Cours Ada 95
pour le programmeur C++

V 1.0a (16/12/2004)

Auteur : Quentin Ochem (quentin.ochem@theatreux.org)

Note sur le présent document

Ce document a été rédigé pour permettre aux programmeurs connaissant le C++ d'apprendre rapidement les bases du langage Ada. De solides connaissances en C++ sont donc requises pour sa compréhension. Seul un tour d'horizon du langage Ada est proposé ici. Pour plus de détails, vous pouvez vous référer aux liens donnés dans le chapitre titré « références ».

Toutes les remarques ou suggestions sont largement appréciées. Vous pouvez les envoyer à cette adresse : quentin.ochem@theatreux.org.

1. PREFACE AU LECTEUR SCEPTIQUE 5

2. GENERALITES 6

3. STRUCTURE DES FICHIERS DE CODE 7

4. SYNTAXE GENERALE 9

4.1. DECLARATIONS 9
4.2. CONDITIONS 9
4.3. BOUCLES 10

5. TYPES 12

5.1. TYPAGE FORT 12
5.2. CONSTRUCTION DE NOUVEAUX TYPES 12
5.3. ATTRIBUTS 14
5.4. TABLEAUX ET CHAINES DE CARACTERES 15
5.5. LES POINTEURS 16

6. PROCEDURES ET FONCTIONS 18

6.1. FORME GENERALE 18
6.2. SURNOMMAGE (OU SURDEFINITION) 19

7. PAQUETAGES 20

7.1. PROTECTION DES DECLARATIONS 20
7.2. PAQUETAGES HIERARCHIQUES 20

8. CLASSES 22

8.1. LE TYPE RECORD 22
8.2. DERIVATION ET LIAISON DYNAMIQUE 23
8.3. CLASSES ABSTRAITES 24

9. GENERICITE 25

9.1. MODELE GENERAL 25
9.2. PARAMETRES DE GENERICITE 25
9.3. PAQUETAGES GENERIQUES 26

10. EXCEPTIONS 28

10.1. EXCEPTIONS STANDARD 28
10.2. EXCEPTIONS ETENDUES (ADA 95) 28

11. PROGRAMMATION CONCURRENTE ET TEMPS REEL 30

11.1. AVERTISSEMENT 30
11.2. TACHES 30
11.3. RENDEZ-VOUS 31
11.4. RENDEZ-VOUS MULTIPLE 32
11.5. OBJETS PROTÉGÉS 33

12. LES PETITS BONHEURS DE ADA 35

12.1. INTRODUCTION 35
12.2. GESTION DES DONNEES BIT A BIT 35
12.3. SERIALISATION D'ENREGISTREMENTS 35

CONCLUSION 37

13. REFERENCES 38

13.1. COURS ET RESSOURCES EN LIGNE 38
13.2. COURS EN LIBRAIRIE 38
13.3. OUTILS 38
13.4. DIVERS 38

1. Préface au lecteur sceptique

Parler de Ada et de C++, c'est un peu comme parler de Linux et de Windows, de Firefox et d'Internet Explorer, ou encore d'OpenOffice et de Word. On sombre vite dans une guerre de religion sans fin entre deux camps qui s'ignorent mutuellement et qui se plaisent à poser argument après argument. Nous éviterons donc dans ce document de tomber dans un fanatisme informatique, par ailleurs si tentant et si ludique, pour présenter Ada de la façon la plus neutre possible, en le comparant plutôt qu'en l'opposant à C++ (voir le chapitre « références »).

Mais pourquoi faire de l'Ada ? Pourquoi recommencer l'apprentissage d'un langage, par ailleurs réputé si austère, lorsque l'on maîtrise parfaitement un langage orienté objet comme C++ ? Ce C++ qui, entre autres avantages, est un standard connu et reconnu de tous ? Si vous lisez ces lignes, c'est que vous avez sans doute, au moins partiellement, répondu à la question. Passer de C++ à Ada, c'est faire un gros investissement. Bien que les paradigmes objets soient relativement proches entre ces deux langages, leur philosophie est complètement différente. En tant que débutant Ada, vous mettrez plus de temps à écrire votre code, et une fois écrit, vous mettrez encore plus de temps à lui faire passer la compilation. Vous aurez des difficultés structurelles à accéder à la mémoire bas niveau, vous ne pourrez pas faire de conversions implicites, vous serez contraints et forcés de mille manières différentes. Cerise sur le gâteau, vous ne parlerez pas le même langage que vos clients et concurrents (sauf si eux aussi travaillent en Ada), vous aurez donc des difficultés à réutiliser nombre de bibliothèques, vous aurez même des fonctions à interfacer vous-même. En Ada, pas de compilateur Borland, pas d'outil de compilation Microsoft, et de trop rares mentions sur les C.V. de vos recrues. Et pour couronner le tout, vos programmes compilés vous paraîtront énormes. Alors pourquoi Ada ?

