Quelques outils pour les projets

Thierry Lacoste

27 octobre 2007

Chapitre 1

Les chaînes de caractères

1.1 Le type `String`

Le type prédéfini `String` permet de représenter les chaînes de caractères. Il est défini dans le paquetage `Standard` par :

```
type String is array (Positive range <>) of Character;
```

Par conséquent, il est impossible de déclarer une variable de ce type sans fournir ses contraintes d'indices.

⋆

La manipulation des chaînes de caractères peut donc s'avèrer laborieuse. En effet, il faut dimensionner une chaîne à la taille maximale qu'on lui prévoit. Par exemple, si on veut une chaîne qui ne peut contenir plus de 80 caractère, on déclare :

```
S : String (1 .. 80);
```

1.1.1 Saisie des chaînes

Le paquetage `Text_IO` contient essentiellement deux procédures de saisie d'une chaîne, dont les spécifications sont :

```
procedure Get (Item : out String);
procedure Get_Line (Item : out String; Last : out Natural);
```

L'appel `Get (S)` déclenche l'attente de la saisie de 80 caractères exactement (les retours à la ligne sont ignorés).

⋆

Si on veut que la saisie s'arrête au retour à la ligne, il faut avoir une variable (disons `N`) de type `Integer` de façon à faire l'appel `Get_Line (S, N)` qui range les caractères entrés au clavier au début de `S` et retourne dans `N` le nombre de caractères saisie. Il faudra utiliser les tranches pour récupérer la partie significative de `S` (la "fin" de `S` contient n'importe quoi).

⋆

Exemple : le programme strings.adb[1] illustre la manipulation des chaînes ; il permet de visualiser à l'écran le contenu d'un fichier.

1.2 Le type `Unbounded_String` (*en construction*)

La paquetage `Ada.Strings.Unbounded` fournit le type `Unbounded_String` des chaînes de caractères non bornées.

Ce type est privé ; il est réalisé essentiellement par pointeur sur type `String`.

Les chaînes sont ainsi allouées dans le pool, ce qui permet de redimensionner une chaîne (chose impossible avec le type `String`).

En réalité, il s'agit d'un **type contrôlé**, qui permet une gestion automatique du pool.

Si vous utilisez GNAT, vous trouverez le spécification du paquetage `Ada.Strings.Unbounded` dans le fichier a-strunb.ads.

Chapitre 2

Les fichiers

2.1 Les fichiers séquentiels

2.1.1 Les fichiers textes

Le paquetage `Ada.Text_IO` contient la définition d'un type limité privé `File_Type` représentant les fichiers textes.

Le type énuméré `File_Mode` permet de traîter les trois types de fichiers suivants :

- `In_File` pour un fichier en lecture ;
- `Out_File` pour un fichier en écriture ; et
- `Append_File` pour un fichier à la fin duquel on veut rajouter du texte.

Le programme cp.adb[2] illustre l'utilisation du type `File_Type` ; Il réalise la copie d'un fichier texte.

⋆

Attention ! Le type `File_Type` du paquetage `Text_IO` ne permet pas de lire les caractères de contrôle (comme le retour chariot) qui sont interprètés par le système d'exploitation.

Pour s'en convaincre il suffit de faire la copie d'un éxécutable à l'aide du programme précédent ; cette copie n'est en générale pas correcte.

2.1.2 Les autres fichiers

Le paquetage générique `Ada.Sequential_IO` contient la définition d'un type limité privé `File_Type` représentant les fichiers séquentiels d'objets de

type quelconque.

Soit `T` un type queconque. Il suffit d'instancier le paquetage ci-dessus pour manipuler des fichiers d'objets de type `T`.

Voyons deux exemples :

1. une instanciation avec le type prédéfini `Integer` ; et
2. une instanciation avec un type composé défini par le programmeur.

Types prédéfinis

Le programme create.adb[3] créé un fichier d'entiers. Ce fichier s'appelle `create.out` ; il est illisible car c'est la représentation binaire des entiers qui s'y trouve.

Le programme read.adb[4] lit un fichier d'entiers ; il permet donc de relire le fichier `create.out`.

⋆

Remarque. Nous avons vu que le type `File_Type` du paquetage `Text_IO` ne permet pas de lire les caractères de contrôle.

