Ecole d'ingénieurs du Canton de Vaud

Les informations de ce document servent de complément au livre:

Programmation séquentielle avec Ada 95

pour les étudiants des classes *Informatique Logiciel* de première année; il ne présente donc pas un structure propre rigoureuse!

Molliet J.-P.

ût 2003

Hes·so
Haute Ecole Spécialisée de Suisse occidentale

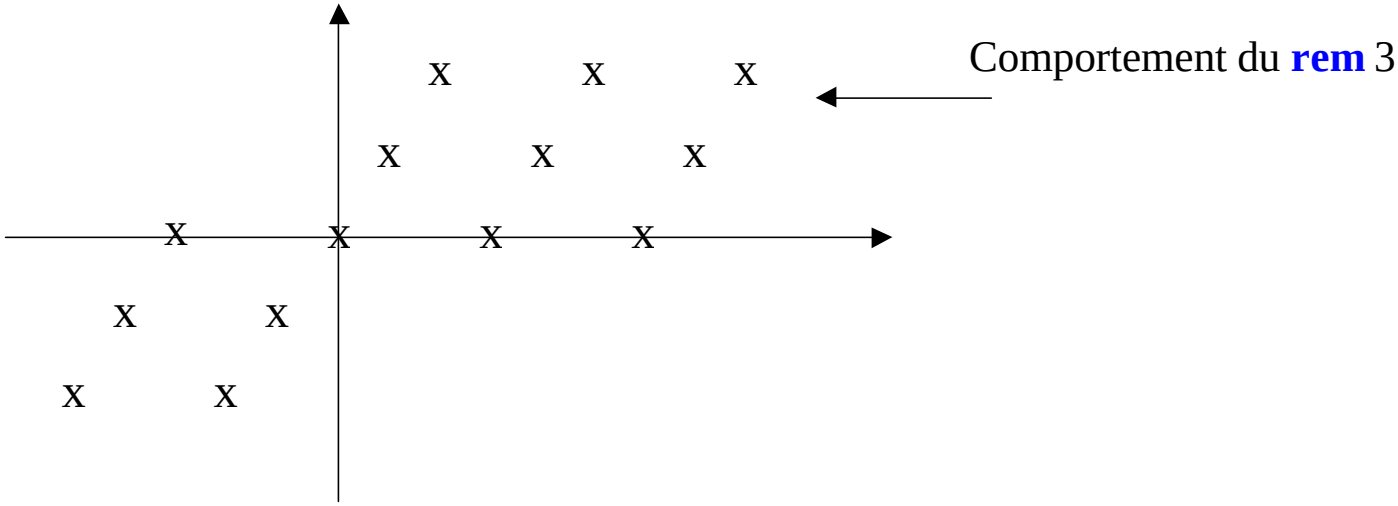
Table des matières

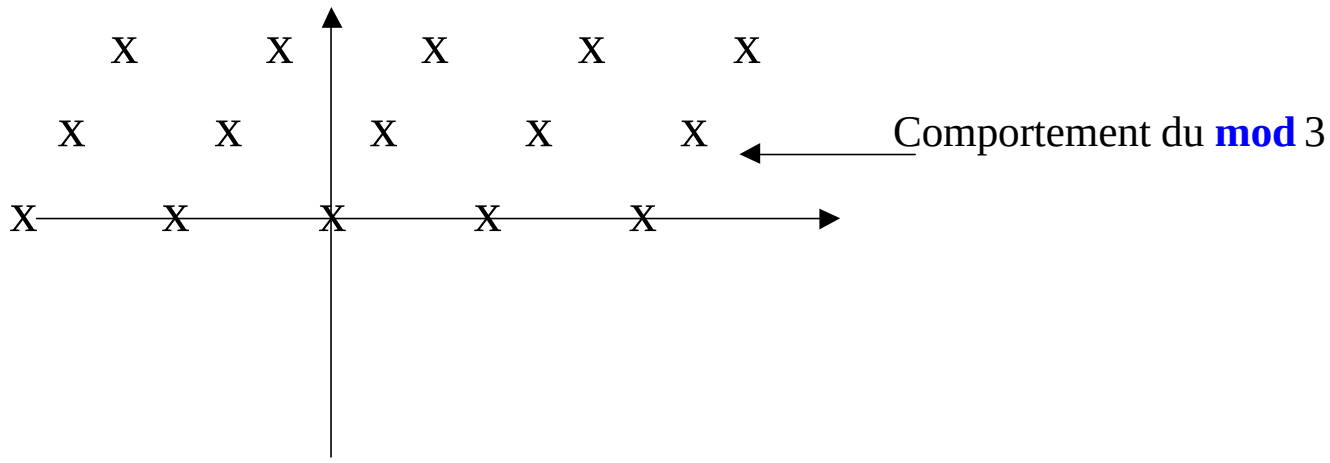
Opérateurs rem et mod

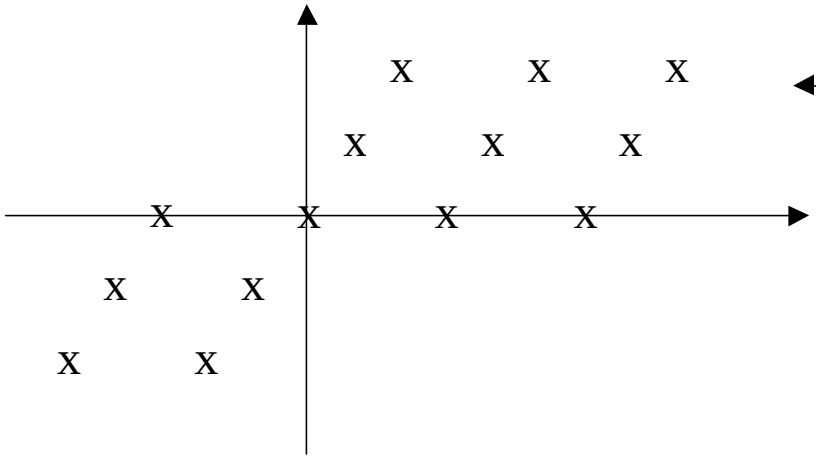
Vous avez souvent quelques difficultés à différencier le comportement de l'opérateur **rem** de l'opérateur **mod**. Plutôt que de longues explications, voici une petite table qui devrait vous aider à le comprendre!

I	I mod 3	I mod -3	I rem 3	I rem -3
-4	2	-1	-1	-1
-3	0	0	0	0
-2	1	-2	-2	-2
-1	2	-1	-1	-1
0	0	0	0	0
1	1	-2	1	1
2	2	-1	2	2
3	0	0	0	0

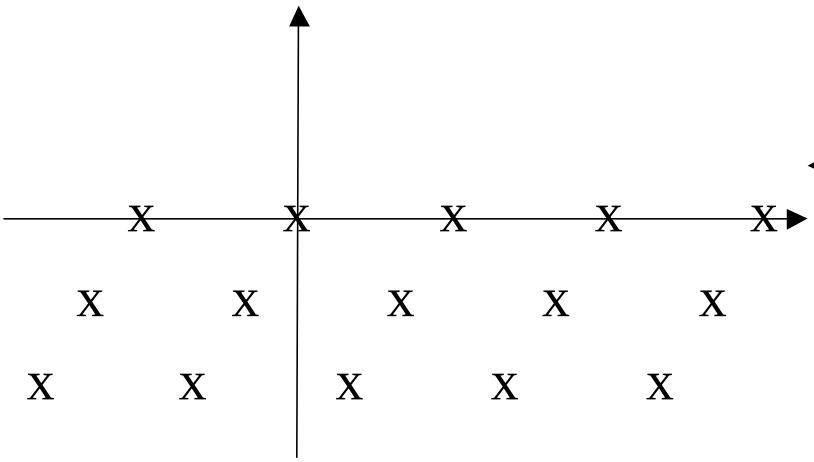
Graphiquement, c'est souvent plus parlant:







Comportement du **rem** -3



Comportement du **mod** -3

Les effets de bords

Pourquoi faut-il les éviter?

Par définition une fonction livre un résultat par l'intermédiaire de son nom (son appel). Nous avons des habitudes en arithmétique usuelle, ainsi on sait que:

A + A = 2 * A

ceci peut ne plus être vrai dans le cas de fonctions avec effets de bord:

Si les fonctions avaient des paramètres de sortie:

Les règles de l'arithmétique élémentaire ne sont plus valables:

```
A : Integer := 1;
…
function F1 ( A : in Integer ) return Integer is
begin
    return A + 3;
end F1;
…
function F2 ( A : "in out" Integer ) return Integer is
begin
    A := A * 10;
    return A + 3;
end F2;
…
Begin
    ...
    Put ( F1 (A) + F2 (A) );     -- 4 + 13 = 17
    A := 1;
    Put ( F2 (A) + F1 (A) );     -- 13 + 13 = 26
```

Commutativité:
A + B = B + A

Associativité:
(A + B) + C = A + (B + C)

F1 (A) + F2 (A) /= F2 (A) + F1 (A)

De plus: **FX (A) + FX (A) /= 2 * FX (A)**

- Il en va de même si les fonctions utilisent des variables globales (effets de bord):

```
A : Integer := 1;
…
function F1 return Integer is
begin
    return A + 3;
end F1;
…
function F2 return Integer is
begin
    A := A * 10;
    return A + 3;
end F2;
…
Begin
    ...
    Put ( F1 + F2 );       -- 4 + 13 = 17
    A := 1;
    Put ( F2 + F1 );       -- 13 + 13 = 26
```

Il ne faut jamais utiliser de variables globales, mais les transmettre en paramètres.

ici: **F1 + F2 /= F2 + F1**

- Tout ceci est quelque peu gênant et même dangereux!

Le **très mauvais** programme exemple qui suit illustre bien le danger d'une telle situation et montre donc ce qu'il ne faut surtout pas faire!

```
-- Ce qu'il ne faut pas faire...
with Ada.Text_Io; use Ada.Text_Io;
with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
procedure Test_Param is

     Valeur : Integer := 10;            -- Pour l'exemple

  ------------------------------------------------------
  -- Démnontre l'effet du passage de paramètres par recopie
  procedure Bidon ( Copie : in out Integer ) is

  begin

     Put_Line ( "Dans le sous-programme" );
     Put ( Copie ); New_Line;
     Copie := 2 * Copie;
     Put ( Copie ); New_Line;
     Put ( Valeur ); New_Line ( 3 );

  end Bidon;
```

```
begin -- Test_Param

    -- Introduction
    Put_Line ( "Demontre le passage de paramètre par copie" );
    Put_Line ( "Avant l'appel du sous-programme" );
    Put ( Valeur ); New_Line ( 3 );
    Bidon ( Valeur );
    Put_Line ( "Apres l'appel du sous-programme" );
    Put ( Valeur ); New_Line ( 3 );

end Test_Param;
```

… et ce programme donne les résultats:

Demontre le passage de paramètre par copie
Avant l'appel du sous-programme
10

Dans le sous-programme
10
20
10

Apres l'appel du sous-programme
20

Encore quelques attributs

L'attribut **Size** s'applique à n'importe quel type:

```
Integer'Size
```

Ou à une variable de n'importe quel type:

```
Ma_Variable'Size
```

Il livre sous forme d'une valeur entière universelle la taille en bits de la variable ou des objets déclarés de ce type.

Pour n'importe quel type scalaire:

```
Type'Max ( V1, V2 )
Type'Min ( V1, V2 )
```

Livre la plus grande, respectivement petite valeur de V1 ou V2

V1, V2 et le résultat livré par l'attribut sont du type *Type*

Exemple:

…

```
Put ( Integer'Max ( 5, 21 ) );  New_line;
```

affichera: 21

Pour n'importe quel type scalaire:

```
Type'Width
```

Livre la taille maximale de la chaîne qui sera restituée par l'utilisation de l'attribut Image appliqué à un objet du type *Type*

Types numériques: complément

Types entiers

Les remarques ci-dessous s'appliquent aussi, pour l'essentiel, aux réels flottants.

Lors des déclarations de types entiers, le compilateur choisit le type à disposition au niveau matériel le plus adapté: **le plus petit type satisfaisant notre contrainte**. Ceci implique qu'à chaque modification d'une variable de ce type, un test sur la contrainte est réalisé, donc du code a été généré (place, temps). Si toutefois on désire réduire le code et le temps induit par ces contraintes, on peut passer par une double déclaration du genre:

```
type Anonyme is range -1E5 .. 1E5;

type Entier is range Anonyme'Base'First .. Anonyme'Base'Last;
```

On notera sur cet exemple l'attribut ***Base*** qui s'utilise ici en association avec d'autres attributs, dans l'exemple: First qui s'applique à tout type ou sous-type pour indiquer que l'on se réfère au type de base ayant permis la construction de ce sous-type! Les autres attributs usuels sont utilisables sur ce type:

```
T_Entier'Base'First

T_Entier'Base'Last

T_Entier'Base'Range
```

Les opérations arithmétiques sont toujours réalisées sur le type de base cela peut impliquer des problèmes de portabilité car un calcul intermédiare peut provoquer un débordement sur une machine, alors que ce ne sera pas le cas sur une autre; le résultat final théorique étant lui tout à fait compatible avec la déclaration du type. La solution pour obtenir le même résultat dans tous les environnement, c'est à dire une erreur (exception) dans tous les cas:

```
function Base_Add ( Left, Right : T_Entier )
                          return T_Entier renames "+";

function "+" ( Left, Right : T_Entier ) return T_Entier is

begin
        return Base_Add ( Left, Right );
end "+";
```

Type décimal

Un type décimal est une forme particulière de point-fixe:

type Identificateur **is delta** Erreur **digits** Nb_Digits;

où

`Erreur` (statique) représente l'erreur tolérée; dans ce cas c'est toujours une puissance de 10

`Nb_Digits` représente le nombre de chiffres significatifs désirés pour notre type décimal.

Exemple:

```
type Francs is delta 0.01 digits 6;
```

Les opérations possibles applicables à de tels types sont les mêmes que pour le type `point-fixe` avec quelques particularités.

Les valeurs sont toujours arrondies par troncature.

Les entrées-sorties sont possibles grâce au paquetage `Ada.Text_IO.Decimal_IO` utilisé de manière semblable à celle présentée pour les types point-flottant.

Type article complément

Cas limite:

- Un article peut être vide, mais il faut le dire explicitement:

```
type Bidon is
  record
    null;
  end record;
```

- Contrairement à ce que l'on pourrait croire à priori cette situation est suffisamment fréquente (programmation objet) pour que l'on mette à disposition une formulation abrégée:

```
type Bidon is null record;
```

Exceptions, compléments

Dans un traite exception, la même exception peut apparaître 2 fois dans la même branche; cette possibilité, à première vue étrange s'avère utile lors de surnommage, car dans ce cas là deux noms différents peuvent désigner la même exception.

Les instructions d'un traite exception peuvent être presque quelconques, la seule restriction réside dans l'instruction **goto** qui ne peut pas redonner le contrôle à l'intérieur du bloc qui a provoqué l'exception. Vous pouvez donc:

- Appeler un sous-programme.
- Déclarer des blocs.
- Utiliser un **return** si vous êtes dans un sous-programme.
- Utiliser un **exit** si le bloc traitant l'exception est dans une boucle.
- Utiliser un **goto** redonnant le contrôle en dehors du bloc ayant provoqué l'exception (mais cette possibilité vous ne l'utiliserez pas!!!).

Si dans un traite exception une nouvelle exception est levée, elle est immédiatement propagée au niveau supérieur, à moins que le traitement de l'exception se fasse dans un bloc comportant son propre traite exception.

Les exceptions sont des objets un peu particuliers:

- Il n'est pas possible de déclarer un tableau d'exceptions.
- Il n'est pas possible de déclarer un champs d'article du type exception.
- Dans les appels récursifs, contrairement aux autres objets qui sont recréés pour chaque appel, les exceptions déclarées localement n'existerons qu'à un seul exemplaire. Si ce n'était pas le cas, leur gestion deviendrait impossible.
- Elles ne peuvent pas être passées en paramètres à des sous-programmes.

Le paquetage Ada.Exceptions (ne pas confondre avec Ada.Io_Exceptions) peut nous aider à aller un peu plus loin! Voici tout d'abord son contenu:

```
package Ada.Exceptions is
  type Exception_Id is private;
  Null_Id : constant Exception_Id;
  function Exception_Name        ( Id : Exception_Id ) return String;
  type Exception_Occurrence is limited private;
  type Exception_Occurrence_Access is access all Exception_Occurrence;
  Null_Occurrence : constant Exception_Occurrence;
  procedure Raise_Exception      ( E : in Exception_Id;
                                   Message : in String := "" );
  function Exception_Message     ( X : Exception_Occurrence ) return String;
  procedure Reraise_Occurrence   ( X : in Exception_Occurrence );
  function Exception_Identity    ( X : Exception_Occurrence )
                                              return Exception_Id;
  function Exception_Name        ( X : Exception_Occurrence ) return String;
  -- Same as Exception_Name      ( Exception_Identity ( X ) )
  function Exception_Information ( X : Exception_Occurrence ) return String;
  procedure Save_Occurrence      ( Target : out Exception_Occurrence;
                                   Source : in Exception_Occurrence );
```

```
  function Save_Occurrence       ( Source : Exception_Occurrence )
                                       return Exception_Occurrence_Access;
private
  ... -- not specified by the language
end Ada.Exceptions;
```

Il faut bien reconnaître que l'utilisation des exceptions n'est pas la chose la plus cohérente du langage Ada. Imaginons le type Exception_Id comme un type énuméré qui serait construit à partir de toutes les exceptions utilisées dans le programme; donc ce type ne pourrait être construit qu'à l'édition de liens. Les exceptions existent donc durant toute la durée de vie du programme.

L'objet Nom que l'on obtient en le précisant dans une branche du traite exception:

when Nom : Mon_Exception => …

correspondrait à une constante du type Exception_Occurence.

Tout ce qui suit est valable aussi bien pour les exceptions prédéfinies que pour nos propres exceptions.

Donnons d'abord un programme exemple "bidon" permettant d'illustrer l'essentiel des possibilités:

```
-- Programme exemple "bidon" montrant l'utilisation des exceptions
with Ada.Text_Io; use Ada.Text_Io;
with Ada.Exceptions; use Ada.Exceptions;
procedure Exceptions2 is

  Mon_Exception : exception;     -- Exemple de définition d'exception
  Sauve_Exception : Exception_Occurrence;-- Pour sauver une exception

begin   -- Exceptions2

  Put_Line ( "Presentation des exceptions.");
  New_Line (3);
  --
  -- Les informations sur une exception standard
  begin
    raise Constraint_Error;
  exception
    when Erreur1 : Constraint_Error =>
      --
      -- Sauver l'exception pour usage ultérieur
      Save_Occurrence ( Sauve_Exception, Erreur1 );
      Put_Line ( "Name:" );
      Put_Line ( Exception_Name ( Erreur1 ) ); New_Line;
      Put_Line ( "Message:" );
      Put_Line ( Exception_Message ( Erreur1 ) ); New_Line;
      Put_Line ( "Information:" );
      Put_Line ( Exception_Information ( Erreur1 ) ); New_Line;
    when others =>              -- Autre erreur éventuelle
      Put_Line ( "Etrange!!!");
   end;
```

```
   --
   -- La même chose pour nos propres exceptions
   begin
     raise Mon_Exception;
   exception
     when Erreur2 : Mon_Exception =>
       Put_Line ( "Name:" );
       Put_Line ( Exception_Name ( Erreur2 ) ); New_Line;
       Put_Line ( "Message:" );
       Put_Line ( Exception_Message ( Erreur2 ) ); New_Line;
       Put_Line ( "Information:" );
       Put_Line ( Exception_Information ( Erreur2 ) ); New_Line;
     when others =>              -- Autre erreur éventuelle
       Put_Line ( "Etrange!!!");
   end;
   --
   -- Lever une exception en la complétant par un message
   begin
     Raise_Exception ( Constraint_Error'Identity, "C'est pas beau!" );
   exception
     when Erreur3 : Constraint_Error =>
       Put_Line ( "Information:" );
       Put_Line ( Exception_Information ( Erreur3 ) ); New_Line;
     when others =>              -- Autre erreur éventuelle
       Put_Line ( "Etrange!!!");
   end;
   --
   -- Traiter l'exception qui avait ete sauvee
   begin
     Reraise_Occurrence ( Sauve_Exception );
   exception
     when Erreur4 : Constraint_Error =>
       Put_Line ( "Information:" );
       Put_Line ( Exception_Information ( Erreur4 ) ); New_Line;
     when others =>              -- Autre erreur éventuelle
       Put_Line ( "Etrange!!!");
   end;

end Exceptions2;
```

Voici les résultats que donne l'exécution de ce programme dans notre environnement:

Presentation des exceptions.

```
Name:
Constraint_Error

Message:


Information:
Constraint_Error ()
Exception traceback:
  23   exceptions2                    resulfichier.ada
Exception handled at:
  25   exceptions2                    resulfichier.ada

Name:
Exceptions2.Mon_Exception

Message:


Information:
Exceptions2.Mon_Exception ()
Exception traceback:
  41   exceptions2                    resulfichier.ada
Exception handled at:
  43   exceptions2                    resulfichier.ada

Information:
Constraint_Error () C'est pas beau!
Exception traceback:
  45   ada.exceptions.raise_exception          except$3.bdy
  56   exceptions2                    resulfichier.ada
Exception handled at:
  58   exceptions2                    resulfichier.ada

Information:
Constraint_Error ()
Exception traceback:
  23   exceptions2                    resulfichier.ada
  85   ada.exceptions.reraise_occurrence       except$3.bdy
  67   exceptions2                    resulfichier.ada
Exception handled at:
  69   exceptions2                    resulfichier.ada
```

Il faut tout d'abord signaler que le contenu des messages dépend de l'environnement et dans le notre à priori ce n'est pas très riche!

Au moment où nous levons une exception, nous pouvons associer un message spécifique à cette occurrence de l'exception. Dans ce cas nous n'utilisons pas l'instruction **raise** pour la lever, mais nous appelons la procédure Raise_Exception qui a 2 paramètres:

- Le premier est du type Exception_Id; nous devons donc convertir notre exception dans ce type. Ceci s'obtient par l'utilisation de l'attribut 'Identity:

  Mon_Exception'Identity

- Le deuxième est du type String et représente le message que l'on désire associer. Ce message sera restitué par les fonction Exception_Message et Exception_Information.

On peut sauver une exception par la procédure Save_Occurrence qui a 2 paramètres:

- Le premier est un objet d'un type limité Exception_Occurrence. Généralement nous ne voulons pas sauver une seule exception, mais certainement toute une série, donc nous déclarerons dans ce cas un tableau dont les éléments sont du type Exception_Occurrence.
- Le second paramètre est aussi du type Exception_Occurrence, c'est l'exception que nous voulons sauver.

Notons qu'il existe aussi une fonction Save_Occurrence qui a un paramètre du type Exception_Occurrence et qui livre comme résultat un pointeur sur l'objet qu'elle aura créé pour sauver notre exception.

Nous pouvons en tout temps à nouveau lever une exception qui a été sauvée, en utilisant la procédure Reraise_Exception, qui a un seul paramètre du type Exception_Occurrence. Il est à noter que cet appel ne provoque pas une nouvelle occurrence de l'exception, puisque nous "rejouons" ultérieurement une exception qui est déjà survenue.

Finalement la fonction Exception_Identity permet de convertir un objet de type Exception_Occurrence en un objet de type Exception_Id.

Après cela j'espère que vous comprendrez que l'utilisation des exceptions doit rester dans le cadre de situations strictement exceptionnelles!

Les fichiers, compléments

Ada.Io_Exceptions:

Le paquetage Ada.Io_Exception est utilisé par tous les autres paquetage d'entrées/soties; son contenu est simple, il s'agit de la définition des exceptions liées aux entrées/sorties:

```
package Ada.IO_Exceptions is
    pragma Pure ( IO_Exceptions );
    Status_Error : exception;
    Mode_Error   : exception;
    Name_Error   : exception;
    Use_Error    : exception;
    Device_Error : exception;
    End_Error    : exception;
    Data_Error   : exception;
    Layout_Error : exception;
end Ada.IO_Exceptions;
```

A titre d'exemple:

STATUS_ERROR	est levée si l'on tente de fermer un fichier (close) qui n'aurait pas été ouvert au préalable.
USE_ERROR	est levée si l'environnement ne permet pas d'ouvrir un fichier dans le mode spécifié.
NAME_ERROR	est levée si le nom spécifié pour un fichier externe n'est pas valide.
END_ERROR	est levée si l'on demande une lecture alors que l'on a déjà épuisé les éléments du fichier (fin de fichier).
DATA_ERROR	est levée si l'on demande la lecture d'un objet d'un certain type et que les données sur le fichier ne peuvent être interprétées comme étant de ce type.
MODE_ERROR	est levée si l'on tente d'écrire sur un fichier préparé en lecture.
DEVICE_ERROR	est levée si le périphérique utilisé présente un problème physique.
LAYOUT_ERROR	est levée si l'on écrit dans une chaîne trop courte pour contenir l'ensemble des caractères.