Ada est un choix à faire sur le long terme, qui ne peut être rentable que s'il est assumé dans la durée. Il s'agit d'un langage en symbiose avec un certain nombre de concepts génie logiciel, souvent mal acceptés parce qu'ils imposent une réflexion plus poussée. Le code sera moins agréable à écrire, mais il gagnera en lisibilité. Il sera plus difficile à compiler mais une fois passé cette phase, il subira moins d'erreurs d'exécution. Il sera plus facile à maintenir, à réutiliser, et de très grosses erreurs auront été détectées tôt dans le développement là où, dans d'autres langages, elles peuvent parfois dormir des années… Pour exploser la veille de noël ou dix minutes avant une présentation. Passer à Ada ne sera pas forcément une partie de plaisir, mais ceux qui en ont fait l'effort affirment que leur efficacité a été décuplée. La question est : en avez-vous besoin ?

L'objectif de ce document n'est pas de vous apprendre à programmer en Ada. Pour étudier vraiment le langage, il faudrait au moins dix fois plus de pages. Mais il a pour but de vous en présenter les principales caractéristiques, vous permettant ainsi de réaliser facilement vos premiers programmes.

2. Généralités

Ada 95 est un langage qui implémente la quasi-totalité des notions que vous avez l'habitude d'utiliser en C++ : classes, héritage, polymorphisme, généricité. Les aficionados de Ada ont coutume de prétendre qu'il les implémente mieux, nous nous contenterons de dire qu'il les implémente différemment.

Tout le langage C++ est conçu pour permettre au programmeur de coder le plus facilement et de la façon la plus compacte possible, quitte à rendre le programme incompréhensible ou très difficilement débuggable en cas de problème. Cette tendance est actuellement contrebalancée par l'ajout de plus en plus systématique de warnings et de normes de codage. Mais personne n'est à l'abri d'une erreur d'inattention non contrôlée, et non vérifiable (cf critique du C++, en référence).

Le langage Ada part du principe exactement inverse. Le compilateur effectue de très nombreux tests qui bloquent le programme. Il rajoute même des tests à l'exécution (test de dépassements de tableaux par exemple). Le langage est très verbeux, on peut compter deux ou trois lignes Ada pour une ligne C++ en moyenne, et les mots clés sont explicites (begin et end au lieu des accolades ouvrantes et fermantes par exemple).

Contrairement à C++, Ada ne reconnaît pas la casse. Les variables VAR, var et VaR sont donc identiques. Ada accepte même les accents dans les identificateurs, bien que cette possibilité ne fasse pas l'unanimité. Certains craignent, entre autres, des incompatibilités entre des codes écrits en Ascii simple et en UTF-8.

Ada est voulu portable. En conséquence, les compilateurs subissent des tests poussés avant d'être autorisés à porter le label du langage, et n'ont qu'une marge très limitée pour étendre le langage, au travers de « directives de compilation », aussi appelées pragmas. En théorie, donc, lorsque vous écrivez un code pour un compilateur et que vous le recompilez sur un autre, il fonctionnera de la même façon. Si vous écrivez un programme pour un système, il fonctionnera pour un autre, sauf si vous faites des appels à des procédures exotiques, aux frontières du langage.

Ada est aussi rapide à l'exécution que C++, les programmes sont plus gros parce qu'ils rajoutent des tests, mais ces tests sont placés suffisamment intelligemment pour être effectués le plus rarement possible.

