~~drawing~~ vaut ~~False~~, et que la variable ~~updating~~ vaut ~~False~~. En fait, l'accès est un peu surprotégé, mais c'est pour illustrer. Durant l'exécution du code, les accès concurrents sont mis en attente : aucune autre tâche ne peut donc exécuter ce code en même temps.

```
79       entry Update
80          when ready and
81               (not drawing) and
82               (not updating) is
83       begin
84          updating := True;
85          label.Set_Pixmap(pix);
86          label.Repaint;
87          Process_Events;
88          updating := False;
89       end Update;
90    end Pixmap;
```

Le point d'entrée ~~Update()~~ fonctionne de la même manière. La protection des données est effectuée ainsi :

- si une tâche obtient l'accès à ~~Draw_Point()~~, aucune autre ne peut y accéder ; la variable drawing est alors placée à True : toute tâche demandant l'accès à ~~Update()~~ sera mise en attente, du fait du gardien (deuxième condition, ligne 81) ; ainsi on ne risque plus d'afficher l'image pendant qu'on la modifie ;
- de façon symétrique, lorsqu'une tâche obtient l'accès à ~~Update()~~, les accès à ~~Draw_Point()~~ seront mis en attente du fait du gardien ligne 61 : on ne risque plus de modifier l'image pendant qu'on l'affiche.

Cette technique des objets protégés permet une conception de haut niveau, le compilateur se chargeant d'ajouter le code nécessaire aux mises en attente et aux synchronisations. Il suffit alors de créer une instance de notre type protégé et de modifier légèrement le corps de notre tâche de calcul :

```
92    fract_image: Fractal_Image(1024, 768);
93
94    pix: Pixmap;
95
96    task body Fractal_Task is
97       min_x: Positive;
98       max_x: Positive;
99    begin
100      accept Set_Interval(x_min: in Positive;
101                          x_max: in Positive)
102      do
103         min_x := x_min;
104         max_x := x_max;
105      end Set_Interval;
106      for x in min_x..max_x
107      loop
108         for y in 1..fract_image.height
109         loop
110            fract_image.Compute_Pixel(x, y);
111            pix.Draw_Point(x,
112                           y,
113                           fract_image.Get_Pixel(x, y));
114         end loop;
115         pix.Update;
116      end loop;
117   end Fractal_Task;
```

Les appels à l'instance de l'objet protégé (déclarée ligne 94) apparaissent lignes 111-113 et 115. Le premier appel dessine un point, le second rafraîchit l'affichage. La procédure principale doit naturellement débuter par un appel à ~~pix.Setup~~, afin d'initialiser les données de l'objet protégé.

Voici ce que cela donne en cours de calcul :

/img-articles/lm/87/art-7/fig-3.jpg

Remarquez que les deux moitiés ne s'affichent pas à la même vitesse.
C'est parfaitement normal et même voulu : l'image n'est pas symétrique, la partie droite demande beaucoup plus de calculs que la partie gauche.
La seconde tâche progresse donc moins vite que la première, preuve indirecte que l'on a bien deux flots d'instructions

qui s'exécutent.
Voici les temps obtenus : (voir tableau 3)
Catastrophe, les durées explosent ! C'est normal, l'affichage répété d'une image aussi grande est extrêmement coûteux. Il serait donc préférable de limiter les rafraîchissements d'écran. Une solution consiste à utiliser...

/img-articles/lm/87/art-7/t3.jpg

Un chronomètre

L'idée est de ne rafraîchir l'affichage qu'à certains intervalles de temps, par exemple toutes les deux secondes. Pour cela, on va définir une nouvelle tâche chargée de cette action. Sa déclaration est toute simple :

```
  118    task type Timer is
  119       entry Start(tout: in Duration);
  120       entry Stop;
  121    end Timer;
```

Le point d'entrée ~~Start()~~ démarre le chronomètre, ~~Stop()~~ l'arrête. Le type prédéfini Duration représente une durée en secondes. Plus intéressante est l'implémentation de cette tâche :

```
  123    task body Timer is
  124       timeout: Duration;
  125       stopped: Boolean := False;
  126    begin
  127       accept Start(tout: in Duration)
  128       do
  129          timeout := tout;
  130       end Start;
  131       while not stopped
  132       loop
  133          select
  134             delay timeout;
  135             pix.Update;
  136          or
  137             accept Stop
  138             do
  139                stopped := True;
  140             end Stop;
  141          end select;
  142       end loop;
  143    end Timer;
```

L'exécution commence par l'attente du point d'entrée ~~Start()~~, avec en paramètre l'intervalle de temps à appliquer. Puis s'exécute une boucle jusqu'à ce que l'ordre d'arrêt soit donné, par le point d'entrée ~~Stop()~~.

Mais regardez le contenu de la boucle, lignes 133 à 141. Au sein d'une tâche, la structure introduite par select (ligne 133) permet d'effectuer un choix entre diverses portions de code, selon le premier événement qui survient – un peu comme une structure de choix multiple case. Ici, la ligne 135 ne sera exécutée qu'après l'expiration du délai demandé (l'attente ayant lieu ligne 134, par le mot-clef delay suivi d'une durée en secondes), à moins que durant cette attente le point d'entrée Stop() ne soit invoqué. En bouclant sur ce choix, on obtient bien un chronomètre.

À noter que l'attente ligne 134, bien qu'elle rende inactive la tâche, n'est pas bloquante : si ~~Stop()~~ est invoqué avant la fin de l'attente, alors le code correspondant reprend la main.

L'appel à ~~pix.Update~~ doit simplement être retiré du corps de la tâche de calcul. Le corps du programme principal doit enfin être légèrement adapté :

```
...
  149    ret   : Integer;
  150    chrono: Timer;
  151
  152 begin
  153    app.Setup;
...
  159    start := Clock;
  160    declare
  161       task_1: Fractal_Task;
  162       task_2: Fractal_Task;
  163    begin
  164       task_1.Set_Interval(1,
  165                           fract_image.width/2);
  166       task_2.Set_Interval(fract_image.width/2+1,
  167                           fract_image.width);
```

```
168       chrono.Start(2.0);
169       Put_Line(Standard_Error, "Computing...");
170    end;
171    chrono.Stop;
172    pix.Update;
173    stop := Clock;
```

...

Le chronomètre est déclaré ligne 150 et déclenché ligne 168. À l'issu du calcul, il est arrêté (ligne 171), un rafraîchissement de l'affichage étant nécessaire car il est possible (et même fort probable) que l'arrêt survienne alors que la fin de l'image n'a pas encore été affichée.
Voici ce que cela donne en termes de performances : (voir tableau 4, page suivante)
C'est mieux, une amélioration supplémentaire serait de ne dessiner dans l'image qu'aux moments choisis.

/img-articles/lm/87/art-7/t4.jpg

À propos de Qt4Ada

La bibliothèque ~~Qt4Ada~~ a été initiée par votre serviteur, pour les besoins de cet article et d'autres à venir.
Ce n'est encore qu'un projet au début de sa vie, mais au moment où ces lignes sont écrites les six premiers tutoriels de Qt4 ont été ré-implémentés en Ada avec succès. Vous trouverez Qt4Ada avec les codes sources de cet article [1].
Cette bibliothèque s'appuie, d'une part, sur la version 4.1.4 de Qt, d'autre part, sur le compilateur GNAT dans sa dernière mouture [3].
Pour compiler la bibliothèque, il est nécessaire de définir quelques variables d'environnement :

- ~~QTDIR~~ qui doit pointer vers l'endroit où vous avez installé ~~Qt4~~ ;
- ~~GNAT_HOME~~ qui doit pointer vers le répertoire dans lequel vous avez installé le compilateur GNAT ;
- ~~QT4ADA~~ qui doit pointer vers le répertoire contenant la bibliothèque ~~Qt4Ada~~ ;
- ~~PATH~~ doit impérativement commencer par ~~$GNAT_HOME/bin:$QTDIR/bin~~ ;
- ~~LD_LIBRARY_PATH~~ doit impérativement commencer par ~~$GNAT_HOME/lib:$QTDIR/lib:$QT4ADA/lib~~.

Vous pouvez utiliser le fichier ~~set_env.sh~~ à la racine de ~~Qt4Ada~~ pour vous faciliter la vie. Cela fait, un simple ~~make~~ devrait suffire à compiler la bibliothèque.
La compilation des deux derniers exemples de cet article, qui font appels justement à ~~Qt4Ada~~, se fait dans leur répertoire respectif par la commande ~~gnatmake -P fract.gpr~~.
Lorsqu'elle sera suffisamment avancée (à ce propos, toute aide est la bienvenue), cette bibliothèque fera très certainement l'objet d'un article spécifique.

Conclusion

On pourrait poursuivre longuement sur les tâches en Ada. Nous n'avons en réalité fait qu'effleurer la surface des possibilités offertes : points d'entrées conditionnels, contrôle de l'ordonnancement et des priorités entre tâches... Consultez le standard Ada pour plus de détails.
Le prochain article sera consacré aux structures de données génériques faisant désormais partie intégrante de la bibliothèque standard du langage Ada. Nous y retrouverons les traditionnels tableaux dynamiques et listes chaînées, mais également des tables de hachage et les itérateurs afférents. Si vous êtes un adepte de la bibliothèque générique standard du langage C++ (STL), vous vous retrouverez en pays connu, bien que peut-être plus attrayant.

Références :

- [1] Codes sources de l'article : http://kafka.fr.free.fr/articles/ada/sources_14.tar.bz2
- [2] XaoS : http://wmi.math.u-szeged.hu/~kovzol/xaos/doku.php
- [3] Le compilateur GNAT : https://libre2.adacore.com/
- [4] HyperThreading : http://fr.wikipedia.org/wiki/Hyperthreading

~~Retrouvez cet article dans : Linux Magazine 87~~

Le langage Ada – 15 : conteneurs

Yves Bailly

EN DEUX MOTS **Pour Ada comme pour tout langage de programmation, les types fondamentaux deviennent rapidement insuffisants. Des structures plus sophistiquées comme les listes chaînées ou les dictionnaires sont indispensables pour tout développement d'envergure. La norme Ada2005 enrichit la bibliothèque standard de telles structures.**

C'était en effet un manque important du langage Ada95 (c'est-à-dire, le langage Ada tel que défini par la norme de 1995). Chacun réinventait alors dans son coin sa propre roue, ou bien utilisait des ensembles de paquetages tels que Booch [1] ou Simple Components [2]. Situation évidemment loin d'être idéale. Aussi, la dernière version de la norme Ada (2005) inclut-elle désormais un ensemble de structures de données répondant à ces besoins.