Les fichiers de texte:

Le paquetage Ada.Text_Io lui-même:

```
with Ada.Io_Exceptions;
package Ada.Text_Io is
  type File_Type is limited private;
  type File_Mode is ( In_File, Out_File, Append_File );
  type Count     is range 0 .. Implementation-Defined;
  subtype Positive_Count is Count range 1 .. Count'Last;
  Unbounded : constant Count := 0;    -- line and page length
  subtype Field is Integer range 0 .. Implementation-Defined;
  subtype Number_Base is Integer range 2 .. 16;
  type Type_Set is ( Lower_Case, Upper_Case );
  -- File Management
  procedure Create ( File : in out File_Type;
                     Mode : in File_Mode := Out_File;
                     Name : in String    := "";
                     Form : in String    := "" );
  procedure Open   ( File : in out File_Type;
                     Mode : in File_Mode;
                     Name : in String;
                     Form : in String    := "" );
  procedure Close  ( File : in out File_Type );
  procedure Delete ( File : in out File_Type );
  procedure Reset  ( File : in out File_Type; Mode : in File_Mode );
  procedure Reset  ( File : in out File_Type );

  function  Mode    ( File : in File_Type ) return File_Mode;
  function  Name    ( File : in File_Type ) return String;
  function  Form    ( File : in File_Type ) return String;
  function  Is_Open ( File : in File_Type ) return Boolean;
  -- Control of default input and output files
  procedure Set_Input  ( File : in File_Type );
  procedure Set_Output ( File : in File_Type );
  procedure Set_Error  ( File : in File_Type );

  function Standard_Input  return File_Type;
  function Standard_Output return File_Type;
  function Standard_Error  return File_Type;
  function Current_Input   return File_Type;
  function Current_Output  return File_Type;
  function Current_Error   return File_Type;

  type File_Access is access constant File_Type;
  function Standard_Input  return File_Access;
  function Standard_Output return File_Access;
  function Standard_Error  return File_Access;
  function Current_Input   return File_Access;
  function Current_Output  return File_Access;
  function Current_Error   return File_Access;
  -- Buffer control
  procedure Flush ( File : in out File_Type );
  procedure Flush;
```

```
   -- Specification of line and page lengths
   procedure Set_Line_Length ( File : in File_Type; To : in Count );
   procedure Set_Line_Length ( To   : in Count );
   procedure Set_Page_Length ( File : in File_Type; To : in Count );
   procedure Set_Page_Length ( To   : in Count );
   function  Line_Length ( File : in File_Type ) return Count;
   function  Line_Length return Count;
   function  Page_Length ( File : in File_Type ) return Count;
   function  Page_Length return Count;
   --  Column, Line, and Page Control
   procedure New_Line    ( File    : in File_Type;
                           Spacing : in Positive_Count := 1 );
   procedure New_Line    ( Spacing : in Positive_Count := 1 );
   procedure Skip_Line   ( File    : in File_Type;
                           Spacing : in Positive_Count := 1 );
   procedure Skip_Line   ( Spacing : in Positive_Count := 1 );
   function  End_Of_Line ( File    : in File_Type ) return Boolean;
   function  End_Of_Line return Boolean;

   procedure New_Page   ( File : in File_Type );
   procedure New_Page;
   procedure Skip_Page  ( File : in File_Type );
   procedure Skip_Page;

   function  End_Of_Page ( File : in File_Type ) return Boolean;
   function  End_Of_Page return Boolean;
   function  End_Of_File ( File : in File_Type ) return Boolean;
   function  End_Of_File return Boolean;
   procedure Set_Col ( File  : in File_Type; To : in Positive_Count );
   procedure Set_Col  ( To   : in Positive_Count );
   procedure Set_Line ( File : in File_Type;
                        To   : in Positive_Count );
   procedure Set_Line ( To   : in Positive_Count );
   function Col ( File  : in File_Type ) return Positive_Count;
   function Col  return Positive_Count;
   function Line ( File : in File_Type ) return Positive_Count;
   function Line return Positive_Count;
   function Page ( File : in File_Type ) return Positive_Count;
   function Page return Positive_Count;
   -- Character Input-Output
   procedure Get ( File : in  File_Type; Item : out Character );
   procedure Get ( Item : out Character );
   procedure Put ( File : in  File_Type; Item : in Character );
   procedure Put ( Item : in  Character );

   procedure Look_Ahead    ( File : in  File_Type;
                             Item : out Character;
                             End_Of_Line : out Boolean );
   procedure Look_Ahead    ( Item : out Character;
                             End_Of_Line : out Boolean );
   procedure Get_Immediate ( File : in  File_Type;
                             Item : out Character );
   procedure Get_Immediate ( Item : out Character );
```

```
procedure Get_Immediate ( File      : in  File_Type;
                          Item      : out Character;
                          Available : out Boolean );
procedure Get_Immediate ( Item      : out Character;
                          Available : out Boolean);
-- String Input-Output
procedure Get ( File : in  File_Type; Item : out String );
procedure Get ( Item : out String );

procedure Put ( File : in  File_Type; Item : in String );
procedure Put ( Item : in  String );
procedure Get_Line ( File : in  File_Type;
                     Item : out String;
                     Last : out Natural);
procedure Get_Line ( Item : out String;    Last : out Natural );
procedure Put_Line ( File : in  File_Type; Item : in  String  );
procedure Put_Line ( Item : in  String );

-- Generic packages for Input-Output of Integer Types
generic
  type Num is range <>;
package Integer_IO is
  Default_Width : Field := Num'Width;
  Default_Base  : Number_Base := 10;
  procedure Get ( File  : in  File_Type;
                  Item  : out Num;
                  Width : in  Field := 0 );
  procedure Get ( Item  : out Num;
                  Width : in  Field := 0 );
  procedure Put ( File  : in  File_Type;
                  Item  : in  Num;
                  Width : in  Field := Default_Width;
                  Base  : in  Number_Base := Default_Base );
  procedure Put ( Item  : in  Num;
                  Width : in  Field := Default_Width;
                  Base  : in  Number_Base := Default_Base );
  procedure Get ( From  : in  String;
                  Item  : out Num;
                  Last  : out Positive );


  procedure Put ( To   : out String;
                  Item : in  Num;
                  Base : in  Number_Base := Default_Base );

end Integer_IO;
```

```
generic
  type Num is mod <>;
  package Modular_IO is
  Default_Width : Field := Num'Width;
  Default_Base  : Number_Base := 10;
  procedure Get ( File  : in  File_Type;
                  Item  : out Num;
                  Width : in  Field := 0 );
  procedure Get ( Item  : out Num;
                  Width : in  Field := 0 );

  procedure PuT ( File  : in  File_Type;
                  Item  : in  Num;
                  Width : in  Field       := Default_Width;
                  Base  : in  Number_Base := Default_Base );
  procedure Put ( Item  : in  Num;
                  Width : in  Field       := Default_Width;
                  Base  : in  Number_Base := Default_Base );
  procedure Get ( From  : in  String;
                  Item  : out Num;
                  Last  : out Positive );
  procedure Put ( To    : out String;
                  Item  : in   Num;
                  Base  : in Number_Base  := Default_Base );

end Modular_IO;

-- Generic packages for Input-Output of Real Types
generic
  type Num is digits <>;
package Float_IO is
  Default_Fore : Field := 2;
  Default_Aft  : Field := Num'Digits-1;
  Default_Exp  : Field := 3;
  procedure Get ( File  : in  File_Type;
                  Item  : out Num;
                  Width : in  Field := 0 );
  procedure Get ( Item  : out Num;
                  Width : in  Field := 0 );
  procedure Put ( File  : in  File_Type;
                  Item  : in  Num;
                  Fore  : in  Field := Default_Fore;
                  Aft   : in  Field := Default_Aft;
                  Exp   : in  Field := Default_Exp );
  procedure Put ( Item  : in  Num;
                  Fore  : in  Field := Default_Fore;
                  Aft   : in  Field := Default_Aft;
                  Exp   : in  Field := Default_Exp );
  procedure Get ( From  : in  String;
                  Item  : out Num;
                  Last  : out Positive );
  procedure Put ( To    : out String;
                  Item  : in  Num;
                  Aft   : in  Field := Default_Aft;
                  Exp   : in  Field := Default_Exp );
end Float_IO;
```

```
generic
  type Num is delta <>;
package Fixed_IO is
  Default_Fore : Field := Num'Fore;
  Default_Aft  : Field := Num'Aft;
  Default_Exp  : Field := 0;
  procedure Get ( File  : in  File_Type;
                  Item  : out Num;
                  Width : in  Field := 0 );
  procedure Get ( Item  : out Num;
                  Width : in  Field := 0 );
  procedure Put ( File  : in  File_Type;
                  Item  : in  Num;
                  Fore  : in  Field := Default_Fore;
                  Aft   : in  Field := Default_Aft;
                  Exp   : in  Field := Default_Exp );
  procedure Put ( Item  : in  Num;
                  Fore  : in  Field := Default_Fore;
                  Aft   : in  Field := Default_Aft;
                  Exp   : in  Field := Default_Exp );
  procedure Get ( From  : in  String;
                  Item  : out Num;
                  Last  : out Positive );
  procedure Put ( To    : out String;
                  Item  : in  Num;
                  Aft   : in  Field := Default_Aft;
                  Exp   : in  Field := Default_Exp);
end Fixed_IO;
```

```
generic
  type Num is delta <> digits <>;
package Decimal_IO is
  Default_Fore : Field := Num'Fore;
  Default_Aft  : Field := Num'Aft;
  Default_Exp  : Field := 0;
  procedure Get ( File  : in  File_Type;
                  Item  : out Num;
                  Width : in  Field := 0 );
  procedure Get ( Item  : out Num;
                  Width : in  Field := 0 );
  procedure Put ( File  : in  File_Type;
                  Item  : in  Num;
                  Fore  : in  Field := Default_Fore;
                  Aft   : in  Field := Default_Aft;
                  Exp   : in  Field := Default_Exp );
  procedure Put ( Item  : in  Num;
                  Fore  : in  Field := Default_Fore;
                  Aft   : in  Field := Default_Aft;
                  Exp   : in  Field := Default_Exp );
  procedure Get ( From  : in  String;
                  Item  : out Num;
                  Last  : out Positive );
  procedure Put ( To    : out String;
                  Item  : in  Num;
                  Aft   : in  Field := Default_Aft;
                  Exp   : in  Field := Default_Exp );
end Decimal_IO;

-- Generic package for Input-Output of Enumeration Types
generic
  type Enum is (<>);
package Enumeration_IO is
  Default_Width   : Field := 0;
  Default_Setting : Type_Set := Upper_Case;
  procedure Get ( File  : in  File_Type;
                  Item  : out Enum );
  procedure Get ( Item  : out Enum );
  procedure Put ( File  : in File_Type;
                  Item  : in Enum;
                  Width : in Field    := Default_Width;
                  Set   : in Type_Set := Default_Setting );
  procedure Put ( Item  : in Enum;
                  Width : in Field    := Default_Width;
                  Set   : in Type_Set := Default_Setting );
  procedure Get ( From  : in  String;
                  Item  : out Enum;
                  Last  : out Positive );
  procedure Put ( To    : out String;
                  Item  : in  Enum;
                  Set   : in  Type_Set := Default_Setting );
end Enumeration_IO;
```

```
   -- Exceptions
   Status_Error : exception renames IO_Exceptions.Status_Error;
   Mode_Error   : exception renames IO_Exceptions.Mode_Error;
   Name_Error   : exception renames IO_Exceptions.Name_Error;
   Use_Error    : exception renames IO_Exceptions.Use_Error;
   Device_Error : exception renames IO_Exceptions.Device_Error;
   End_Error    : exception renames IO_Exceptions.End_Error;
   Data_Error   : exception renames IO_Exceptions.Data_Error;
   Layout_Error : exception renames IO_Exceptions.Layout_Error;
private
   ... -- not specified by the language
end Ada.Text_Io;
```

Quelques explications:

Gestion globale

- **procedure** Reset (File : **in out** File_Type; Mode : **in** File_ Mode);
- **procedure** Reset (File : **in out** File_ Type);

  - Permet de repositionner le fichier sur son premier enregistrement.
  - Dans la forme 1, le nouveau mode est précisé explicitement, alors que dans la forme 2, le mode dépend du mode initial de traitement du fichier. Bien entendu le fichier doit déjà être ouvert pour pouvoir être traité par Reset, sinon Status_Error est levé.

- **procedure** Delete (File : **in out** File_Type);

  - Ferme le fichier File, comme le Close, mais en plus le fichier externe est détruit. Le fichier File doit au préalable être ouvert pour pouvoir lui appliquer la procédure Delete.

- **function** Mode (File : **in** File_Type) **return** File_Mode;

  - Livre le mode actuel de traitement du fichier (In_File, Out_File, Append_File). Le fichier doit être ouvert, sinon Status_Error est levée.

- **function** Name (File : **in** File_Type) **return** Sring;

  - Livre une chaîne de caractères correspondant au nom externe du fichier. Ce nom doit être le plus complet possible et sa forme dépend du système sur lequel on travaille! Le fichier doit être ouvert.

- **function** Form (File : **in** File_Type) **return** String;

  - Livre une chaîne de caractères décrivant les caractéristiques associées au fichier; celles-ci dépendent du système sur lequel on travaille. Cette spécification doit être la plus complète possible, c'est-à-dire qu'elle doit fournir aussi les spécifications par défaut.

- **function** Is_Open (File : **in** File_Type) **return** Boolean;

  - Livre la valeur True si le fichier File est ouvert, donc associé à un fichier externe et False sinon.

- **procedure** Set_Input (File : **in** File_Type);

  - Le fichier File devient l'entrée par défaut; donc dans toutes les procédures ou fonctions d'entrée pour lesquelles on ne précisera pas de fichier, c'est File qui sera utilisé. A noter que le type File_Type est défini dans le paquetage comme un type limité privé.

- **procedure** Set_Output (File : **in** File_Type);

  - Idem à Set_Input, mais pour le fichier de sortie: Output.

- **procedure** Set_Error (File : **in** File_Type);

  - Idem à Set_Input, mais pour le fichier des erreurs: Error.

- **function** Standard_Input **return** File_Type;
- **function** Standard_Input **return** File_Access;

  - Retournent le fichier d'entrée standard, donc le fichier par défaut au départ de l'application.

- **function** Standard_Output **return** File_Type;
- **function** Standard_Output **return** File_Access;

  - Retournent le fichier de sortie standard, donc le fichier par défaut au départ de l'application.

- **function** Standard_Error **return** File_Type;
- **function** Standard_Error **return** File_Access;

  - Retournent le fichier d'erreur standard, donc le fichier par défaut au départ de l'application.

- **function** Current_Input **return** File_Type;
- **function** Current_Input **return** File_Access;

  - Retournent le fichier d'entrée actuel par défaut qui est égal à Standard_Input s'il n'y a pas eu de modification.

- **function** Current_Output **return** File_Type;
- **function** Current_Output **return** File_Access;

  - Retournent le fichier actuel de sortie par défaut est égal à Standard_Output s'il n'y a pas eu de modification.

- **function** Current_Error **return** File_Type;
- **function** Current_Error **return** File_Access;

  - Retournent le fichier actuel par défaut des erreurs.

- **procedure** Flush (File : **in out** File_Type);
- **procedure** Flush;

  - Pour un fichier d'entrée, synchronise la variable de fichier interne avec le fichier externe: vide le(s) tampon(s).

Contrôle des lignes

- **procedure** Set_Line_Length (File : **in** File_Type; To : **in** Count);
- **procedure** Set_Line_Length (To : **in** Count);

  - Permettent de fixer la longueur maximale des lignes du fichier. Le paramètre To de type Count, représente le nombre de caractères maximum des lignes. Une valeur de 0 (valeur initiale par défaut) implique que l'on n'impose pas de limite à la longueur des lignes. Dans les cas où la longueur est fixée, une fin de ligne sera générée automatiquement si nos ordres d'écriture impliquent que l'on a atteint cette limite.

- **function** Line_Length (File : **in** File_Type) **return** Count;
- **function** Line_Length **return** Count;

  - Livrent la longueur actuelle maximale d'une ligne.

Contrôle des pages

- **procedure** Set_Page_Length (File : **in** File_Type; To : **in** Count);
- **procedure** Set_Page_Length (To : **in** Count);

  ➢ Fixent le nombre maximum de lignes par page à la valeur spécifiée par To. Une marque de fin de page est générée automatiquement si le nombre de lignes écrites par notre programme a atteint cette valeur. Par défaut (0) notre document ne comporte qu'une seule grande page.

- **function** Page_Length (File : **in** File_Type) **return** Count;
- **function** Page_Length **return** Count;

  ➢ Livrent le nombre de lignes maximum actuellement valable par page du fichier.

Traitement des lignes

- **procedure** New_Line (File : **in** File_Type; Spacing : **in** Positive_Count := 1);
- **procedure** New_Line (Spacing : **in** Positive_Count := 1);

  ➢ Permettent d'ajouter dans un fichier de sortie Spacing termine-lignes, donc de vider la ligne courante et d'introduire Spacing-1 lignes vides. Si nécessaire, soit si l'on atteint une fin de page, un termine-page est ajouté automatiquement au bon endroit.

- **procedure** Skip_Line (File : **in** File_Type; : **in** Positive_Count := 1);
- **procedure** Skip_Line (Spacing : **in** Positive_Count := 1);

  ➢ Permettent dans un fichier d'entrée de sauter des caractères jusqu'à ce que Spacing termine-lignes soient lus. L'opération se poursuit au-delà d'un termine-page si nécessaire; le numéro de page courante est alors augmenté en conséquence. La colonne courante devient la première colonne de la nouvelle ligne courante.

- **function** End_Of_Line (File : **in** File_Type) **return** Boolean;
- **function** End_Of_Line **return** Boolean;
-

  ➢ Livrent la valeur True si le fichier est positionné sur un termine-ligne (marque de fin de ligne) et False sinon.

Traitement des pages

- **procedure** New_Page (File : **in** File_Type);
- **procedure** New_Page;

  - Provoquent l'enregistrement d'un termine-page précédé éventuellement d'un termine-ligne si la ligne courante n'est pas terminée ou si la page courante est vide (ce qui laisserait une page blanche !). Le numéro de la page courante est donc augmenté de 1 et les numéros de colonne et de ligne courantes sont mis à 1.

- **procedure** Skip_Page (File : **in** File_Type);
- **procedure** Skip_Page;

  - Permettent de lire dans le fichier jusqu'à un termine-page. Le numéro de la page courante est donc augmenté de 1 et les numéros de colonne et de ligne courantes sont mis à 1.

- **function** End_Of_Page (File : **in** File_Type) **return** Boolean;
- **function** End_Of_Page **return** Boolean;

  - Livrent la valeur True si le fichier est positionné sur la combinaison d'un termine-ligne et d'un termine-page (marque de fin de page) ou d'un termine-fichier (marque de fin de fichier). Livre False sinon.

- **function** End_Of_File (File : **in** File_Type) **return** Boolean;
- **function** End_Of_File **return** Boolean;

  - Livrent la valeur True si le fichier est positionné sur un termine-fichier ou la combinaison d'un termine-ligne, termine-page et d'un termine-fichier. Livre False sinon.

Opérations sur les colonnes, lignes et pages

- **procedure** Set_Col (File : **in** File_Type; To : **in** Positive_Count);
- **procedure** Set_Col (To : **in** Positive_Count);

  - **En sortie:**
    - Si To représente une valeur plus grande que la position actuelle dans la ligne courante, on ajoute autant de caractères espaces que nécessaire pour arriver à une position courante égale à To.
    - Si To est égal à la position actuelle aucun effet.
    - Si To plus petit que la position actuelle, on vide la ligne (New_Line) et l'on commence une nouvelle ligne avec To - 1 espaces.

- Lorsque la longueur maximum d'une ligne a été fixée (Set_Line_Length), la valeur de To ne doit pas dépasser cette grandeur, sinon Laayout_Error est levée.

➢ **En entrée:**

- Si To est égal à la position actuelle aucun effet.
- Dans les autres cas, on saute les caractères, sans tenir compte des éventuels termine-lignes et termine-pages jusqu'à ce que l'on puisse se positionner dans une ligne à une colonne de numéro To.

- **function** Col (File : **in** File_Type) **return** Positive_Count;
- **function** Col **return** Positive_Count;

- Livrent le numéro de la colonne courante.

- **procedure** Set_Line (File : **in** File_Type; To : **in** Positive_Count);
- **procedure** Set_Line (To : **in** Positive_Count);

➢ **En sortie:**

- Si To est plus grand que le numéro de la ligne courante, on ajoute des lignes (New_Line) jusqu'à ce que le numéro courant de la ligne soit égal à To.
- Si To est égal au numéro de la ligne courante, pas d'effet.
- Si To est plus petit que le numéro de la ligne courante, on ajoute un termine-page (New_Page) et l'on commence une nouvelle page avec To - 1 lignes vides (New_Line).

➢ **En entrée:**

- Si To est égal au numéro de la ligne courante pas d'effet.
- Dans tous les autres cas, on saute des lignes sans tenir compte des termine-pages, jusqu'à ce que l'on trouve une page comportant une ligne de numéro égal à To. Les pages "courtes" sont donc sautées!

- **function** Line (File : **in** File_Type) **return** Positive_Count;
- **function** Line **return** Positive_Count;

- Livrent le numéro de la ligne courante.

- **function** Page (File : **in** File_Type) **return** Positive_Count;
- **function** Page **return** Positive_Count;

- On ne peut pas positionner explicitement le fichier à une page spécifique, c'est à nous de le programmer par une boucle si on le désire! Toutefois, on peut connaître le numéro de la page courante par les fonctions Page.

Traitement des caractères

- **procedure** Get (File : **in** File_Type; Item : **out** Character);
- **procedure** Get (Item : **out** Character);
-
  - Pour lire un caractère, sautent les éventuels fin de lignes et de pages

- **procedure** Put (File : **in** File_Type; Item : **in** Character);
- **procedure** Put (Item : **in** Character);
  - Pour écrire un caractère, ajoutent si nécessaire une fin de ligne et de page.

- **procedure** Look_Ahead (File: **in** File_Type; Item: **out** Character;End_Of_Line:**out** Boolean);
- **procedure** Look_Ahead (Item : **out** Character; End_Of_Line : **out** Boolean);
  - Permettent d'obtenir le caractère suivant sans le "consommer", c'est à dire qu'il reste dans le tampon d'entré et sera utilisé lors de la prochaine lecture. Ceci est très utile pour prendre des décisions sur la suite des lectures; par exemple pouvoir réaliser nos propres lectures qui se comportent comme les procédures standards, qui s'arrêtent dès que l'on a un caractère qui n'est plus valide.

- **procedure** Get_Immediate (File : **in** File_Type; Item : **out** Character);
- **procedure** Get_Immediate (Item : **out** Character);
- **procedure** Get_Immediate(File:**in** File_Type;Item: **out** Character;Available:**out** Boolean);
- **procedure** Get_Immediate (Item : **out** Character; Available : **out** Boolean);
  - Permettent de lire le caractère suivant qui se trouve dans le tampon, sans attendre qu'une ligne entière soit introduite (pour les applications interactives). Dans la forme avec le paramètre Available, aucune attente n'a lieu et si aucun caractère n'est disponible, on revient immédiatement avec Available à False et Item qui est alors indéfini. Note: les Get_Immediate ne modifient pas les numéros de ligne, colonne et page courante!).

Traitement des chaînes

- **procedure** Get (File : **in** File_Type; Item : **out** String);
- **procedure** Get (Item : **out** String);
  - Reviennent à faire un nombre de Get caractère égal à la longueur de chaîne.

- **procedure** Put (File : **in** File_Type; Item : **in** String);
- **procedure** Put (Item : **in** String);

  - Reviennent à faire un nombre de Put caractère égal à la longueur de chaîne.

- **procedure** Get_Line (File : **in** File_Type; Item : **out** String; Last : **out** Natural);
- **procedure** Get_Line (Item : **out** String; Last : **out** Natural);

  - Lisent caractère à caractère sur le fichier d'entrée et réduit les éléments lus dans les caractères successifs de la chaîne Item. La lecture s'arrête lorsque l'on atteint la longueur de la chaîne, ou un termine-ligne dans le fichier. Dans ce dernier cas un passage à la ligne (Skip_Line) est encore réalisé et les caractères non remplacés dans Item sont indéfinis. Si des caractères ont été lus, on retourne dans Last la position dans Item du dernier caractère lu. Si aucun caractère n'a été lu (on était déjà sur un termine-ligne), Last contient une valeur plus petite que Item'First.
    Attention: si on lit une ligne qui a un nombre de caractères égal à la longueur de la chaîne, le Skip_Line n'est pas réalisé!

- **procedure** Put_Line (File : **in** File_Type; Item : **in** String);
- **procedure** Put_Line (Item : **in** String);

  - Ecrivent la chaîne Item et passe à la ligne suivante. Agit donc comme un Put sur une chaîne suivi de l'opération New_Line.

Les paquetages génériques de Text_Io:

Les entiers

— **Integer_Io et Modular_Io**

- **Les mêmes caractéristiques son valables pour les 2 paquetages.**

— En admettant que notre type entier s'appelle Num, nous pourrons instancier un paquetage spécifique par exemple sous la forme:

```
package Num_Io is new Ada.Text_Io.Integer_Io ( Num );
use Num_Io;
```

— **subtype** Number_Base **is** Integer **range** 2 .. 16;

- Sur les entiers nous pourrons travailler (lire et écrire) des valeurs dans n'importe quelle base comprise entre 2 et 16

— Default_Width : Field := Num'Width;

— Default_Base : Number_Base := 10;

- Permettent de définir la taille du champ par défaut pour l'écriture (à noter l'utilisation de l'attribut: Num'Width, sur Num paramètre générique) et la base utilisée, par défaut bien entendu la base 10! Notez bien qu'il s'agit de variables, donc vous pouvez changer à votre convenance ces valeurs par défaut.

- **En lecture**

- **procedure** Get (File : **in** File_Type; Item : **out** Num; Width : **in** Field := 0);
- **procedure** Get (Item : **out** Num; Width : **in** Field := 0);

  - Si Width vaut 0 (valeur par défaut) tous les espaces, termine-lignes et termine-pages de tête sont sautés; ensuite on reconstruit un nombre caractère à caractère. Ce nombre doit respecter la syntaxe d'un littéral entier qui peut être basé.
  - Si Width > 0 on lit exactement Width caractères à moins qu'un termine-ligne apparaisse avant, auquel cas on arrête la lecture. On reconstruit alors la valeur de la même manière qu'avant.
  - L'utilisateur peut donner sa valeur dans n'importe quelle base entre 2 et 16, pour autant qu'il respecte la notation pour les littéraux basés, par exemple: 8#10#.

- **procedure** Get (From : **in** String; Item : **out** Num; Last : **out** Positive);

  - Lit une valeur non pas dans un fichier, mais dans la chaîne From, dont la fin de la chaîne se comporte comme une fin de fichier. Pour la lecture, cette procédure se comporte comme le Get normal, mais livre en plus dans Last l'indice du dernier caractère utilisé dans la chaîne

- **En écriture**

- **procedure** Put (File : **in** File_Type;
  Item : **in** Num;
  Width : **in** Field := Default_Width;
  Base : **in** Number_Base := Default_Base);

- **procedure** Put (Item : **in** Num;
  Width : **in** Field := Default_Width;
  Base : **in** Number_Base := Default_Base);

  - Permettent d'écrire dans le fichier la valeur du paramètre Item exprimé dans la base (2 à 16) donnée par le paramètre Base. Par défaut Base vaut 10. La grandeur par défaut du champ utilisé dépend du type Num. La valeur est écrite avec éventuellement, si nécessaire, des espaces en tête, un signe moins et les chiffres du nombre, sans zéro de tête, sans trait bas ni exposant. Si Base > 10, les éventuelles lettres utilisées pour représenter le nombre sont écrites en majuscules. Note: étrangement, mais cela n'a pas beaucoup d'importance, on affichera une valeur dans une base donnée sous sa forme "basée" (exemple: 8#10#), mais en base 10 nous aurons toujours la forme usuelle!

- **procedure** Put (To : **out** String;
  Item : **in** Num;
  Base : **in** Number_Base := Default_Base);

  - Ecrit la valeur de Item non pas dans un fichier, mais dans la chaîne de caractères To, dont la longueur sert implicitement de paramètre "Width" non présent ici. La valeur est donc ajustée à droite dans le chaîne. Pour le reste, elle se comporte comme un Put habituel.

Les réels

– Float_Io, Fixed_Io, et Decimal_Io

- Le principe est le même pour les 3 paquetages liés aux réels.

– Les éléments utilisés pour les flottants:

```
Default_Fore  : Field := 2;
Default_Aft   : Field := Num'Digits–1;
Default_Exp   : Field := 3;
```

– Les mêmes éléments pour les fixes et les décimaux, mais avec d'autres valeurs d'initialisation:

```
Default_Fore  : Field := Num'Fore;
Default_Aft   : Field := Num'Aft;
Default_Exp   : Field := 0;
```

– D'où la représentation des nombres:

- Fore . Aft
- Fore . Aft E Exp

- **En lecture**

- **procedure** Get (File : **in** File_Type; Item : **out** Num; Width : **in** Field := 0);
- **procedure** Get (Item : **out** Num; Width : **in** Field := 0);

  - Lisent une valeur sur le fichier d'entrée et livrent cette valeur dans le paramètre de sortie Item.
  - Si Width = 0 tous les blancs, termine-lignes et termine-pages de tête sont sautés; ensuite peut venir un éventuel signe (+ ou -) enfin une chaîne de caractères dont la syntaxe doit correspondre à celle d'un littéral (éventuellement basé).

    - Si Width /= 0 on lit exactement Width caractères à moins qu'un termine-ligne qui arrête la lecture ne survienne avant. Les blancs de tête (compris dans la valeur Width) sont ignorés.

  – Les nombres peuvent prendre les formes suivantes:

    [+|–]numeric_literal
    [+|–]numeral.[exponent]
    [+|–].numeral[exponent]
    [+|–]base#based_numeral.#[exponent]
    [+|–]base#.based_numeral#[exponent]

- **procedure** Get (From : **in** String; Item : **out** Num; Last : **out** Positive);

  - Se comporte comme le Get "normal", simplement le fichier source est remplacé par la chaîne From dont la fin correspond à une fin de fichier. De plus, elle livre dans Last une valeur telle que From (Last) représente le dernier caractère utilisé pour la lecture.

- **En écriture**

- **procedure** Put (File : **in** File_Type;
  Item : **in** Num;
  Fore : **in** Field := Default_Fore;
  Aft : **in** Field := Default_Aft;
  Exp : **in** Field := Default_Exp);

- **procedure** Put (Item : **in** Num;
  Fore : **in** Field := Default_Fore;
  Aft : **in** Field := Default_Aft;
  Exp : **in** Field := Default_Exp);

- Permettent d'écrire dans le fichier la valeur de Item avec les conventions de formatage fixées par les paramètres.
- Si Item < 0.0 un signe - est compris dans la partie Fore.
- Si Exp = 0 la partie entière comportera autant de chiffres que nécessaire (plus que Fore s'il le faut); l'impression a alors la forme Fore.Aft
- Si Exp > 0 alors la partie entière ne comprendra qu'un seul chiffre. La partie Exp comporte toujours un signe et si nécessaire elle est complétée par des zéro de tête.
- Si nécessaire, la partie Fore est complétée par des espaces de tête.

- **procedure** Put (To : **out** String;
  Item : **in** Num;
  Aft : **in** Field := Default_Aft;
  Exp : **in** Field := Default_Exp);

  - Agit comme le Put "normal" mais le fichier est remplacé par la chaîne To. La valeur de Fore est ici fixée implicitement, de telle sorte que toute la chaîne soit utilisée.

Les types énumérés

- Notons la présence dans ce paquetage du type énuméré:
  - **type** Type_Set **is** (Lower_Case, Upper_Case);

- et des variables:
  - Default_Width : Field := 0;
    - représente la largeur du champ utilisé (nombre de caractères) pour écrire une valeur du type énuméré. L'initialisation par défaut à 0 implique l'utilisation de la place strictement nécessaire, mais cette valeur par défaut peut être changée puisqu'il s'agit d'une variable.
  - Default_Setting : Type_Set := Upper_Case;
    - permet, par un paramètre des procédures Put de préciser si l'on désire écrire les identificateurs du type énuméré en majuscules ou en minuscules (par défaut : majuscules, mais cette valeur par défaut peut être changée puisqu'il s'agit d'une variable).

- **En lecture**

- **procedure** Get (File : **in** File_Type; Item : **out** Enum);
- **procedure** Get (Item : **out** Enum);

  - Sautent les espaces, termine-lignes et termine-pages de tête puis lit un élément de type Enum qu'elle livrent dans Item. Les valeurs peuvent indifféremment être données avec des majuscules et/ou des minuscules, sauf si l'on traite des caractères sous cette forme.

- **procedure** Get (From : **in** String; Item : **out** Enum; Last : **out** Positive);

  - Fonctionne comme le Get sur un fichier, mais dans la chaîne From dont elle considère la fin comme une fin de fichier. Au retour Last contient une valeur telle que From (Last) soit le dernier caractère utilisé pour la lecture.