Pour obtenir un programme qui traîte tous les caractères, il faut utiliser le paquetage `Ada.Sequential_IO` et l'instancier avec le type `Character` (voire la nouvelle version de notre programme cp.adb[5] de copie).

Autre types

Soit le type `Person` définit dans le paquetage `Persons`[6] .

Le programme create.adb[7] créé un fichier `create.out` d'objets de ce type.

Le programme read.adb[8] permet de relire ce fichier.

2.2 Les fichiers directs

Le paquetage générique `Ada.Direct_IO` contient la définition d'un type limité privé `File_Type` représentant les fichiers à accès direct d'objets de type quelconque.

Soit `T` un type queconque. Il suffit d'instancier le paquetage ci-dessus pour manipuler des fichiers directs d'objets de type `T`.

Le type énuméré `File_Mode` permet de traîter les trois types de fichiers suivants :

- `In_File` pour un fichier en lecture ;
- `Out_File` pour un fichier en écriture ; et
- `Inout_File` pour un fichier en lecture/écriture.

Le programme create.adb[9] montre la création d'un fichier direct d'entiers.

⋆

Le programme read1.adb[10] montre que la lecture peut se faire de façon séquentielle, et les programmes read2.adb[11] et read3.adb[12] illustrent des moyens de lecture plus spécifiques.

Enfin le programme modify.adb[13] montre l'utilisation du mode `Inout_File`.

Chapitre 3

Récupération de la ligne de commande

Voire la procédure `testCL`[14] , issue de l'ensemble de sources Ada fournis dans le répertoire `examples` de la distribution de GNAT.

Notes

[1] fichier : Chaines/Ada/strings.adb

```
with Text_IO;
procedure Strings is

   Name : String (1 .. 20); -- le nom du fichier
   Line : String (1 .. 80); -- pour lire les lignes du fichiers
   Last : Natural;

   My_File : Text_IO.File_Type;

begin
   Text_IO.Put ("Nom du fichier : ");
   Text_IO.Get_Line (Name, Last);
   --  Pour ouvrir le fichier, on utilise la tranche Name (1 .. Last)
   --  qui représente la chaîne da caractères effectivement
   --  tapée par l'utilisateur :
   Text_IO.Open (My_File, Text_IO.In_File, Name (1 .. Last));

   while not Text_IO.End_Of_File (My_File) loop
      Text_IO.Get_Line (My_File, Line, Last);
      --  Pour imprimer la ligne lue il faut à nouveau utiliser une tranche :
      Text_IO.Put_Line (Line (1 .. Last));
   end loop;

   Text_IO.Close (My_File);

end Strings;
```

[2] fichier : Fichiers/Ada/Text/cp.adb

```
--  Copie d'un fichier texte dans un autre
--  Illustre le type File_Type du paquetage Text_IO
--  et le paquetage Ada.Command_Line qui permet
--  de récupérer les arguments sur la ligne de commande.
--  Cet exemple est très succint !
with Ada.Command_Line;
with Text_IO;
procedure Cp is
```

```
   --  La source et la destination de la copie :
   Source, Target : Text_IO.File_Type;

   Line : String (1 .. 80); -- pour lire les lignes du fichiers
   Last : Natural;

begin

   if Ada.Command_Line.Argument_Count /= 2 then
      Text_IO.Put_Line ("cp source destination");
   else
      --  Assignation de fichiers physiques aux fichiers logiques
      --  Les noms sont récupérés sur la ligne de commande
      Text_IO.Open (Source, Text_IO.In_File, Ada.Command_Line.Argument (1));
      Text_IO.Create (Target, Text_IO.Out_File, Ada.Command_Line.Argument (2));
      --  Boucle de lecture/écriture
      while not Text_IO.End_Of_File (Source) loop
         Text_IO.Get_Line (Source, Line, Last);
         Text_IO.Put_Line (Target, (Line (1 .. Last)));
      end loop;
      --  Fermeture des fichiers
      Text_IO.Close (Source);
      Text_IO.Close (Target);
   end if;

end Cp;
```