Organisation

L'ensemble des conteneurs Ada prend racine dans le paquetage `Ada.Containers`. Celui-ci ne définit rien d'autre qu'une paire de types : `Hash_Type` pour représenter une valeur de hachage (issue d'une fonction de hachage) et `Count_Type` pour représenter le nombre d'éléments dans un conteneur. Ces deux types dépendent de l'implémentation, toutefois `Count_Type` est un type entier modulaire.

On retrouve les grandes familles usuelles des conteneurs, présentées dans le tableau suivant avec leur correspondance en langage C++ (dans le tableau, `AC` représente `Ada.Containers`).

	Ada		C++	
	Type	Paquetage	Classe	En-tête
Liste doublement chaînée	`List`	`AC.Doubly_Linked_Lists`	`std::list`	`<list>`
Vecteur (tableau linéaire)	`Vector`	`AC.Vectors`	`std::vector`	`<vector>`
Ensemble ordonné	`Set`	`AC.Ordered_Sets`	`std::set`	`<set>`
Ensemble non ordonné	`Set`	`AC.Hashed_Sets`	`std::unordered_set`	`<unordered_set>`
Dictionnaire ordonné	`Map`	`AC.Ordered_Maps`	`std::map`	`<map>`
Dictionnaire non ordonné	`Map`	`AC.Hashed_Maps`	`std::unordered_map`	`<unordered_map>`

Dans les structures ordonnées, les éléments sont rangés selon une relation de type « inférieur à » (généralement dans un arbre), tandis que les structures non ordonnées reposent sur une fonction de hachage pour placer les éléments dans une table.

Il convient de noter que les classes `unordered_set` et `unordered_map`, déclarées dans l'espace de nommage `std::tr1`, ne sont pas encore normalisées pour le C++.

Voyons maintenant comment tout cela s'utilise.

Le type Vector

Pour commencer, nous allons nous pencher sur le type `Vector`, qui représente un tableau linéaire dynamique. Les fonctionnalités présentées sont généralement également disponibles pour le type `List` (liste doublement chaînée), à l'exception de tout ce qui concerne l'indexation des éléments, la notion d'indice n'existant pas pour une liste.

La déclaration du paquetage est celle-ci :

```
generic
   type Index_Type is range <>;
   type Element_Type is private;
   with function "=" (Left, Right : Element_Type)
      return Boolean is <>;
package Ada.Containers.Vectors is
...
    type Vector is tagged private;
...
end  Ada.Containers.Vectors;
```

Il s'agit donc d'un paquetage générique, ce qui est bien le moins que l'on puisse attendre d'un tel outil. Les paramètres génériques sont :

- `Index_Type` représente le type utilisé comme indice pour ce vecteur ; cela peut être n'importe quel type entier : comme les tableaux prédéfinis, il est possible d'indexer les éléments par des nombres négatifs, le premier indice n'étant pas forcément 1 (ou 0) ;
- `Element_Type` est le type des éléments à stocker ; ce type ne doit pas être limité, ce qui signifie qu'il doit être possible d'affecter des valeurs entre elles par l'affectation `:=` ;

- Enfin, la fonction "=" est celle qui sera utilisée pour comparer deux éléments, notamment pour la recherche ; la présence de `is <>` à la fin de la déclaration du paramètre signale que celui-ci est optionnel, l'égalité standard étant utilisée par défaut.

Remarquez que le type `Vector` est un type qualifié par `tagged` : on peut donc lui appliquer toutes les techniques de la programmation objet, en particulier le dériver en un nouveau type, l'étendre en lui ajoutant des données ou des opérations.

Voyons immédiatement un exemple simple d'utilisation :

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Containers.Vectors;
procedure Vector_1 is
   package Int_Vects is
      new Ada.Containers.Vectors(Index_Type => Positive,
                                 Element_Type => Positive);
```

Il est tout d'abord nécessaire d'instancier le paquetage générique. Nous créons ici un nouveau paquetage, nommé `Int_Vects`, permettant de déclarer des vecteurs d'entiers positifs indexés par des entiers positifs (pour mémoire, le type `Positive` est défini comme un sous-type de `Integer` limité à l'intervalle `1..Integer'Last`).

```
   vect: Int_Vects.Vector;
begin
   vect.Set_Length(10);
```

Nous pouvons alors déclarer une variable de notre « nouveau » type `Vector`, défini dans notre « nouveau » paquetage `Int_Vects`. L'opération `Set_Length()` définit le nombre d'éléments contenus dans le vecteur, ce nombre pouvant être modifié par la suite.

Le type `Vector` distingue la *taille* de la *capacité*. La taille représente le nombre d'éléments effectivement contenus dans le conteneur. Elle est fixée par `Set_Length()`, donnée par `Length()` et `Is_Empty()` retourne un booléen à `True` ou `False` selon que cette taille est nulle ou non.

Par contre, la capacité représente le nombre d'éléments qui *pourraient* être stockés sans provoquer une nouvelle allocation de mémoire, autrement dit, la taille maximale que l'on peut demander sans que cela donne lieu à une allocation de mémoire supplémentaire (et incidemment, la potentielle duplication des données existantes dans la nouvelle zone mémoire). Elle est fixée par `Reserve_Capacity()` et donnée par `Capacity()`. Si `v` est un vecteur, on a donc toujours `v.Length() <= v.Capacity()`. Notez, par ailleurs, que l'on peut avoir un vecteur vide (`v.Length()` retourne `0`, ou `v.Is_Empty()` retourne `True`) sans que la capacité soit nulle.

```
   for i in 1..10
   loop
      vect.Replace_Element(i, i**2);
   end loop;
```

La boucle précédente remplit le vecteur – ou plutôt, elle remplace les éléments existants par d'autres. En effet, l'appel à `Set_Length()` a implicitement créé 10 éléments non initialisés. Notre tableau est donc déjà plein, sauf que les valeurs nous sont inconnues. Pour placer un élément à un indice `i`, on utilise `Replace_Element()` avec en paramètres l'indice en question et la valeur voulue. Remarquez la précision de la sémantique utilisée : on ne *stocke* pas une valeur dans le vecteur, on *remplace* une valeur existante par une autre.

```
   for i in 1..10
   loop
      Put(Positive'Image(vect.Element(i)));
   end loop;
   New_Line;
end Vector_1;
```

Enfin, on affiche les valeurs contenues dans le vecteur, l'élément situé à un indice donné étant obtenu par l'opération `Element()`.

Opérateurs sur les vecteurs

Le type `Vector` étant simplement privé, il est possible d'affecter entre eux des vecteurs, même de tailles différentes (mais contenant des éléments de même type), par l'opérateur `:=`. Par ailleurs, deux vecteurs peuvent être comparés pour l'égalité au moyen de l'opérateur `=`, ce qui revient à comparer les éléments des vecteurs.

Plus amusant est l'opérateur de concaténation `&`, que nous connaissons déjà sur les tableaux et les chaînes de caractères. Par exemple :

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Containers.Vectors;
procedure Vector_2 is
   package Int_Vects is
      new Ada.Containers.Vectors(Index_Type => Positive,
                                 Element_Type => Positive);
   use type Int_Vects.Vector;
```

Le début est similaire à l'exemple précédent. On a simplement ajouté une clause `use type`, ce qui permet de rendre visibles toutes les opérations associées à un type donné (donc les opérateurs, ce qui nous intéresse ici) sans pour autant rendre visible tout le paquetage dans lequel est déclaré ce type.

```
   procedure Put_Line(v: in Int_Vects.Vector) is
   begin
      for i in 1..v.Length
      loop
         Put(Positive'Image(v.Element(Positive(i))) & " ");
      end loop;
      New_Line;
   end Put_Line;
```

Pour nous simplifier la tâche, on crée une procédure permettant d'afficher le contenu d'un vecteur. Le transtypage (en rouge) est nécessaire, car l'indice de boucle `i` est implicitement de type `Ada.Containers.Count_Type` (le type retourné par l'opération `Length()`), alors que l'opération `Element()` attend un paramètre de type `Positive` (en fait, du type générique formel `Index_Type`).

```
   v1: Int_Vects.Vector;
   v2: Int_Vects.Vector;
begin
   v1 := 1 & 2;
   Put_Line(v1);
```

Premier exemple, la concaténation de deux valeurs produit un vecteur. Naturellement, cela ne fonctionne que si le type des deux opérandes est celui des éléments du vecteur, et que le récepteur du résultat (ici, l'opération d'affectation) est bien du type vecteur ainsi créé.

On pourrait enchaîner ainsi les concaténations, par exemple quelque chose comme :

```
Put_Line(10 & 20 & 30 & 40);
```

mais ce n'est pas recommandé, étant donné que la création puis la destruction des résultats intermédiaires peuvent s'avérer coûteuses.

```
   v2 := v1 & 3;
   Put_Line(v2);
   v1 := 4 & v1;
   Put_Line(v1);
```

Il est également possible d'étendre un vecteur en le concaténant avec une valeur. La première ligne est équivalente à :

```
v2 := v1;
v2.Append(3);
```

tandis que la deuxième (qui modifie `v1` en y plaçant le résultat) est équivalente à :

```
v1.Prepend(4);
```

Les opérations `Append()` (ajouter à la fin) et `Prepend()` (ajouter au début) prennent un paramètre supplémentaire optionnel indiquant le nombre de copies de la valeur ajoutée.

```
   v2 := v2 & v1;
   Put_Line(v2);
end Vector_2;
```

Enfin, il est également possible de concaténer deux vecteurs pour en produire un nouveau, comme le montre le code précédent. Cette commande est équivalente à :

```
v2.Append(v1);
```

Les opérations `Append()` et `Prepend()` peuvent en effet recevoir un vecteur en entrée, plutôt qu'un simple élément.