3. Structure des fichiers de code

En C++, on a tendance à utiliser deux types de fichiers : les fichiers d'entête (.h) qui servent plus ou moins à définir les spécifications et les fichiers de corps (.cxx ou .cpp dans la plupart des cas) qui définissent les implémentations. Ces règles ne sont absolument pas imposées, et il n'est pas rare de voir une fonction « inline » qui traîne dans un .h, ou dans un fichier annexe inclus dans ce .h.

Avec Ada, la structure du code est toujours séparée en deux unités : une spécification et un corps (la spécification pouvant être omise dans de très rares cas). Nous allons présenter l'architecture choisie par le compilateur GNAT, qui est très certainement le premier compilateur que vous essaierez, pour la simple raison qu'il est gratuit (voir référence en annexes). Par la suite, nous utiliserons sa convention de nommage. Sachez qu'au-delà de l'architecture des fichiers, tout le code est strictement identique quel que soit le compilateur.

Sous GNAT, on sépare ces deux parties dans deux fichiers distincts, un .ads pour la spécification et un .adb pour le corps. Il y a deux types d'unités de compilation : les paquetages et les sous-programmes. Un paquetage est une sorte de bibliothèque, qui contient plusieurs procédures (mais pas de « main »), alors qu'une procédure est une unité qui peut servir à créer un programme exécutable.

Voilà la syntaxe de la spécification d'un paquetage (.ads) :

```
package Nom_Paquetage is
-- code
private
-- code
end Nom_Paquetage;
```

Et voilà la syntaxe d'implémentation d'un paquetage (.adb) :

```
package body Nom_Paquetage is
-- code
end Nom_Paquetage;
```

Vous remarquerez également qu'un mot clé **private** sépare en deux parties la spécification d'un paquetage. Il s'agit de la séparation entre les objets publics, c'est-à-dire accessibles par les fichiers utilisant ce paquetage, et les objets privés accessibles uniquement par le paquetage lui-même (et quelques ayant droits, voir le chapitre sur les paquetages hiérarchiques). Cette façon de protéger des données est assez différente de C++. Alors que le C++ possède une partie publique et une partie privée pour chaque classe, Ada différencie ces parties au niveau du paquetage. Notez que la notion de donnée protégée est inexistante en Java, même s'il est possible d'avoir un principe quelque peu similaire.

Notez également que le nom du fichier doit impérativement être le même que celui de l'unité de compilation. On aurait donc ici respectivement Nom_Paquetage.ads et Nom_Paquetage.adb.

Pour utiliser les entités déclarées dans un paquetage, il faut l'invoquer comme il suit :

```
with Nom_Paquetage;
```

```
#include Nom_Paquetage
```

D'un certain point de vue, on peut également considérer le paquetage comme un espace de nom (namespace) C++, c'est-à-dire que les unités décrites à l'intérieur sont à préfixer du nom de ce paquetage. En C++ ce préfixe est lié au nom de l'entité par un opérateur de résolution de portée (::). En Ada, un simple point suffit. Il est cependant possible de s'abstraire de cette notation préfixée. En C++, on utiliserait « using namespace ». En Ada, on utilise :

```ada
use Nom_Paquetage;
```

```cpp
using namespace Nom_Paquetage;
```

Parlons enfin des unités de compilation procédures. Ce sont elles qui peuvent servir de point d'entrée à un programme (fonction main), même si on peut les invoquer à partir d'un autre programme via une directive with. Comme pour les paquetages, le nom du ficher qui contient la procédure principale doit être le même que celui de la procédure, suivi d'un .adb. Lorsque l'unité de compilation est une procédure, il n'y a pas besoin de fichier de spécifications (même si on peut en fournir un).

Voici un exemple de procédure qui affiche « hello world » à l'écran. Cette procédure utilise un paquetage prédéfini, le paquetage Ada.Text_IO, qui réalise les opérations d'entrée / sortie sur texte :

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Principale is
begin
    Put_Line ("Hello World");
end Principale;
```