  - **En écriture**

- **procedure** Put (File : **in** File_Type;
  Item : **in** Enum;
  Width : **in** Field := Default_Width;
  Set : **in** Type_Set := Default_Setting);

- **procedure** Put (Item : **in** Enum;
  Width : **in** Field := Default_Width;
  Set : **in** Type_Set := Default_Setting);

  - Ecrivent dans le fichier la valeur de Item sous la forme d'un littéral d'énumération, en majuscules ou minuscules suivant le paramètre Set. Si Width est plus grand que la valeur nécessaire pour l'affichage, des espaces sont ajoutés en queue.

- **procedure** Put (To : **out** String;
  Item : **in** Enum;
  Set : **in** Type_Set := Default_Setting);

  - Agit comme le Put pour un fichier, mais écrit la valeur dans la chaîne To dont la longueur fixe implicitement la valeur de Width.

Remarques complémentaires

- Le type Boolean étant un type énuméré, on peut instancier Enumeration _Io avec lui.

- Les types "entiers", bien que discrets, ne doivent pas être utilisés pour instancier Enumeration_Io.

- L'appel:

  Ada.Text_Io.Put ('A');

  a pour effet d'écrire le caractère A; mais si nous avons:

```
package Char_Io is new  Ada.Text_Io.Enumeration_Io ( Character );
```

ce qui est possible, le type Character étant un type énuméré, et que nous écrivons ensuite:

```
Char_Io.Put ( 'A' );
```

nous imprimons en fait 'A' (avec les apostrophes!).

- Nous pouvons aussi avec les procédures Get de Enumeration _Io lire un littéral caractère, mais l'utilisateur doit alors le mettre entre apostrophes.
- Il existe un paquetage Ada.Wide_Text_Io dont le contenu est totalement identique à celui de Ada.Text_Io, mais qui travail, bien entendu sur le type Wide_Character au lieu de travailler sur le type Character.

Pour l'utilisation, se référer aux exemples de programmes.

Le paquetage Ada.Sequential_Io:

```
with Ada.Io_Exceptions;
generic
  type Element_Type(<>) is private;
package Ada.Sequential_Io is
  type File_Type is limited private;
  type File_Mode is (In_File, Out_File, Append_File);
  -- File management
  procedure Create( File : in out File_Type;
                    Mode : in File_Mode := Out_File;
                    Name : in String := "";
                    Form : in String := "" );
  procedure Open  ( File : in out File_Type;
                    Mode : in File_Mode;
                    Name : in String;
                    Form : in String := "" );
  procedure Close ( File : in out File_Type );
  procedure Delete( File : in out File_Type );
  procedure Reset ( File : in out File_Type; Mode : in File_Mode );
  procedure Reset ( File : in out File_Type );

  function Mode   ( File : in File_Type ) return File_Mode;
  function Name   ( File : in File_Type ) return String;
  function Form   ( File : in File_Type ) return String;
  function Is_Open( File : in File_Type ) return Boolean;
  -- Input and output operations
  procedure Read  ( File : in File_Type; Item : out Element_Type );
  procedure Write ( File : in File_Type; Item : in Element_Type );

  function End_Of_File(File : in File_Type) return Boolean;
  -- Exceptions
  Status_Error : exception renames Io_Exceptions.Status_Error;
  Mode_Error   : exception renames Io_Exceptions.Mode_Error;
  Name_Error   : exception renames Io_Exceptions.Name_Error;
  Use_Error    : exception renames Io_Exceptions.Use_Error;
  Device_Error : exception renames Io_Exceptions.Device_Error;
  End_Error    : exception renames Io_Exceptions.End_Error;
  Data_Error   : exception renames Io_Exceptions.Data_Error;
private
  ... -- not specified by the language
end Ada.Sequential_Io;
```

Ce paquetage est générique et doit donc être instancié avec le type des éléments (des enregistrements) du fichier.

Notons simplement, comme complément, que l'on retrouve, comme dans Ada.Text_Io:

- Le type des fichiers, File_Type, comme étant limité privé (MAIS C'EST UN AUTRE TYPE!).

- De même le mode de traitement, File_Mode, reste défini par un type énuméré (In_File, Out_File, Append_File).

- Le renommage des exceptions: Status_Error, Mode_Error, Name_Error, Use_Error,

Device_Error, End_Error, Data_Error

- On retrouve également avec les mêmes paramètres et la même fonctionnalité, les procédures: Create, Open, Close, Delete, Reset (pour ce dernier, toujours avec ses deux formes surchargées).

- Les fonctions: Mode, Name, Form, Is_Open, End_Of_File

- Les lectures/écritures se font respectivement par Read/Write et non Put et Get.

Pour l'utilisation, se référer aux exemples de programmes.

Le paquetage Ada.Direct_Io:

```
with Ada.Io_Exceptions;
generic
  type Element_Type is private;
package Ada.Direct_Io is
  type File_Type is limited private;
  type File_Mode is ( In_File, Inout_File, Out_File );
  type Count     is range 0 .. Implementation-Defined;
  subtype Positive_Count is Count range 1 .. Count'Last;
  -- File management
  procedure Create( File : in out File_Type;
                    Mode : in File_Mode := Inout_File;
                    Name : in String := "";
                    Form : in String := "" );
  procedure Open  ( File : in out File_Type;
                    Mode : in File_Mode;
                    Name : in String;
                    Form : in String := "" );
  procedure Close ( File : in out File_Type );
  procedure Delete( File : in out File_Type );
  procedure Reset ( File :in out File_Type; Mode :in File_Mode );
  procedure Reset ( File : in out File_Type );

  function Mode   ( File : in File_Type ) return File_Mode;
  function Name   ( File : in File_Type ) return String;
  function Form   ( File : in File_Type ) return String;
  function Is_Open( File : in File_Type ) return Boolean;
  -- Input and output operations
  procedure Read  ( File : in File_Type; Item : out Element_Type;
                    From : in Positive_Count );
  procedure Read  ( File : in File_Type; Item :out Element_Type );

  procedure Write ( File : in File_Type; Item : in  Element_Type;
                    To   : in Positive_Count );
  procedure Write ( File : in File_Type; Item :in Element_Type );
  procedure Set_Index ( File : in File_Type; To : in Positive_Count );
  function Index  ( File : in File_Type ) return Positive_Count;
  function Size   ( File : in File_Type ) return Count;
  function End_Of_File( File : in File_Type ) return Boolean;
  -- Exceptions
  Status_Error : exception renames Io_Exceptions.Status_Error;
  Mode_Error   : exception renames Io_Exceptions.Mode_Error;
  Name_Error   : exception renames Io_Exceptions.Name_Error;
  Use_Error    : exception renames Io_Exceptions.Use_Error;
  Device_Error : exception renames Io_Exceptions.Device_Error;
  End_Error    : exception renames Io_Exceptions.End_Error;
  Data_Error   : exception renames Io_Exceptions.Data_Error;

private
  ... -- not specified by the language
end Ada.Direct_Io;
```

Ce paquetage est générique et doit donc être instancié avec le type des éléments (des enregistrements) du fichier.

Notons, comme complément, que l'on retrouve, comme dans Ada.Text_Io:

- Le type des fichiers, File_Type, comme étant limité privé (MAIS C'EST ENCORE UN AUTRE TYPE!).

- Le mode de traitement, File_Mode, reste défini par un type énuméré, In_File, Out_File, Inout_File; on peut donc pour un tel fichier lire ou écrire n'importe quel enregistrement, d'où le mode Inout_File; le mode Append_File n'a donc pas de raison d'exister puisqu'il suffira de se positionner en fin de fichier et d'écrire.

- Le renommage des exceptions: Status_Error, Mode_Error, Name_Error, Use_Error, Device_Error, End_Error, Data_Error

- On retrouve également avec les mêmes paramètres et la même fonctionnalité, les procédures: Create, Open, Close, Delete, Reset (pour ce dernier, toujours avec ses deux formes surchargées).

- Les fonctions: Mode, Name, Form, Is_Open, End_Of_File

- Les lectures/écritures se font respectivement par Read/Write, les 2 procédures sont surchargées, les formes sans les paramètres From, respectivement To permettent des lectures, respectivement des écritures séquentielles à partir de la position courante; après l'opération l'enregistrement courant devient simplement l'enregistrement suivant. La forme avec les paramètres From, respectivement To permettent elles de préciser dans l'opération de lecture, respectivement d'écritue le numéro de l'enregistrement qui doit être traité.

- La procédure Set_Index permet simplement de fixer le numéro de l'enregistrement courant, celui qui sera utilisé lors de la prochaine lecture, respectivement écriture sans la paramètre From, respectivement To.

- La fonction Index livre en retour le numéro de l'enregistrement suivant.

- Finalement la fonction Size nous livre en retour la taille du fichier en nombre d'enregistrements, donc indirectement le numéro du dernier enregistrement.

Ada.Streams et Ada.Stream.Stream_Io:

La notion de Stream peut être approchée par le principe de "flot de données". Jusqu'à présent nos fichiers étaient tous homogènes, c'est-à-dire que tous les enregistrements du fichier contenaient des informations du même type. Le but des « streams » vise à permettre d'enregistrer des données hétérogènes. Toutefois pour que le fichier reste utilisable, non seulement le créateur du fichier doit savoir ce qu'il enregistre, mais aussi l'utilisateur, celui qui relit doit aussi en connaître la structure, ou tout au moins il doit avoir le moyen de la retrouver dans le fichier lui-même.

Les techniques mises en place pour arriver à ce résultat sont relativement complexes, font appel à la programmation objet que nous ne connaissons pas, mais fort heureusement Ada nous offre des outils devant nous faciliter le travail. En fait il y aura des conversions implicites, à l'écriture: d'une donnée structurée d'un type spécifique vers une suite de bytes et inversement à la lecture: d'une suite de bytes vers une donnée structurée du type approprié.

Voici le contenu du paquetage Ada.Streams (mais ne vous attardez pas trop sur le détail):

```
package Ada.Streams is
  pragma Pure(Streams);
  type Root_Stream_Type is abstract tagged limited private;
  type Stream_Element is mod implementation-defined;
  type Stream_Element_Offset is range implementation-defined;
  subtype Stream_Element_Count is
          Stream_Element_Offset range 0..Stream_Element_Offset'Last;
  type Stream_Element_Array is
          array(Stream_Element_Offset range <>) of Stream_Element;

  procedure Read ( Stream : in out Root_Stream_Type;
                   Item   : out Stream_Element_Array;
                   Last   : out Stream_Element_Offset) is abstract;
  procedure Write( Stream : in out Root_Stream_Type;
                   Item   : in Stream_Element_Array) is abstract;
private
  ... -- not specified by the language
end Ada.Streams;
```

Et le contenu du paquetage enfant Ada.Streams.Stream_Io (mais ne vous attardez pas trop sur le détail):

```
with Ada.Io_Exceptions;
package Ada.Streams.Stream_IO is
  type Stream_Access is access all Root_Stream_Type'Class;
  type File_Type     is limited private;
  type File_Mode     is (In_File, Out_File, Append_File );
  type Count is range 0 .. implementation-defined;
  subtype Positive_Count is Count range 1 .. Count'Last;
  -- Index into file, in stream elements.
```

```
  procedure Create ( File : in out File_Type;
                     Mode : in File_Mode := Out_File;
                     Name : in String    := "";
                     Form : in String    := "" );
  procedure Open   ( File : in out File_Type;
                     Mode : in File_Mode;
                     Name : in String;
                     Form : in String := ""  );
  procedure Close  ( File : in out File_Type );
  procedure Delete ( File : in out File_Type );
  procedure Reset  ( File : in out File_Type; Mode : in File_Mode );
  procedure Reset  ( File : in out File_Type );

  function Mode ( File        : in File_Type ) return File_Mode;
  function Name ( File        : in File_Type ) return String;
  function Form ( File        : in File_Type ) return String;
  function Is_Open ( File     : in File_Type ) return Boolean;
  function End_Of_File ( File : in File_Type ) return Boolean;
  function Stream ( File      : in File_Type ) return Stream_Access;
  -- Return stream access for use with T'Input and T'Output

  -- Read array of stream elements from file
  procedure Read ( File : in  File_Type;
                   Item : out Stream_Element_Array;
                   Last : out Stream_Element_Offset;
                   From : in  Positive_Count );
  procedure Read ( File : in  File_Type;
                   Item : out Stream_Element_Array;
                   Last : out Stream_Element_Offset );

  -- Write array of stream elements into file
  procedure Write ( File : in File_Type;
                    Item : in Stream_Element_Array;
                    To   : in Positive_Count );
  procedure Write ( File : in File_Type;
                    Item : in Stream_Element_Array );
  -- Operations on position within file
  procedure Set_Index ( File : in File_Type; To : in Positive_Count );

  function Index  ( File   : in File_Type ) return Positive_Count;
  function Size   ( File   : in File_Type ) return Count;
  procedure Set_Mode( File : in out File_Type; Mode : in File_Mode );
  procedure Flush ( File   : in out File_Type );
  -- exceptions
  Status_Error : exception renames Io_Exceptions.Status_Error;
  Mode_Error   : exception renames Io_Exceptions.Mode_Error;
  Name_Error   : exception renames Io_Exceptions.Name_Error;
  Use_Error    : exception renames Io_Exceptions.Use_Error;
  Device_Error : exception renames Io_Exceptions.Device_Error;
  End_Error    : exception renames Io_Exceptions.End_Error;
  Data_Error   : exception renames Io_Exceptions.Data_Error;

private
  ... -- not specified by the language
end Ada.Streams.Stream_Io;
```

Comme déjà indiqué, tout ceci est bien compliqué, notons toutefois qu'au niveau de la gestion globale des fichiers, tout se passe comme dans les autres paquetages déjà étudiés. Pour introduire les autres aspects, basons nous sur un exemple pour comprendre les points essentiels:

```
with Ada.Text_Io;           use Ada.Text_Io;
with Ada.Integer_Text_Io;   use Ada.Integer_Text_Io;
with Ada.Float_Text_Io;     use Ada.Float_Text_Io;
with Ada.Streams.Stream_Io; use Ada.Streams.Stream_Io;

procedure Stream is

  Fichier : Ada.Streams.Stream_Io.File_Type; -- Fichier de flot
  Pointeur_Sur_Fichier : Stream_Access;
  type T_Tab is array ( Positive range <> ) of Float;
  Passage              : Positive := 1;
  Taille_Tableau       : Natural;

begin  -- Stream

  Put_Line ( " Programme Stream " );
  -- Préparation du fichier pour l'écriture
  Create ( Fichier, Out_File, "Test" );
  Pointeur_Sur_Fichier := Stream ( Fichier );

  -- Traiter tous les tableaux de l'utilisateur
  loop
    Put ( "Donnez la taille du tableau (0 pour terminer): ");
    Get ( Taille_Tableau ); Skip_Line;
    exit when Taille_Tableau = 0; -- L'utilisateur en a assez
    --
    -- Bloc pour créer le tableau
    declare
      subtype T_Tab_Contraint is T_Tab ( 1..Taille_Tableau );
      Le_Tableau : T_Tab_Contraint := ( others => Float (Passage));
    begin
      Integer'Write ( Pointeur_Sur_Fichier, Taille_Tableau );
      T_Tab_Contraint'Write ( Pointeur_Sur_Fichier,Le_Tableau );
      Passage := Passage + 1;
    end;
  end loop;

  Close ( Fichier );

  -- Préparation du fichier pour la relecture
  Open ( Fichier, In_File, "Test" );
  Pointeur_Sur_Fichier := Stream ( Fichier );

  -- Traiter tous les éléments du fichier
  while not End_Of_File ( Fichier )  loop
    Integer'Read ( Pointeur_Sur_Fichier, Taille_Tableau );
```

```
    --
    -- Bloc pour lire le tableau
    declare
      subtype T_Tab_Contraint is T_Tab ( 1..Taille_Tableau );
      Le_Tableau : T_Tab_Contraint;
    begin
      T_Tab_Contraint'Read (  Pointeur_Sur_Fichier, Le_Tableau );
      -- Afficher le tableau lu
      for I in Le_Tableau'Range loop
        Put ( Le_Tableau ( I ),2, 1, 0 );
      end loop;
      New_Line;
    end;
  end loop;

  Put_Line ( " Fin du programme, Pressez <Enter> pour terminer " );
  Skip_Line;

end Stream;
```

Que faut-il en déduire:

- Les fichiers de flots se traitent séquentiellement.
- Le paquetage enfant nous met à disposition:

```
type Stream_Access is access all Root_Stream_Type'Class;
function Stream ( File : in File_Type ) return Stream_Access;
```

La fonction Stream prend un fichier et nous livre un pointeur sur ce fichier, pointeur qui sera utilisé par les attributs mis à disposition pour gérer ces flots. C'est pour cela que dans notre programme, en plus de la variable Fichier du type File_Type, nous avons déclaré la variable:

```
Pointeur_Sur_Fichier : Stream_Access;
```

à laquelle, après une création sous forme usuelle du fichier:

```
Create ( Fichier, Out_File, "Test" );
```

nous associons, par l'intermédiaire de la fonction Stream, un pointeur sur ce fichier:

```
Pointeur_Sur_Fichier := Stream ( Fichier );
```

- Les attributs Read et Write nous permettent respectivement de lire et d'écrire dans le fichier désigné par ce pointeur. Ils correspondent à la définition:

```
procedure  T'Read ( Stream  : access Root_Stream_Type'Class;
                    Item  : out T );

procedure  T'Write ( Stream : access Root_Stream_Type'Class;
                    Item  : in T );
```

Ces attributs existent pour tous les types non limités; c'est pourquoi dans notre programme nous avons enregistré dans notre fichier binaire une valeur entière représentant la taille du tableau et tout le tableau d'éléments réels par les instructions:

```
Integer'Write ( Pointeur_Sur_Fichier, Taille_Tableau );
T_Tab_Contraint'Write ( Pointeur_Sur_Fichier, Le_Tableau );
```

Notons bien que nous avons choisi, dans ce petit exemple artificiel, d'enregistrer sous forme d'un entier la taille, ou plutôt ici le nombre, des éléments du tableau qui va être enregistré ensuite. Les lectures elles étant faites par:

```
Integer'Read ( Pointeur_Sur_Fichier, Taille_Tableau );
…
T_Tab_Contraint'Read ( Pointeur_Sur_Fichier, Le_Tableau );
```

- La fermeture d'un flot se fait de manière usuelle:

```
Close ( Fichier );
```

- Encore quelques compléments:
  - La lecture (respectivement l'écriture) d'un type structuré correspond automatiquement à une lecture de chacun de ses composants; ainsi, pour un type T_Date usuel:

```
type T_Date is record
     Jour  : T_Jour;
     Mois  : T_Mois;
     Annee : T_Annee;
end record;
```

La lecture:

```
T_Date'Read ( Pointeur_Sur_Fichier, La_Date );
```

va correspondre à:

```
T_Jour'Read  ( Pointeur_Sur_Fichier, La_Date.Jour  );
T_Mois'Read  ( Pointeur_Sur_Fichier, La_Date.Mois  );
T_Annee'Read ( Pointeur_Sur_Fichier, La_Date.Annee );
```

  - Les attributs Read et Write peuvent être redéfinis par l'utilisateur; toutefois cette redéfinition ne peut se faire que dans la même spécification que la définition du type lui-même. Ceci implique que nous ne pouvons pas redéfinir ces attributs pour les types de base, mais nous pourrions le faire pour les types dérivés de types de base. Exemple d'une telle redéfinition, dans l'optique d'enregistrer le mois non pas sous la forme d'un objet de type énuméré (T_Mois), mais sous celle d'un numéro de mois:

```
procedure  Ecrire ( Stream : access Root_Stream_Type'Class;
                    Item   : in T_Date );

for  T_Date'Write use Ecrire;     -- Clause d'utilisation

procedure  Ecrire ( Stream : access Root_Stream_Type'Class;
                    Item : in T )is
begin
 T_Jour'Write   ( Stream, Item.Jour );
 Positive'Write ( Stream,
                  T_Mois'Pos ( Item.Mois ) + 1 );
 T_Annee'Write  ( Stream, Item.Annee );
end;
```

Evidemment la lecture doit être adaptée en conséquence si l'on veut obtenir quelque chose de cohérent:

```
procedure  Lire ( Stream : access Root_Stream_Type'Class;
                  Item   : out T_Date );

for  T_Date'Read use Lire;  -- Clause d'utilisation

procedure  Lire ( Stream : access Root_Stream_Type'Class;
                  Item   : out T_Date)is

 Mois_Positif : Positive range 1..12;

begin
 T_Jour'Read   ( Stream, Item.Jour );
 Positive'Read ( Stream, Mois_Positif );
 Item.Mois := T_Mois'Val ( Mois_Positif - 1 );
 T_Annee'Read  ( Stream, Item.Annee );
end;
```

Il faut être prudent sur ce point car, à moins d'être sur une fin de fichier, la lecture pourra toujours se faire (c'est des bits!), mais le résultat sera aléatoire, y compris une valeur non valide pour le type en question.
Dans notre redéfinition, seuls les objets complets de type T_Date seront écrits avec des mois sous forme numérique. Si nous désirons que tous les objets de type T_Mois soient écrits sous cette forme, y compris ceux faisant partie du champ Mois d'un objet de type T_Date, c'est l'attribut Write pour le type T_Mois que nous devons redéfinir:

```
procedure  Ecrire ( Stream : access Root_Stream_Type'Class;
                    Item   : in T_Mois );

for  T_Mois'Write use Ecrire;     -- Clause d'utilisation


procedure  Ecrire ( Stream : access Root_Stream_Type'Class;
                    Item   : in T_Mois )is
begin
 Positive'Write ( Stream, T_Mois'Pos ( Item ) + 1 );
end;
```

Modifions partiellement notre programme exemple pour illustrer cette possibilité:

```
with Ada.Text_Io;              use Ada.Text_Io;
with Ada.Integer_Text_Io;      use Ada.Integer_Text_Io;
with Ada.Float_Text_Io;        use Ada.Float_Text_Io;
with Ada.Streams.Stream_Io;    use Ada.Streams.Stream_Io;

procedure Stream is

  Fichier : Ada.Streams.Stream_Io.File_Type; -- Fichier de flot
  Pointeur_Sur_Fichier : Stream_Access;
  type T_Tab is array ( Positive range <> ) of Float;
  Passage              : Positive := 1;
  Taille_Tableau       : Natural;

  ------------------------------------------------------------------
  -- Pour la redéfinition de l'attribut d'écriture
  procedure Ecrire ( Stream : access Ada.Streams.Root_Stream_Type'Class;
                     Item   : in T_Tab );
  for T_Tab'Write use Ecrire;

  procedure Ecrire ( Stream : access Ada.Streams.Root_Stream_Type'Class;
                     Item   : in T_Tab )is
  begin
    for I in Item'range loop
      Float'Write ( Stream, Item ( I ) * 10.0 );
    end loop;
  end;

begin  -- Stream

  Put_Line ( " Programme Stream " );
  -- Préparation du fichier pour l'écriture
  Create ( Fichier, Out_File, "Test" );
  Pointeur_Sur_Fichier := Stream ( Fichier );

  -- Traiter tous les tableaux de l'utilisateur
  loop
    Put ( "Donnez la taille du tableau (0 pour terminer): " );
    Get ( Taille_Tableau ); Skip_Line;
    exit when Taille_Tableau = 0; -- L'utilisateur en a assez
    --
    -- Bloc pour créer le tableau
    declare
      subtype T_Tab_Contraint is T_Tab ( 1..Taille_Tableau );
      Le_Tableau : T_Tab_Contraint := ( others => Float ( Passage ) );

    begin
      Integer'Write ( Pointeur_Sur_Fichier, Taille_Tableau );
      T_Tab'Write   ( Pointeur_Sur_Fichier, Le_Tableau );
      Passage := Passage + 1;
    end;
  end loop;

  Close ( Fichier );
```

```
  -- Préparation du fichier pour la relecture
  Open ( Fichier, In_File, "Test" );
  Pointeur_Sur_Fichier := Stream ( Fichier );

  -- Traiter tous les éléments du fichier
  while not End_Of_File ( Fichier )  loop
    Integer'Read ( Pointeur_Sur_Fichier, Taille_Tableau );
    --
    -- Bloc pour lire le tableau
    declare
      subtype T_Tab_Contraint is T_Tab ( 1..Taille_Tableau );
      Le_Tableau : T_Tab_Contraint;
    begin
      T_Tab_Contraint'Read (  Pointeur_Sur_Fichier, Le_Tableau );
      -- Afficher le tableau lu
      for I in Le_Tableau'Range loop
        Put ( Le_Tableau ( I ),4, 1, 0 );
      end loop;
      New_Line;
    end;
  end loop;

  Put_Line ( " Fin du programme, Pressez <Enter> pour terminer " );
  Skip_Line;

end Stream;
```

Nous pouvons encore introduire une autre forme d'utilisation, les attributs Input et Output liés aux flots, mais pour l'utilisation d'objets non contraints. Toutefois ils sont aussi utilisables pour les objets contraints. Nous aurions pu en faire usage dans la première version de notre programme exemple, mais ce ne serait pas réellement une bonne solution. Modifions quelque peu ce programme exemple pour illustrer ces autres possibilités:

```
with Ada.Text_Io;           use Ada.Text_Io;
with Ada.Integer_Text_Io;   use Ada.Integer_Text_Io;
with Ada.Float_Text_Io;     use Ada.Float_Text_Io;
with Ada.Streams.Stream_Io; use Ada.Streams.Stream_Io;

procedure Stream is

   Fichier : Ada.Streams.Stream_Io.File_Type; -- Fichier de flot
   Pointeur_Sur_Fichier : Stream_Access;
   type T_Tab is array ( Positive range <> ) of Float;
   Passage : Positive := 1;

begin  -- Stream

  Put_Line( " Programme Stream " );

  -- Préparation du fichier pour l'écriture
  Create ( Fichier, Out_File, "Test" );
  Pointeur_Sur_Fichier := Stream ( Fichier );
```

```
  -- Traiter tous les tableaux de l'utilisateur
  for Taille_Tableau in 1..4 loop
    --
    -- Bloc pour créer le tableau
    declare
      subtype T_Tab_Contraint is T_Tab ( 1..Taille_Tableau );
      Le_Tableau : T_Tab_Contraint := ( others => Float(Passage));
    begin
      T_Tab'Output (  Pointeur_Sur_Fichier,Le_Tableau );
      Passage := Passage + 1;
    end;
  end loop;

  -- Préparation du fichier pour la relecture
  Reset ( Fichier, In_File );

  -- Traiter tous les éléments du fichier
  while not End_Of_File ( Fichier ) loop
    --
    -- Bloc pour lire le tableau
    declare
      Le_Tableau : T_Tab := T_Tab'Input (  Pointeur_Sur_Fichier );
    begin
      -- Afficher le tableau lu
      for I in Le_Tableau'Range loop
        Put ( Le_Tableau ( I ),2, 1, 0 );
      end loop;
      New_Line;
    end;
  end loop;

  Put_Line ( " Fin du programme, Pressez <Enter> pour terminer " );
  Skip_Line;

end Stream;
```

Notez bien que si l'attribut Output réagit comme une procédure:

```
T_Tab'Output (  Pointeur_Sur_Fichier,Le_Tableau );
```

L'attribut Input lui a le comportement d'une fonction:

```
Le_Tableau := T_Tab'Input (  Pointeur_Sur_Fichier );
```

Ceci s'explique par le fait que le résultat de la lecture étant non contraint, on ne connaît pas la taille résultante!
En écriture: pour un tableau Output enregistre d'abord les bornes dans le fichier puis appelle Write (utilise l'attribut!) pour écrire le tableau proprement dit; pour un article à discriminant, le discriminant ou les discriminants sont enregistrés en premier dans le fichier, puis les composants de l'article.
La forme et la valeur retournées par Input dépendent des bornes ou des discriminants.

Ada.Text_Io.Text_Streams

Finalement, il est possible de mélanger un flot (binaire) dans un fichier texte; dans cette optique, Ada nous met à disposition le paquetage enfant de Ada.Text_Io, Ada.Text_Io.Text_Streams, dont voici la spécification très simple:

```
with Ada.Streams;
package Ada.Text_IO.Text_Streams is
  type Stream_Access is access all
                Streams.Root_Stream_Type'Class;
  function Stream (File : in File_Type)
                                 return Stream_Access;
end Ada.Text_IO.Text_Streams;
```

Les fonctionnalités au niveau flot sont les même que pour Ada.Streams.Stream_Io et, pour la partie texte, les mêmes que dans Ada.Text_Io; toutefois, étant donnée le mélange que l'on va faire de binaire et de texte, les notions telles que le numéro de ligne ou de colonne n'a plus de sens!