Ce programme affiche tout simplement :

```
 1  2
 1  2  3
 4  1  2
 1  2  3  4  1  2
```

Il existe également une série d'opérations `Insert()` qui permettent d'insérer une valeur au milieu d'un vecteur, en redimensionnant celui-ci au besoin. Mais notez que ce genre d'action est généralement coûteux, car cela implique un déplacement en mémoire des éléments se trouvant après l'élément inséré. Si vous devez effectuer fréquemment de telles insertions (ou suppressions avec `Delete()`), considérez plutôt le type `List`.

Les curseurs

Nous avons jusqu'ici manipulé les éléments par le biais de leur indice. Mais il existe un autre moyen d'y accéder, disponible sur tous les types de conteneurs : le *curseur*, représenté par le type privé `Cursor`. Il s'agit là de l'équivalent des itérateurs de la bibliothèque standard du langage C++.

Voici un exemple déterminant quelques nombres premiers par l'algorithme du crible d'Ératosthène s'appuyant sur une liste (le programme est prodigieusement inefficace, mais c'est pour l'exemple) :

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Containers.Doubly_Linked_Lists;
procedure Sieve is
   package Int_Lists is
      new Ada.Containers.Doubly_Linked_Lists
        (Element_Type => Positive);
   use type Int_Lists.Cursor;
```

Pour commencer, on réalise l'instanciation du paquetage `Doubly_Linked_Lists`, en ne donnant cette fois que le type des éléments (il n'y a pas d'indice et la fonction de comparaison est celle par défaut). La clause `use type` nous facilitera l'accès au type `Cursor` de ce nouveau paquetage.

```
   procedure Put_Line(l: in Int_Lists.List) is
      c: Int_Lists.Cursor;
   begin
      Put("(");
      c := l.First;
      while c /= Int_Lists.No_Element
       loop
         Put(Positive'Image(Int_Lists.Element(c)));
         c := Int_Lists.Next(c);
      end loop;
      Put_Line(" )");
   end Put_Line;
```

Commençons par une simple procédure affichant la liste. Un curseur est tout d'abord déclaré pour parcourir celle-ci. On lui affecte le résultat de l'opération `First()`, qui donne un curseur « pointant sur » le premier élément de la liste. Puis, on boucle, jusqu'à ce que notre curseur reçoive la valeur `No_Element`.

Cette valeur constante, de type `Cursor`, est déclarée dans tous les paquetages de conteneurs. Elle représente un curseur ne pointant sur rien. Dès qu'un curseur reçoit cette valeur, il ne peut tout simplement plus être utilisé, à moins de lui affecter une nouvelle valeur. On a donc une sémantique assez différente de celle du C++, où on peut avoir un itérateur pointant juste un cran au-delà des bornes d'un conteneur.

L'élément associé au curseur est donné par la fonction `Element()`. Remarquez que comme le type `Cursor` n'est pas qualifié par `tagged`, on ne peut pas utiliser la notation pointée sur une instance de ce type : il est donc nécessaire de préfixer `Element()` par le nom du paquetage concerné.

Le passage à l'élément suivant se fait par la fonction `Next()`, qui retourne un curseur pointant sur l'élément suivant ou `No_Element` si on est déjà à la fin de la liste. Cette fonction existe également sous la forme d'une procédure modifiant son paramètre (passé en mode `in out`).

Voyons maintenant comment nous pouvons manipuler la liste :

```
procedure Prime(l: in out Int_Lists.List;
                p:    out Positive) is
   first: Positive;
   last : Positive;
begin
   first := l.First_Element;
   last := l.Last_Element;
```

Les opérations `First_Element()` et `Last_Element()` retournent respectivement le premier et le dernier élément du conteneur. À ne pas confondre avec `First()` et `Last()`, qui retournent des *curseurs* pointant sur ces éléments.

```
   if first*first > last
   then
      l.Delete_First;
```

`Delete_First()` a simplement pour effet de retirer le premier élément de la liste, qui se trouve ainsi plus courte. L'opération symétrique `Delete_Last()` retire le dernier élément. Ces deux opérations sont également disponibles sur les vecteurs, mais, sur ceux-ci, `Delete_First()` est relativement coûteuse.

```
   else
      declare
         curs: Int_Lists.Cursor;
         next: Int_Lists.Cursor;
         elem: Positive;
      begin
         curs := l.First;
         while curs /= Int_Lists.No_Element
         loop
            next := Int_Lists.Next(curs);
            elem := Int_Lists.Element(curs);
            if (elem mod first) = 0
            then
               l.Delete(curs);
            end if;
            curs := next;
         end loop;
      end;
```

Ce qui précède est le cœur du crible : on recherche tous les multiples du premier élément et on les retire de la liste. On utilise pour cela l'opération `Delete()`, qui prend en paramètre (outre le conteneur concerné) un curseur pointant sur le premier élément à supprimer et éventuellement un nombre d'éléments à supprimer. À l'issu de cet appel, le curseur vaut `No_Element` : dans notre cas, nous devons donc en sauvegarder une copie pour avancer dans la liste.

```
   end if;
   p := first;
end Prime;

lst: Int_Lists.List;
p  : Positive;
begin
   for i in 1..25
   loop
      lst.Append(2*i+1);
   end loop;
   while not lst.Is_Empty
   loop
      Put_Line(lst);
      Prime(lst, p);
      Put_Line(Positive'Image(p));
   end loop;
end Sieve;
```

Le programme se termine en affichant simplement le nombre premier trouvé et l'état de la liste à mesure que le crible avance. On obtient l'affichage suivant :

```
( 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49 51 )
 3
( 5 7 11 13 17 19 23 25 29 31 35 37 41 43 47 49 )
 5
( 7 11 13 17 19 23 29 31 37 41 43 47 49 )
 7
( 11 13 17 19 23 29 31 37 41 43 47 )
 11
( 13 17 19 23 29 31 37 41 43 47 )
 13
( 17 19 23 29 31 37 41 43 47 )
 17
( 19 23 29 31 37 41 43 47 )
 19
( 23 29 31 37 41 43 47 )
 23
( 29 31 37 41 43 47 )
 29
( 31 37 41 43 47 )
 31
( 37 41 43 47 )
 37
( 41 43 47 )
 41
( 43 47 )
 43
( 47 )
 47
```

Ce qui illustre bien les réductions successives de la liste.

Les types ensemblistes

Les types `Hashed_Set` et `Ordered_Set` représentent le concept mathématique d'ensemble, c'est-à-dire une collection de valeurs sans duplication. La différence essentielle est que le premier repose sur une fonction de hachage pour organiser ses éléments, tandis que le second repose sur une relation d'ordre entre les éléments. Par exemple :

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Containers.Hashed_Sets;
with Ada.Containers.Ordered_Sets;
procedure Set_1 is
   function Int_Hash(i: in Integer)
      return Ada.Containers.Hash_Type is
   begin
      return Ada.Containers.Hash_Type(abs i);
   end Int_Hash;

   package Int_Hash_Sets is
      new Ada.Containers.Hashed_Sets
         (Element_Type        => Integer,
         Hash                 => Int_Hash,
         Equivalent_Elements => "=");
   use type Int_Hash_Sets.Cursor;
```

Première instanciation, celle du paquetage `Hashed_Sets` pour contenir des entiers. La

fonction de hachage consiste simplement à renvoyer la valeur absolue d'une valeur (paramètre générique `Hash` du paquetage), tandis que l'équivalence entre deux valeurs est obtenue par la simple égalité (paramètre générique `Equivalent_Elements`).

```
procedure Put_Line(hs: in Int_Hash_Sets.Set) is
   curs: Int_Hash_Sets.Cursor;
begin
...
end Put_Line;
```

La procédure précédente affiche le contenu d'un ensemble. Nous ne la détaillerons pas, elle est tout à fait similaire à celle vue pour les listes.

```
package Int_Ordered_Sets is
   new Ada.Containers.Ordered_Sets
      (Element_Type => Integer);
use type Int_Ordered_Sets.Cursor;

procedure Put_Line(hs: in Int_Ordered_Sets.Set) is
   curs: Int_Ordered_Sets.Cursor;
begin
...
end Put_Line;
```

Ensuite on instancie le paquetage `Ordered_Sets`, toujours pour contenir des entiers. Cette fois, il suffit de donner le type des éléments, la fonction de comparaison étant par défaut l'opérateur "<" prédéfini.

Voyons maintenant ce que cela donne, en insérant les mêmes valeurs dans le même ordre dans chacun des deux conteneurs :

```
   hashed : Int_Hash_Sets.Set;
   ordered: Int_Ordered_Sets.Set;
begin
   hashed.Insert(-1);
   hashed.Insert(-2);
   hashed.Insert(1);
   hashed.Insert(2);
   Put_Line(hashed);
   ordered.Insert(-1);
   ordered.Insert(-2);
   ordered.Insert(1);
   ordered.Insert(2);
   Put_Line(ordered);
end Set_1;
```

L'opération `Insert()` permet d'ajouter un élément à un ensemble. Résultat de l'exécution :

```
[ 1 -1  2 -2 ]
[-2 -1  1  2 ]
```

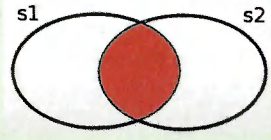
Vous pouvez constater qu'un ensemble issu de `Hashed_Sets` n'est pas ordonné, tandis que `Ordered_Sets` offre des ensembles ordonnés. Savoir lequel des deux types utiliser dépend de vos besoins. D'une manière générale, un ensemble s'appuyant sur une fonction de hachage sera plus efficace pour les opérations d'insertion ou de recherche pour un grand nombre de valeurs, tandis qu'un ensemble basé sur un arbre présente l'intérêt de maintenir l'ordre.

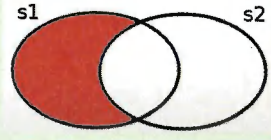
Opérations ensemblistes

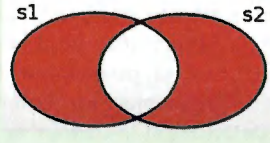
Voyons maintenant quelques opérations communes. Elles sont généralement disponibles sous trois formes : une procédure modifiant son premier paramètre, une fonction de même nom retournant un nouvel ensemble, ou un opérateur surdéfini.