```cpp
#include <iostream>

void main (void)
{
    cout << "Hello World" << endl;
}
```

Nous verrons dans un prochain chapitre les détails liés à l'écriture de procédures et de fonctions.

4. Syntaxe générale

4.1. Déclarations

Comme en C++, l'indication de fin d'instruction est le point virgule. Un bloc ne se déclare pas avec deux accolades, mais entre deux mots clés **begin** et **end**. Les variables ne peuvent être déclarées que dans une partie déclarative, avant un code (comme en C) c'est-à-dire avant un **begin**, mais il est possible de créer un bloc à l'intérieur d'un autre bloc. L'affectation est réalisée via le symbole :=, qui, contrairement au C++, n'est pas un opérateur (on ne pourra donc ni le redéfinir, ni le mettre dans une condition, ni écrire quelque chose du genre A := B := C). Il n'y a pas d'incrémentation raccourcie (++). Les principaux types primitifs sont Integer, Boolean Float et Character pour int, bool, float et char.

Prenons par exemple ce code qui réalise la déclaration d'un entier puis son incrémentation :

```
procedure Principale is
   Variable : Integer := 0;
begin
   Variable := Variable + 1;
end Principale;
```

```
int main (void)
{
    int Variable = 0;
    Variable++;
}
```

Vous remarquerez que pour déclarer une variable, on utilise son nom suivi de deux points et de son type, ce qui est l'inverse du C++. Il est possible de déclarer plusieurs variables du même type en séparant leurs noms par des virgules. Cependant, s'il y a une valeur initiale, elle sera la même pour toutes les variables de la déclaration en cours (alors qu'en C++ il faut définir une valeur pour chaque variable). Notez que si cette valeur initiale est calculée par un appel de fonction, alors la fonction sera appelée une fois pour chaque variable (ce qui peut être gênant dans le cas de fonctions à effet de bord).

Les commentaires sont introduits par la chaîne --, et définissent le texte comme commenté jusqu'à la fin de la ligne. Il n'y a pas d'équivalent pour les commentaires de bloc /* */ du C++.

Il est possible de créer un nouveau bloc à l'intérieur d'un bloc déjà existant en utilisant le mot clef **declare** :

```
declare
   Variable : Integer := 0;
begin
   Variable := Variable + 1;
end;
```

Attention également, en Ada, un bloc sans instruction ne peut pas exister. Si vous avez besoin de créer un bloc sans rien écrire à l'intérieur, parce que cela sera fait à l'avenir par exemple, il faut quand même au minimum y écrire l'instruction **null**, qui ne fait rien.

4.2. Conditions

Une condition en Ada s'énonce **if** condition **then** instructions **elsif** condition **then** instructions **else** instructions **end if**. Il n'y a pas besoin d'entourer la condition de parenthèses comme en C++, puisque le mot clef **then** (obligatoire !) marque la fin de la condition. Les opérateurs de comparaison sont les mêmes qu'en C++, à l'exception de l'égalité (== devient =) et de la différence (!= devient /=). Les opérateurs booléens sont, eux, à écrire en toutes lettres (!, &&, || deviennent **not**, **and**, **or**). Les opérateurs **and** et **or** ont la particularité de tester systématiquement les deux opérandes, alors qu'en C++, le and ne teste

l'opérande de gauche que si celle de droite est fausse, et le or que si elle est vraie. Il est possible d'obtenir le même résultat en Ada en utilisant les opérateurs **and then** et **or else**.

Voilà un petit exemple de code qui teste si une valeur est plus grande que zéro, et qui affiche un texte en conséquence :

```
if Variable > 0 then
    Put_Line (" > 0 ");
else
    Put_Line (" <= 0 ");
end if;
```

```
if (Variable > 0)
{
    cout << " > 0 " << endl;
}
else
{
    cout << " <= 0 " << endl;
}
```

Il existe un autre schéma de condition, les conditions par cas. En C++, elles s'écrivent avec un switch case. En Ada, on utilisera **case when**. Comme en C++, on ne peut en Ada faire de case que sur des entiers ou des types énumératifs. Ada impose que tous les cas soient envisagés. Si on ne sait pas réagir à certaines valeurs, alors il faut mettre un choix par défaut : ce qui s'écrivait default en C++ s'écrit **when others** en Ada. Autre subtilité : lorsqu'un choix est validé, on ne cherche pas à exécuter les choix suivants, inutile donc de mettre d'instruction break. Dernière chose : Ada permet d'utiliser des plages de valeurs, ce qui permet de mettre un grand nombre de valeurs derrière un cas, et résout la plupart des problèmes qui avaient poussé les concepteurs du C à amener un cas à exécuter le cas suivant. Voici un exemple de code :