Donnons encore une dernière version modifiée de notre petit programme de démonstration pour illustrer ces nouvelles possibilités:

```
with Ada.Text_Io;                use Ada.Text_Io;
with Ada.Integer_Text_Io;        use Ada.Integer_Text_Io;
with Ada.Float_Text_Io;          use Ada.Float_Text_Io;
with Ada.Text_Io.Text_Streams;   use Ada.Text_Io.Text_Streams;

procedure Stream is

  Fichier : Ada.Text_Io.File_Type;      -- Variable fichier de flot
  Pointeur_Sur_Fichier : Stream_Access;
  type T_Tab is array ( Positive range <> ) of Float;
  Passage              : Positive := 1;
  Taille_Tableau       : Natural;
  Longueur_Ligne       : constant := 80;
  Ligne : String ( 1..Longueur_Ligne ); -- Pour les parties texte
  Longueur_Courante    : Natural;       -- Longueur lue

begin  -- Stream

  Put_Line ( " Programme Stream " );

  -- Préparation du fichier pour l'écriture
  Create ( Fichier, Out_File, "Test.txt" );
  Pointeur_Sur_Fichier := Stream ( Fichier );

  -- Traiter tous les tableaux de l'utilisateur
  loop
    Put ( "Donnez la taille du tableau (0 pour terminer): ");
    Get ( Taille_Tableau ); Skip_Line;
    exit when Taille_Tableau = 0; -- L'utilisateur en a assez
```

```
    --
    -- Bloc pour créer le tableau
    declare
      subtype T_Tab_Contraint is T_Tab ( 1..Taille_Tableau );
      Le_Tableau : T_Tab_Contraint := ( others => Float(Passage));
    begin
      Put_Line(Fichier,"Element suivant "&Natural'Image(Passage));
      Integer'Write ( Pointeur_Sur_Fichier, Taille_Tableau );
      T_Tab_Contraint'Write (  Pointeur_Sur_Fichier, Le_Tableau );
      Passage := Passage + 1;
    end;
  end loop;

  Close ( Fichier );

  -- Préparation du fichier pour la relecture
  Open ( Fichier, In_File, "Test.txt" );
  Pointeur_Sur_Fichier := Stream ( Fichier );

  -- Traiter tous les éléments du fichier
  while not End_Of_File ( Fichier )  loop
    Get_Line ( Fichier, Ligne, Longueur_Courante );
    Put_Line ( Ligne ( 1..Longueur_Courante ) );
    Integer'Read ( Pointeur_Sur_Fichier, Taille_Tableau );
    --
    -- Bloc pour créer le tableau
    declare
      subtype T_Tab_Contraint is T_Tab ( 1..Taille_Tableau );
      Le_Tableau : T_Tab_Contraint;
    begin
      T_Tab_Contraint'Read (  Pointeur_Sur_Fichier,Le_Tableau );
      -- Afficher le tableau lu
      for I in Le_Tableau'Range loop
        Put ( Le_Tableau ( I ),2, 1, 0 );
      end loop;
      New_Line;
    end;
  end loop;

  Put_Line ( " Fin du programme, Pressez <Enter> pour terminer " );
  Skip_Line;

end Stream;
```

A noter simplement qu'il n'y a plus de clause de contexte pour Ada.Streams.Stream_Io, mais par contre il y a la clause:

```
with Ada.Text_Io.Text_Streams; use Ada.Text_Io.Text_Streams;
```

et le mélange des attribut Write et Read (binaires) avec les Put_Line et Get_Line (texte) qui accèdent au même fichier.

Complément au type access

Partons des déclarations suivantes:

```
type Cellule;
type Pointeur is access Cellule;
type Pointeur_2 is access Cellule;
type Cellule is
        record
                Les_Valeurs : Info;
                Lien : Pointeur;
        end record;

Tete, Autre_Tete : Pointeur;
Tete_2 : Pointeur_2;
```

- Si nous avons dans les instructions:

```
if Tete = Autre_Tete then
        ...
```

nous nous demandons si les 2 pointeurs désignent la même cellule; par contre si nous désirons savoir si deux cellules pointées contiennent les mêmes informations, nous écrirons:

```
if  Tete.all = Autre_Tete.all then
        ...
```

la comparaison se faisant sur l'ensemble des champs.

Et si la comparaison d'un seul champ nous intéresse:

```
if  Tete.Lien = Autre_Tete.Lien then
        ...
```

qui signifierait ici : "Est-ce que les 2 liens pointent sur la même cellule?".

Nous ne pouvons **pas** écrire:

```
Tete_2 := Tete;
```

les types des 2 pointeurs n'étant pas identiques, mais par contre nous pouvons écrire:

```
Tete_2.all := Tete.all;
```

les variables pointées étant de type identique.

Bien qu'à première vue cela puisse paraître étrange, nous pouvons définir une constante de type **access**:

```
Tete_Vide : constant Pointeur := new Cellule'( 0, null );
```

Le pointeur désignera toujours la même cellule, donc ne pourra pas changer de valeur, mais le contenu de la variable pointée lui peut changer. Nous avons donc tout à fait le droit d'écrire:

```
Tete_Vide.Les_Valeurs := 33;
```

ou

```
Tete_Vide.Lien := Tete_Vide;
```

ou encore:

```
Tete_Vide.all := Tete.all;
```

Type access et tableaux:

Si nous avons un type tableau non contraint:

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

Nous pouvons déclarer un type **access** sur un tel type non contraint, ainsi que des variables de ce type:

```
type Pt_Vecteur is access T_Vecteur;
Pointeur : Pt_Vecteur;
```

Nous pouvons alors créer dynamiquement de tels objets:

```
Pointeur := new T_Vecteur ( 1..10 );
```

Ou, si nous voulons initialiser l'objet créé:

```
Pointeur := new T_Vecteur'( 3..9 => 0 );
```

ici les bornes sont déduites de l'expression d'initialisation.

Par contre les bornes devant être connues au moment de la création de l'objet, nous ne pouvons **pas** écrire:

```
Pointeur := new T_Vecteur;
```

L'accès aux éléments d'un tel tableau se fait de manière usuelle, la déréférençage étant automatique:

```
Pointeur (4) := 5;
```

mais nous pouvons aussi utiliser la forme complète:

```
Pointeur.all (4) := 5;
```

Par contre si nous voulons désigner tout le tableau, le **.all** est obligatoire:

```
Pointeur.all := ( 3..9 => 0 );
```

Nous pouvons aussi créer des sous-types de notre type Pt_Vecteur, permettant d'accéder uniquement à des vecteurs de dimension fixée, par exemple:

```
subtype Pt_Vecteur2 is Pt_Vecteur ( 1..2 );
Pointeur2 : Pt_Vecteur2;
```

Pointeur2 ne pourra contenir que l'adresse d'un vecteur à 2 éléments!

Un exemple limite, mais pratique tiré de J. Barnes:

```
type Une_Chaine is access String;

function "+" ( S : String ) return Une_Chaine is
begin
    return new String' ( S );
end "+";

type Tableau_Une_Chaine is array ( Positive range <> ) of Une_Chaine;

Zoo : constant Tableau_Une_Chaine := ( +"antilope", +"babouin", …, +"zebre" );
```

Nous avons ainsi, par un artifice, construit un tableau de chaînes de longueurs variables!

Ce qu'il ne faut pas faire:

Un objet du type access ne peut pas contenir (à priori) l'adresse d'un objet dont la durée de vie serait plus courte que celle du pointeur, car alors on risquerait d'utiliser une référence à un objet qui n'existe plus! A titre d'illustration de ce propos, l'extrait de code ci-après n'est pas correcte:

```
type Pt is access all Float;
Pointeur : Pt;
…
declare
  R : aliased Float;
  …
begin
  …
  Pointeur := R'Access;
  …
```

Toutefois si l'utilisateur est certain de ce qu'il fait et qu'il est convaincu qu'il ne risque pas de se mettre dans une situation dangereuse, il peut utiliser l'attribut Unchecked_Access en lieu et place de Access, pour lever les contrôles qui sont réalisés par le compilateur:

```
Pointeur := R'Unchecked_Access;
```

mais dans ce cas il ne faudra pas qu'il écrive, après le **end** du bloc interne:

```
Pointeur.all := 3.2;
```

Car alors il y aurait de grands risques qu'il détruise d'autres objets de son programme!!!

Accès à des sous-programmes, complément:

a) Unchecked_Access ne peut pas s'appliquer à des sous-programmes, ceci deviendrait bien trop dangereux!

b) On peut construire des tableaux contenant des adresses de sous-programmes, ceci par exemple pour permettre la réalisation d'une séquence d'opérations répétitives:

```
Nb_Opérations : constant := …;
type T_Operation is access procedure;  -- s'il n'y a pas de paramètre
Les_Operations : array ( 1..Nb_Operation ) of T_Operation;
```

Après initialisation du tableau avec les adresses des sous-programmes désirés, on peut imaginer la séquence d'instructions:

```
for I in Les_Operations'Range loop
  ….
  Les_Operations (I).all;
  …
end loop;
```

c) De même un champ d'article peut lui aussi être un pointeur sur un sous-programme, ce qui permet par exemple d'associer une action à un objet!!

Paramètres access

Lorsque nous avons étudié les sous-programmes, nous avons dit qu'il existait 3 modes de passage des paramètres: **in**, **out** et **in out**; en réalité il existe une autre possibilité: **access**.

Bien que cette possibilité offre un intérêt réel lorsqu'elle est associée à la programmation objet, donnons ici un exemple et les règles principales d'utilisation, ce qui nous permettra entre autre de montrer comment détourner le fait qu'une fonction ne peut pas avoir de paramètre de sortie, mais attention, cela ne veut pas dire qu'il faut abuser de cette possibilité, car rappelons le, dans ce cas les règles élémentaires de l'arithmétique (commutativité, associativité) ne sont plus vraies...

Exemple:

```
-- Utilisation de paramètres access, quelques "mauvais" exemples!
with Ada.Text_Io;              use Ada.Text_Io;
with Ada.Integer_Text_Io;      use Ada.Integer_Text_Io;
procedure Access_Sp is

  N : aliased Natural := 5;      -- Valeur que l'on veut modifier
  type Pt_Natural is access all Natural;

  -- Fonction "pas focément belle"
  function Douteuse ( Pt : access Natural ) return Natural is
                      Local : Pt_Natural := Pt_Natural (Pt);
  begin  -- Douteuse

    New_Line;
    Put ( "Dans la foncttion, avant modification:" );
    Put ( Pt.all );
    New_Line;
    Pt.all := Pt.all * 10;
    return Local.all;

  end Douteuse;

begin  -- Access_Sp

  Put_Line ("Exemple de parametre access ");
  Put ( "N avant l'appel:" );
  Put ( N );
  New_Line;
  Put ( "Le resultat de la fonction:" );
  Put ( Douteuse ( N'Access ));
  New_Line;
  Put ( "N apres l'appel:" );
  Put ( N );
  New_Line;

end Access_Sp;
```

Et oui, cela fonctionne et donne comme résultat:

```
Exemple de parametre access
N avant l'appel:      5
Le resultat de la fonction:
Dans la foncttion, avant modification:      5
     50
N apres l'appel:      50
```

Tout d'abord, le "mode" utilisé est donc **access**:

```
function Douteuse ( Pt : access Natural ) return Natural is
```

notons bien que cela implique un type anonyme, ce qui nous oblige dans la déclaration de notre variable locale:

```
Local : Pt_Natural := Pt_Natural (Pt);
```

à utiliser une conversion de type pour lui affecter la valeur du paramètre.

Dans le sous-programme nous pouvons utiliser l'objet désigné par les pointeurs:

```
Put ( Pt.all );
Pt.all := Pt.all * 10;
return Local.all;
```

ici nous sommes obligés d'utiliser le .**all** puisque nous nous référons à un objet simple; par contre si notre type access désignait un article, nous pourrions l'utiliser sans cela!

Signalons encore quelques points:

a) Nous ne pouvons pas passer un pointeur **null** comme paramètre effectif, ceci est contrôlé automatiquement, ce qui nous dispense de faire explicitement un tel contrôle dans nos sous-programmes.
b) Le paramètre **access** est considéré comme une constante, nous ne pouvons pas modifier sa valeur dans le sous-programmes, mais comme nous avons pu le constater, nous pouvons modifier l'objet désigné par ce pointeur.
c) Ce que nous venons de présenter pour les paramètres de fonctions, s'applique aussi bien entendu aux paramètres des procédures.
d) Nous n'avions pas besoin du mode access pour réaliser cette opération; nous aurions déjà pu, par le passé, utiliser un pointeur au sens habituel.

Type accès et discriminants

Les objets de type **access** peuvent pointer sur des objets à discriminants. La pré déclaration du type de l'objet pointé doit préciser aussi le ou les discriminants.

Reprenons en le modifiant quelque peu l'exemple du type bateau que nous avons déjà utilisé pour les articles à discriminants:

```
type T_Bateau  ( Classe : Genre );

type T_Lien is access T_Bateau;

type T_Bateau ( Classe: Genre ) is
    record
        Suivant              : T_Lien;   -- Pour chaîner les bateaux
        Longueur             : Foat;     -- Longueur du bateau
        Nombre_De_Places : Integer range 0 .. Nb_Max_ De_Places;
        --
        -- Partie variante dépendant de la classe du bateau
        --
        case Classe is
        ...                  -- Pas de changements
        end case;

    end record;
```

On constate, dans cette nouvelle forme, par rapport à l'ancienne, que l'on a ajouté un champ Suivant de type T_Lien, ceci dans l'idée que l'on pourrait par exemple gérer un port sous la forme d'une liste simplement chaînée.

Ensuite, on a supprimé la valeur initiale de défaut du discriminant Classe; ceci n'est pas indispensable, mais il faut bien comprendre que son utilité n'a plus d'importance dans ce contexte. Lorsque nous demandons des objets de ce type par l'allocateur **new**, nous devons explicitement donner la valeur de la contrainte:

- soit pour elle-même:

```
Un_ Bateau: T_Lien:= new T_Bateau  ( Moteur );
```

- soit par l'intermédiaire d'une expression d'initialisation:

```
Un_ Bateau : T_Lien := new T_Bateau ' ( Classe => Rame,
                                        Suivant => null,
                                        Longueur => 7.2,
                                        Nombre _ De_Places => 6 );
```

Notons encore une fois la différence entre les 2 formes ci-dessus, la deuxième comporte l'apostrophe puisqu'il s'agit en fait d'une expression qualifiée.

Il faut savoir, c'est le point essentiel de ce paragraphe, qu'un objet créé par l'allocateur **new** ne peut jamais voir la valeur de son discriminant changer, même par une affectation globale, ce qui n'est pas le cas des variables habituelles.

En pratique, on passera souvent par une déclaration de sous-type pour les différentes classes de bateaux:

```
subtype T_Bateau _A_Voile     is T_Bateau  ( Voile  );

subtype  T_Bateau _A_ Moteur is T_Bateau  ( Moteur );
```

...etc

On peut alors écrire:

```
Un_ Bateau : T_Lien:= new T_Bateau _A_ Voile;
```

ou encore:

```
Un_ Bateau : T_Lien:= new T_Bateau _A_ Moteur (Moteur, null, 6.8, 8, 70 );
```

mais jamais:

```
Un_ Bateau: T_Lien := new T_Bateau;
```

Gestion du temps - CALENDAR

Les problèmes de gestion du temps sont très étroitement liés à la programmation en temps réel et seront développés dans le cadre du cours qui concerne ces aspects. Ici nous allons nous contenter de donner quelques notions.

Delay

En Ada nous pouvons temporairement suspendre un programme (ou une tâche) par l'instruction:

delay Temps;

Temps est une expression de type Duration qui est en fait un type point fixe, défini dans le paquetage Standard; ces caractéristiques dépendent de l'implémentation mais Ada impose:

- Que l'on puisse définir un intervalle de plus ou moins un jour (+/-86400 secondes, la valeur étant exprimée en secondes!).
- Que la plus petite valeur positive ne soit pas supérieure à 20 millisecondes; sur tout système cette valeur s'obtient par:

  Duration'Small

Ainsi l'instruction:

delay 2.5;

suspend le programme pour deux secondes et demi au moins.

Note:

Pour suspendre temporairement un programme, **delay** est préférable à une boucle d'attente active, car il n'y aura pas ainsi de temps machine utilisé inutilement, et le nombre d'itérations nécessaires ne dépendra pas des caractéristiques du processeur.

Il existe une deuxième forme de l'instruction **delay**:

delay until Temps;

- Dans cette deuxième forme Temps est une expression du type Time qui vient du paquetage Ada.Calendar.

Exemple d'utilisation:

```
declare
    use Ada.Calendar;
    Next_Time : Time := Clock + Period;       -- Period de type Duration
begin
    loop          -- Répéter chaques Period secondes
      delay until Next_Time;
      ... -- opérations
      Next_Time := Next_Time + Period;
    end loop;
end;
```

Note: Dans certains cas l'exception Time_Error peut être levée si la durée donnée pour un **delay** dépasse 90 jours!

Paquetage CALANDAR

Le paquetage CALENDAR nous aide dans certains problèmes liés à la gestion du temps. En voici la spécification:

```
package Ada.Calendar is

  type Time is private;
  subtype Year_Number  is Integer range 1901 ..  2099;
  subtype Month_Number is Integer range 1 ..  12;
  subtype Day_Number   is Integer range 1 ..  31;
  subtype Day_Duration is Duration range 0.0 ..  86_400.0;
  function Clock return Time;
  function Year    ( Date : Time ) return Year_Number;
  function Month   ( Date : Time ) return Month_Number;
  function Day     ( Date : Time ) return Day_Number;
  function Seconds ( Date : Time ) return Day_Duration;

  procedure Split  ( Date      : in  Time;
                     Year      : out Year_Number;
                     Month     : out Month_Number;
                     Day       : out Day_Number;
                     Seconds   : out Day_Duration );
  function Time_Of ( Year      : Year_Number;
                     Month     : Month_Number;
                     Day       : Day_Number;
                     Seconds : Day_Duration := 0.0 ) return Time;

  function "+"  ( Left : Time;     Right : Duration ) return Time;
  function "+"  ( Left : Duration; Right : Time     ) return Time;
  function "-"  ( Left : Time;     Right : Duration ) return Time;
  function "-"  ( Left : Time;     Right : Time ) return Duration;
  function "<"  ( Left, Right  : Time ) return Boolean;
  function "<=" ( Left, Right  : Time ) return Boolean;
```

```
  function ">"  ( Left, Right  : Time ) return Boolean;
  function ">=" ( Left, Right  : Time ) return Boolean;

  Time_Error : exception;
private
  ... -- not specified by the language
end Ada.Calendar;
```

On peut remarquer les points suivants:

- Définition du type Time (privé) avec des sous-types pour l'année, le mois, le jour et le nombre de secondes dans le jour; Time en lui-même comporte toutes ces informations.

- La fonction Clock qui rend la valeur courante de "Time".

- Les 4 fonctions suivantes (Year, Month, Day, Seconds) permettent d'extraire d'un objet de type TIME les informations relatives respectivement à l'année, au mois, au jour et au nombre de secondes dans le jour.

- La procédure Split permet, à partir d'un objet de type Time, d'obtenir directement les 4 informations : année, mois, jour et secondes.

- Inversement la fonction Time_Of rend une valeur de type Tme à partir des 4 valeurs de base (année, mois, jour et secondes) transmises en paramètres.

- Vient ensuite la définition des opérations "+" et "-" entre des objets de types Time et Duration (à noter que toutes les combinaisons de types des opérandes ne sont pas raisonnables).

- Définition des opérateurs de relation ("<", "<=", ">", ">=") entre des opérandes de type Time. Time étant privé (et non limité privé), les opérations "=", et "/=" sont disponibles par définition (de même que l'affectation).

- Finalement, définition de l'exception Time_Error qui peut être levée par exemple lors d'un appel à Time_Of avec des paramètres ne permettant pas de construire une date valable!

- La spécification se termine par la partie privée, dépendante de l'implémentation.

Voici pour terminer un petit programme permettant d'illustrer certains points vus ci-dessus.

Il affiche simplement la date et l'heure courante à l'écran.

```
--
-- Exemple d'utilisation de certsins outils de gestion du temps.
-- Affiche la date et l'heure courante à l'aide de Ada.Calendar
--
with Ada.Text_Io;             use Ada.Text_Io;
with Ada.Integer_Text_Io;     use Ada.Integer_Text_Io;
with Ada.Calendar;            use Ada.Calendar;
procedure Date is
   --
   --- Pour les entrées/sorties du type DDuration
   package Day_Duration_Io is new Fixed_Io ( Day_Duration );
   use Day_Duration_Io;
   --
   -- Le type MOIS et ses entrees/sorties
   type Mois is ( Janvier, Fevrier, Mars, Avril, Mai, Juin, Juillet,
                  Aout, Septembre, Octobre, Novembre, Decembre );
   --
   -- Pour l'affichage du mois
   package Mois_Io is new Enumeration_Io ( Mois ); use Mois_Io;
   --
   -- Les differentes variables pour traiter la date
   Date_Courante : Ada.Calendar.Time ;  -- La date rendue par le système
   L_Année : Ada.Calendar.Year_Number;  -- Décomposition de la date
   Le_Mois : Ada.Calendar.Month_Number; --    "
   Le_Jour : Ada.Calendar.Day_Number;   --    "
   L_Heure : Integer range 0 .. 23;     -- ... et de l'heure
   Les_Minutes  : Integer range 0 .. 59;      --       "
   Les_Secondes : Ada.Calendar.Day_Duration;  --       "
   --
   -- Pour transformer une DURATION en heures, minutes et secondes
   Minute : constant := 60;         -- 1 minute = 60 secondes
   Heure : constant := 3_600;       -- 1 heure = 3600 secondes

begin   -- Date
   --
   -- Demander la date courante au système
   Date_Courante := Clock;
   --
   -- Décomposer la date courante
   Split ( Date_Courante, L_Année, Le_Mois, Le_Jour, Les_Secondes );
   --
   -- Extraire les heures et mettre à jour les secondes
   L_Heure      := Integer ( Les_Secondes ) / Heure ;
   Les_Secondes := Les_Secondes - Day_Duration ( L_Heure * Heure );
   --
   -- Extraire les minutes et mettre à jour les secondes
   Les_Minutes := Integer ( Les_Secondes ) / Minute ;
   Les_Secondes :=
             Les_Secondes - Day_Duration ( Les_Minutes * Minute );
   --
   -- Afficher la date
   Put ( Le_Jour, 3 );
   Put ( ' ' );
   Put ( Mois'Val ( Le_Mois - 1 ) );  -- Affiche le mois en clair
   Put ( L_Année, 5 );
   New_Line;
```

```
   --
   -- Affiche l'heure
   Put ( L_Heure, 3 );
   Put ( " : " );              --  Séparateur et traiter les minutes
   if Les_Minutes < 10 then    -- 0 de tête nécessaire?
     Put ( '0' );              -- Pour avoir toujours 2 chiffres
     Put ( Les_Minutes, 1 );
   else
     Put ( Les_Minutes, 2 );
   end if;
   Put ( " : " );                -- Séparateur et traiter les secondes
   if Les_Secondes < 10.0 then   -- 0 de tête necessaire?
     Put ( '0' );                -- Pour avoir toujours 2 chiffres
     Put ( Les_Secondes, 1, 2 );
   else
     Put ( Les_Secondes, 2, 2 );
   end if;
   New_Line;

end Date;
```

Complements sur les generiques:

Généralités

Objet générique:

Moule non utilisable directement.

With…;
Pas de **use**!

Paquetage
Procédure
Fonction

Les paramètres:

- Objets
- Types
- Sous-programmes
- Paquetages

Instanciation:

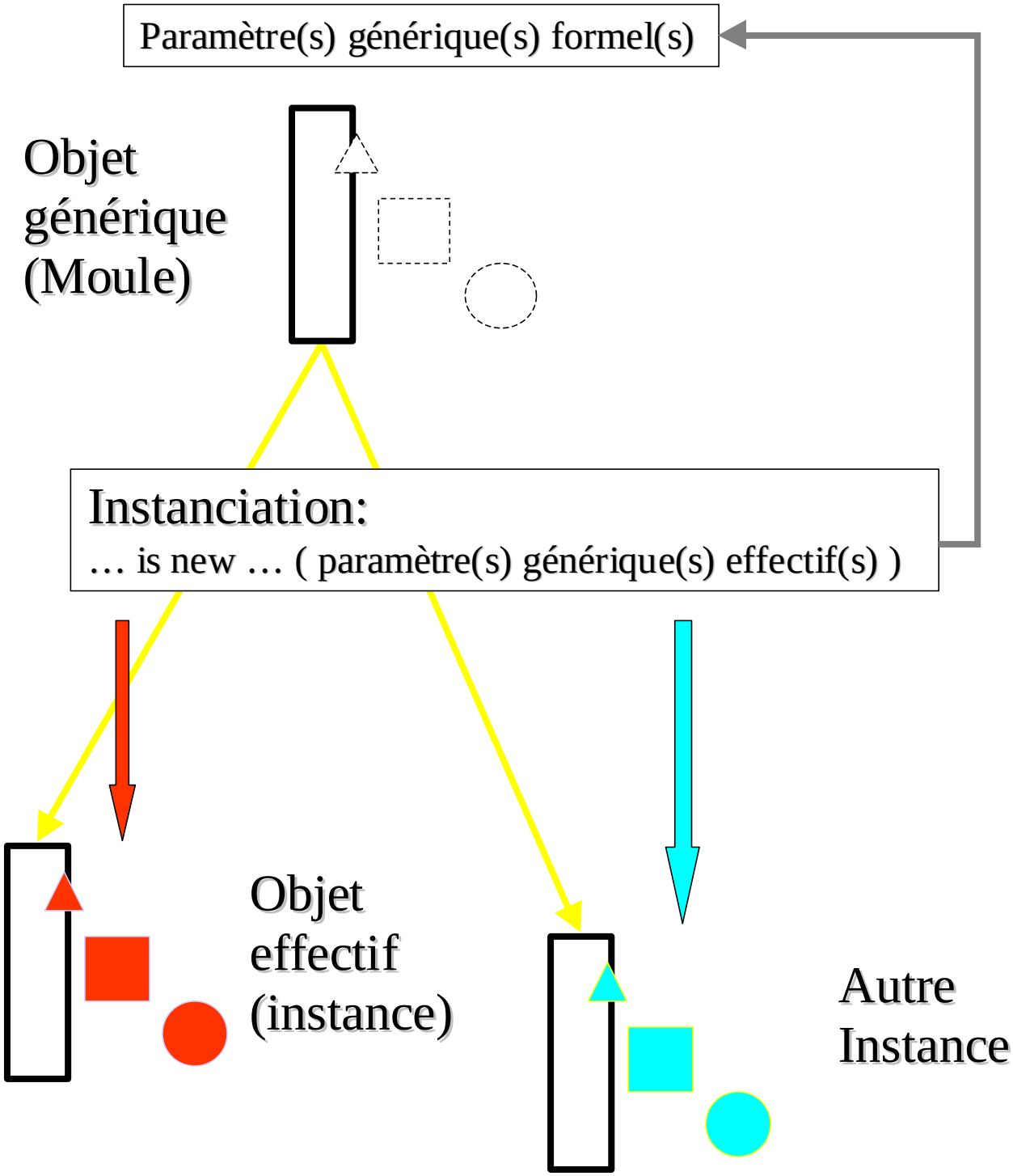
transforme le module, en fonction des paramètres de la généricité, en un objet utilisable.

Eventuellement un **use**…;

Objet Utilisable

package
procesdure
function

Nom **is new** Nom_Du_Generique (parametres effectifs);


Paramètre(s) générique(s) formel(s)
Objet
générique
(Moule)
Instanciation:
… is new … (paramètre(s) générique(s) effectif(s))
Objet
effectif
(instance)
Autre
Instance

- Paramétrage réalisé à la compilation.
- La logique est souvent indépendante des types.
- Exemple simple:

```
procedure Echange ( V1, V2 : in out T_Info ) is

  Temp : T_Info := V1;

begin -- Echange

  V1 := V2 ;
  V2 := Temp;

end Echange;
```

- **On rend facilement une telle procédure générique:**

```
generic
   type T_Info is private;
procedure Echange ( V1, V2 : in out T_Info );

procedure Echange ( V1, V2 : in out T_Info ) is

  Temp : T_Info := V1;

begin -- Echange

  V1 := V2 ;
  V2 := Temp;

end Echange;
```

- Et pour utiliser ce moule:

```
procedure Echanger is new Echange ( Integer );
procedure Echanger is new Echange ( T_Info => T_Date );
```

- **Une unité générique peut utiliser une autre unité elle même générique:**

```
with Echange;
generic
  type T is private;
procedure Double ( V1, V2, V3 : in out T );

procedure Double ( V1, V2, V3 : in out T ) is

  procedure Echanger is new Echange ( T_Info => T );

begin -- Double

  Echanger ( V1, V2 );
  Echanger ( V1, V3 );

end Double ;
```

Paquetages comme paramètres génériques

Les paquetages représentent la dernière possibilité de paramètres génériques. Ici, le fait de passer un paquetage complet en paramètre va permettre d'éviter une (longue) liste de paramètres génériques formée d'entités correspondant à celles exportées du paquetage. Un cas simple est constitué des opérations d'entrées-sorties sur des valeurs d'un type entier comme paramètre générique, illustré dans l'exemple ci-dessous. Le type entier `T_Entier` est un paramètre générique pour que le paquetage puisse fournir des nombres rationnels dont les numérateur et dénominateur sont d'un ordre de grandeur défini par l'utilisateur du paquetage. Indépendamment des entrées-sorties, la généricité de `T_Entier` a nécessité l'introduction du sous-type `T_Entier_Positif` pour assurer que tous les dénominateurs des nombres rationnels seront strictement positifs.