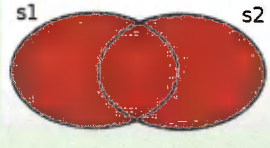
Si `s1` et `s2` sont deux ensembles de même type, les opérations suivantes sont disponibles (données le cas échéant en utilisant la notation traditionnelle et la notation pointée) :

Opération	Sous-programme

intersection
$s1 \cap s2$
 
 	
	Procédure (modifie `s1`) :
	`Intersection(s1, s2);`
	`s1.Intersection(s2);`
	Fonction :
	`s3 := Intersection(s1, s2);`
	`s3 := s1.Intersection(s2);`
	Opérateur :
	`s3 := s1 and s2;`

Différence
$s1 \setminus s2$
 
 	
	Procédure (modifie `s1`) :
	`Difference(s1, s2);`
	`s1.Difference(s2);`
	Fonction :
	`s3 := Difference(s1, s2);`
	`s3 := s1.Difference(s2);`
	Opérateur :
	`s3 := s1 - s2`

Différence symétrique
$(s1 \cup s2) \setminus (s1 \cap s2)$ ou $(s1 \setminus s2) \cup (s2 \setminus s2)$
 
 	
	Procédure (modifie `s1`) :
	`Symmetric_Difference(s1, s2);`
	`s1.Symmetric_Difference(s2);`
	Fonction :
	`s3 := Symmetric_Difference(s1, s2);`
	`s3 := s1.Symmetric_Difference (s2);`
	Opérateur :
	`s3 := s1 xor s2;`

Union
$s1 \cup s2$
 
 	
	Procédure (modifie `s1`) :
	`Union(s1, s2);`
	`s1.Union(s2);`
	Fonction :
	`s3 := Union(s1, s2);`
	`s3 := s1.Union(s2);`
	Opérateur :
	`s3 := s1 or s2;`

Ces opérations ne peuvent s'appliquer qu'entre ensembles de même type et de même nature. Par ailleurs, la norme Ada2005 recommande (mais n'impose pas) que la complexité moyenne des opérations d'insertion, de suppression ou de recherche soit en $O(\log(N))$ pour les dictionnaires de `Hashed_Sets` ou au pire en $O(\log(N)^2)$ pour les dictionnaires de `Ordered_Sets`,

lorsqu'elles impliquent des valeurs. Ces complexités devraient par contre être en O(1) lorsque les opérations sont effectuées par l'intermédiaire de curseurs.

Les dictionnaires

Les dictionnaires permettent de réaliser l'indexation de valeurs d'un type (presque) quelconque par des valeurs d'un autre type (presque) quelconque. Les valeurs indexées sont les *éléments*, les valeurs qui indexent sont les *clefs*. Un exemple typique sont des salaires (valeurs numériques) indexés par des noms (chaînes de caractères). Un dictionnaire est comparable à un tableau, sauf que l'indice n'est pas forcément un nombre entier. Comme les ensembles que nous venons de voir, les types « dictionnaire », nommés `Map`, sont fournis par deux paquetages selon la façon dont sont organisées les clefs :

- `Ada.Containers.Hashed_Maps` pour une organisation utilisant une fonction de hachage sur les valeurs des clefs ;
- `Ada.Containers.Ordered_Maps` pour une organisation fondée sur un ordonnancement des valeurs des clefs.

On pourrait dire qu'un dictionnaire est un ensemble de paires *(clef, valeur)* n'opérant que sur la clef. Par exemple :

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;
with Ada.Strings.Unbounded.Hash;
with Ada.Containers.Hashed_Maps;
with Ada.Containers.Ordered_Maps;
procedure Map_1 is
   package Salaries_Hashed is
      new Ada.Containers.Hashed_Maps(
         Key_Type        => Unbounded_String,
         Element_Type    => Integer,
         Hash            => Ada.Strings.Unbounded.Hash,
         Equivalent_Keys => "=");
   use type Salaries_Hashed.Cursor;
```

On commence par instancier un paquetage `Salaries_Hashed` fournissant un type dictionnaire indexant des entiers `Integer` par des chaînes de caractères `Unbounded_String`. Ce type, déclaré dans `Ada.Strings.Unbounded`, représente une chaîne de caractères dynamique pouvant s'étendre ou se réduire selon les besoins : il est donc beaucoup plus proche du type C++ standard `std::string` que le type usuel `String`, qui n'est en fait qu'un tableau de caractères.

La raison pour laquelle nous utilisons ce type est que dans tous les conteneurs que nous avons rencontrés jusqu'ici, les types stockés doivent être *définis*. Or, le type `String` est déclaré ainsi :

```
type String is array(Positive range <>) of Character;
```

Il s'agit d'un type tableau non contraint : en lui-même, le type `String` ne définit pas les bornes de ce tableau. On dit alors que ce type est *indéfini*. C'est également le cas pour les types « enregistrement » (`record`) présentant un discriminant (revoyez éventuellement le cinquième article de cette série, dans *Linux Magazine* 75).

De tels types indéfinis ne peuvent pas être stockés dans les conteneurs que nous avons vus jusqu'ici (mais nous verrons bientôt comment faire). C'est pourquoi nous utilisons le type `Unbounded_String`, qui est un type défini, le tableau de caractères interne étant manipulé par la mémoire dynamique.

Comme pour le type ensemble déclaré dans `Hashed_Sets`, nous devons fournir une fonction de hachage sur notre type de clefs – c'est-à-dire sur le type `Unbounded_String`. Heureusement, une telle fonction est disponible dans la bibliothèque standard, `Ada.Strings.Unbounded.Hash`. L'équivalence entre clefs est définie par l'égalité entre les chaînes de caractères.

Voyons comment écrire une simple procédure affichant le contenu d'un dictionnaire, tel que défini par notre paquetage :

```
   procedure Put_Line(m: in Salaries_Hashed.Map) is
      crs: Salaries_Hashed.Cursor;
   begin
      Put("{ ");
      crs := m.First;
      while crs /= Salaries_Hashed.No_Element
      loop
         Put("(");
         Put(To_String(Salaries_Hashed.Key(crs)));
         Put(" -> ");
         Put(Integer'Image(Salaries_Hashed.Element(crs)));
         Put(") ");
         Salaries_Hashed.Next(crs);
      end loop;
      Put_Line("}");
   end Put_Line;
```

Comme tous les conteneurs, les dictionnaires possèdent un type `Cursor` permettant de les parcourir. Sauf qu'ici un curseur pointe sur une paire *(clef, valeur)* plutôt que sur une simple valeur. La clef est donnée par la fonction `Key()`, la valeur par la fonction `Element()`. En dehors de cette petite subtilité, la procédure précédente est tout à fait similaire à celles que nous avons déjà vues.

Poursuivons avec un paquetage de dictionnaires ordonnés :

```
   package Salaries_Ordered is
      new Ada.Containers.Ordered_Maps(
         Key_Type     => Unbounded_String,
         Element_Type => Integer);
   use type Salaries_Ordered.Cursor;

   procedure Put_Line(m: in Salaries_Ordered.Map) is
...
   end Put_Line;
```

Il suffit cette fois de donner les types pour les clefs et les valeurs. La procédure d'affichage `Put_Line()` n'est pas détaillée, elle ressemble parfaitement à la précédente.

Utilisons maintenant tout cela :

```
   m_hash: Salaries_Hashed.Map;
   m_ord : Salaries_Ordered.Map;
begin
   m_hash.Insert(To_Unbounded_String("wxyz"), 10);
   m_hash.Insert(To_Unbounded_String("abcd"), 20);
   m_hash.Insert(To_Unbounded_String("mnop"), 30);
   Put_Line(m_hash);
   m_ord.Insert(To_Unbounded_String("wxyz"), 10);
   m_ord.Insert(To_Unbounded_String("abcd"), 20);
   m_ord.Insert(To_Unbounded_String("mnop"), 30);
   Put_Line(m_ord);
end Map_1;
```

L'ajout d'une paire *(clef, valeur)* dans un dictionnaire se fait par l'opération `Insert()`.

La fonction `To_Unbounded_String()` convertit une chaîne de type `String` en une chaîne de type `Unbounded_String`, opération nécessaire étant donné que les écritures littérales comme `"abcd"` sont interprétées comme des chaînes `String`.

Voici enfin ce que ce programme affiche :

```
{ (wxyz -> 10) (abcd -> 20) (mnop -> 30) }
{ (abcd -> 20) (mnop -> 30) (wxyz -> 10) }
```

Comme vous pouvez le constater, on retrouve chez les dictionnaires certaines caractéristiques des ensembles : ceux fondés sur une fonction de hachage stockent leurs éléments dans un ordre quelconque, tandis que ceux fondés sur la comparaison des clefs ordonnent celles-ci.

Sans contraintes

L'exemple précédent a montré que nous ne pouvions stocker dans les conteneurs que des types définis. Cela signifie, par exemple, qu'il est impossible de déclarer une liste de chaînes de caractères de type `String`... du moins avec ce que nous avons vu.

Car c'est en réalité bel et bien possible. Six autres paquetages sont disponibles, dont les noms sont ceux que nous venons de voir préfixés par `Indefinite_`. Ils proposent les mêmes structures de données, mais pouvant contenir des types indéfinis – par exemple, des tableaux non contraints... Voici un exemple d'une liste chaînée de chaînes de caractères, sans passer par le type `Unbounded_String` :

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Containers.Indefinite_Doubly_Linked_Lists;
procedure Indef_List is
   package String_Lists is
      new Ada.Containers.Indefinite_Doubly_Linked_
Lists(
         Element_Type => String);
   use type String_Lists.Cursor;
```

Le paquetage qui nous intéresse ici est `Indefinite_Doubly_Linked_Lists`. Si vous retirez les parties « `Indefinite_` » dans ce qui précède, l'instanciation du paquetage sera refusée par le compilateur, car `Doubly_Linked_Lists` ne peut accepter que des types définis – ce que n'est pas le type `String`.

