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,