Spécification (partielle) d'un paquetage générique de gestion de nombres rationnels:

```
-- Paquetage générique de calcul avec les nombres rationnels
with Ada.Text_Io;      use Ada.Text_Io;
generic
  type T_Entier is range <>;
  with package Entier_Io is new Integer_Io ( T_Entier );
package Nombres_Rationnels_G is

  type T_Rationnel is private;     -- Le type privé
  Zéro : constant T_Rationnel;     -- Une constante différée
  Division_Par_0 : exception;      -- Exception si division par 0
  -- Pour que le denominateur soit toujours positif
  subtype T_Entier_Positif is T_Entier range 1..T_Entier'Last;
  ------------------------------------------------------------
  -- Construction d'un nombre rationnel
  function "/" ( Numerateur : T_Entier;
                 Denominateur:T_Entier_Positif ) return T_Rationnel;
  ------------------------------------------------------------
  -- Autres operations…    ...
  ------------------------------------------------------------
  -- Lecture d'un nombre rationnel
  procedure Get ( X : out T_Rationnel );
  ------------------------------------------------------------
  -- Ecriture d'un nombre rationnel
  procedure Put ( X : in T_Rationnel );
  ------------------------------------------------------------
private
  type T_Rationnel is                  -- Déclaration complete du type
    record
      Numerateur : T_Entier;
      Denominateur : T_Entier_Positif; -- Le signe au numérateur
    end record;
  Zéro : constant T_Rationnel:=(0, 1); -- La constante complète

end Nombres_Rationnels_G;
```

`Entier_Io` est le paquetage passé comme paramètre générique et va permettre l'implémentation des corps des procédures `Get` et `Put`. Il remplace avantageusement deux autres paramètres: les procédures de lecture et d'écriture d'une valeur entière nécessaires dans `Get` et `Put`.

De manière générale, le ou les paramètres effectifs (T_Entier dans l'exemple) mentionnés dans la déclaration du paquetage passé comme paramètre générique (Integer_Io dans l'exemple) peuvent naturellement être différents du ou des éventuels paramètres génériques du paquetage en cours de définition (Nombres_Rationnels_G dans l'exemple).

Corps (partiel) d'un paquetage générique de gestion de nombres rationnels:

```
-- Ce paquetage générique permet le calcul avec les nombres rationnels
package body Nombres_Rationnels_G is
  use Entier_IO; -- Evite de prefixer Get et Put sur les entiers
  ------------------------------------------------------------------
  -- Lecture d'un nombre rationnel
  procedure Get ( X : out T_Rationnel ) is

  begin -- Get

    Put ( "Numerateur: "  );
    Get ( X.Numerateur    );
    Put ( "Denominateur: ");
    Get ( X.Denominateur  );

  end Get;


  ------------------------------------------------------------------
  -- Ecriture d'un nombre rationnel
  procedure Put ( X : in T_Rationnel ) is

  begin -- Put

    Put ( X.Numerateur, 1);
    Put ( " /" );
    Put ( X.Denominateur, 1);

  end Put;

  ------------------------------------------------------------------
  -- Autres corps où les noms des types ont été adaptés
   ...
  ------------------------------------------------------------------
end Nombres_Rationnels_G;
```

Il faut aussi insister sur le fait que non seulement le paquetage dont `Entier_Io` est l'instanciation (ici `Ada.Text_Io.Integer_Io`) doit être un paquetage générique mais encore que le paramètre *effectif* correspondant à `Entier_Io` devra être un paquetage qui a été obtenu par une instanciation de `Ada.Text_IO.Integer_Io`.

Instanciations du paquetage Nombres_Rationnels_G.

```
-- Ada.Integer_Text_Io est un paquetage instancié avec le type Integer
package Nombres_Rationnels is new
                    Nombres_Rationnels_G ( Integer, Ada.Integer_Text_IO );


-- Exemple avec un type entier non prédéfini
type Entier_16_Bits is range –2**15 .. 2**15 – 1;
package ES_Entier_16_Bits is new
                    Ada.Text_IO.Integer_IO ( Entier_16_Bits );
package Nombres_Rationnels_16_Bits is new
                    Nombres_Rationnels_G ( Entier_16_Bits, ES_Entier_16_Bits );
```

Ici en pratique le paquetage `Nombres_Rationnels_G` aurait pu en fait se passer du paramètre `Entier_Io`. Il aurait suffit d'instancier le paquetage générique `Ada.Text_IO.Integer_Io` avec le type `T_Entier` au début du corps de `Nombres_Rationnels_G`. Mais dans ce cas cette instanciation doublonnerait avec celle, souvent nécessaire, placée dans l'unité utilisatrice, alors qu'avec le paquetage en paramètre, le paquetage instancié n'existe qu'en un seul exemplaire, celui réalisé par l'utilisateur.

Cette présentation des paquetages comme paramètres génériques se termine par un dernier exemple inspiré de [BAR 97]. Il s'agit de regrouper des types pour les traiter comme un tout, en particulier s'ils sont tous candidats à devenir des paramètres génériques. C'est typiquement le cas pour les types nécessaires à la déclaration de tables, de vecteurs ou de matrices. Le groupement en paquetage générique et son utilisation comme paramètre sont présentés dans l'exemple ci-dessous.

Types nécessaires à la formation de tables génériques.

```
generic
  type T_Indice is (<>);              -- Les indices sont de type discret, les
  type T_Element is private;          -- éléments les plus généraux possibles
  type T_Table is array (T_Indice range <>) of T_Element;

package Tables_G is end Tables_G; -- Partie declarative vide!

-- Fonction générique de recherche d'un élément dans une table. Elle
-- retourne la position de l'élément cherché
generic
  with package Tables is new Tables_G ( <> );
function Recherche_G (     Table:Tables.T_Table;
                           Element:Tables.T_Element) return Tables.T_Indice;
```

Cet exemple illustre encore une autre forme de déclaration pour un paquetage comme paramètre générique, forme où le symbole <> remplace la liste des paramètres:

```
with package Tables is new Tables_G ( <> );
```

La déclaration ainsi écrite n'impose pas les paramètres pour le paquetage instancié qui sera mentionné comme paramètre effectif. Par contre il est possible et souvent nécessaire avec cette variante d'utiliser les paramètres génériques du paquetage `Tables_G` dans la fonction `Recherche_G` en cours de définition.

Génériques et exceptions

Lorsque l'on déclare une exception dans un générique:

```
generic
  type T_Réel is digits <>;
package Nombres_Complexes_G is

  type T_Complexe is private;
  Complexe_Erreur : exception; -- Exception spécifique
  ... -- Autres déclarations et opérations
private
  ...
end Nombres_ Complexes _G;
```

Chaque instanciation crée une nouvelle version des objets déclarés dans le paquetage, donc chaque instanciation crée une nouvelle exception *Complexe_Erreur*
L'utilisateur ne peut plus gérer les exceptions, car il devrait tenir compte de chacune d'elles spécifiquement.
Pour cette raison, on englobe souvent le paquetage générique dans un autre paquetage permettant de déclarer les exceptions:

```
package Nombres_Complexes is

  Complexe_Erreur : exception; -- Exception spécifique

  generic
    type T_Réel is digits <>;
  package Nombres_Complexes_G is

    type T_Complexe is private;
    ... -- Autres declarations et opérations
  private
    ...
  end Nombres_Complexes_G;

end Nombres_ Complexes;
```

Génériques et paquetages enfants

Un paquetage non générique peut avoir un ou des enfants génériques. Dans ce cas l'enfant générique peut être instancié partout où il est visible.
Si un paquetage parent est générique, chacun de ses enfants, privé ou non, doit être générique. Dans ce cas l'enfant peut être instancier sans aucun problème dans le corps de son parent; par contre, dans le monde extérieur (les unités utilisatrices), l'instanciation de l'enfant ne peut se faire qu'après celle de son parent, ce qui finalement s'avère très logique!

Imaginons la situation suivante:
Le parent générique:

```
generic
  type Elément is private;
package Parent is …
```

Ses enfants doivent donc être génériques, même s'ils n'ont pas de paramètres:

```
generic
  package Parent.Enfant is …
```

Dans le programme où l'on a mis la clause de contexte nécessaire, on instancie d'abord le parent:

```
package Exemple is new Parent ( Float );
```

Ensuite on peut instancier l'enfant générique sur la base de l'instance de son parent, donc attention, c'est le nom de l'instance du parent qu'il faut utiliser, et non pas le nom du générique:

```
package Suite is new Exemple.Enfant;
```

Finalement, relevons qu'un enfant privé d'un parent générique doit lui aussi être générique; attention alors à la syntaxe, le mot **private** vient avant le mot **generic**:

```
private generic
  … -- Les paramètres génériques s'il y en a!
package Parent.Enfant is
  …
```

Divers

- On ne peut pas transmettre un type String (non contraint) pour un paramètre générique formel "**private**" simple. Si l'on veut obtenir cette possibilité, il faut utiliser un type privé avec discriminant inconnu.

  type T_Type (<>) **is private**;

- Notons encore qu'un paramètre générique formel privé peut avoir des discriminants connus; il prend alors la forme:

  type T_Type (P1 : T1, P2 : T2, …) **is private**;

  Dans ce cas le paramètre générique effectif doit avoir des discriminants correspondant à ceux du paramètre formel. Le paramètre formel ne peut pas avoir de valeurs par défaut, alors que le paramètre effectif lui le peut!

- Pour les paramètres des sous-programmes en paramètres génériques, les éventuelles contraintes des paramètres formels sont ignorées, seules comptent les contraintes des paramètres effectifs.

- La forme de paramètre effectif sous-programme avec valeur par défaut:

  with procedure Machin (P1 : T1, …) **is** Chose;

  n'est possible que lorsque :

  - les paramètres du sous-programme ne sont pas eux-mêmes d'un type venant des (autres) paramètres formels générique.
  - le sous-programme par défaut (ici Chose) est lui-même un paramètre formel générique.

Etiquette et Goto

Cela existe aussi en ADA !!!
Mais nous ne donnerons pas d'exemple !!!

Une étiquette se met devant n'importe quelle instruction et est délimitée par "<<" et ">>".

<<PAS_BEAU>>

On renvoie à cette étiquette par l'instruction:

goto PAS_BEAU;

On ne peut pas utiliser une étiquette pour transférer le contrôle à l'intérieur d'une instruction structurée, ni pour passer d'une branche à l'autre d'un **if** ou d'un **case** ou d'un traite exception, ni pour transférer brutalement le contrôle à l'intérieur d'un sous-programme.

Les justifications de la présence d'une instruction **goto** en Ada:

- La traduction automatique d'un autre langage vers Ada.
- La sortie brutale d'une structure très imbriquée.

Evolution du langage

Tout langage de programmation raisonnable, s'il ne disparaît pas, va évoluer au cours du temps, ceci afin de refléter l'évolution des technologies et celle des besoins des utilisateurs. Ces évolutions se font en général en prenant un maximum de précautions pour rester le plus possible compatible avec les versions précédentes du langage. Pour diminuer les problèmes, il arrive souvent que dans la définition d'une nouvelle norme on annonce que des éléments qui existaient dans la version précédente, existent encore dans cette version mais ne devraient plus être utilisés, car ils disparaîtront dans la norme suivante.

Ada ne fait pas exception à ce principe et après la première normalisation de 1983, une nouvelle norme a été introduite en 1995.

Un informaticien se doit de comprendre cette évolution, tout comme il doit, de manière générale, comprendre l'ensemble de l'évolution de l'informatique; c'est encore l'un des meilleurs moyens pour anticiper l'avenir. Pour lui, il ne faut pas simplement connaître les nouveautés, les modifications ou les disparitions, mais il devrait aussi comprendre les raisons de ces transformations.

Dans cette optique, nous allons présenter dans ce chapitre quelques uns des points de l'évolution entre Ada 83 et Ada 95. Le but n'est pas d'être exhaustif, ni de détailler les points, mais simplement de sensibiliser. Nous nous contenterons pour l'essentiel "d'énumérer" les points dans le but de provoquer une réflexion.

Nous ne parlerons pas des aspects qui n'ont pas été abordés dans notre cours, même si pour certains ils représentent une part importante de cette évolution; ils feront pour vous l'objet d'autres cours par la suite. Nous n'aborderont pas par conséquence la programmation objet, ni les nouveautés importantes introduites dans le contexte des tâches.

Généralités

- Les conventions typographiques ont changées entre Ada 83 et Ada 95:

  - Ada 83: mots clés en minuscules, identificateurs entièrement en majuscules.
  - Ada 95: mots clés en minuscules, première lettre de chaque partie d'identificateurs en majuscules.

  Bien entendu cette modification n'a aucune incidence sur la compilation d'un programme, ni sur son exécution. L'objectif vise simplement à améliorer la lisibilité des programmes au moment de la relecture.

- De nouveaux mots réservés sont apparus en Ada 95:

  abstract, **aliased**, **protected**, **requeue**, **tagged**, **until**

  la majorité ont trait aux tâches ou à la programmation objet, nous ne les aborderons pas ici; pour **aliased**, nous l'avons traité dans la partie consacrée aux types **access** et pour **until**, nous l'avons vu dans le cadre de l'instruction **delay**.

- Pour des raisons simples d'environnement de travail, certains caractères pouvaient être utilisés comme alternatives à d'autres (le # par exemple!); cette possibilité, bien qu'elle soit encore autorisée en Ada 95, fait partie des éléments obsolètes, qui disparaîtront avec la prochaine norme. Il ne faut donc plus les utiliser et de fait, il n'y a plus aucune raison de le faire!

- En Ada 83 il n'était pas possible de déclarer des types ou des objets après des corps (sous-programmes, paquetages). Ada 95 introduit une liberté complète dans l'ordre des déclarations, tout en respectant le principe que "quelque chose" ne peut pas être utilisé avant sa déclaration.

Type Character

- En Ada 83 utilisait un jeu de caractères normalisé sur 7 bits (ASCII), donc 128 caractères, ce qui ne permet pas de représenter les lettres accentuées. Ada 95 a introduit un jeu de caractères normalisé sur 8 bits donc 256 caractères (ISO Latin1, dont les 128 premiers caractères sont les mêmes que pour le code ASCII).

- En Ada 95 a introduit, en plus du type Character, le type Wide_Character sur 16 bits, dont les 256 premiers caractères sont les mêmes que ceux du type Character. Cela permet bien entendu d'étendre considérablement les caractères utilisés et de traiter les langues nécessitant une telle variété.

- En Ada 83 le paquetage Ada.Characters.Latin n'existait pas; sa création a rendu obsolète le paquetage ASCII (interne à Standard).

Boucle for

- En Ada 83 on ne pouvait pas écrire une boucle **for** de la forme:

  for I **in** -1 .. 12 **loop** ….

  sans qualifier l'expression définissant la borne inférieure afin d'en fixer le type; en levant cette barrière on a simplifié l'écriture et finalement rendu la formulation plus naturelle dans des cas relativement courants.

Les sous-programmes

- En Ada 83 la surcharge de l'opérateur "=" (donc indirectement "/=") ne pouvait se faire que pour des types privés et ne pouvait retourner qu'un résultat booléen. Ceci va dans le sens d'une plus grande généralisation et d'une plus grande souplesse.

- En Ada 83 les paramètres en mode **out** ne pouvaient pas être lus dans le corps du sous-programme. L'introduction de cette possibilité en Ada 95 évite de déclarer une variable locale que l'on affecte au paramètre de sortie juste avant la fin du sous-programme. Par contre elle est moins "rigoureuse" et présente des dangers si l'on tente d'utiliser la valeur avant sa première affectation; de plus la situation est particulière avec des pointeurs transmis en paramètres.

- Le glissement des bornes d'un tableau n'était pas autorisé en Ada 83 pour les paramètres et les résultats de fonctions. Ceci obligeait le programmeur à réaliser un contrôle beaucoup plus rigoureux et parfois à faire une "gymnastique" complexe pour ramener les bornes dans le bon intervalle.

- Ada 83 ne pouvait pas fournir un corps de sous-programme par surnomage

Types scalaires

- Les attributs Min et Max, qui s'appliquent à tous les types scalaires, n'existent pas en Ada 83; pour obtenir ce résultat il fallait donc effectuer les différentes comparaisons menant au bon résultat, d'où une simplification du codage dans des situations tout de même relativement fréquentes.

- En Ada 95 les attributs Succ et Pred ont été étendus à l'ensemble de types scalaires et donc plus uniquement aux types discrets. C'est une généralisation qui met en évidence la différence entre les types réels traités en informatique et en mathématiques.

Types Numériques

- Les types entiers modulo n'existaient pas en Ada 83, pour réaliser ce genre d'opérations le programmeur était obligé de les coder lui-même explicitement.

- Les types réels point-fixe décimal n'existaient pas en Ada 83, Le programmeur devait donc utiliser soit des types flottant, soit des types point-fixe standard; soulignons toutefois que les applications qui nécessitent des types point-fixe décimal sont bien spécifiques.

- L'attribut Base ne pouvait s'utiliser qu'en conjonction avec d'autres attributs en Ada 83. En Ada 95 nous pouvons donc utiliser plus facilement la représentation réelle des nombres.

- Lors de multiplications ou de divisions sur des objets point-fixe, le résultat devait être explicitement converti dans le type désiré en Ada 83. On a ainsi considérablement simplifié la tâche du programmeur et amélioré la lisibilité globale du programme.

- La valeur initiale d'un nombre nommé (constante universelle) ne pouvait être donnée que par d'autres valeurs universelles en Ada 83.

- De nombreux autres points, que nous ne détaillerons pas ici, différencient Ada 83 d'Ada 95 sur ces aspects numériques en particulier de nombreux attributs différents existent, certains ont fait leur apparition, d'autres disparaissent.

Exceptions

- En Ada 83 il existait une exception de base supplémentaire: Numeric_Error qui devait être levée par exemple lors de division par 0 ou de débordement dans un calcul numérique. La situation était ambiguë avec Constraint_Error. Numeric_Error disparaîtra avec la prochaine norme et pour la version Ada 95 elle constitue un surnom de Constraint_Error.

- En Ada 83 un traite exception ne pouvait pas comporter 2 fois la même exception, ce qui rendait délicat le surnomage d'une exception. Attention, cela ne signifie pas que 2 branches distinctes peuvent comporter le même nom d'exception!

- La notion d'occurrence d'exception n'existait pas en Ada 83; cette adjonction permet une gestion plus fine des exceptions, mais aux prix de difficultés pas possibles, ce qui n'est certainement pas satisfaisant!

Types Articles

- En Ada 83 la forme abrégée pour les articles vides:

  type Bidon **is null reccord**;

  n'existait pas; ceci n'est réellement pas très important, mais montre indirectement l'importance prise par la programmation objet.

- Un discriminant ne pouvait pas être du type **access** en Ada 83.

- La notion de discriminant inconnu n'existait pas en Ada 83.

- Les types dérivés ne pouvaient pas avoir leurs propres discriminants en Ada 83. Ils gardaient donc les discriminants spécifiques du type parent.

Types tableaux

- En Ada 83 lors de la déclaration d'une variable tableau, les bornes de l'objet ne pouvaient pas être déduites de l'expression d'initialisation (agrégat), alors que cette possibilité existait pour les constantes tableaux. Cette incohérence injustifiée entre variable et constante a été levée en Ada 95.

- En Ada 83 on ne pouvait pas utiliser un agrégat par nom avec une partie **others** après un signe d'affectation. Cette subtilité simplifiait peut être le travail du compilateur, mais dans certains cas compliquait de manière injustifiable la tâche du programmeur.

- Les règles pour la concaténation des tableaux unidimensionnels étaient beaucoup plus strictes en Ada 83; une fois de plus ceci va dans le sens d'une facilité d'utilisation.

Type String

- Puisque le type String n'est à priori rien d'autre qu'un tableau non contraint de Character, Ada 95 en introduisant le type Wide_Character a aussi introduit le type Wide_String qui n'est rien d'autre qu'un tableau non contraint de Wide_Character. Il ne s'agit là que d'une simple question logique de cohérence.

Paquetages

- Les paquetages enfants n'existaient pas en Ada 83; par conséquence pour étendre la fonctionnalité d'un paquetage existant il fallait soit réécrire le paquetage, avec les inconvénients que cela pouvait comporter pour la modification des programmes existant, soit définir un nouveau paquetage utilisant l'ancien. Dans les 2 cas la solution n'offrait pas la souplesse désirée et les problèmes de visibilité ne pouvaient pas toujours être résolus de manière satisfaisante.

Clause use type

- Elle n'existait pas en Ada 83; il fallait donc, pour pouvoir utiliser de manière non fonctionnelle les opérateurs, obligatoirement insérer une clause **use**. Rappelons qu'après une clause **use** rien nous interdit de donner systématiquement le chemins complet pour accéder aux autres éléments, mais les habitudes simplificatrices font que bien souvent on ne le fait pas!

Type access

- En Ada 83 le type **access** généralisé n'existait pas, les pointeurs ne pouvaient donc s'utiliser qu'avec des variables dynamiques. L'introduction de cette possibilité nous vient d'une volonté d'être compatible avec C, et des besoins introduits par la programmation objet!

- En Ada 83 le type **access** à un sous-programme n'existait pas, on avait estimé que les mécanismes de la généricité étaient suffisants pour résoudre ce genre de situations. C'est effectivement le cas, mais ces mécanismes s'avèrent lourds et pénalisant pour ce type de situations.

- En Ada 83 les paramètres **access** n'existaient pas. Leur introduction vient essentiellement des besoins de la programmation objet.

Types limités

- Un type ne pouvait être limité que s'il était privé en Ada 83. La programmation objet a nécessité d'étendre cette notion aux types articles.

- Seul un sous-programme déclaré dans le même paquetage que la définition d'un type limité privé pouvait avoir un paramètre de ce type en mode **out**.

La généricité

- Les types modulo et décimal n'existant pas en Ada 83, il n'y avait pas de formes spécifiques pour les paramètres formels génériques permettant de les représenter.

- Les paramètres paquetages n'existaient pas en Ada 83; à priori le programmeur devait donc définir un paramètre spécifique pour chacun des sous-programmes qu'il voulait utiliser.

Les fichiers

- Les paquetages préinstanciés pour les entrées/soties des type Integer et Float n'existaient pas en Ada 83; l'objectif de cet introduction vise à simplifier une opération relativement courante.

- Les sous-programmes Get_Immediat, Look_Ahead, Flush n'existaient pas en Ada 83; leur introduction a pour but évident de faciliter le développement d'applications interactives.

- En Ada 83 le Get pour des valeurs d'un type réel n'acceptait que des littéraux réels; en Ada 95 on simplifie la vie de l'utilisateur essentiellement en lui permettant éventuellement de donner ces valeurs sous forme d'entiers.

- Le mode Append_File n'existait pas en Ada 83; c'était donc au programmeur de réaliser l'opération par recopie de fichier!

- Le type File_Access et les fonctions livrant un résultat de ce type n'existaient pas en Ada 83; l'introduction de cette possibilité est une suite logique de l'apparition du type **access** généralisé.

- Le fichier "erreur" n'existait pas en Ada 83; son introduction peut permettre une gestion plus propre des sorties.

Autres éléments de bibliothèque

- Les paquetages Characters et Strings ainsi que leurs enfants n'existaient pas en Ada 83.

- Les paquetages Numerics et ses enfants n'existaient pas en Ada 83. Par contre il existait un paquetage générique Math_Lib mettant à disposition les fonctions mathématique élémentaires.

- Les paquetages Command_line et Interfaces n'existaient pas en Ada 83.

- Le paquetage System n'avait pas d'enfant en Ada 83.

Rappels sur les conversions

Nous avons déjà vu, qu'entre des types numériques ou des tableaux d'éléments de même type, il est possible de demander des conversions de type. Ainsi si Entier est une variable déclarée Integer nous pouvons écrire:

```
Entier := Integer ( 3.5 );
```

en utilisant le nom de type comme fonction. Ceci se déroule correctement dans la mesure du raisonnable, c'est-à-dire pour autant qu'il n'y ait pas de débordement!

Une deuxième possibilité existe entre les types scalaires et le type String; il s'agit des fonctions attributs:

Type'Image et Type'Value

Type'Image

Imaginons que I est une variable de type Integer qui contient la valeur 127, nous pouvons alors écrire:

```
Ada.Text_Io.Put ( "RESULTAT" & Integer'Image ( I ) );
```

Donc le type utilisé (ici Integer) doit être celui du paramètre, le résultat est toujours une chaîne de caractères représentant la valeur du paramètre. Ceci s'applique aussi aux types énumérés, qui font partie des types scalaires.

Type'Value

C'est la fonction réciproque d'Image qui convertit le paramètre (de type String) dans le type scalaire approprié. Par exemple si Car est de type Character, l'expression:

```
Car = Character'Value ( Character'Image ( Car ) ) ;
```

est toujours vraie.

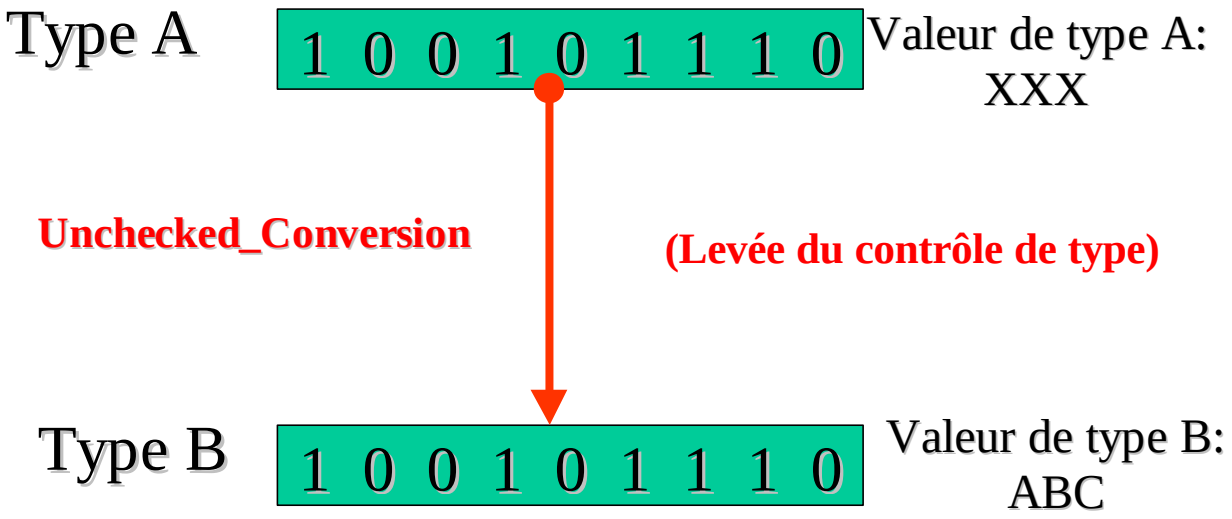
Rappelons aussi que la fonction attribut Type'Pos et sa fonction réciproque Type'Val font aussi partie des possibilités de conversion d'un type discret à un type entier et réciproquement!

UNCHECKED_CONVERSION

Finalement une troisième possibilité de conversion existe entre des types n'ayant à priori aucun lien, mais possédant en principe la même représentation physique en mémoire (même nombre de bits). Cette possibilité s'avère très souvent utile pour la programmation système, requêtes système et accès au matériel.

Ceci peut se réaliser grâce à la fonction générique Unchecked_Conversion dont voici la spécification:

```
generic
   type Source(<>) is limited private;
   type Target(<>) is limited private;
function Ada.Unchecked_Conversion ( S : Source ) return Target;
pragma Convention ( Intrinsic, Ada.Unchecked_Conversion );
pragma Pure ( Ada.Unchecked_Conversion );
```



```
-- Utilisation de Unchecked_Conversion; on demande une
--   valeur entière et on affiche la configuration de bits
--   correspondant à la représentation interne de cette valeur
with Ada.Text_Io;     use Ada.Text_Io;
with Ada.Unchecked_Conversion;
procedure Unc_Conv is

  package Entier_Io is new Integer_Io ( Integer ); use Entier_Io;
  --
  -- Pour la représentation des bits
  type T_Zéro_Un  is range 0 .. 1;       -- Un bit
  for T_Zéro_Un'Size use 1;   -- Force l'utilisation d'un seul bit
  package Zero_Un_Io is new Integer_Io ( T_Zéro_Un  );
  use Zero_Un_Io;
  --
  -- Pour la correspondance entre tableau de bits et entiers
  type T_Table_De_Bits is array ( 1..Integer'Size ) of T_Zéro_Un;
  pragma Pack ( T_Table_De_Bits  );  -- Force la compactification
  --
  -- Instancier la fonction de conversion
  function Transformer is new
          Ada.Unchecked_Conversion ( Integer, T_Table_De_Bits  );
  --
  -- Les variables courantes:
  Valeur_Lue : Integer;            -- Donnée par l'utilisateur
  Les_Bits   : T_Table_De_Bits ;   -- Pour la transformation

begin   -- Unc_Conv

  -- Demander la valeur et la lire
  Put_Line ( "-- Configuration de bits d'un entiers" );
  Put ( "DONNEZ LA VALEUR ENTIERE: " );
  Get ( Valeur_Lue );
  -- Transformer en bits
  Les_Bits := Transformer ( Valeur_Lue );
  --
  -- Afficher le résultat ( les bits )
  for I in reverse 1.. Integer'Size loop
    Put ( Les_Bits ( I ), 1 );
  end loop;
  New_Line;

end Unc_Conv;
```

Une exécution type donne le résultat suivant:

PROGRAMME DONNANT LA CONFIGURATION DE BITS D'UN ENTIER
DONNEZ LA VALEUR ENTIERE: 13
00000000000000000000000000001101

Il faut bien voir qu'en réalité il n'y a aucune conversion physique; ceci nous permet simplement d'affecter une configuration de bits à un objet d'un type donné.