La procédure d'affichage de la liste n'est pas plus compliquée que précédemment :

```
   procedure Put_Line(l: in String_Lists.List) is
      crs: String_Lists.Cursor;
   begin
      crs := l.First;
      Put("[ ");
      while crs /= String_Lists.No_Element
      loop
         Put("""");
         Put(String_Lists.Element(crs));
         Put(""" ");
         String_Lists.Next(crs);
      end loop;
      Put_Line("]");
   end Put_Line;
   lst: String_Lists.List;
begin
   lst.Append("efgh");
   lst.Append("ijkl");
   lst.Prepend("abcd");
   Put_Line(lst);
end Indef_List;
```

Pour enfoncer le clou, voyons un exemple de dictionnaire plus sophistiqué. Supposons que nous devions stocker les patients d'un médecin, indexés par leur nom. Pour chaque patient, on mémorise quelques caractéristiques, maladies, etc. On pourrait commencer ainsi :

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Calendar; use Ada.Calendar;
with Ada.Containers.Indefinite_Hashed_Maps;
with Ada.Strings.Hash;
procedure Indef_Map is
   type Sexe_Type is (Homme, Femme);
   type Patient_Type(Sexe: Sexe_Type) is
   record
      naissance: Time;
      case Sexe is
         when Homme =>
            hydrocele: Boolean;
         when Femme =>
            nb_enfants: Natural;
      end case;
   end record;
```

Le type `Patient_Type` est paramétré selon le sexe du patient (`Sexe_Type`). Selon ce sexe, des informations différentes sont stockées (si vous ignorez le sens de *hydrocèle*, disons qu'il s'agit d'un petit ennui que les femmes ne peuvent pas connaître). Un tel type paramétré est par essence un type indéfini, donc impossible à stocker dans les conteneurs que nous avons vus au début. Mais il existe un type dictionnaire indéfini :

```
   package Patients_Maps is
      new Ada.Containers.Indefinite_Hashed_Maps(
         Key_Type        => String,
         Element_Type    => Patient_Type,
         Hash            => Ada.Strings.Hash,
         Equivalent_Keys => "=");
```

Nous instancions le paquetage `Indefinite_Hashed_Maps`, en donnant comme type de clef le type (indéfini car non contraint) `String` et comme type d'élément le type (indéfini car paramétré) `Patient_Type`. Malgré cette particularité d'avoir clefs et valeurs de types indéfinis, les dictionnaires fournis par ce paquetage s'utilisent tout à fait naturellement :

```
   patients: Patients_Maps.Map;
begin
   patients.Insert("Untel",
                   (Sexe      => Homme,
```

```
                  naissance => Time_Of(1974, 2, 21),
                  hydrocele => False));
   patients.Insert("Unetelle",
                  (Sexe      => Femme,
                  naissance  => Time_Of(1980, 4, 3),
                  nb_enfants => 0));
end Indef_Map;
```

Il s'agit là d'un moyen puissant pour construire des structures de données hétérogènes, sans passer par une modélisation objet sophistiquée.

Étant donné les avantages manifestes des conteneurs acceptant des types indéfinis, pourquoi ne pas alors les utiliser constamment ? Et à quoi servent finalement les conteneurs « contraints » ?

La réponse tient à la performance. L'utilisation d'un type défini est plus simple, donc plus rapide, qu'un type indéfini. Prenons un exemple trivial consistant à ajouter des éléments (des entiers) à un vecteur, par l'opération `Append()`, sans réservation préalable. À chaque ajout, le conteneur doit s'agrandir pour accueillir le nouvel élément. Voici le programme tout simple :

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Command_Line; use Ada.Command_Line;
```

Le paquetage `Ada.Command_Line` permet de récupérer les paramètres passés par la ligne de commande.

```
with Ada.Calendar; use Ada.Calendar;
```

Dans `Ada.Calendar` est déclaré un type `Time` représentant une date, ainsi que quelques opérations pratiques pour mesurer un intervalle de temps.

```
with Ada.Containers.Vectors;
with Ada.Containers.Indefinite_Vectors;
procedure Bench is
   package Int_Vectors is
      new Ada.Containers.Vectors(
         Index_Type => Positive,
         Element_Type => Integer);
   package Indef_Int_Vectors is
      new Ada.Containers.Indefinite_Vectors(
         Index_Type => Positive,
         Element_Type => Integer);
```

On instancie les deux paquetages de vecteur, le premier prenant des types définis, le second des types indéfinis. Remarquez que ce dernier peut accepter également des types définis.

```
   nb     : Natural := 10000;
   def    : Int_Vectors.Vector;
   indef  : Indef_Int_Vectors.Vector;
   t_start: Time;
   t_end  : Time;
begin
   if Argument_Count > 0
   then
      nb := Natural'Value(Argument(1));
   end if;
   Put_Line(Natural'Image(nb));
   -- vecteur "défini"
   t_start := Clock;
   for i in 1..nb
   loop
      def.Append(i);
   end loop;
   t_end := Clock;
   def.Clear;
   Put_Line(Duration'Image(t_end - t_start));
```

On affiche d'abord le temps d'exécution (en secondes) de la boucle pour le vecteur de types définis, puis...

```
   -- vecteur "indéfini"
   t_start := Clock;
   for i in 1..nb
   loop
      indef.Append(i);
   end loop;
   t_end := Clock;
   indef.Clear;
   Put_Line(Duration'Image(t_end - t_start));
end Bench;
```

...le temps d'exécution de la boucle pour le vecteur de types indéfinis. Compilé sans aucune optimisation, le remplissage par vingt millions de valeurs nécessite (environ) 1.6s pour le premier cas, 3.6s pour le second : le vecteur « indéfini » est donc deux fois plus lent. Compilé en optimisant par `-O2`, on obtient 0.9s pour le premier cas et 3.1s pour le second : l'écart est donc encore plus important.

Moralité : prenez soin d'utiliser la « bonne » version des conteneurs en fonction de vos besoins.

Conclusion

Voilà pour cette présentation générale des structures de données disponibles dans la bibliothèque standard Ada2005. Comme toujours, de nombreux aspects ont été passés sous silence : consultez le standard du langage pour plus d'informations. Mais vous devriez déjà être capable de construire des structures amusantes, comme un dictionnaire de listes d'ensembles...

La prochaine fois, nous reviendrons sur les types fondamentaux que sont les caractères et les chaînes, que nous utilisons largement sans vraiment connaître leurs possibilités.

Yves Bailly,

http://www.kafka-fr.net

RÉFÉRENCES

- [1] Booch Components : http://booch95.sourceforge.net
- [2] Simple Components : http://www.dmitry-kazakov.de/ada/components.htm

Programmation

Le langage Ada – 16 Les caractères des chaînes

17/12/2008

Posté par La rédaction | Signature : Yves Bailly
Tags : GLMF
0 Commentaire | Ajouter un commentaire

Retrouvez cet article dans : Linux Magazine 90

Tout au long des articles précédents, nous n'avons cessé d'utiliser des chaînes de caractères. Celles-ci n'étaient alors connues que comme de simples tableaux. Mais Ada propose bien plus que cela : c'est un ensemble très complet d'outils, qui est disponible et qui n'a que peu de choses à envier à d'autres langages.

Modérons dès maintenant cette introduction : un reproche fréquemment fait à Ada est l'absence notable d'expressions régulières. Il existe toutefois des paquetages comblant ce manque, notamment les paquetages GNAT.Regexp et GNAT.Regpat fournis avec le compilateur GNAT. Ceux-ci ne faisant pas partie intégrante du langage, nous ne les aborderons pas ici.

Les caractères

Le type caractère de base, correspondant au type char du langage C, est Character. Il y a toutefois une différence fondamentale : contrairement à char, le type Character n'est pas un type numérique : c'est un type énuméré, fournissant exactement 256 valeurs, définies très précisément comme étant les 256 caractères du jeu de caractères Latin-1 (la ligne 0 du standard ISO/IEC 10646:2003). N'utilisez donc pas le type Character pour représenter des petites valeurs entières tenant dans un unique octet. Pour cela, utilisez plutôt :

```
type Petit_Entier is new Integer range -128..127;
for Petit_Entier'Size use 8;
```

L'attribut 'Size, qui donne habituellement la taille d'un type en bits (et non en octets), est ici utilisé pour spécifier la taille voulue. Naturellement, vous êtes libre de définir l'intervalle que vous voulez, du moment que cela tient dans 8bits – dans le cas contraire, le compilateur vous signalera une erreur. Par ailleurs, la taille demandée n'est pas obligatoirement un multiple de 8, ce qui peut s'avérer pratique dans des environnements à la quantité de mémoire très limitée.

Pour mémoire, cette écriture signifie " Ada Reference Manual ", annexe A, section 3. Vous trouverez le standard Ada 2005 sur http://www.adaic.com/standards/ada05.html.

Mais revenons à nos caractères. Étant un type énuméré, Character possède tous les attributs de ces types, notamment 'Pos (pour connaître la position d'une valeur) et 'Val (pour obtenir la valeur à une position donnée). Par exemple, Character'Pos('A') retourne l'entier 65, tandis que Character'Val(65) retourne le caractère 'A'. Ada étant un langage à vocation universelle, ce seul type de caractères ne peut suffire. Aussi, en existe-il deux autres, Wide_Character et Wide_Wide_Character. Le premier représente les caractères encodés sur 16bits (UCS-2), le second les caractères sur 32bits (UCS-4). Encore une fois, ces deux types sont considérés comme étant des types énumérés, pas des types numériques : inutile d'essayer de les utiliser dans des calculs. Chaque " grand " type contient les caractères du type " plus petit ". Ainsi, les caractères de Character sont inclus dans ceux de Wide_Character, dont les caractères sont inclus dans ceux de Wide_Wide_Character :

/img-articles/lm/90/art-7/fig-1.jpg

Attention, cela ne signifie pas que les valeurs d'un type sont contenues dans l'ensemble des valeurs d'un autre type : les inclusions sont sémantiques, mais chaque type possède son propre ensemble de valeurs.
La bibliothèque Ada standard propose trois paquetages liés directement aux caractères :

- Ada.Characters.Conversions contient des sous-programmes pour convertir entre les différents types de caractères, ou pour savoir si un caractère d'un " grand " type est également un caractère d'un type plus " petit " ;
- Ada.Characters.Handling offre des fonctions de classifications des caractères de type Character, pour distinguer entre majuscules et minuscules, ponctuation, chiffres, etc. ;
- Ada.Characters.Latin_1, enfin, contient des constantes représentant chacun des caractères du jeu Latin-1.