[3] fichier : Fichiers/Ada/Sequential/create.adb

```
--  Création d'un fichier d'entiers
--  Illustre le paquetage Sequential_IO (pour entrées/sorties séquentielles)
with Sequential_IO; -- paquetage générique
procedure Create is

   --  Pour l'exemple, on veut stocker les onzes premières factorielles
   --  dans un fichier. On aura donc besoin de la fonction :
   function Fact (N : Natural) return Natural;

   --  Il faut instancier Sequential_IO avec le type Integer
   package Int_Io is new Sequential_IO (Integer);

   Target : Int_Io.File_Type; -- le fichier destination

   --  Voici le corps de Fact :
```

```
   function Fact (N : Natural) return Natural is
   begin
      if N = 0 then return 1;
      else return N * Fact (N - 1);
      end if;
   end Fact;

begin

   --  Assignation du fichier physique "create.out" au fichier logique Target
   Int_Io.Create (Target, Int_Io.Out_File, "create.out");
   --  Boucle d'écriture
   for I in Integer range 0 .. 10 loop
      Int_Io.Write (Target, Fact (I));
   end loop;
   --  Fermeture du fichier
   Int_Io.Close (Target);

end Create;
```

[4] fichier : Fichiers/Ada/Sequential/read.adb

```
--  Lecture d'un fichier d'entiers
--  Illustre le paquetage Sequential_IO (pour entrées/sorties séquentielles)
with Sequential_IO; -- paquetage générique
with Io; -- pour imprimer les entiers à l'écran
procedure Read is

   --  Il faut instancier Sequential_IO avec le type Integer
   package Int_Io is new Sequential_IO (Integer);

   Source : Int_Io.File_Type; -- le fichier source

   I : Integer;

begin

   --  Assignation du fichier physique "create.out" au fichier logique Source
   Int_Io.Open (Source, Int_Io.In_File, "create.out");
   --  Boucle de lecture
   while not Int_Io.End_Of_File (Source) loop
      Int_Io.Read (Source, I);
      Io.Put (I);
   end loop;
```

```
   --  Fermeture du fichier
   Int_Io.Close (Source);

end Read;
```

[5] fichier : Fichiers/Ada/Sequential/cp.adb

```
--  Copie d'un fichier dans un autre
--  Illustre le paquetage Sequential_IO (pour entrées/sorties séquentielles)
--  et le paquetage Ada.Command_Line (pour récupérer les arguments sur la ligne
--  de commande).
--  Cet exemple est très succint !
with Ada.Command_Line;
with Sequential_IO; -- paquetage générique pour les fichiers séquentiels
with Text_IO;
procedure Cp is

   package Char_Io is new Sequential_IO (Character);

   --  La source et la destination de la copie :
   Source, Target : Char_Io.File_Type;

   In_Char : Character;

begin

   if Ada.Command_Line.Argument_Count /= 2 then
      Text_IO.Put_Line ("cp source destination");
   else
      --  Assignation de fichiers physiques aux fichiers logiques
      --  Les noms sont récupérés sur la ligne de commande
      Char_Io.Open (Source, Char_Io.In_File, Ada.Command_Line.Argument (1));
      Char_Io.Create (Target, Char_Io.Out_File, Ada.Command_Line.Argument (2));
      --  Boucle de lecture/écriture
      while not Char_Io.End_Of_File (Source) loop
         Char_Io.Read (Source, In_Char);
         Char_Io.Write (Target, In_Char);
      end loop;
      --  Fermeture des fichiers
      Char_Io.Close (Source);
      Char_Io.Close (Target);
   end if;

end Cp;
```

[6] fichier : Fichiers/Ada/Sequential/Persons/persons.ads

```
package Persons is

   type Person is record
      Name : String (1 .. 20);
      Age : Natural;
   end record;

end Persons;
```

[7] fichier : Fichiers/Ada/Sequential/Persons/create.adb

```
--  Création d'un fichier de personnes
--  Illustre le paquetage Sequential_IO (pour entrées/sorties séquentielles)
with Persons;
with Sequential_IO; -- paquetage générique
procedure Create is

   --  Il faut instancier Sequential_IO avec le type Person
   package Person_Io is new Sequential_IO (Persons.Person);

   Target : Person_Io.File_Type; -- le fichier destination

   use Persons;
   P : Person;

begin

   --  Assignation du fichier physique "create.out" au fichier logique Target
   Person_Io.Create (Target, Person_Io.Out_File, "create.out");
   --  On écrit trois personnes
   P.Name := "toto                ";
   P.Age := 20;
   Person_Io.Write (Target, P);
   P.Name := "titi                ";
   P.Age := 40;
   Person_Io.Write (Target, P);
   P.Name := "tutu                ";
   P.Age := 60;
   Person_Io.Write (Target, P);

   --  Fermeture du fichier
```

```
   Person_Io.Close (Target);

end Create;
```