On notera sur notre exemple qu'en principe ce programme s'adapte à n'importe quelle machine par l'utilisation de:

Integer'Size

pour fixer le nombre de bits à utiliser.

Pragmas

En fait un pragma consiste en une directive au compilateur sur la manière dont il doit travailler. Il en existe plusieurs, définis par le manuel de référence. La forme générale des pragmas est la suivante:

pragma Nom_Du_Pragma (Liste_D_Arguments);

La Liste_D_Arguments entre parenthèses dépend du type de pragma et dans certains cas elle n'existe pas.

L'endroit où l'on a le droit d'introduire un pragma dépend du type de pragma; disons qu'en règle générale, si l'on est logique et raisonnable, il n'y a pas de problème. En plus des pragmas donnés par le manuel de référence, une implémentation peut en introduire d'autres. Si une implémentation ne reconnaît pas un pragma, cela ne doit pas avoir d'autres conséquences sur le programme que sa non-application et éventuellement un message d'avertissement; c'est ce que dit le manuel de référence, mais en pratique il en va souvent autrement!

Nous reviendrons par la suite sur la notion de pragma; pour l'instant nous regardons simplement un cas particulier, lié à notre exemple.

Pour le pragma Pack, son unique argument est le nom d'un type *article* ou *tableau* et indique au compilateur que pour les objets de ce type il faut essayer d'économiser au maximum la place mémoire. Dans ce cas, il est bien évident que le pragma doit obligatoirement venir après la déclaration du type concerné, mais dans la même zone déclarative!

Dans notre exemple:

pragma Pack (T_Table_De_Bits);

pour paqueter le tableau de bits! Attention, ceci peut ne pas avoir d'effet sur certaines machines; c'est là que les règles théoriques ne correspondent plus toujours à la réalité!

Les clauses de représentation

Une telle directive impose des contraintes au compilateur. Ces clauses sont diverses suivant la nature des éléments auxquels on les applique. Elles doivent obligatoirement figurer dans la même partie déclarative que l'entité à laquelle elles s'appliquent. Une implémentation peut ne pas prendre en considération certaines clauses; mais si elle les accepte, elle garantit que tout se passe normalement vu de l'extérieur, c'est-à-dire comme s'il n'y avait pas de clause de représentation. Il s'agit d'un outil important pour la programmation système, qui peut également apporter une aide dans l'optimisation de programmes.

Clauses de longueur

Sa forme générale:

for Attribut **use** Expression;

EXPRESSION doit être une expression statique à valeur entière, elle indique le nombre maximum de bits à utiliser pour la représentation d'un objet du type donné. Bien sûr la valeur de l'expression doit être raisonnable pour le type considéré.

Spécification de taille d'objets, dans ce cas:

Attribut : Type'Size

Par exemple:

for T_Zéro_Un'Size **use** 1;

pour forcer l'utilisation d'un seul bit.

Les miracles n'existant pas la valeur donnée doit être raisonnable en fonction du type!

Par exemple il ne serait **pas possible** d'écrire:

```
type Entier is range 0..255;
for Entier'Size use 2;
```

Ni d'ailleurs:

```
subtype Mes_Entiers is Integer range –128..127;
for Mes_Entiers'Size use 8;
```

car un sous-type est construit sur son type parent et hérite de ses propriétés.

Clause de représentation d'énumérations

Elle permet de fixer les valeurs des représentations internes des constantes du type énuméré.

Sa forme générale:

for Type_Enumeratif **use** *agrégat*;

Exemple:

```
type Français is ( Bleu,  Blanc,  Rouge );

for Français use ( Bleu => 5,  Blanc => 7,  Rouge => 9 );
```

Bien sûr si la clause existe, c'est que nous ne devons pas nécessairement donner toutes les valeurs, en partant de 0 et sans laisser de trous, ce qui se passe normalement par défaut.

Les valeurs entières doivent être données par des expressions statiques, elles doivent respecter la structure d'ordre imposée par la déclaration du type. Signalons que malgré la contrainte imposée par la clause, les attributs Succ, Pred, Val et Pos fonctionnent normalement. Attention: Pos donne la valeur de la position de la constante dans le type énuméré et non pas la valeur de sa représentation interne.

Clauses de représentation d'article

La forme générale se décompose en 2 parties:

- Première partie:

```
for Type_Article'Alignment use Expression_Statique_Entière;
```

expression_statique_entière étant une valeur entière, elle indique que l'adresse à laquelle sera réduit en mémoire un tel objet doit être un multiple de cette valeur; dans l'exemple ci-dessous, à une adresse divisible par 4. Sur certaines architectures d'ordinateurs, ceci peut être important pour l'efficacité.

- Deuxième partie:

```
for Type_Article use
    record
        Composant at Position range First_Bit .. Last_Bit;
        ...;
    end record;
```

Pour chaque composant de l'article on peut préciser:

- L'adresse mémoire du composant par rapport au début de l'article (**at** Position); la première adresse est toujours 0.

- La position des bits utilisés dans l'unité de mémoire (**range** First_Bit..Last_Bit). Le premier bit dans l'unité est le bit 0. Des bits peuvent rester inutilisés. Dans certaines implémentations, mais ce n'est pas toujours le cas, on peut demander qu'un composant chevauche 2 zones mémoires. Il n'est pas nécessaire de donner une forme spécifique pour chaque composant d'un article, on peut laisser la liberté au compilateur de faire certains choix. Notons que le recouvrement n'est pas possible.

Exemple:

L'article tel qu'on le définirait normalement:

```
type Référence is range 0 .. 999;
type Classe is ( C1, C2, C3 );

type Artificiel is
record
        La_Référence : Référence;
        La_Classe : Classe;
end record;
```

Les clauses de représentation d'articles, qui doivent venir dans la même partie déclarative que la définition du type elle-même:

```
for Artificiel'Alignment use 4;

for Artificiel use
        record
                La_ Référence at 0 range 2 .. 11;
                La_Classe      at 1 range 4 .. 5;
        end record;
```

Chacune des 2 clauses peut être présente indépendamment l'une de l'autre!

Les attributs suivants peuvent être utilisés pour connaître différentes caractéristiques:

- **Alignement**: pour connaître la valeur de l'alignement mémoire d'un objet.
- **Position**: pour connaître la position d'un champ dans un article.
- **First_Bit**: pour connaître la position du premier bit d'un champ d'article.
- **Last_Bit**: pour connaître la position du dernier bit d'un champ d'article.

Un tout petit programme pour illustrer ces possibilités:

```
with Ada.Text_Io;              use Ada.Text_Io;
with Ada.Integer_Text_Io;      use Ada.Integer_Text_Io;
procedure Exemple is
  type T_Référence is range 0 .. 999;
  type T_Classe    is ( C1, C2, C3 );

  type T_Artificiel is
    record
      La_Référence : T_Référence;
      La_Classe    : T_Classe;
    end record;

  for T_Artificiel'Alignment use 4;
  for T_Artificiel use record
    La_Référence at 0 range 2 .. 11;
    La_Classe    at 1 range 4 .. 5;
  end record;
  Valeur : T_Artificiel;

Begin  -- Exemple

  Put ( "Alignment: ");
  Put ( Valeur'Alignment ); New_Line;
  Put ( "Position: ");
  Put ( Valeur.La_Classe'Position ); New_Line;
  Put ( "First_Bit: ");
  Put ( Valeur.La_Classe'First_Bit ); New_Line;
  Put ( "Last_Bit: ");
  Put ( Valeur.La_Classe'Last_Bit ); New_Line;

end Exemple;
```

De plus, notons la situation quelque peu particulière de type dérivé:

```
type Descriptor is
        record
                -- …
        end record;
type Packed_Descriptor is new Descriptor;
for Packed_Descriptor use
        record
                -- …
        end record;
```

Si nous déclarons:

```
D : Descriptor;
P : Packed_Descriptor;
```

Nous pouvons dans le code convertir explicitement l'un des objet dans le type de l'autre:

```
P := Packed_Descriptor ( D );     -- pack D
D := Descriptor ( P );            -- unpack P
```

Clause de représentation des tableaux

Nous disposons aussi d'une clause de représentation pour les éléments d'un tableau, basée sur l'attribut: Component'Size qui s'applique à un type ou un objet tableau.

Exemple:

```
type T_Bits is array ( 1 .. 15 ) of Boolean;
Le_Tableau : T_Bits;
for Le_Tableau'Component'Size use 8;
```

L'objectif d'une telle clause de représentation est bien évidemment d'augmenter les performances, en admettant que sur un système donné un tel accès sera plus efficace.

Note: Dans notre environnement de travail cet attribut n'est pas implémenté!

Clause d'adresse

Il existe aussi une clause d'adresse dont nous ne faisons qu'introduire l'idée ici.

Sa forme générale:

for Nom'Address **use** *Expression*;

Expression doit correspondre à une adresse, dont la forme dépend de l'implémentation; le paquetage System met a disposition des fonctions de conversion.

Nom peut être un nom de: constante, variable, sous-programme, tâche, paquetage ou entrée.

Cette clause permet de préciser l'adresse physique à laquelle on veut que l'objet se trouve en mémoire.

Exemple:

```
for Interruption_1'Address use 16#0028#;
```

Ceci se révèle important pour pouvoir accéder à des zones de mémoire physiques précises!

Problème lié:

Reprenons les déclarations:

```
type Français is ( Bleu,  Blanc,  Rouge );
for Français use ( Bleu => 5,  Blanc => 7,  Rouge => 9 );
Valeur : Français;
for Valeur'Address use 16#0028#;
```

rien ne prouve qu'à cette adresse là se trouve une valeur valide pour notre type!

Nous pouvons à tout moment dans le code contrôler cette situation par l'intermédiaire de l'attribut *Valid*:

if Valeur'Valide **then** . . .

On peut également se mettre dans une telle situation (obtenir une valeur non valide pour le type) tout simplement avec une variable non initialisée ou lors de l'utilisation de Unchecked_Conversion!

Clause de représentation pour point fixe

Sans précaution particulière, pour la représentation d'un réel point fixe, votre compilateur peut choisir un incrément quelconque, pour autant qu'il respecte la condition que vous avez imposé. Ainsi, si vous lui demandez un **delta** de 0.1, il y a de fortes chances qu'il choisisse un incrément de 1/16 (binaire) soit 0.0625 qui satisfait votre condition de 0.1, mais qui peut donner des surprises par la suite dans les calculs. Pour remédier à ce problème, nous disposons d'un clause de représentation spécifique:

```
for Type'Small use Valeur;
```

Exemples:

```
for T_Reel'Small use 0.1;
```

```
with Ada.Text_Io;       use Ada.Text_Io;
procedure Pointfixe is

  type T_Réel is delta 0.1 range - 10000.0 .. 10000.0;
  for T_Réel'Small use 0.1;
  package Es_P_Fixe is new Fixed_Io ( T_Réel );
  use Es_P_Fixe;
  Increment : constant T_Réel := 0.1;
  Valeur    :          T_Réel := 0.0;

Begin

  Put_Line ( "Debut" );
  for I in 1..10000 loop
    Valeur := Valeur +Increment;
  end loop;
  Put ( Valeur );
  New_Line;
  Put ( Incrément * 10000.0 );
  New_Line;
  Skip_Line;

end Pointfixe;
```

Le programme ci-dessus, dans l'environnement GNAT nous donne, sans la clause de représentation:

625.0
625.0

et avec la clause de représentation, tel qu'il est représenté:

1000.0
1000.0

Ce qui est certainement bien ce que nous espérions!

Optimisation

Différents éléments peuvent intervenir dans l'optimisation d'un programme:

a) Tout d'abord le pragma Optimize, qui peut venir partout où un pragma est permis et dont la portée se termine à la fin de la portée du bloc où il apparaît. Sa forme générale est:

pragma Optimize (Identificateur);

Les valeurs possibles pour Indentificateur sont:

- **Time**: le compilateur doit alors mettre l'accent sur l'optimisation du temps d'exécution.
- **Space**: le compilateur doit alors mettre l'accent sur la minimisation de l'espace mémoire utilisé.
- **Off**: le compilateur ne doit faire aucune optimisation; ceci peut être important dans une phase de développement par exemple.

Bien entendu, une seule de ces 3 valeurs peut être utilisée à la fois; il n'y a pas de miracles!

Exemple: **pragma** Optimize (Off);

b) Une autre possibilité consiste à demander au compilateur de ne pas générer des appels pour certains sous-programmes, mais de développer complètement le code de ces sous-programmes à la place de son appel. Il en résultera des performances meilleures en temps, mais bien sur un code globalement plus important en taille dès que l'on aura plusieurs appels.

pragma Inline (Nom {, Nom });

Le pragma peut s'appliquer à des sous-programmes ou a des génériques. Dans le cas de sous-programmes simples, le pragma doit venir dans la même zone déclarative que le sous-programme. Dans le cas de génériques le pragma s'applique à toutes les instances du sous-programme. Si le nom donné dans le pragma peut désigner plusieurs sous-programmes (surcharge) le pragma s'applique à tous les sous-programmes correspondants.

c) Suppression des vérifications, c'est une technique qui peut être efficace, mais qui s'avère aussi très dangereuse (politique de l'autruche!). Le code que doit générer le compilateur pour détecter les conditions qui lèveront des exceptions peut être supprimé par l'utilisation du pragma Suppress.

pragma Suppress (Identificateur [, [On =>] Nom]);

Identificateur est l'un des noms de contrôle que l'on peut supprimer et dont vous avez la liste ci-dessous. Dans le cas où nous ne donnons qu'un identificateur, sans la partie optionnelle Nom,

le pragma s'applique alors dans toutes les situations; par contre, si l'on précise un nom et que ce nom est celui d'un objet, le pragma ne s'applique qu'à cet objet; finalement si le nom est celui d'un type ou d'un sous-type, le pragma s'applique à tous les objets de ce type.

Liste des contrôles que l'on peut supprimer:

Les éléments liés à Constraint_Error:

Access_Check
Contrôle que, lors de l'accès à un objet par un pointeur, celui-ci n'est pas **null**.

Discriminant_Check
Contrôle que le discriminant est valable pour un objet contraint et que lors de l'accès à un champ, celui-ci existe effectivement pour la valeur actuelle du discriminant.

Division_Check
Contrôle que le deuxième opérande ne vaut pas 0 pour: /, **rem** et **mod**.

Index_Check
Contrôle que lors de la déclaration d'un tableau les bornes données sont bien dans la plage de validité des indices, et lors de l'accès d'un élément que chaque indice est bien dans les bornes fixées du tableau.

Length_Check
Contrôle que la grandeur des tableaux concorde lors de conversions de type ou lors d'opérations sur des tableaux unidimensionnels de booléens.

Overflow_Check
Contrôle que les valeurs (scalaires) restent bien dans le type de base lors d'opérations.

Range_Check
Contrôle que les contraintes d'intervalles sont satisfaites, aussi bien lors de déclarations qu'à l'exécution.

Tag_Check Pas abordé ici!

Les éléments liés à Program_Error:

Elaboration_Check
Contrôle que lors de l'appel d'un sous-programme ou de l'instanciation d'un générique (il y a encore d'autres situations) le corps correspondant a déjà été élaboré.

Accessibility_Check Pas abordé ici!

Elément lié à Storage_Error:

Storage_Check
Contrôle que lors d'une allocation de mémoire, la place nécessaire est disponible.

Toutes les erreurs:

All_Checks
Supprime tous les contrôles.

Exemples d'utilisation:

```
pragma Suppress ( Range_Check );
pragma Suppress ( Index_Check, On => Table );
pragma Suppress ( Index_Check, Table );
pragma Suppress ( All_Checks );
```

e) Rappelons encore qu'une clause de représentation, suivant les environnements, peut aussi être une méthode pour optimiser un programme.

Classification des types

<table>
<tr><td colspan="6">simples</td></tr>
<tr><td rowspan="3">accès</td><td colspan="5">scalaires</td></tr>
<tr><td colspan="2">discrets</td><td colspan="3">réels</td></tr>
<tr><td>énumérés</td><td colspan="2">entiers</td><td rowspan="2">point-flottants</td><td colspan="2">point-fixes</td></tr>
<tr><td></td><td>signés</td><td>modulo</td><td>ordinaires</td><td>décimals</td></tr>
<tr><td colspan="5">composés</td><td></td></tr>
<tr><td>tableaux</td><td>articles</td><td colspan="2">protégés</td><td>tâches</td><td></td></tr>
</table>

Classification des instructions

Simple		Composée	
Séquentielle	Contrôle	Séquentielle	Contrôle
vide	exit	if	accept
affectation	goto	case	select
appel de procédure	raise	loop	
appel d'entrée	return	bloc	
requeue			
delay			
abort			
code			

Paquetage Standard

Le paquetage Standard joue un rôle très particulier et même généralement n'est pas présent en tant que paquetage dans une implémentation. En tout cas il ne faut pas le faire apparaître dans une clause **with**, ses éléments étant toujours disponibles; on peut dire qu'il s'agit plus d'un concept que d'une entité réelle.

```
package Standard is
   pragma Pure ( Standard );
   type Boolean is ( False, True );
   -- The predefined relational operators for this type are as follows:
    -- function "="  ( Left, Right : Boolean ) return Boolean;
    -- function "/="  ( Left, Right : Boolean ) return Boolean;
    -- function "<"   ( Left, Right : Boolean ) return Boolean;
    -- function "<="  ( Left, Right : Boolean ) return Boolean;
    -- function ">"   ( Left, Right : Boolean ) return Boolean;
    -- function ">="  ( Left, Right : Boolean ) return Boolean;

   -- The predefined logical operators and the predefined logical
   -- negation operator are as follows:
    -- function "and" ( Left, Right : Boolean ) return Boolean;
    -- function "or"  ( Left, Right : Boolean ) return Boolean;
    -- function "xor" ( Left, Right : Boolean ) return Boolean;
    -- function "not" ( Right : Boolean ) return Boolean;

   -- The integer type root_integer is predefined.
   -- The corresponding universal type is universal_integer.
   type Integer is range Implementation-Defined;
   subtype Natural  is Integer range 0 .. Integer'Last;
   subtype Positive is Integer range 1 .. Integer'Last;
   -- The predefined operators for type Integer are as follows:

    -- function "="  ( Left, Right : Integer'Base ) return Boolean;
    -- function "/=" ( Left, Right : Integer'Base ) return Boolean;
    -- function "<"  ( Left, Right : Integer'Base ) return Boolean;
    -- function "<=" ( Left, Right : Integer'Base ) return Boolean;
    -- function ">"  ( Left, Right : Integer'Base ) return Boolean;
    -- function ">=" ( Left, Right : Integer'Base ) return Boolean;

    -- function "+"   ( Right : Integer'Base ) return Integer'Base;
    -- function "–"   ( Right : Integer'Base ) return Integer'Base;
    -- function "abs" ( Right : Integer'Base ) return Integer'Base;
    -- function "+"   ( Left, Right : Integer'Base ) return Integer'Base;
    -- function "–"   ( Left, Right : Integer'Base ) return Integer'Base;
    -- function "*"   ( Left, Right : Integer'Base ) return Integer'Base;
    -- function "/"   ( Left, Right : Integer'Base ) return Integer'Base;
    -- function "rem" ( Left, Right : Integer'Base ) return Integer'Base;
    -- function "mod" ( Left, Right : Integer'Base ) return Integer'Base;
    -- function "**"  ( Left : Integer'Base; Right : Natural ) return Integer'Base;
```

```
-- The specification of each operator for the type
-- root_integer, or for any additional predefined integer
-- type, is obtained by replacing Integer by the name of the type
-- in the specification of the corresponding operator of the type
-- Integer.  The right operand of the exponentiation operator
-- remains as subtype Natural.

-- The floating point type root_real is predefined.
-- The corresponding universal type is universal_real.
type Float is digits Implementation-Defined;
-- The predefined operators for this type are as follows:
 -- function "="   ( Left, Right : Float ) return Boolean;
 -- function "/="  ( Left, Right : Float ) return Boolean;
 -- function "<"   ( Left, Right : Float ) return Boolean;
 -- function "<="  ( Left, Right : Float ) return Boolean;
 -- function ">"   ( Left, Right : Float ) return Boolean;
 -- function ">="  ( Left, Right : Float ) return Boolean;

 -- function "+"   ( Right : Float ) return Float;
 -- function "–"   ( Right : Float ) return Float;
 -- function "abs" ( Right : Float ) return Float;
 -- function "+"   ( Left, Right : Float ) return Float;
 -- function "–"   ( Left, Right : Float ) return Float;
 -- function "*"   ( Left, Right : Float ) return Float;
 -- function "/"   ( Left, Right : Float ) return Float;

 -- function "**"  ( Left : Float; Right : Integer'Base ) return Float;
-- The specification of each operator for the type  root_real,  or  for
-- any  additional  predefined  floating  point  type,  is  obtained  by
-- replacing  Float by the name of  the type in the specification of the
-- corresponding  operator  of  the  type  Float.
-- In  addition,  the  following operators are predefined for the root
-- numeric types:

function "*" ( Left : Root_Integer; Right : Root_Real ) return Root_Real;
function "*" ( Left : Root_Real;    Right : Root_Integer ) return Root_Real;
function "/" ( Left : Root_Real;    Right : Root_Integer ) return Root_Real;
-- The type universal_fixed is predefined.
-- The only multiplying operators defined between
-- fixed point types are

function "*" ( Left : Universal_Fixed; Right : Universal_Fixed ) return Universal_Fixed;
function "/" ( Left : Universal_Fixed; Right : Universal_Fixed ) return Universal_Fixed;

-- The declaration of type Character is based on the standard ISO 8859-1 character set.
-- There are no character literals corresponding to the positions forcontrol characters.
-- They are indicated in italics in this definition.  See 3.5.2.

type Character is
```

```
( Nul, Soh,  Stx,  Etx,  Eot,  Enq,  Ack,  Bel,  -- 0 (16#00#) .. 7 (16#07#)
 Bs,  Ht,  Lf,  Vt,  Ff,  Cr,  So,  Si,  -- 8 (16#08#) .. 15 (16#0F#)

 Dle,  Dc1,  Dc2,  Dc3,  Dc4,  Nak,  Syn,  Etb,  -- 16 (16#10#) .. 23 (16#17#)
 Can,  Em,  Sub,  Esc,  Fs,  Gs,  Rs,  Us,  -- 24 (16#18#) .. 31 (16#1F#)

 ' ',  '!',  '"',  '#',  '$',  '%',  '&',  ''',  -- 32 (16#20#) .. 39 (16#27#)
 '(',  ')',  '*',  '+',  ',',  '-',  '.',  '/',  -- 40 (16#28#) .. 47 (16#2F#)

 '0',  '1',  '2',  '3',  '4',  '5',  '6',  '7',  -- 48 (16#30#) .. 55 (16#37#)
 '8',  '9',  ':',  ';',  '<',  '=',  '>',  '?',  -- 56 (16#38#) .. 63 (16#3F#)

 '@',  'A',  'B',  'C',  'D',  'E',  'F',  'G',  -- 64 (16#40#) .. 71 (16#47#)
 'H',  'I',  'J',  'K',  'L',  'M',  'N',  'O',  -- 72 (16#48#) .. 79 (16#4F#)

 'P',  'Q',  'R',  'S',  'T',  'U',  'V',  'W',  -- 80 (16#50#) .. 87 (16#57#)
 'X',  'Y',  'Z',  '[',  '\',  ']',  '^',  '_',  -- 88 (16#58#) .. 95 (16#5F#)

 '`',  'a',  'b',  'c',  'd',  'e',  'f',  'g',  -- 96 (16#60#) .. 103 (16#67#)
 'h',  'i',  'j',  'k',  'l',  'm',  'n',  'o',  -- 104 (16#68#) .. 111 (16#6F#)

 'p',  'q',  'r',  's',  't',  'u',  'v',  'w',  -- 112 (16#70#) .. 119 (16#77#)
 'x',  'y',  'z',  '{',  '|',  '}',  '~',  Del,  -- 120 (16#78#) .. 127 (16#7F#)

 Reserved_128,  Reserved_129,  Bph,  Nbh,  -- 128 (16#80#) .. 131 (16#83#)
 Reserved_132,  Nel,  Ssa,  Esa,  -- 132 (16#84#) .. 135 (16#87#)

 Hts,  Htj,  Vts,  Pld,  Plu,  Ri,  Ss2,  Ss3,  -- 136 (16#88#) .. 143 (16#8F#)

 Dcs,  Pu1,  Pu2,  Sts,  Cch,  Mw,  Spa,  Epa,  -- 144 (16#90#) .. 151 (16#97#)

 Sos,  Reserved_153,  Sci,  Csi,  -- 152 (16#98#) .. 155 (16#9B#)
 St,  Osc,  Pm,  Apc,  -- 156 (16#9C#) .. 159 (16#9F#)

' ',  '¡',  '¢',  '£',  '¤',  '¥',  '¦',  '§'  -- 160 (16#A0#) .. 167 (16#A7#)
'¨',  '©',  'ª',  '«',  '¬',  '-',  '®',  '¯'  -- 168 (16#A8#) .. 175 (16#AF#)

'°',  '±',  '²',  '³',  '´',  'µ',  '¶',  '·'  -- 176 (16#B0#) .. 183 (16#B7#)
'¸',  '¹',  'º',  '»',  '¼',  '½',  '¾',  '¿'  -- 184 (16#B8#) .. 191 (16#BF#)

'À',  'Á',  'Â',  'Ã',  'Ä',  'Å',  'Æ',  'Ç'  -- 192 (16#C0#) .. 199 (16#C7#)
'È',  'É',  'Ê',  'Ë',  'Ì',  'Í',  'Î',  'Ï'  -- 200 (16#C8#) .. 207 (16#CF#)

'Ð',  'Ñ',  'Ò',  'Ó',  'Ô',  'Õ',  'Ö',  '×'  -- 208 (16#D0#) .. 215 (16#D7#)
'Ø',  'Ù',  'Ú',  'Û',  'Ü',  'Ý',  'Þ',  'ß'  -- 216 (16#D8#) .. 223 (16#DF#)

'à',  'á',  'â',  'ã',  'ä',  'å',  'æ',  'ç'  -- 224 (16#E0#) .. 231 (16#E7#)
'è',  'é',  'ê',  'ë',  'ì',  'í',  'î',  'ï'  -- 232 (16#E8#) .. 239 (16#EF#)

'ð',  'ñ',  'ò',  'ó',  'ô',  'õ',  'ö',  '÷'  -- 240 (16#F0#) .. 247 (16#F7#)
'ø',  'ù',  'ú',  'û',  'ü',  'ý',  'þ',  'ÿ' );  -- 248 (16#F8#) .. 255 (16#FF#)
```

```
   -- The predefined operators for the type Character are the same as for
   -- any enumeration type.

   -- The declaration of type Wide_Character is based on the standard
   --    ISO 10646 BMP character set.
   -- The first 256 positions have the same contents as type Character.  See 3.5.2.


   type Wide_Character is ( Nul, Soh ... Fffe, Ffff );

   package Ascii is ... end Ascii;  -- Obsolescent; see J.5

   -- Predefined string types:

   type String is array ( Positive range <> ) of Character;
   pragma Pack ( String );
   -- The predefined operators for this type are as follows:
     -- function "="  Left, Right: String ) return Boolean;
     -- function "/=" ( Left, Right: String ) return Boolean;
     -- function "<"  ( Left, Right: String ) return Boolean;
     -- function "<=" ( Left, Right: String ) return Boolean;
     -- function ">"  ( Left, Right: String ) return Boolean;
     -- function ">=" ( Left, Right: String ) return Boolean;

     -- function "&" ( Left: String;    Right: String )    return String;
     -- function "&" ( Left: Character; Right: String )    return String;
     -- function "&" ( Left: String;    Right: Character ) return String;
     -- function "&" ( Left: Character; Right: Character ) return String;
   type Wide_String is array ( Positive range <> ) of Wide_Character;
   pragma Pack( Wide_String );

   -- The predefined operators for this type correspond to those for String
   type Duration is delta Implementation-Defined range Implementation-Defined;
   -- The predefined operators for the type Duration are the same as for
   -- any fixed point type.
   -- The predefined exceptions:
   Constraint_Error : exception;
   Program_Error    : exception;
   Storage_Error    : exception;
   Tasking_Error    : exception;

end Standard;
```

Si l'on résume, on y trouve les notions principales suivantes:

- Définition du type Boolean et de ses opérateurs associés.

- Définition du type Integer et de ses opérateurs associés.

Suivant les implémentations d'autres types entiers peuvent être définis. En plus, il y a ce qu'il faut pour le type *entier universel* qui n'est pas accessible à l'utilisateur en temps que type.

- Définition des sous-types Natural et Positive

- Définition du type Float et de ses opérateurs associés. Suivant l'implémentation d'autres types réels sont introduits.

- Définition du type Character, en fait un type énuméré.

- Définition du type Wide_Character qui étend le jeu de caractères, en utilisant un codage sur 16 bits.

- Le paquetage interne Ascii est obsolète, il ne faut donc plus l'utiliser, il reste présent dans cette version pour des raisons de compatibilité avec Ada 83.

- Définition du type String et de ses opérateurs associés.