Consultez le standard Ada pour plus d'informations (ARM-A.31).

Les chaînes de base

En Ada, une chaîne de caractères est un tableau non contraint de caractères. Avec tout ce que cela implique comme contraintes et opérations disponibles sur les tableaux. Pour être précis, les trois types chaîne sont définis ainsi :

```
subtype Positive is Integer range 1..Integer'Last;
type String      is array(Positive range <>) of Character;
type Wide_String      is array(Positive range <>) of Wide_Character;
type Wide_Wide_String is array(Positive range <>) of Wide_Wide_Character;
```

Il existe donc un type de chaîne pour chacun des types de caractères, chaque type étant un tableau dont l'indice est de type Positive, lui-même une restriction du type entier général Integer aux entiers strictement positifs. Par convention (identique au langage Pascal), l'indice du premier caractère d'une chaîne est 1.
Le fait que les chaînes soient des tableaux non contraints imposent des contraintes (sic) parfois gênantes. Par exemple, vous ne pouvez pas déclarer une variable ainsi :

```
chaine: String;
```

Car il est interdit de déclarer une variable d'un type non contraint sans lui donner une restriction ou une valeur initiale. Par exemple :

```
chaine_1: String(1..7);
chaine_2: String := "Monde";
```

Il n'est alors possible d'affecter à ces variables que des chaînes d'exactement la même longueur, ainsi :

```
chaine_1 := "Bonjour";   -- ok
chaine_1 := "Coucou";    -- erreur
```

La deuxième affectation provoquera une erreur de compilation, car la chaîne "Coucou" ne contient que six caractères, alors que la variable chaine_1 a été déclarée comme devant en contenir sept. Cela vaut également pour chaine_2, qui est implicitement une chaîne de cinq caractères. Une façon de contourner le problème est de réaliser l'affectation sur une partie de la chaîne :

```
chaine_1(1..5) := "Hello";
chaine_1(6..7) := (others => ' ');
```

Mais ce n'est guère satisfaisant. Heureusement, la bibliothèque Ada standard nous fournit différents moyens de nous simplifier la vie, comme nous le verrons plus loin.
Dernier mot, la concaténation de chaînes se fait comme pour les tableaux, à l'aide de l'opérateur &.

Manipulations sur les chaînes

Le paquetage Ada.Strings.Fixed contient tout un ensemble de sous-programmes pour manipuler les chaînes de caractères String (les paquetages Ada.Strings.Wide_Fixed et Ada.Strings.Wide_Wide_Fixed offrent les mêmes fonctionnalités sur les autres types de chaînes). Son paquetage parent, Ada.Strings, contient divers types et constantes permettant de spécifier, par exemple, l'alignement à gauche ou à droite, si une recherche doit s'effectuer vers la gauche ou la droite, etc. (voir ARM-A.4 et les sections suivantes).
On trouve ainsi dans Ada.Strings.Fixed, entre autres, une procédure Move que l'on pourrait utiliser ainsi :

```
Move("Coucou", chaine_1);
```

Ceci réalise l'affectation de la chaîne "Coucou" dans chaine_1, ce qui était une erreur en utilisant directement l'opérateur :=. Si la chaîne source est plus courte que la chaîne cible (ce qui est le cas ici), par défaut les caractères sont " calés " à gauche (paramètre optionnel Justify, de type Ada.Strings.Alignment, valeur par défaut Left) et l'espace restant comblé par des espaces (paramètre optionnel Pad de type Character). Dans le cas contraire, où la chaîne cible est plus courte que la chaîne source, un cinquième paramètre optionnel (Drop, de type Ada.Strings.Truncation) précise comment la chaîne source doit être tronquée, voire s'il faut lever l'exception Length_Error (cas par défaut).

Ce paquetage fournit également de nombreux sous-programmes de recherche d'une sous-chaîne dans une chaîne (nommés Index, tous des fonctions) : supprimer une sous-chaîne (Delete), remplacer des caractères (Overwrite), en insérer d'autres (Insert), extraire une partie d'une chaîne (Trim, Tail et Head), etc. Mais un opérateur particulièrement utile, "*", permet de construire une chaîne comme étant la répétition d'un même caractère ou d'une même chaîne, par exemple :

```
with Ada.Strings.Fixed;
use Ada.Strings.Fixed;
...
s1: String := 5*':';
s2: String := 2*"---8<--";
```

À l'issu de ces déclarations, s1 contient ":::::" et s2 contient "---8<-----8<--". On retrouve là une fonctionnalité existant entre autres en Python.

Chaînes dynamiques

Le paquetage Ada.Strings.Unbounded définit le type privé Unbounded_String permettant de manipuler des chaînes de Character dynamiques, c'est-à-dire dont la taille peut varier au cours de l'exécution. Pour les chaînes basées sur les autres types de caractères, il existe les paquetages Ada.Strings.Wide_Unbound et Ada.Strings.Wide_Wide_Unbound, définissant respectivement les types Wide_Unbounded_String et Wide_Wide_Unbounded_String, aux fonctionnalités similaires.
La création d'une chaîne dynamique se fait tout simplement à partir d'une chaîne classique, à l'aide de la fonction To_Unbounded_String :

```
us1: Unbounded_String := To_Unbounded_String("Bonjour");
```

La conversion inverse est obtenue par To_String.
L'intérêt de ces chaînes est que l'on peut leur affecter n'importe quelle autre chaîne, quelle qu'en soit la longueur :

```
us1 := To_Unbounded_String("Une autre chaîne");
```

ou encore, ce qui est plus efficace, car cela évite la création d'un objet temporaire :

```
Set_Unbounded_String(us1, "Une autre chaîne");
```

Le paquetage fournit en plus les mêmes facilités que Ada.Strings.Fixed pour ce qui est des manipulations, transformations, recherches, et ainsi de suite.

Conclusion

Selon la formule consacrée, nous n'avons fait ici que survoler toutes les possibilités des chaînes de caractères en Ada. En particulier, ont été négligés les outils permettant de convertir un ensemble de caractères en un autre (paquetage Ada.Strings.Maps), ou les chaînes à taille variable, mais dont une longueur maximale est fixée, plus efficaces que les chaînes Unbounded_String (paquetage Ada.Strings.Bounded). Plongez-vous dans le standard Ada pour avoir plus de détails.

Retrouvez cet article dans : Linux Magazine 90

Yves Bailly

Le langage Ada : un peu d'assembleur

Le langage Ada est un langage généraliste de haut niveau. Mais l'une de ses vocations premières était l'informatique embarquée, au plus proche du matériel. Comme tous les langages offrant la possibilité de « toucher à l'électronique », Ada permet d'intégrer directement du code assembleur au sein d'un programme plus vaste.

Naturellement, la manière d'intégrer l'assembleur ainsi que le contenu même de ce code dépendent fortement de la plate-forme visée, aussi bien de l'architecture matérielle que du compilateur utilisé. Ce que nous allons voir ici concerne les machines à base de processeur de la famille 80x86, sous environnement GNU/Linux, en utilisant le compilateur Ada GNAT tel qu'il se trouve intégré à la suite GCC.

Certains s'interrogent peut-être sur la pertinence d'utiliser l'assembleur, surtout de nos jours où le moindre morceau de puce peut être manipulé par une API en langage C, généralement fournie par le fabricant. On pourrait alors se contenter d'interfacer l'API C en Ada, comme nous l'avons vu précédemment (voir *Linux Magazine* 85).

Mais, il est des situations où l'assembleur peut s'avérer indispensable. Il serait (théoriquement) parfaitement possible d'écrire le pilote d'une carte vidéo en Ada, bénéficiant ainsi de toute la puissance et robustesse du langage – mais dans des cas de ce genre, les petites latences induites par une couche d'interface avec les fonctions C sont tout simplement inacceptables, car dégradant les performances. Plus généralement, dans une situation où la vitesse d'exécution est primordiale (imaginez la simulation d'un modèle météorologique ou astronomique), le recours à l'assembleur peut permettre d'optimiser très fortement certaines portions du programme bien plus efficacement qu'un compilateur – nous verrons un exemple dans cet article.

Enfin, imaginez vouloir écrire un système d'exploitation en Ada. Le recours à l'assembleur est incontournable à certains endroits clefs – même un système écrit en C comme Linux n'y échappe pas. Et si vous pensez qu'écrire un système en Ada est une idée farfelue et irréaliste pour autre chose qu'une sonde martienne, jetez un œil au projet *Toy Lovelace* **[1]**.

Intégrer l'assembleur

Commençons par l'intégration d'une instruction assembleur au sein d'une procédure Ada (dans un programme `rien.adb`) :

```
with System.Machine_Code;
use System.Machine_Code;
procedure Rien is
begin
   Asm("nop");
end Rien;
```

L'assembleur est intégré au code Ada au moyen de la procédure `Asm()`, déclarée dans le paquetage `System.Machine_Code`. Le premier paramètre de `Asm()` est une chaîne de caractères contenant les instructions à exécuter. Ici, nous invoquons l'instruction `nop`, synonyme de *no operation*, c'est-à-dire très précisément « ne rien faire ».

L'assembleur étant ce qu'il est, il peut s'avérer fort utile de pouvoir consulter le code généré pour l'ensemble du programme, afin de voir précisément comment les instructions que nous écrivons sont intégrées au reste – surtout quand cela ne fonctionne pas. Pour cela, exécutez simplement :

```
$ gcc -c -S -gnatp rien.adb
```

Ceci va générer un fichier `rien.s` contenant le code assembleur équivalent à l'ensemble du programme. Les paramètres passés à `gcc` sont :

- `-c` pour seulement compiler, sans chercher à effectuer d'édition des liens ;
- `-S` pour générer, justement, le fichier `rien.s` ;
- `-gnatp` désactive (presque) tous les mécanismes de contrôle internes au langage Ada (vérification des intervalles de valeurs, ce genre de choses), ce qui allège le code généré.