[8] fichier : Fichiers/Ada/Sequential/Persons/read.adb

```
--  Lecture d'un fichier de personnes
--  Illustre le paquetage Sequential_IO (pour entrées/sorties séquentielles)
with Persons;
with Sequential_IO; -- paquetage générique
with Io; -- pour imprimer les entiers à l'écran
procedure Read is

   --  Il faut instancier Sequential_IO avec le type Person
   package Person_Io is new Sequential_IO (Persons.Person);

   Source : Person_Io.File_Type; -- le fichier source

   use Persons;
   P : Person;

begin

   --  Assignation du fichier physique "create.out" au fichier logique Source
   Person_Io.Open (Source, Person_Io.In_File, "create.out");
   --  Boucle de lecture
   while not Person_Io.End_Of_File (Source) loop
      Person_Io.Read (Source, P);
      Io.Put (P.Name);
      Io.Put (P.Age);
      Io.New_Line;
   end loop;
   --  Fermeture du fichier
   Person_Io.Close (Source);

end Read;
```

[9] fichier : Fichiers/Ada/Direct/create.adb

```
--  Création d'un fichier direct d'entiers
--  Illustre le paquetage Direct_IO
--  (manipulant les fichiers à accès direct)
with Direct_IO; -- paquetage générique
```

```
procedure Create is

   --  Pour l'exemple, on veut stocker les onzes premières factorielles
   --  dans un fichier. On aura donc besoin de la fonction :
   function Fact (N : Natural) return Natural;

   --  Il faut instancier Direct_IO avec le type Integer
   package Int_Io is new Direct_IO (Integer);

   Target : Int_Io.File_Type; -- le fichier destination

   --  Voici le corps de Fact :
   function Fact (N : Natural) return Natural is
   begin
      if N = 0 then return 1;
      else return N * Fact (N - 1);
      end if;
   end Fact;

begin

   --  Assignation du fichier physique "create.out" au fichier logique Target
   Int_Io.Create (Target, Int_Io.Out_File, "create.out");
   --  Boucle d'écriture
   for I in Integer range 0 .. 10 loop
      Int_Io.Write (Target, Fact (I));
   end loop;
   --  Fermeture du fichier
   Int_Io.Close (Target);

end Create;
```

[10] fichier : Fichiers/Ada/Direct/read1.adb

```
--  Lecture d'un fichier direct d'entiers
--  Illustre le paquetage Direct_IO
--  (manipulant les fichiers à accès direct)

--  Montre une lecture séquentielle équivalente
--  à celle sur fichier séquentiel
--  Voire les autre procédures Read2, Read3 pour
--  des exemples de manipulations spécifiques.

with Direct_IO; -- paquetage générique
```

```
with Io; -- pour imprimer les entiers à l'écran
procedure Read1 is

   --  Il faut instancier Direct_IO avec le type Integer
   package Int_Io is new Direct_IO (Integer);

   Source : Int_Io.File_Type; -- le fichier source

   I : Integer;

begin

   --  Assignation du fichier physique "create.out" au fichier logique Source
   Int_Io.Open (Source, Int_Io.In_File, "create.out");
   --  Boucle de lecture
   while not Int_Io.End_Of_File (Source) loop
      Int_Io.Read (Source, I);
      Io.Put (I);
   end loop;
   --  Fermeture du fichier
   Int_Io.Close (Source);

end Read1;
```