- Définition du type Wide_String qui a les mêmes caractéristiques que le type String, mais dont les éléments sont du type Wide_Character.

- Définition du type Duration en type réel point fixe.

- Définition des 4 exceptions de base:
Constraint_Error, Program_Error, Storage_Error, Tasking_Error

Paquetage System et ses enfants

Chaque implémentation comporte un paquetage de bibliothèque appelé System qui définit les caractéristiques liées à la configuration. Certains éléments du paquetage sont imposés par le manuel de référence, d'autres peuvent être ajoutés par chaque implémentation.

```
package System is
        pragma Preelaborate ( System );
        type Name is Implementation-Defined-Enumeration-type;
        System_Name                 : constant Name := Implementation-Defined;
        -- System-Dependent Named Numbers:
        Min_Int                     : constant := Root_Integer'First;
        Max_Int                     : constant := Root_Integer'Last;
        Max_Binary_Modulus          : constant := Implementation-Defined;
        Max_Nonbinary_Modulus       : constant := Implementation-Defined;
        Max_Base_Digits             : constant := Root_Real'digits;
        Max_Digits                  : constant := Implementation-Defined;
        Max_Mantissa                : constant := Implementation-Defined;
        Fine_Delta                  : constant := Implementation-Defined;
        Tick                        : constant := Implementation-Defined;
        -- Storage-related Declarations:
        type Address is Implementation-Defined;
        Null_Address                : constant Address;

        Storage_Unit                : constant := Implementation-Defined;
        Word_Size                   : constant := Implementation-Defined * Storage_Unit;
        Memory_Size                 : constant := Implementation-Defined;
        -- Address Comparison:
        function "<"   ( Left, Right : Address ) return Boolean;
        function "<=" ( Left, Right : Address ) return Boolean;
        function ">"   ( Left, Right : Address ) return Boolean;
        function ">=" ( Left, Right : Address ) return Boolean;
        function "="   ( Left, Right : Address ) return Boolean;
        --      function "/=" ( Left, Right : Address ) return Boolean;
        -- "/=" is implicitly defined
        pragma Convention( Intrinsic, "<" );
          ... -- and so on for all language-defined subprograms in this package
        -- Other System-Dependent Declarations:
        type Bit_Order is ( High_Order_First, Low_Order_First );
        Default_Bit_Order : constant Bit_Order;
        -- Priority-related declarations (see D.1):
        subtype Any_Priority is Integer range Implementation-Defined;
        subtype Priority is Any_Priority range Any_Priority'First .. Implementation-Defined;
        subtype Interrupt_Priority is Any_Priority range Priority'Last+1 .. Any_Priority'Last;
        Default_Priority : constant Priority := ( Priority'First + Priority'Last )/2;
        private
        ... -- not specified by the language
end System;
```

La majorité des caractéristiques ne sont pas données explicitement puisqu'elles dépendent de l'implémentation. L'annexe M du manuel de référence, qui est spécifique à chaque environnement, donne les valeurs particulières valables pour l'implémentation. Une implémentation peut définir d'autres grandeurs dans le paquetage System et ses enfants en plus de celles données dans le manuel de référence.
Quelques éléments de ce paquetage, avec les valeurs pour ObjectAda sous NT:

- **type** Name **is** (S370, I80X86, I80386, MC680X0, Vax, Transputer, RS_6000, MIPS);

Le nom des machines pour lesquelles l'environnement est configuré.

- System_Name : **constant** Name := I80386;

Le nom de la machine courante pour laquelle l'environnement est configuré.

- Min_Int : **constant** :=-2**31;
- Max_Int : **constant** := 2**31 - 1;

Les constantes Min_Int et Max_Int fixent les limites des valeurs entières utilisables sur le système (attention elles peuvent certainement être plus petite, respectivement plus grande que Integer'First, Integer'Last).

- Max_Binary_Modulus : **constant** := 2**32;
- Max_Nonbinary_Modulus : **constant** := 2**31 - 1;

Max_Binary_Modulus et Max_Nonbinary_Modulus fixent la plus grande valeur utilisable pour un type modulo. Pour Max_Binary_Modulus, toutes les autres puissances positives de 2 plus petite ou égales peuvent être utilisées; pour Max_Nonbinary_Modulus, toutes les autres valeurs positives plus petite ou égales peuvent être utilisées.

- Max_Base_Digits : **constant** := 15;

Le plus grand nombre de chiffres décimaux pour un type réel flottant.

- Max_Digits : **constant** := 15;

Le plus grand nombre de chiffres décimaux pour un type réel flottant pour lequel on ne précise pas une plage de validité. Il est plus petit ou égal à Max_Base_Digits.

- Max_Mantissa : **constant** := 31;

Le nombre de chiffres binaires d'un réel point fixe.

- Fine_Delta : **constant** := 2.0 ** (-31);

Egal à 2.0 ** (-Max_Mantissa)

- Tick : **constant** := 0.015_625;

Fréquence de l'horloge en secondes.

- **type** Address **is access** Integer;
- Null_Address : **constant** Address := **null**;

Le type adresse sur le système et l'adresse "nulle".

- Storage_Unit : **constant** := 8;

Le nombre de bits pour un élément mémoire.

- Word_Size : **constant** := 4 * Storage_Unit;

Le nombre de bits d'un mot.

- Memory_Size : **constant** := 2**32;

La taille de la mémoire, exprimée en éléments de mémoire.

- **type** Bit_Order **is** (High_Order_First, Low_Order_First);
- Default_Bit_Order : **constant** Bit_Order := Low_Order_First;

L'ordre dans lequel le système traite les bits.

- **subtype** Any_Priority **is** Integer **range** 1 .. 7;
- **subtype** Priority **is** Any_Priority **range** Any_Priority'first .. Any_Priority'last;
- **subtype** Interrupt_Priority **is** Any_Priority **range** Priority'last + 1 .. Any_Priority'last;
- Default_Priority : **constant** Priority := (Priority'first + Priority'last)/2;

Pour la gestion des différents niveaux de priorité.

Paquetage enfant System.Storage_Elements

```
package System.Storage_Elements is
        pragma Preelaborate ( System.Storage_Elements );
        type Storage_Offset is range Implementation-Defined;
        subtype Storage_Count is Storage_Offset range 0..Storage_Offset'Last;
        type Storage_Element is mod Implementation-Defined;
        for Storage_Element'Size use Storage_Unit;
        type Storage_Array is array  ( Storage_Offset range <> ) of aliased Storage_Element;
        for Storage_Array'Component_Size use Storage_Unit;

        -- Address Arithmetic:
        function "+" ( Left : Address; Right : Storage_Offset ) return Address;
        function "+" ( Left : Storage_Offset; Right : Address ) return Address;
        function "–" ( Left : Address; Right : Storage_Offset ) return Address;
        function "–" ( Left, Right : Address )return Storage_Offset;
        function "mod" ( Left : Address; Right : Storage_Offset ) return Storage_Offset;

        -- Conversion to/from integers:
        type Integer_Address is Implementation-Defined;
        function To_Address  ( Value : Integer_Address ) return Address;
        function To_Integer  ( Value : Address ) return Integer_Address;
        pragma Convention ( Intrinsic, "+" );
        -- ...and so on for all language-defined subprograms declared in this package.
end System.Storage_Elements;
```

Ce paquetage enfant contient les éléments nécessaires pour manipuler des adresses. Storage_Element représente un élément mémoire, alors que Storage_Array désigne une zone contiguë de celle-ci. Les opérateurs arithmétiques permettent de faire du calcul sur des

adresses et 2 fonctions convertissent des adresses en entiers et inversement.

Les valeurs spécifiques, dans notre environnement ObjectAda sous Windows NT:

```
type Storage_Offset is range Min_Int .. Max_Int;
subtype Storage_Count is Storage_Offset range 0..Storage_Offset'last;
-- Should be unsigned
type Storage_Element is mod 2 ** Storage_Unit;
for Storage_Element'size use Storage_Unit;
type Storage_Array is array ( Storage_Offset range <> ) of aliased Storage_Element;
for Storage_Array'component_Size use Storage_Unit;
type Integer_Address is new Integer;
```

Paquetage enfant System.Adress_To_Access_Conversions

```
generic
        type Object ( <>) is limited private;
package System.Address_To_Access_Conversions is
        pragma Preelaborate ( Address_To_Access_Conversions );
        type Object_Pointer is access all Object;
        function To_Pointer ( Value : Address ) return Object_Pointer;
        function To_Address ( Value : Object_Pointer ) return Address;
        pragma Convention ( Intrinsic, To_Pointer );
        pragma Convention ( Intrinsic, To_Address );
end System.Address_To_Access_Conversions;
```

Comme son nom le laisse supposer, ce paquetage enfant générique permet de convertir des adresses en pointeurs et inversement.

Paquetage Ada.Numerics et ses enfants

Le paquetage parent Numerics ne contient que peu de chose, puisque l'on y trouve une exception qui peut être levée lorsque les paramètres n'ont pas des valeurs raisonnables, ainsi que les 2 constantes usuelle "Pi" et "e". Notez pour ces 2 constantes, qui sont des réels universels, la précision donnée; n'oublions pas que certains enfants de Numerics sont des génériques qui peuvent être instanciés avec n'importe quel type réel, donc théoriquement une précision qui peut être très grande, pour autant que notre système le supporte!

```
package Ada.Numerics is
        pragma Pure  ( Numerics );
        Argument_Error : exception;
        Pi : constant :=

        3.14159_26535_89793_23846_26433_83279_50288_41971_69399_37511;
        e  : constant :=

        2.71828_18284_59045_23536_02874_71352_66249_77572_47093_69996;
end Ada.Numerics;
```

Génération de valeurs aléatoires

Deux paquetages permettent de générer des grandeurs aléatoires.

Le paquetage **Ada.Numerics.Float_Random** pour la génération de valeurs aléatoires du type Float comprises entre 0.0 et 1.0 (plus précisément le sous-type: Uniformly_Distributed).

```
package Ada.Numerics.Float_Random is
    -- Basic facilities
    type Generator is limited private;
    subtype Uniformly_Distributed is Float range 0.0 .. 1.0;
    function Random ( Gen : Generator ) return Uniformly_Distributed;
    procedure Reset ( Gen      : in Generator; Initiator : in Integer );
    procedure Reset ( Gen      : in Generator );

    -- Advanced facilities
    type State is private;
    procedure Save  ( Gen      : in  Generator; To_State   : out State );
    procedure Reset ( Gen      : in  Generator; From_State : in  State );
    Max_Image_Width : constant := Implementation-Defined Integer Value;
    function Image ( Of_State    : State )  return String;
    function Value ( Coded_State : String ) return State;

    private
    ... -- not specified by the language
end Ada.Numerics.Float_Random;
```

Quelques explications:

- Le type Generator étant limité privé, nous ne disposons pour travailler avec lui que des fonctionnalités mises à disposition par le paquetage. Ce travail commencera par la déclaration d'au moins une variable du type Generator.
- Ensuite il serait raisonnable d'initialiser notre générateur. Pour cette opération nous disposons de 2 procédures (en fait une surchargée!). Il s'agit de la procédure Reset; sa version avec un seul paramètre initialise le générateur sur une base de temps fournie par l'horloge; dans ce cas, chaque nouvelle initialisation (appel à Reset) nous mettra dans une situation nouvelle, non reproductible. Par contre l'utilisation du Reset avec un deuxième paramètre du type Integer, nous permet, lors d'une réinitialisation avec la même valeur, d'obtenir une série identique, donc de reproduire la même situation. Finalement si l'on appelle pas Reset, on obtiendra toujours la même série.
- Les appels successifs à Random qui a comme paramètre le générateur choisi, nous fourniront la série de valeurs pseudo aléatoires désirée, dans l'intervalle 0.0 .. 1.0.
- Les points ci-dessus forment la fonctionnalité de base et c'est celle que nous utilisons le plus souvent; il existe toutefois quelques compléments:
    - Le type Generator étant limité privé, on ne peut pas sauver l'état courant d'un travail, pour le reprendre ultérieurement par exemple. Le type State lui n'étant que privé, permet l'affectation, donc le sauvetage. La procédure Save permet de préserver l'état interne d'un générateur et la procédure Reset (troisième forme) rétablit la génération (le générateur) en son état lors du Save. Une simulation peut ainsi être arrêtée, analysée et reprise à volonté, depuis l'endroit où elle avait été interrompue. Pour la procédure Save, le paramètre d'entrée est une variable de type Generator et le paramètre de sortie est une variable de type State. Pour la procédure Reset, les deux paramètres d'entrée sont : la variable de type Generator et une variable de type State.
    - Les fonctions Image et Value offrent la possibilité de convertir un état en chaîne et réciproquement, pour une sauvegarde externe facilité; la longueur maximum de la chaîne est donnée par la constante Max_Image_Width, elle dépend de l'implémentation. Pour la fonction Image, le paramètre d'entrée est une variable de type State et elle retourne une chaîne de caractères. Pour la fonction Value, le paramètre d'entrée est une variable de type String et elle retourne l'état de type State. Le contenu de la chaîne de caractères n'est pas compréhensible et dépend de l'implémentation.

Une version générique équivalente existe. Elle doit être instanciée avec un type discret. Ce type définira la plage de valeurs qui pourront être rendues par la fonction random. Voici la spécification de cette version générique:

```
generic
        type Result_Subtype is (<>);
package Ada.Numerics.Discrete_Random is
        -- Basic facilities
        type Generator is limited private;
        function Random ( Gen : Generator ) return Result_Subtype;
        procedure Reset ( Gen  : in Generator; Initiator : in Integer );
        procedure Reset ( Gen  : in Generator );

        -- Advanced facilities
        type State is private;
        procedure Save  ( Gen : in  Generator; To_State   : out State );
```

```
      procedure Reset ( Gen : in  Generator, From_State : in  State );
      Max_Image_Width : constant := Implementation-Defined Integer Value;
      function Image ( Of_State : State )  return String;
      function Value ( Coded_State : String ) return State;

      private
      ... -- not specified by the language
end Ada.Numerics.Discrete_Random;
```

Un exemple d'utilisation:

```
-- Programme simulant le jet d'un dé
with Ada.Numerics.Discrete_Random;
with Ada.Text_Io;            use Ada.Text_Io;
with Ada.Integer_Text_Io;    use Ada.Integer_Text_Io;
with Ada.Float_Text_Io;      use Ada.Float_Text_Io;

procedure Jeu_De_Dé is
  subtype T_Dé is Integer range 1 .. 6;   -- Pour les 6 faces du dé
  -- Pour la création du simulateur de jet du dé
  package Random_Dé is new Ada.Numerics.Discrete_Random (T_Dé);
  use Random_Dé;
  Générateur : Generator;
  -- Pour compter le nombre d'apparitions de chaques faces
  Compteurs : array ( T_Dé ) of Natural := ( others => 0 );
  Nb_Lancés : Positive;      -- Nombre de jets de dé à simuler
  Face : T_Dé;               -- Face du dé tirée par le générateur
begin    -- Jeu_De_Dé

  Reset (Générateur);  -- Initialise le générateur
  Put ( "Combien de jet de des voulez-vous simuler: ");
  Get ( Nb_Lancés );
  -- Simuler tous les jets
  for I in 1 .. Nb_Lancés loop
    -- Faire le jet et incrémenter le compteur correspondant
    Face := Random ( Générateur );
    Compteurs ( Face ) := Compteurs ( Face ) + 1 ;
  end loop;
  -- Afficher les résultats
  New_Line (3);
  Put_Line ( "Les resultats de la simulation:" );
  -- Traiter les 6 faces du dé
  for I in T_Dé loop
    Put ( "Le" ); Put ( I, 3 );
    Put ( " est apparu: " ); Put ( Compteurs ( I ), 1 );
    Put ( " soit: " );
    Put ( Float(Compteurs(I)) / Float(Nb_Lancés) * 100.0, 3, 1, 0 );
    New_Line;
  end loop;

end Jeu_De_Dé;
```

Fonctions mathématiques élémentaires

A nouveau il existe 2 versions avec les mêmes fonctionnalités, l'une générique qu'il faudra instancier avec le type réel désiré, l'autre, non générique, dédiée au type Float. Nous ne donnons ci-dessous que la version générique de la spécification du paquetage:

```
generic
   type Float_Type is digits <>;
package Ada.Numerics.Generic_Elementary_Functions is
   pragma Pure       ( Generic_Elementary_Functions );
   function Sqrt     ( X           : Float_Type'Base )          return Float_Type'Base;
   function Log      ( X           : Float_Type'Base )          return Float_Type'Base;
   function Log      ( X, Base     : Float_Type'Base )          return Float_Type'Base;
   function Exp      ( X           : Float_Type'Base )          return Float_Type'Base;
   function "**"     ( Left, Right : Float_Type'Base )          return Float_Type'Base;

   function Sin      ( X           : Float_Type'Base )          return Float_Type'Base;
   function Sin      ( X, Cycle    : Float_Type'Base )          return Float_Type'Base;
   function Cos      ( X           : Float_Type'Base )          return Float_Type'Base;
   function Cos      ( X, Cycle    : Float_Type'Base )          return Float_Type'Base;
   function Tan      ( X           : Float_Type'Base )          return Float_Type'Base;
   function Tan      ( X, Cycle    : Float_Type'Base )          return Float_Type'Base;
   function Cot      ( X           : Float_Type'Base )          return Float_Type'Base;
   function Cot      ( X, Cycle    : Float_Type'Base )          return Float_Type'Base;
   function Arcsin   ( X           : Float_Type'Base )          return Float_Type'Base;
   function Arcsin   ( X, Cycle    : Float_Type'Base )          return Float_Type'Base;
   function Arccos   ( X           : Float_Type'Base )          return Float_Type'Base;
   function Arccos   ( X, Cycle    : Float_Type'Base )          return Float_Type'Base;
   function Arctan   ( Y           : Float_Type'Base;
                       X           : Float_Type'Base := 1.0 )   return Float_Type'Base;
   function Arctan   ( Y           : Float_Type'Base;
                       X           : Float_Type'Base := 1.0;
                       Cycle       : Float_Type'Base )          return Float_Type'Base;
   function Arccot   ( X           : Float_Type'Base;
                       Y           : Float_Type'Base := 1.0 )   return Float_Type'Base;
   function Arccot   ( X           : Float_Type'Base;
                       Y           : Float_Type'Base := 1.0;
                       Cycle       : Float_Type'Base )          return Float_Type'Base;
   function Sinh     ( X           : Float_Type'Base )          return Float_Type'Base;
   function Cosh     ( X           : Float_Type'Base )          return Float_Type'Base;
   function Tanh     ( X           : Float_Type'Base )          return Float_Type'Base;
   function Coth     ( X           : Float_Type'Base )          return Float_Type'Base;
   function Arcsinh  ( X           : Float_Type'Base )          return Float_Type'Base;
   function Arccosh  ( X           : Float_Type'Base )          return Float_Type'Base;
   function Arctanh  ( X           : Float_Type'Base )          return Float_Type'Base;
   function Arccoth  ( X           : Float_Type'Base )          return Float_Type'Base;
end Ada.Numerics.Generic_Elementary_Functions;
```

Quelques explications:

- Notons l'utilisation du type Float_Type'Base pour les paramètres et les résultats des fonctions, ce qui évite les contrôles de contraintes.
- L'exception Argument_Error sera levée si l'on appelle Sqrt avec un paramètre négatif (elle vient de son parent: Ada.Numerics!)
- La fonction Log avec un seul paramètre donne comme résultat le logarithme népérien, alors que celle avec 2 paramètres utilise le deuxième comme base du logarithme.
- La fonction "**" permet l'élévation à une puissance réelle (non négative), alors que l'opérateur de base ne traite que les entiers.
- Les fonction trigonométriques de base (Sin, Cos, Tan, Cot) sont aussi toutes surchargées, leur version avec un seul paramètre implique que l'angle est donné en radians, alors qu'avec la version à 2 paramètres, le deuxième permet à l'utilisateur de fixer n'importe quelle unité en donnant la valeur d'un cycle complet; par exemple pour travailler en degré, on donnera 360.0.
- La fonction Arcsin livre un résultat dans l'intervalle -Pi/2..Pi/2 et Arccos dans l'intervalle 0..Pi.
- Les fonction Arc… sont toutes surchargées, offrant ou non, comme les autres, la notion de cycle.
- Finalement on trouve les fonctions hyperboliques usuelles (Sinh, Cosh, Tanh, Coth, Arcsinh, Arccosh, Arctanh et Arccoth), pour lesquelles il n'y a rien de particulier à signaler.

Voici un petit exemple d'utilisation:

```
-- Programme exemple d'utilisation du paquetage générique
-- Ada.Numerics.Generic_Elementary_Functions
with Ada.Numerics.Generic_Elementary_Functions;
with Ada.Text_Io;     use Ada.Text_Io;
procedure Ex_Math_Lib is
  --
  -- Premier type réel et ses paquetages associés
  type Réel_1 is digits 6;
  package Réel_1_Io is new Float_Io ( Réel_1 ); use Réel_1_Io;
  package Math_Lib_1 is new
            Ada.Numerics.Generic_Elementary_Functions ( Réel_1 );
  --
  -- Deuxième type réel et ses paquetages associés
  type Réel_2 is digits 10;
  package Réel_2_Io is new Float_Io ( Réel_2 ); use Réel_2_Io;
  package Math_Lib_2 is new
           Ada.Numerics.Generic_Elementary_Functions ( Réel_2 );
```

```
begin   -- Ex_Math_Lib
  --
  -- Afficher la racine des nombres de 1 à 10
  for I in 1 .. 10 loop
    --
    -- Affichage non raisonnable en fonction du type, mais ...
    Put ( Math_Lib_1.Sqrt ( Réel_1 ( I ) ), 1, 9 );
    Put ( "     " );
    Put ( Math_Lib_2.Sqrt ( Réel_2 ( I ) ), 1, 9 );
    New_Line;
  end loop;

end Ex_Math_Lib;
```

L'exécution de ce programme nous donne les résultats suivants:

```
 1.000000000E+00   1.000000000E+00
 1.414213538E+00   1.414213562E+00
 1.732050776E+00   1.732050808E+00
 2.000000000E+00   2.000000000E+00
 2.236068010E+00   2.236067977E+00
 2.449489832E+00   2.449489743E+00
 2.645751238E+00   2.645751311E+00
 2.828427076E+00   2.828427125E+00
 3.000000000E+00   3.000000000E+00
 3.162277699E+00   3.162277660E+00
```

Réflexions sur ces résultats:

1) Bien entendu, il n'est pas raisonnable de travailler avec des nombres de 6 chiffres et d'afficher des résultats sur 9 chiffres. Les 3 chiffres supplémentaires demandés à l'affichage n'ont aucune signification, soyez-en convaincu!
2) Pourtant on obtient les mêmes valeurs sur plusieurs environnements et systèmes différents, même pour les valeurs calculées sur 6 chiffres et affichées sur 9!

Réfléchissez à ce que cela peut bien cacher!

Les nombres complexes

L'annexe G du manuel de référence nous donne un bon exemple des possibilités d'imbrication des paquetages et de l'utilisation des paquetages génériques en paramètre de la généricité. Nous avons construit dans ce cours plusieurs exemples de paquetages pour manipuler des nombres complexes. En fait cette annexe G nous met à disposition toute une série de paquetages, pour la manipulation des nombres complexes. Nous vous en donnons ci-dessous les spécifications, sans autres explications complémentaires puisqu'ils correspondent pour l'essentiel a des notions déjà étudiées dans d'autres contextes.

1) Tout d'abord le paquetage de base lui-même comportant la définition des types et des fonctions de base, certainement bien plus riche que l'ébauche de paquetage que nous avions construit:

```
generic
        type Real is digits <>;
package Ada.Numerics.Generic_Complex_Types is
        pragma Pure ( Generic_Complex_Types ) ;
        type Complex is record
                Re, Im : Real'Base;
        end record;
        type Imaginary is private;
        I : constant Imaginary;
        J : constant Imaginary;
        function Re ( X : Complex )    return Real'Base;
        function Im ( X : Complex )    return Real'Base;
        function Im ( X : Imaginary )  return Real'Base;

        procedure Set_Re ( X  : in out Complex; Re : in  Real'Base ) ;
        procedure Set_Im ( X  : in out Complex; Im : in  Real'Base ) ;
        procedure Set_Im ( X  :    out Imaginary; Im : in  Real'Base ) ;
        function Compose_From_Cartesian ( Re, Im : Real'Base )  return Complex;
        function Compose_From_Cartesian ( Re : Real'Base )  return Complex;
        function Compose_From_Cartesian ( Im : Imaginary )  return Complex;

        function Modulus   ( X       : Complex )  return Real'Base;
        function "abs"     ( Right   : Complex )  return Real'Base renames Modulus;
        function Argument  ( X       : Complex )    return Real'Base;
        function Argument  ( X       : Complex; Cycle : Real'Base )  return Real'Base;
        function Compose_From_Polar ( Modulus, Argument : Real'Base )  return Complex;
        function Compose_From_Polar ( Modulus, Argument, Cycle : Real'Base )  return Complex;

        function "+"     ( Right       : Complex )  return Complex;
        function "–"     ( Right       : Complex )  return Complex;
        function Conjugate ( X         : Complex )  return Complex;
        function "+" ( Left, Right     : Complex )  return Complex;
        function "–" ( Left, Right     : Complex )  return Complex;
        function "*" ( Left, Right     : Complex )  return Complex;
```

```
        function "/" ( Left, Right      : Complex )  return Complex;

        function "**"       ( Left : Complex; Right : Integer )   return Complex;
        function "+"        ( Right : Imaginary )                 return Imaginary;
        function "–"        ( Right : Imaginary )                 return Imaginary;
        function Conjugate  ( X     : Imaginary )                 return Imaginary renames "–";
        function "abs"      ( Right : Imaginary )                 return Real'Base;
        function "+"        ( Left, Right : Imaginary )           return Imaginary;
        function "–"        ( Left, Right : Imaginary )           return Imaginary;
        function "*"        ( Left, Right : Imaginary )           return Real'Base;
        function "/"        ( Left, Right : Imaginary )           return Real'Base;

        function "**"       ( Left : Imaginary; Right : Integer )       return Complex;
        function "<"        ( Left, Right : Imaginary )                 return Boolean;
        function "<="       ( Left, Right : Imaginary )                 return Boolean;
        function ">"        ( Left, Right : Imaginary )                 return Boolean;
        function ">="       ( Left, Right : Imaginary )                 return Boolean;
        function "+"        ( Left : Complex;   Right : Real'Base )     return Complex;
        function "+"        ( Left : Real'Base; Right : Complex )       return Complex;
        function "–"        ( Left : Complex;   Right : Real'Base )     return Complex;
        function "–"        ( Left : Real'Base; Right : Complex )       return Complex;
        function "*"        ( Left : Complex;   Right : Real'Base )     return Complex;
        function "*"        ( Left : Real'Base; Right : Complex )       return Complex;
        function "/"        ( Left : Complex;   Right : Real'Base )     return Complex;
        function "/"        ( Left : Real'Base; Right : Complex )       return Complex;

        function "+"        ( Left : Complex;   Right : Imaginary )     return Complex;
        function "+"        ( Left : Imaginary; Right : Complex )       return Complex;
        function "–"        ( Left : Complex;   Right : Imaginary )     return Complex;
        function "–"        ( Left : Imaginary; Right : Complex )       return Complex;
        function "*"        ( Left : Complex;   Right : Imaginary )     return Complex;
        function "*"        ( Left : Imaginary; Right : Complex )       return Complex;
        function "/"        ( Left : Complex;   Right : Imaginary )     return Complex;
        function "/"        ( Left : Imaginary; Right : Complex )       return Complex;

        function "+"        ( Left : Imaginary; Right : Real'Base )     return Complex;
        function "+"        ( Left : Real'Base; Right : Imaginary )     return Complex;
        function "–"        ( Left : Imaginary; Right : Real'Base )     return Complex;
        function "–"        ( Left : Real'Base; Right : Imaginary )     return Complex;
        function "*"        ( Left : Imaginary; Right : Real'Base )     return Imaginary;
        function "*"        ( Left : Real'Base; Right : Imaginary )     return Imaginary;
        function "/"        ( Left : Imaginary; Right : Real'Base )     return Imaginary;
        function "/"        ( Left : Real'Base; Right : Imaginary )     return Imaginary;

private
        type Imaginary is new Real'Base;
        I : constant Imaginary := 1.0;
        J : constant Imaginary := 1.0;
end Ada.Numerics.Generic_Complex_Types;
```