Sur notre exemple, on obtient ceci :

```
        .file   "rien.adb"
        .text
.globl _ada_rien
        .type   _ada_rien, @function
_ada_rien:
.LFB3:
        pushl   %ebp
.LCFI0:
        movl    %esp, %ebp
.LCFI1:
#APP
        nop
#NO_APP
        popl    %ebp
        ret
.LFE3:
        .size   _ada_rien, .-_ada_rien
...
```

Plus encore un certain nombre de lignes propres à l'utilisation du *runtime* Ada par Gcc. Ce qui nous intéresse commence à la ligne `_ada_rien` : on retrouve le nom de notre procédure principale, préfixé par `_ada_`. La plupart des instructions ne concernent que la gestion de la pile (elles n'apparaîtraient pas si nous avions passé l'option `-fomit-frame-pointer` à `gcc`).

Finalement, notre code assembleur se trouve encadré par les lignes #APP et #NO_APP : il est donc facilement repérable – du moins dans un petit programme.

Mais passons à quelque chose de plus intéressant.

Entrées et sorties

L'instruction assembleur div effectue une division entière, en retournant le quotient et le reste. Normalement, pour obtenir ces deux valeurs, il est nécessaire d'effectuer deux opérations. Écrivons une procédure qui invoque div pour les obtenir directement :

```
1  with Ada.Text_IO;
2  with System.Machine_Code;
3  with Interfaces;
4  procedure Div is
5     use ASCII;
6     use Interfaces;
7     use System.Machine_Code;
8     use Ada.Text_IO;
9     procedure Div(dividende: in     Integer;
10                  diviseur : in     Integer;
11                 quotient :    out Integer;
12                 reste    :    out Integer) is
13     quotient_interne: Integer_32 := 0;
14     reste_interne   : Integer_32 := 0;
15   begin
16     Asm("movl $0, %%edx " & LF & HT &
17         "div %%ebx "      & LF & HT &
18         "movl %%eax, %0 " & LF & HT &
19         "movl %%edx, %1 ",
20         Inputs => (Integer_32'Asm_Input("a",
           Integer_32(dividende)),
21                    Integer_32'Asm_Input("b",
                      Integer_32(diviseur))),
22         Outputs => (Integer_32'Asm_Output("=m",
           quotient_interne),
23                     Integer_32'Asm_Output("=m",
                       reste_interne)),
24         Clobber => ("edx"));
25     quotient := Integer(quotient_interne);
26     reste    := Integer(reste_interne);
27   end Div;
28...
29 -- etc.
```

Voilà qui est sensiblement plus compliqué. Voyons cela étape par étape.

Pour commencer, le type Integer_32 introduit lignes 13 et 14 représente un entier signé codé sur 32 bits. Ce type est déclaré dans le paquetage Interfaces. On s'assure ainsi de parfaitement maîtriser le format des données que l'on va échanger avec l'assembleur. En effet, le type standard Integer est simplement défini comme étant « un type entier ». Sa taille (voire sa représentation) peut varier selon la plate-forme, aussi est-il indispensable d'utiliser un type parfaitement délimité quand on manipule l'assembleur, qui supporte assez mal l'approximation dans ce domaine.

Ensuite, vous constatez que la chaîne de caractères contenant l'assembleur est littéralement construite morceau par morceau (lignes 16 à 19), en insérant ce curieux & LF & HT entre deux lignes. Ces deux symboles, déclarés de type Character dans le paquetage ASCII, représentent les caractères ASCII « retour chariot » (*Line Feed*) et « tabulation horizontale » (*Horizontal Tab*). Ils sont introduits afin d'améliorer la présentation du code assembleur généré. Si nous avions écrit simplement ceci :

```
Asm("movl $0, %%edx " &
    "div %%ebx "      &
    "movl %%eax, %0 " &
    "movl %%edx, %1 ",
```

Voici ce que nous aurions obtenu en examinant le code source assembleur obtenu par gcc -S :

```
...
        movl    32(%esp), %ebx
#APP
        movl $0, %edx div %ebx movl %eax, 4(%esp) movl %edx, (%esp)
#NO_APP
        movl    4(%esp), %eax
...
```

Ce qui est, convenez-en, parfaitement illisible – et généralement refusé par l'assembleur GNU. L'ajout de ces deux caractères donne un résultat bien plus sympathique et passe-partout :

```
...
        movl    32(%esp), %ebx
#APP
        movl $0, %edx
        div %ebx
        movl %eax, 4(%esp)
        movl %edx, (%esp)
#NO_APP
        movl    4(%esp), %eax
...
```

Les entrées

Les entrées sont données par le paramètres Inputs de la procédure Asm() (lignes 20 et 21). Ce paramètre attend une valeur, ou un tableau de valeurs, d'un type retourné par l'attribut 'Asm_Input. Cet attribut, tout comme le contenu du paquetage System.Machine_Code, est dépendant de l'implémentation – ici, nous ne traitons que du compilateur GNAT, basé sur GCC. Répétons-le, un compilateur différent utilisera des moyens (types, procédures et attributs) différents pour obtenir un effet équivalent.

Le premier paramètre de l'attribut est une chaîne de caractères donnant une *contrainte* (*constraint*) sur la valeur que l'on souhaite passer à l'assembleur, laquelle valeur est donnée en deuxième paramètre. On retrouve là les éléments qui sont communiqués à l'instruction asm() de GCC. Pour reprendre un exemple de la documentation de GNAT, l'instruction suivante dans un code en C :

```
asm("fsinx %1 %0" : "=f" (resultat) : "f" (angle));
```

...se présenterait ainsi en Ada (du moins pour le compilateur GNAT) :

```
Asm("fsinx %1 %0",
    Outputs => Float'Asm_Output ("=f", resultat),
    Inputs  => Float'Asm_Input  ("f",  angle));
```

C'est plus long à écrire, mais cela semble plus lisible.

Naturellement, le type de la valeur donnée en deuxième paramètre doit être le même que le type utilisé en préfixe de l'attribut :Ada est un langage au typage strict. C'est pourquoi nous effectuons ici un transtypage, `dividende` et `diviseur` étant de type `Integer`, alors que nous voulons un type `Integer_32`. On obtient ainsi un minimum de vérifications, donc de robustesse.

Nous ne nous étendrons pas sur le contenu de la chaîne de caractères. Il correspond à la syntaxe propre à GCC (voir le chapitre « *Extensions to the C Language Family/Assembler Instructions with C Expression Operands* » et le suivant dans la documentation de GCC). Disons simplement qu'ici nous demandons à ce que la valeur (transtypée) de `dividende` soit placée dans le registre `EAX`, tandis que la valeur (transtypée) de `diviseur` est placée dans le registre `EBX`. Le compilateur se chargera lui-même de ces actions.

Les sorties

Vous l'aurez sans doute deviné, les sorties sont données par le paramètre `Outputs` de la procédure `Asm()`, en utilisant l'attribut `'Asm_Ouput` (lignes 22 et 23). Le principe général est le même que pour les entrées, sauf que le deuxième paramètre de l'attribut doit être une variable et non une simple valeur. C'est pourquoi il a été nécessaire de déclarer deux variables temporaires, `quotient_interne` et `reste_interne`, du type `Integer_32`.

Pour information, la contrainte `"=m"` indique que l'on désigne une sortie (le signe =) qui sera référencée par une adresse mémoire (la lettre `m`). En pratique, cela signifie que les écritures `%0` et `%1` dans le code assembleur seront remplacées par les adresses effectives des variables `quotient_interne` et `reste_interne`.

Intégrer gentiment

Enfin, notre code assembleur commence par placer la valeur 0 dans le registre `EDX` (ligne 16), car l'instruction `div` (dans notre contexte) suppose que la valeur à diviser est contenue dans la paire `EDX:EAX`. De plus, `EDX` est également utilisé pour contenir le reste de la division. Seulement, le registre `EDX` n'est référencé ni dans les paramètres d'entrées, ni dans les paramètres de sortie. Le compilateur ne peut donc pas « deviner » que le registre `EDX` est utilisé et modifié par notre code assembleur, car il n'analyse pas celui-ci. Cela n'a pas grande conséquence ici, mais cela pourrait être catastrophique dans un programme plus vaste : le compilateur pourrait parfaitement utiliser ce registre `EDX` pour ses besoins propres, notre code perturbant alors le déroulement du programme.

Aussi est-il généralement nécessaire de passer un quatrième paramètre à la procédure `Asm()`, nommé `Clobber` (ligne 24). Celui-ci attend une chaîne de caractères contenant les noms des registres utilisés par le code assembleur, séparés par une virgule. Par exemple, `"eax, ebx, ecx"`. Si nécessaire, le compilateur s'adaptera et le programme pourra se dérouler normalement.

Amélioration... ou pas

Généralement, les programmeurs qui utilisent l'assembleur ont le souci constant de l'optimisation. Le code de l'exemple précédent pourrait être rédigé ainsi :

```
Asm("cltd "       & LF & HT &
    "div %%ebx ",
    Inputs => (Integer_32'Asm_Input("a", Integer_32(dividende)),
               Integer_32'Asm_Input("b", Integer_32(diviseur))),
    Outputs => (Integer_32'Asm_Output("=a", quotient_interne),
                Integer_32'Asm_Output("=d", reste_interne)));
```

C'est-à-dire qu'on réduit au minimum les instructions insérées. Le changement essentiel se situe sur les paramètres de sortie : plutôt que d'indiquer une adresse mémoire, on nomme explicitement les registres remplis par l'instruction `div`. Par ailleurs, le paramètre `Clobber` a disparu, devenu inutile (car `EDX` est mentionné dans la deuxième contrainte).

Le tableau suivant compare la partie « intéressante » de l'assembleur généré, à gauche la première version, à droite la version « améliorée » :

```
   movl   $0, 4(%esp)
   movl   $0, (%esp)
   movl   28(%esp), %eax
   movl   32(%esp), %ebx
#APP
   movl $0, %edx
   div %ebx
   movl %eax, 4(%esp)
   movl %edx, (%esp)
#NO_APP
```

```
   movl   $0, 8(%esp)
   movl   $0, 12(%esp)
   movl   28(%esp), %eax
   movl   32(%esp), %ebx
#APP
   cltd
   div %ebx
#NO_APP
   movl   %eax, 8(%esp)
   movl   %edx, 12(%esp)
```

Mis à part l'utilisation de `cltd` pour mettre à 0 le registre `EDX`, ces deux codes sont en fait équivalents. Peut-être peut-on considérer qu'une version est plus « parlante » que l'autre, mais l'autre est moins « dangereuse » que l'une...