[11] fichier : Fichiers/Ada/Direct/read2.adb

```
--  Lecture d'un fichier direct d'entiers
--  Illustre le paquetage Direct_IO
--  (manipulant les fichiers à accès direct)

--  Montre une lecture séquentielle à partir d'une certaine position

with Direct_IO; -- paquetage générique
with Io; -- pour imprimer les entiers à l'écran
procedure Read2 is

   --  Il faut instancier Direct_IO avec le type Integer
   package Int_Io is new Direct_IO (Integer);

   Source : Int_Io.File_Type; -- le fichier source

   I : Integer;

begin
```

```
   --  Assignation du fichier physique "create.out" au fichier logique Source
   Int_Io.Open (Source, Int_Io.In_File, "create.out");
   --  Positionnement de l'index du fichier
   Int_Io.Set_Index (Source, 5);
   --  Boucle de lecture
   while not Int_Io.End_Of_File (Source) loop
      Int_Io.Read (Source, I);
      Io.Put (I);
   end loop;
   --  Fermeture du fichier
   Int_Io.Close (Source);

end Read2;
```

[12] fichier : Fichiers/Ada/Direct/read3.adb

```
--  Lecture d'un fichier direct d'entiers
--  Illustre le paquetage Direct_IO
--  (manipulant les fichiers à accès direct)

--  Montre une lecture aléatoire (ici à l'envers)
--  Illustre aussi l'utilisation de la fonction Size
--  qui retourne une valeur d'un type entier Count
--  donnant la taille du fichier (en nombre d'enregistrements).

with Direct_IO; -- paquetage générique
with Io; -- pour imprimer les entiers à l'écran
procedure Read3 is

   --  Il faut instancier Direct_IO avec le type Integer
   package Int_Io is new Direct_IO (Integer);

   Source : Int_Io.File_Type; -- le fichier source

   I : Integer;

   Source_Size : Int_Io.Count;

begin

   --  Assignation du fichier physique "create.out" au fichier logique Source
   Int_Io.Open (Source, Int_Io.In_File, "create.out");
   --  Positionnement de l'index du fichier (voire la fin)
```

```
   Int_Io.Set_Index (Source, 5);
   --  Récupération de la taille du fichier
   Source_Size := Int_Io.Size (Source);
   --  Boucle de lecture
   for Local_Index in reverse Int_Io.Count range 1 .. Source_Size loop
      Int_Io.Read (File => Source, Item => I, From => Local_Index);
      Io.Put (I);
   end loop;
   Io.New_Line;

   --  Attention! L'index courant a été modifié par la boucle
   Int_Io.Read (File => Source, Item => I);
   Io.Put (I);

   --  Fermeture du fichier
   Int_Io.Close (Source);

end Read3;
```

[13] fichier : Fichiers/Ada/Direct/modify.adb

```
--  Lecture d'un fichier direct d'entiers
--  Illustre le paquetage Direct_IO
--  (manipulant les fichiers à accès direct)

--  Illustre la modification d'un enregistrement d'un fichier direct
--  grace au mode Inout_File

with Direct_IO; -- paquetage générique
procedure Modify is

   --  Il faut instancier Direct_IO avec le type Integer
   package Int_Io is new Direct_IO (Integer);

   Source : Int_Io.File_Type; -- le fichier source

begin

   --  Assignation du fichier physique "create.out" au fichier logique Source
   Int_Io.Open (Source, Int_Io.Inout_File, "create.out");
   --  Boucle de lecture
   Int_Io.Write (File => Source, Item => -1, To => 5);
   --  Fermeture du fichier
   Int_Io.Close (Source);
```

```
end Modify;
```

[14] fichier : Commande/testcl.adb

```
with Ada.Command_Line;
with Gnat.Io; use Gnat.Io;

procedure TestCL is
begin
   --  Writes out the command name (argv[0])
   Put ("      The command name : ");
   Put (Ada.Command_Line.Command_Name);
   New_Line;

   --  Writes out the number of arguments passed to the program (argc)
   Put ("The number of arguments: ");
   Put (Ada.Command_Line.Argument_Count);
   New_Line;

   --  Writes out all the arguments using the Argument function.
   --  (BE CAREFUL because if the number you pass to Argument is not
   --   in the range 1 .. Argument_Count you will get Constraint_Error!)
   for I in 1 .. Ada.Command_Line.Argument_Count loop
      Put ("             Argument ");
      Put (I);
      Put (": ");
      Put (Ada.Command_Line.Argument (I));
      New_Line;
   end loop;
end TestCL;
```