3) Ensuite le paquetage équivalent aux fonctions mathématiques élémentaires, mais adapté au type complexe nouvellement défini. Notez tout particulièrement l'utilisation du paquetage Ada.Numerics.Generic_Complex_Types comme paramètre générique.

```
with Ada.Numerics.Generic_Complex_Types;
generic
        with package Complex_Types is new Ada.Numerics.Generic_Complex_Types (<>);
        use Complex_Types;
package Ada.Numerics.Generic_Complex_Elementary_Functions is
        pragma Pure ( Generic_Complex_Elementary_Functions ) ;
        function Sqrt  ( X : Complex )    return Complex;
        function Log   ( X : Complex )    return Complex;
        function Exp   ( X : Complex )    return Complex;
        function Exp   ( X : Imaginary )  return Complex;
        function "**" ( Left : Complex;   Right : Complex )       return Complex;
        function "**" ( Left : Complex;   Right : Real'Base )     return Complex;
        function "**" ( Left : Real'Base; Right : Complex )       return Complex;

        function Sin      ( X : Complex )     return Complex;
        function Cos      ( X : Complex )     return Complex;
        function Tan      ( X : Complex )     return Complex;
        function Cot      ( X : Complex )     return Complex;
        function Arcsin   ( X : Complex )     return Complex;
        function Arccos   ( X : Complex )     return Complex;
        function Arctan   ( X : Complex )     return Complex;
        function Arccot   ( X : Complex )     return Complex;

        function Sinh     ( X : Complex )     return Complex;
        function Cosh     ( X : Complex )     return Complex;
        function Tanh     ( X : Complex )     return Complex;
        function Coth     ( X : Complex )     return Complex;
        function Arcsinh  ( X : Complex )     return Complex;
        function Arccosh  ( X : Complex )     return Complex;
        function Arctanh  ( X : Complex )     return Complex;
        function Arccoth  ( X : Complex )     return Complex;

  end Ada.Numerics.Generic_Complex_Elementary_Functions;
```

3) Ensuite un paquetage pour les entrées/sorties de valeurs complexes, qui est un enfant de Ada.Text_Io. On y trouve les Put et Get usuels et ce paquetage ne demande certainement pas plus de commentaires. Notons qu'il existe l'équivalent: Ada.Wide_Text_Io.Complex_Io, de contenu identique, où il faut remplacer String par Wide_String.

```
with Ada.Numerics.Generic_Complex_Types;
generic
    with package Complex_Types is new Ada.Numerics.Generic_Complex_Types (<>);
package Ada.Text_Io.Complex_Io is
    use Complex_Types;
    Default_Fore : Field := 2;
    Default_Aft  : Field := Real'digits – 1;
    Default_Exp  : Field := 3;
    procedure Get ( File  : in  File_Type;
                    Item  : out Complex;
                    Width : in  Field := 0 ) ;
    procedure Get ( Item  : out Complex;
                    Width : in  Field := 0 ) ;

    procedure Put ( File  : in File_Type;
                    Item  : in Complex;
                    Fore  : in Field := Default_Fore;
                    Aft   : in Field := Default_Aft;
                    Exp   : in Field := Default_Exp ) ;
    procedure Put ( Item  : in Complex;
                    Fore  : in Field := Default_Fore;
                    Aft   : in Field := Default_Aft;
                    Exp   : in Field := Default_Exp ) ;

    procedure Get ( From  : in  String;
                    Item  : out Complex;
                    Last  : out Positive ) ;
    procedure Put ( To    : out String;
                    Item  : in  Complex;
                    Aft   : in  Field := Default_Aft;
                    Exp   : in  Field := Default_Exp ) ;
end Ada.Text_Io.Complex_Io;
```

Reprenons, à titre d'exemple le programme de test que nous avions utilisé pour la dernière version de notre paquetage Nombres_Complexes. Nous devons bien entendu modifier les clauses de contexte, ajouter les instanciations de Ada.Numerics.Generic_Complex_Types et de Ada.Text_Io.Complex_Io. Notez aussi l'utilisation du sous-type complexe (pseudo surnommage) pour éviter d'autres modifications dans le reste du source.

```
-- Le programme utilisant le paquetage dans un troisième fichier
-- OA 7.1.2 erreur compilation, 7.2 erreur exécution, GNAT ok!!!
with Ada.Text_Io;       use Ada.Text_Io;
with Ada.Numerics.Generic_Complex_Types;
with Ada.Text_Io.Complex_Io;
procedure Pcomplexe is
  package P_Complexe is new
                   Ada.Numerics.Generic_Complex_Types (Float);
  use P_Complexe;
  package Complexe_Io is new Ada.Text_Io.Complex_Io ( P_Complexe );
  use Complexe_Io;
  subtype Complexe is Complex;

  Op1, Op2 : Complexe;                -- Les opérandes
  Oper     : Character;               -- L'opération à réaliser
  Ok       : Boolean := True;         -- Vrai si expression correcte
  Espace   : constant Character := ' ';


begin -- Pcomplexe

  Put_Line ( "JE REALISE LES 4 OPERATIONS: +, -, *, / " );
  Put_Line ( "DONNEZ-MOI LE CALCUL SOUS LA FORME "      );
  Put_Line ( "OPERANDE1 OPERATEUR OPERANDE2 "           );
  Put_Line ( "LES OPERANDES SONT DES COMPLEXES"         );
  Put_Line ( "DE LA FORME ( RR.RR, II.II )"             );
  Get ( Op1 );                  -- Lecture d'un opérande complexe
  loop                          -- Sauter les espaces
    Get ( Oper );               -- Caractère suivant
    exit when Oper /= Espace;   -- Fin normale sur l'opérateur
  end loop;
  Get ( Op2 );                  -- Lecture du deuxième complexe
  --
  -- Réaliser l'opération demandée sur le type complexe
  case Oper is

    when '+' => Op1 := Op1 + Op2;     -- + complexe
    when '-' => Op1 := Op1 - Op2;     -- - complexe
    when '*' => Op1 := Op1 * Op2;     -- * complexe
    when '/' => Op1 := Op1 / Op2;     -- / complexe
    when others => -- Erreur

    Put_Line ( " ERREUR, RECOMMENCEZ" );
    Ok := False;

  end case;
  --
  -- Si le calcul est juste, afficher le résultat
  if Ok then
    Put ( Op1 );
    New_Line;
  end if;

end Pcomplexe;
```

Le paquetage Command_Line

Le paquetage Command_line permet de récupérer les éventuels paramètres transmis lors du lancement du programme. Bien entendu le comportement et la forme exacte de ce que l'on obtient dépend de l'environnement. Dans un environnement Unix pur et dur les éléments que vous ajoutez à la ligne de commande lors du lancement du programme. Sous ObjectAda (NT) vous avez une option qui permet dans une boîte de dialogue de définir les paramètres complémentaires que l'on veut donner. Sous GNAT (NT) une option permet de faire demander automatiquement la ligne de paramètres complémentaires lors de l'exécution. Voici le contenu de ce paquetage:

```
package Ada.Command_Line is
  pragma Preelaborate ( Command_Line );
  function Argument_Count return Natural;
  function Argument (  Number : in     Positive ) return String;
  function Command_Name return String;
  type Exit_Status is Implementation - Defined Integer type;
  Success : constant Exit_Status;
  Failure : constant Exit_Status;

  procedure Set_Exit_Status ( Code : in   Exit_Status );
  private
   ... -- not specified by the languageend Ada.Command_Line;
end Ada.Command_Line;
```

Quelques explications:

- La fonction Argument_Count qui n'a pas de paramètre livre un résultat entier positif ou nul, indiquant combien de paramètres se trouvent sur la ligne de commande. La manière de définir le nombre de paramètres dépend de l'environnement; généralement au moins un espace sépare 2 paramètres; l'éventuel nom de la commande pour lancer le programme et le nom du programme lui-même pouvant ou non être comptés dans cette valeur.
- La fonction Argument livre une chaîne de caractères représentant le contenu du paramètre de la ligne de commande dont le numéro est transmis en paramètre à la fonction.
- La fonction Command_Name qui n'a pas de paramètre, livre une chaîne de caractères représentant la commande qui a lancée le programme; là aussi le contenu peut varier considérablement d'un environnement à l'autre.
- La procédure Set_Exit_Status permet, si le système le supporte, de définir le code d'erreur qui lui sera retourné lors de la terminaison de l'exécution du programme. Ce code d'erreur, transmis en paramètre à la procédure est du type Exit_Status, définit par le paquetage et dépendant de l'implémentation. Le paquetage met également à disposition 2 constantes de ce type (Success et Failure) qui permettent de fixer une terminaison normale, respectivement en erreur.

Pour terminer, donnons un petit exemple rudimentaire d'utilisation de ce paquetage:

```
--
-- Récuperation des paramètres de la ligne de commande
with Ada.Text_Io;          use Ada.Text_Io;
with Ada.Command_Line;     use Ada.Command_Line;
procedure Cmdline is

  Nb_Paramètres : Natural := Argument_Count;

begin   -- CmdLine

  -- Affiche le nombre de paramètres
  Put_Line ( "Nombre de parametres: "
              & Natural'Image ( Nb_Parametres ) );
  New_Line;

  -- Affiche tous les paramètres
  for I in 1 .. Nb_Parametres loop
    Put_Line ( Argument ( I ) );
    New_Line;
  end loop;

  New_Line ( 2 );
  Put_Line ( Command_Name );
  New_Line ( 2 );
  Set_Exit_Status ( -3 );   -- A titre d'exemple

end Cmdline;
```

Traitement des caractères

Le paquetage parent pour le traitement des caractères est vide:

```
package Ada.Characters is
      pragma Pure(Characters);
end Ada.Characters;
```

Le paquetage enfant Ada.Characters.Handling met à disposition des fonctions pour classifier les caractères, ainsi que d'autres pour réaliser des conversions.

Voici sa spécification:

```
package Ada.Characters.Handling is
      pragma Preelaborate( Handling ) ;
      -- Character classification functions
      function Is_Control   ( Item : in Character ) return Boolean;
      function Is_Graphic   ( Item : in Character ) return Boolean;
      function Is_Letter    ( Item : in Character ) return Boolean;
      function Is_Lower     ( Item : in Character ) return Boolean;
      function Is_Upper     ( Item : in Character ) return Boolean;
      function Is_Basic     ( Item : in Character ) return Boolean;
      function Is_Digit     ( Item : in Character ) return Boolean;
      function Is_Decimal_Digit  ( Item : in Character ) return Boolean renames Is_Digit;
      function Is_Hexadecimal_Digit      ( Item : in Character ) return Boolean;
      function Is_Alphanumeric           ( Item : in Character ) return Boolean;
      function Is_Special                ( Item : in Character ) return Boolean;

      -- Conversion functions for Character and String
      function To_Lower   ( Item : in Character ) return Character;
      function To_Upper   ( Item : in Character ) return Character;
      function To_Basic   ( Item : in Character ) return Character;
      function To_Lower   ( Item : in String ) return String;
      function To_Upper   ( Item : in String ) return String;
      function To_Basic   ( Item : in String ) return String;

      -- Classifications of and conversions between Character and ISO 646
      subtype Iso_646 is Character range Character'Val( 0 ) .. Character'Val( 127 ) ;
      function Is_Iso_646  ( Item : in Character )          return Boolean;
      function Is_Iso_646  ( Item : in String )             return Boolean;
      function To_Iso_646 ( Item : in Character; Substitute : in Iso_646:= ' ')   return Iso_646;

      function To_Iso_646 ( Item  : in String; Substitute : in Iso_646 := ' ' )     return String;
      -- Classifications of and conversions between Wide_Character and Character.
      function Is_Character       ( Item  : in Wide_Character )       return Boolean;
      function Is_String          ( Item  : in Wide_String )          return Boolean;
      function To_Character       ( Item  : in Wide_Character;
                                    Substitute : in Character := ' ' )      return Character;
```

```
      function To_String              ( Item      : in Wide_String;
                                Substitute : in Character := ' ' )      return String;
      function To_Wide_Character      ( Item : in Character )          return Wide_Character;
      function To_Wide_String         ( Item : in String )             return Wide_String;
end Ada.Characters.Handling;
```

Quelques considérations sur ce paquetage:

- Is_Control — True si le paramètre est un caractère de contrôle, à savoir: un caractère dont la position se trouve dans les intervalles 0..31 ou 127..159.
- Is_Graphic — True si le paramètre est un caractère graphique, à savoir: un caractère dont la position se trouve dans les intervalles 32..126 ou 160..255.
- Is_Letter — True si le paramètre est une lettre, à savoir ‘A’..’Z’ ou ‘a’..’z’ , ou un caractère dont la position se trouve dans les intervalles 192..214, 216..246, ou 248..255.
- Is_Lower — True si le paramètre est une lettre minuscule, à savoir: ‘a’..’z’, ou un caractère dont la position se trouve dans les intervalles 223..246 ou 248..255.
- Is_Upper — True si le paramètre est une lettre majuscule, à savoir: ‘A’..’A’, ou un caractère dont la position se trouve dans les intervalles 192..214 ou 216.. 222.
- Is_Basic — True si le paramètre est une lettre de base, à savoir, ‘A’..’Z’ ou ‘a’..’z’, ou ‘Æ’, ‘æ’, ‘Ð’, ‘ð’, ‘Þ’, ‘þ’, or ‘ß’.
- Is_Decimal_Digit,
- Is_Digit — True si le paramètre est un chiffre décimal, à savoir ‘0’..’9’.
- Is_Hexadecimal_Digit — True si le paramètre est un chiffre hexadécimal, à savoir ‘0’..’9’ ou ‘A’.. ‘F’ ou ‘a’.. ‘f’.
- Is_Alphanumeric — True si le paramètre est un caractère alphanumérique, à savoir une lettre ou un chiffre.
- Is_Special — True si le paramètre est un caractère graphique, à savoir si ce n'est pas un caractère alphanumérique.

Le paquetage enfant Ada.Character.Latin_1 ne contient qu'une définition symbolique pour chacun des caractères du code ISO Latin_1. Nous vous donnons ci-dessous, simplement à titre indicatif le contenu de ce paquetage (regardez simplement les 16 premiers, qui sont les caractères de contrôle, si vous ne les connaissez pas!):

```
package Ada.Characters.Latin_1 is
      pragma Pure(Latin_1);
      -- Control characters:
      Nul     : constant Character := Character'Val(0);
      Soh     : constant Character := Character'Val(1);
      Stx     : constant Character := Character'Val(2);
      Etx     : constant Character := Character'Val(3);
      Eot     : constant Character := Character'Val(4);
      Enq     : constant Character := Character'Val(5);
      Ack     : constant Character := Character'Val(6);
      Bel     : constant Character := Character'Val(7);
      Bs      : constant Character := Character'Val(8);
      Ht      : constant Character := Character'Val(9);
      Lf      : constant Character := Character'Val(10);
```

```
Vt          : constant Character := Character'Val(11);
Ff          : constant Character := Character'Val(12);
Cr          : constant Character := Character'Val(13);
So          : constant Character := Character'Val(14);
Si          : constant Character := Character'Val(15);

Dle         : constant Character := Character'Val(16);
Dc1         : constant Character := Character'Val(17);
Dc2         : constant Character := Character'Val(18);
Dc3         : constant Character := Character'Val(19);
Dc4         : constant Character := Character'Val(20);
Nak         : constant Character := Character'Val(21);
Syn         : constant Character := Character'Val(22);
Etb         : constant Character := Character'Val(23);
Can         : constant Character := Character'Val(24);
Em          : constant Character := Character'Val(25);
Sub         : constant Character := Character'Val(26);
Esc         : constant Character := Character'Val(27);
Fs          : constant Character := Character'Val(28);
Gs          : constant Character := Character'Val(29);
Rs          : constant Character := Character'Val(30);
Us          : constant Character := Character'Val(31);

-- ISO 646 graphic characters:
Space               : constant Character := ' ';  -- Character'Val(32)
Exclamation         : constant Character : := '!';  -- Character'Val(33)
Quotation           : constant Character := '"';  -- Character'Val(34)
Number_Sign         : constant Character := '#';  -- Character'Val(35)
Dollar_Sign         : constant Character := '$';  -- Character'Val(36)
Percent_Sign        : constant Character := '%';  -- Character'Val(37)
Ampersand           : constant Character := '&';  -- Character'Val(38)
Apostrophe          : constant Character := ''';  -- Character'Val(39)
Left_Parenthesis    : constant Character := '(';  -- Character'Val(40)
Right_Parenthesis   : constant Character := ')';  -- Character'Val(41)
Asterisk            : constant Character := '*';  -- Character'Val(42)
Plus_Sign           : constant Character := '+';  -- Character'Val(43)
Comma               : constant Character := ',';  -- Character'Val(44)
Hyphen              : constant Character := '-';  -- Character'Val(45)
Minus_Sign          : Character renames Hyphen;
Full_Stop           : constant Character := '.';  -- Character'Val(46)
Solidus             : constant Character := '/';  -- Character'Val(47)

-- Decimal digits '0' though '9' are at positions 48 through 57
Colon               : constant Character := ':';  -- Character'Val(58)
Semicolon           : constant Character := ';';  -- Character'Val(59)
Less_Than_Sign      : constant Character := '<';  -- Character'Val(60)
Equals_Sign         : constant Character := '=';  -- Character'Val(61)
Greater_Than_Sign   : constant Character := '>';  -- Character'Val(62)
Question            : constant Character := '?';  -- Character'Val(63)
Commercial_At       : constant Character := '@';  -- Character'Val(64)

-- Letters 'A' through 'Z' are at positions 65 through 90
Left_Square_Bracket         : constant Character := '[';  -- Character'Val(91)
Reverse_Solidus             : constant Character := '\';  -- Character'Val(92)
Right_Square_Bracket        : constant Character := ']';  -- Character'Val(93)
Circumflex                  : constant Character := '^';  -- Character'Val(94)
Low_Line            : constant Character := '_';  -- Character'Val(95)

Grave       : constant Character := '`';  -- Character'Val(96)
Lc_A        : constant Character := 'a';  -- Character'Val(97)
Lc_B        : constant Character := 'b';  -- Character'Val(98)
Lc_C        : constant Character := 'c';  -- Character'Val(99)
Lc_D        : constant Character := 'd';  -- Character'Val(100)
Lc_E        : constant Character := 'e';  -- Character'Val(101)
Lc_F        : constant Character := 'f';  -- Character'Val(102)
Lc_G        : constant Character := 'g';  -- Character'Val(103)
Lc_H        : constant Character := 'h';  -- Character'Val(104)
Lc_I        : constant Character := 'i';  -- Character'Val(105)
Lc_J        : constant Character := 'j';  -- Character'Val(106)
Lc_K        : constant Character := 'k';  -- Character'Val(107)
Lc_L        : constant Character := 'l';  -- Character'Val(108)
```

```
Lc_M      : constant Character := 'm';  -- Character'Val(109)
Lc_N      : constant Character := 'n';  -- Character'Val(110)
Lc_O      : constant Character := 'o';  -- Character'Val(111)

Lc_P      : constant Character := 'p';  -- Character'Val(112)
Lc_Q      : constant Character := 'q';  -- Character'Val(113)
Lc_R      : constant Character := 'r';  -- Character'Val(114)
Lc_S      : constant Character := 's';  -- Character'Val(115)
Lc_T      : constant Character := 't';  -- Character'Val(116)
Lc_U      : constant Character := 'u';  -- Character'Val(117)
Lc_V      : constant Character := 'v';  -- Character'Val(118)
Lc_W      : constant Character := 'w';  -- Character'Val(119)
Lc_X      : constant Character := 'x';  -- Character'Val(120)
Lc_Y      : constant Character := 'y';  -- Character'Val(121)
Lc_Z      : constant Character := 'z';  -- Character'Val(122)
Left_Curly_Bracket        : constant Character := '{';  -- Character'Val(123)
Vertical_Line             : constant Character := '|';  -- Character'Val(124)
Right_Curly_Bracket       : constant Character := '}';  -- Character'Val(125)
Tilde                     : constant Character := '~';  -- Character'Val(126)
Del                       : constant Character := Character'Val(127);

-- ISO 6429 control characters:
Is4       : Character renames Fs;
Is3       : Character renames Gs;
Is2       : Character renames Rs;
Is1       : Character renames Us;
Reserved_128      : constant Character := Character'Val(128);
Reserved_129      : constant Character := Character'Val(129);
Bph               : constant Character := Character'Val(130);
Nbh               : constant Character := Character'Val(131);
Reserved_132      : constant Character := Character'Val(132);
Nel               : constant Character := Character'Val(133);
Ssa               : constant Character := Character'Val(134);
Esa               : constant Character := Character'Val(135);
Hts               : constant Character := Character'Val(136);
Htj               : constant Character := Character'Val(137);
Vts               : constant Character := Character'Val(138);
Pld               : constant Character := Character'Val(139);
Plu               : constant Character := Character'Val(140);
Ri                : constant Character := Character'Val(141);
Ss2               : constant Character := Character'Val(142);
Ss3               : constant Character := Character'Val(143);

Dcs               : constant Character := Character'Val(144);
Pu1               : constant Character := Character'Val(145);
Pu2               : constant Character := Character'Val(146);
Sts               : constant Character := Character'Val(147);
Cch               : constant Character := Character'Val(148);
Mw                : constant Character := Character'Val(149);
Spa               : constant Character := Character'Val(150);
Epa               : constant Character := Character'Val(151);

Sos               : constant Character := Character'Val(152);
Reserved_153      : constant Character := Character'Val(153);
Sci               : constant Character := Character'Val(154);
Csi               : constant Character := Character'Val(155);
St                : constant Character := Character'Val(156);
Osc               : constant Character := Character'Val(157);
Pm                : constant Character := Character'Val(158);
Apc               : constant Character := Character'Val(159);

-- Other graphic characters:
-- Character positions 160 (16#A0#) .. 175 (16#AF#):
No_Break_Space            : constant Character := ' ';  -- Character'Val(160)
Nbsp      : Character renames No_Break_Space;
Inverted_Exclamation      : constant Character := '¡';  -- Character'Val(161)
Cent_Sign                 : constant Character := '¢';  -- Character'Val(162)
Pound_Sign                : constant Character := '£';  -- Character'Val(163)
Currency_Sign             : constant Character := '¤';  -- Character'Val(164)
Yen_Sign                  : constant Character := '¥';  -- Character'Val(165)
Broken_Bar                : constant Character := '¦';  -- Character'Val(166)
```

```
Section_Sign                    : constant Character := '§';  -- Character'Val(167)
Diaeresis                       : constant Character := '¨';  -- Character'Val(168)
Copyright_Sign                  : constant Character := '©';  -- Character'Val(169)
Feminine_Ordinal_Indicator      : constant Character := 'ª';  -- Character'Val(170)
Left_Angle_Quotation            : constant Character := '«';  -- Character'Val(171)
Not_Sign                        : constant Character := '¬';  -- Character'Val(172)
Soft_Hyphen                     : constant Character := '-';  -- Character'Val(173)
Registered_Trade_Mark_Sign      : constant Character := '®';  -- Character'Val(174)
Macron                          : constant Character := '¯';  -- Character'Val(175)

-- Character positions 176 (16#B0#) .. 191 (16#BF#):
Degree_Sign                     : constant Character := '°';  -- Character'Val(176)
Ring_Above : Character renames Degree_Sign;
Plus_Minus_Sign     : constant Character := '±';  -- Character'Val(177)
Superscript_Two     : constant Character := '²';  -- Character'Val(178)
Superscript_Three   : constant Character := '³';  -- Character'Val(179)
Acute               : constant Character := '´';  -- Character'Val(180)
Micro_Sign          : constant Character := 'µ';  -- Character'Val(181)
Pilcrow_Sign        : constant Character := '¶';  -- Character'Val(182)
Paragraph_Sign      : Character renames Pilcrow_Sign;
Middle_Dot          : constant Character := '·';  -- Character'Val(183)
Cedilla             : constant Character := '¸';  -- Character'Val(184)
Superscript_One     : constant Character := '¹';  -- Character'Val(185)
Masculine_Ordinal_Indicator : constant Character := 'º';  -- Character'Val(186)
Right_Angle_Quotation       : constant Character := '»';  -- Character'Val(187)
Fraction_One_Quarter        : constant Character := '¼';  -- Character'Val(188)
Fraction_One_Half           : constant Character := '½';  -- Character'Val(189)
Fraction_Three_Quarters     : constant Character := '¾';  -- Character'Val(190)
Inverted_Question           : constant Character := '¿';  -- Character'Val(191)

-- Character positions 192 (16#C0#) .. 207 (16#CF#):
Uc_A_grave          : constant Character := 'À';  -- Character'Val(192)
Uc_A_acute          : constant Character := 'Á';  -- Character'Val(193)
Uc_A_circumflex     : constant Character := 'Â';  -- Character'Val(194)
Uc_A_tilde          : constant Character := 'Ã';  -- Character'Val(195)
Uc_A_diaeresis      : constant Character := 'Ä';  -- Character'Val(196)
Uc_A_ring           : constant Character := 'Å';  -- Character'Val(197)
Uc_Ae_Diphthong     : constant Character := 'Æ';  -- Character'Val(198)
Uc_C_cedilla        : constant Character := 'Ç';  -- Character'Val(199)
Uc_E_grave          : constant Character := 'È';  -- Character'Val(200)
Uc_E_acute          : constant Character := 'É';  -- Character'Val(201)
Uc_E_circumflex     : constant Character := 'Ê';  -- Character'Val(202)
Uc_E_diaeresis      : constant Character := 'Ë';  -- Character'Val(203)
Uc_I_grave          : constant Character := 'Ì';  -- Character'Val(204)
Uc_I_acute          : constant Character := 'Í';  -- Character'Val(205)
Uc_I_circumflex     : constant Character := 'Î';  -- Character'Val(206)
Uc_I_diaeresis      : constant Character := 'Ï';  -- Character'Val(207)

-- Character positions 208 (16#D0#) .. 223 (16#DF#):
Uc_Icelandic_Eth    : constant Character := 'Ð';  -- Character'Val(208)
Uc_N_tilde          : constant Character := 'Ñ';  -- Character'Val(209)
Uc_O_grave          : constant Character := 'Ò';  -- Character'Val(210)
Uc_O_acute          : constant Character := 'Ó';  -- Character'Val(211)
Uc_O_circumflex     : constant Character := 'Ô';  -- Character'Val(212)
Uc_O_tilde          : constant Character := 'Õ';  -- Character'Val(213)
Uc_O_diaeresis      : constant Character := 'Ö';  -- Character'Val(214)
Multiplication_Sign : constant Character := '×';  -- Character'Val(215)
Uc_O_oblique_stroke         : constant Character := 'Ø';  -- Character'Val(216)
Uc_U_grave                  : constant Character := 'Ù';  -- Character'Val(217)
Uc_U_acute                  : constant Character := 'Ú';  -- Character'Val(218)
Uc_U_circumflex             : constant Character := 'Û';  -- Character'Val(219)
Uc_U_diaeresis              : constant Character := 'Ü';  -- Character'Val(220)
Uc_Y_acute                  : constant Character := 'Ý';  -- Character'Val(221)
Uc_Icelandic_Thorn          : constant Character := 'Þ';  -- Character'Val(222)
Lc_German_Sharp_S           : constant Character := 'ß';  -- Character'Val(223)

-- Character positions 224 (16#E0#) .. 239 (16#EF#):
Lc_A_grave          : constant Character := 'à';  -- Character'Val(224)
Lc_A_acute          : constant Character := 'á';  -- Character'Val(225)
Lc_A_circumflex     : constant Character := 'â';  -- Character'Val(226)
Lc_A_tilde          : constant Character := 'ã';  -- Character'Val(227)
```

```
      Lc_A_diaeresis      : constant Character := 'ä';  -- Character'Val(228)
      Lc_A_ring           : constant Character := 'å';  -- Character'Val(229)
      Lc_Ae_Diphthong     : constant Character := 'æ';  -- Character'Val(230)
      Lc_C_cedilla        : constant Character := 'ç';  -- Character'Val(231)
      Lc_E_grave          : constant Character := 'è';  -- Character'Val(232)
      Lc_E_acute          : constant Character := 'é';  -- Character'Val(233)
      Lc_E_circumflex     : constant Character := 'ê';  -- Character'Val(234)
      Lc_E_diaeresis      : constant Character := 'ë';  -- Character'Val(235)
      Lc_I_grave          : constant Character := 'ì';  -- Character'Val(236)
      Lc_I_acute          : constant Character := 'í';  -- Character'Val(237)
      Lc_I_circumflex     : constant Character := 'î';  -- Character'Val(238)
      Lc_I_diaeresis      : constant Character := 'ï';  -- Character'Val(239)

      -- Character positions 240 (16#F0#) .. 255 (16#FF#):
      Lc_Icelandic_Eth    : constant Character := 'ð';  -- Character'Val(240)
      Lc_N_tilde          : constant Character := 'ñ';  -- Character'Val(241)
      Lc_O_grave          : constant Character := 'ò';  -- Character'Val(242)
      Lc_O_acute          : constant Character := 'ó';  -- Character'Val(243)
      Lc_O_circumflex     : constant Character := 'ô';  -- Character'Val(244)
      Lc_O_tilde          : constant Character := 'õ';  -- Character'Val(245)
      Lc_O_diaeresis      : constant Character := 'ö';  -- Character'Val(246)
      Division_Sign       : constant Character := '÷';  -- Character'Val(247)
      Lc_O_oblique_stroke           : constant Character := 'ø';  -- Character'Val(248)
      Lc_U_grave          : constant Character := 'ù';  -- Character'Val(249)
      Lc_U_acute          : constant Character := 'ú';  -- Character'Val(250)
      Lc_U_circumflex     : constant Character := 'û';  -- Character'Val(251)
      Lc_U_diaeresis      : constant Character := 'ü';  -- Character'Val(252)
      Lc_Y_acute          : constant Character := 'ý';  -- Character'Val(253)
      Lc_Icelandic_Thorn  : constant Character := 'þ';  -- Character'Val(254)
      Lc_Y_diaeresis      : constant Character := 'ÿ';  -- Character'Val(255)
end Ada.Characters.Latin_1;
```