Calculs en virgule flottante

Si vous pratiquez le développement d'applications graphiques en 3D, il est probable que vous ayez fréquemment à calculer le sinus et le cosinus d'un angle donné. La plupart du temps, les deux sont d'ailleurs nécessaires dès que l'on manipule quelque chose ressemblant de près ou de loin à un cercle. Or, il existe une instruction assembleur qui permet justement de calculer ces deux valeurs en une seule fois : `fsincos`. Voilà qui semble un bon candidat si on se trouve dans un objectif d'optimisation.

Réalisons donc un petit programme pour vérifier que l'utilisation de l'assembleur apporte effectivement un gain de temps :

```
with Ada.Text_IO;           use Ada.Text_IO;
with Ada.Real_Time;         use Ada.Real_Time ;
with Ada.Numerics;          use Ada.Numerics;
with Interfaces;            use Interfaces;
with System.Machine_Code;   use System.Machine_Code;
with Ada.Numerics.Long_Elementary_Functions;
use Ada.Numerics.Long_Elementary_Functions;
procedure Sin_Cos is
   procedure Sin_Cos(angle: in     Long_Float;
                     sin_a:    out Long_Float;
                     cos_a:    out Long_Float) is
   begin
      sin_a := Sin(angle);
      cos_a := Cos(angle);
   end Sin_Cos;
   pragma Inline(Sin_Cos);
```

Les fonctions sinus et cosinus pour les flottants en double précisions (`Long_Float` en Ada) sont déclarées dans le paquetage `Ada.Numerics.Long_Elementary_Functions`, aux côtés de nombreuses autres fonctions élémentaires. Pour commencer, on crée une procédure classique réalisant ce que nous voulons, par des moyens « normaux ». Remarquez la directive `Inline` qui suit la procédure, elle informe le compilateur d'étendre le code correspondant « en ligne » afin d'économiser un appel de fonction.

Voyons maintenant la version assembleur :

```
procedure Sin_Cos_Asm(angle: in     Long_Float;
                      sin_a:    out Long_Float;
                      cos_a:    out Long_Float) is
begin
   Asm("fsincos",
     Inputs  => Long_Float'Asm_Input("0", angle),
     Outputs => (Long_Float'Asm_Output("=t", cos_a),
                 Long_Float'Asm_Output("=u", sin_a))
      );
end Sin_Cos_Asm;
pragma Inline(Sin_Cos_Asm);
```

Non, pas d'erreur : le code assembleur se résume au seul appel de `fsincos`. En fait, tout le travail pénible est effectué par les déclaration des entrées/sorties. L'instruction opère sur la valeur sur la pile du co-processeur (`ST0`) en la dépilant, puis empile le sinus *puis* le cosinus – on récupère donc les résultats dans `ST1` et `ST0`, respectivement. Ces résultats sont obtenus par les contraintes `"u"` et `"t"` (respectivement), comme toujours précédées sur signe `"="` signalant qu'il s'agit de valeurs de retour.

Mais comme `fsincos` attend également une valeur en `ST0`, lequel registre est également utilisé pour le résultat, on passe la contrainte `"0"` pour la déclaration de la valeur d'entrée, signalant ainsi qu'elle doit être placée dans le même emplacement que la première valeur de retour.

De fait, toutes les manipulations nécessaires de la pile du co-processeur sont automatiquement assurées par le compilateur, ce que l'on peut vérifier en jetant un œil à l'assembleur généré :

```
$ gcc -c -S -gnatp sin_cos.adb && cat sin_cos.s
...
sin_cos__sin_cos_asm.394:
.LFB5:
        pushl   %ebp
.LCFI12:
        movl    %esp, %ebp
.LCFI13:
        subl    $24, %esp
.LCFI14:
        movl    8(%ebp), %edx
        movl    12(%ebp), %eax
        movl    %eax, -24(%ebp)
        movl    16(%ebp), %eax
        movl    %eax, -20(%ebp)
        fldl    -24(%ebp)
#APP
        fsincos
#NO_APP
        fstpl   -8(%ebp)
        fstpl   -16(%ebp)
        fldl    -16(%ebp)
        fstpl   (%edx)
        fldl    -8(%ebp)
        fstpl   8(%edx)
        movl    %edx, %eax
        leave
        ret     $4
...
```

Naturellement, il convient tout de même de vérifier que l'on ne s'est pas trompé quelque part, par exemple en intervertissant les résultats.

Vous aurez sans doute remarqué que nous avons utilisé directement le type `Long_Float`. Cela fonctionne, car, dans notre situation précise (compilateur GNAT, machine 80x86), ce type correspond exactement aux nombres à virgule flottante sur 64 bits du processeur. Si toutefois vous vous placez sur une architecture différente, avec un compilateur différent, il peut s'avérer nécessaire d'effectuer un transtypage comme nous l'avons fait précédemment. Pour information, le paquetage Interfaces qui accompagne le compilateur GNAT comporte les types `IEEE_Float_32` et `IEEE_Float_64` pour les nombres à virgule flottante sur 32 et 64 bits respectant la norme IEEE.

Côté performances, le calcul de dix millions de sinus/cosinus par la méthode classique prend environ 5.86 secondes sur la machine de votre serviteur, tandis qu'en utilisant l'assembleur cela prend... moins d'une seconde.

Passer des adresses mémoires

Pour finir, voyons comment nous pouvons passer l'adresse de blocs de données depuis l'Ada vers l'assembleur. L'exemple consiste à reproduire une fonctionnalité de la procédure `Put_Line()`, permettant d'afficher une chaîne de caractères. L'affichage sera en fait réalisé par la fonction système `write()`. Enfin, nous pourrons saluer le monde comme il se doit.

Une adresse mémoire est représentée en Ada par le type `Address`, déclaré dans le paquetage `System`, l'adresse d'un objet étant fournie par l'attribut `'Address`. Contrairement au langage C, il ne s'agit pas (forcément) d'un type équivalent à un entier. C'est généralement le cas, en particulier sur plate-formes x86, mais pas forcément. Pour un maximum de portabilité, il est donc nécessaire de convertir une valeur de type `Address` en une valeur d'un type entier reconnaissable directement par l'assembleur.

Le paquetage `System.Storage_Elements` contient justement une fonction `To_Integer()` effectuant cela, retournant une valeur de type `Integer_Address` (déclaré dans `System.Storage_Elements`). Comme pour le type standard `Integer`, la taille de ce type n'est pas réellement connue, car dépendante de la plate-forme, mais, au moins, nous savons qu'il s'agit d'un entier que nous pourrons transtyper vers, par exemple, `Integer_32`.

Le programme commence donc par définir une fonction `To_Int_32()` recevant une adresse mémoire et retournant un `Integer_32` :

```
with System.Machine_Code;
with Interfaces;
with System.Storage_Elements;
procedure Put_Line is
   use ASCII;
   use Interfaces;
   use System.Machine_Code;
   function To_Int_32(addr: in System.Address)
      return Unsigned_32 is
   begin
      return Unsigned_32(System.Storage_
Elements.To_Integer(addr));
   end To_Int_32;
```

Remarquez que contrairement à ce que nous faisons d'habitude, le paquetage `Ada.Text_IO` n'est pas référencé. Voici notre version de `Put_Line()` :

```
   procedure Put_Line(s: in String) is
      new_s: constant String := s & LF;
   begin
      Asm("movl  $1,%%ebx " & LF & HT &
          "movl  $4,%%eax " & LF & HT &
          "int   $0x80 ",
          Inputs => (Unsigned_32'Asm_Input("d",
new_s'Length),
                     Unsigned_32'Asm_Input("c",
To_Int_32(new_s(1)'Address))),
          Clobber => ("eax, ebx")
         );
   end Put_Line;
   pragma Inline(Put_Line);
```

Il n'y a en fait rien de bien particulier par rapport à ce que nous avons déjà vu. La procédure commence par créer une nouvelle chaîne à partir de la première, en lui ajoutant le caractère `LF` (pour le retour à la ligne). Les entrées du code assembleur consistent, dans l'ordre :

- à placer la longueur de la chaîne dans le registre `EDX` ;
- à placer l'adresse du premier caractère de la chaîne dans le registre `ECX`.

C'est là qu'intervient notre fonction `To_Int_32`. La seule petite subtilité consiste à prendre l'adresse du *premier caractère* de la chaîne (par `s(1)'Address`), plutôt que l'adresse de la chaîne elle-même (par `s'Address`). En effet, en Ada, une chaîne est un tableau de caractères, comme en C, mais contrairement au C, un tableau n'est pas qu'une adresse mémoire. Il est donc possible qu'en mémoire les données d'un tableau soient précédées d'informations « administratives », comme la taille du tableau, ses bornes, etc. Prendre directement l'adresse de la chaîne pourrait donc aboutir à l'affichage d'octets indésirables. On évite cela en prenant explicitement l'adresse du premier caractère. Naturellement, si la chaîne est vide, invoquer `s(1)` provoquera une exception : un code véritable devrait donc s'assurer qu'il y a au moins un caractère à adresser.

Enfin, on place dans `EBX` l'identifiant du descripteur de fichier voulu (ici 1, correspondant à la sortie standard `stdout`), puis dans `EAX` l'identifiant de la fonction système voulue (ici 4, la fonction `write()`). La dernière instruction déclenche l'interruption 0x80, utilisée sous Linux pour invoquer les fonctions système du noyau.

On termine le programme par le message consacré :

```
begin
   Put_Line("Bonjour, Monde !");
end Put_Line;
```

Résultat :

```
$ gnatmake put_line.adb && ./put_line
Bonjour, Monde !
$
```

Conclusion

Voilà pour cette petite présentation de l'utilisation de l'assembleur en Ada, du moins dans le cadre du compilateur GNAT utilisé sur une architecture 80x86. Ceci montre bien qu'Ada est un langage véritablement généraliste pouvant être utilisé dans toutes les situations, même les plus exigeantes.

Yves Bailly,

http://www.kafka-fr.net

RÉFÉRENCES

- **[1]** Toy Lovelace : **http://ipnnarval.in2p3.fr/~xavier/**
- **[2]** Codes sources de l'article : **http://www.kafka-fr.net/articles/presentation-ada05_17-sources.tar.bz2**