```
case Variable is
when 0 =>
   Put_Line ("Zéro");
when 1 .. 9 =>
   Put_Line ("Chiffre positif");
when 10 | 12 | 14 | 16 | 18 =>
   Put_Line ("Nombre pair entre 10 et 18");
when others =>
   Put_Line ("Autre chose");
end case;
```

```
switch (Variable)
{
  case 0:
    cout << "Zéro" << endl;
    break;
  case 1: case 2: case 3: case 4: case 5:
  case 6: case 7: case 8: case 9:
    cout << "Chiffre positif" << endl;
    break;
  case 10: case 12: case 14: case 16: case 18:
    cout << "Nombre pair entre 10 et 18"
         << endl;
    break;
  default:
    cout << "Autre chose" << endl;
}
```

4.3. Boucles

En Ada, une boucle commence nécessairement par le mot clef **loop** et se finit par le mot clef **end loop**. L'équivalent du break du C++ est l'instruction **exit**. Cette instruction peut également faire l'objet d'une condition, avec la syntaxe **exit when** condition. Le **loop** qui ouvre la boucle est éventuellement précédé d'une instruction **for** ou **while**.

L'instruction **while** est la plus simple. Comme en C++, elle est suivie d'une condition, à ceci près que la condition n'a pas besoin d'être parenthèsée. Par exemple :

```
while Variable < 10_000 loop
    Variable := Variable * 2;
end loop;
```

```
while (Variable < 10000)
{
    Variable = Variable * 2;
}
```

L'instruction **for** est, par contre, assez différente de son homologue C++. Il s'agit forcément d'un parcours incrémental ou décrémental entre deux bornes discrètes (entières ou énumératives). La variable

de contrôle est implicitement déclarée, et il est impossible de la modifier à l'intérieur de la boucle. La syntaxe générale est **for** variable **in** borne_inferieure .. borne_superieure **loop**. Les deux bornes doivent être mises dans l'ordre croissant, même si le parcours souhaité va du maximum au minimum (au contraire du C++). Si ce n'est pas le cas, l'intervalle sera considéré comme vide et le corps de la boucle ne sera pas exécuté. Pour faire un parcours à l'envers, il faut ajouter le mot clef **reverse** après le mot clef **in**. Voilà un exemple de boucle qui parcourt les chiffres dans l'ordre croissant puis dans l'ordre décroissant :

```
for Variable in 0 .. 9 loop
    Put_Line (Integer'Image (Variable));
end loop;

for Variable in reverse 0 .. 9 loop
    Put_Line (Integer'Image (Variable));
end loop;
```

```
for (Variable = 0; Variable <= 9; Variable++)
{
    cout << Variable << endl;
}

for (Variable = 9; Variable >= 0; Variable--)
{
    cout << Variable << endl;
}
```

Remarquez l'étrange façon que Ada a d'afficher une variable numérique. Cela est lié au fait que l'instruction Put_Line ne sait afficher que des chaînes de caractères, et qu'il faut donc extraire une image textuelle de la valeur numérique avant de l'envoyer dans la procédure. Nous étudierons cette étrange syntaxe dans le chapitre sur les attributs.

5. Types

5.1. Typage fort

L'une des plus notables caractéristiques de Ada, et l'une des moins appréciés par les nouveaux programmeurs, c'est la quasi absence de conversions implicites, et la nécessité de convertir explicitement. On ne peut, par exemple, pas ajouter directement un entier à un flottant. Cela peut paraître terriblement fastidieux au premier abord, mais expliciter ce que l'on veut faire, c'est garantir que c'est bien ce que l'on veut qui sera fait, et pas ce que le compilateur aura interprété. Examinons par exemple ce code qui réalise une opération très simple :

```
procedure Bug is
    Variable_1 : Float := 10.0;
    Variable_2 : Integer := 1009;
    Resultat   : Float;
begin
    Resultat := Float (Variable_2) /
       Variable_1;
end Bug;
```

```
void Bug (void)
{
    float Variable_1 = 10;
    int Variable_2 = 1009;
    float Resultat;

    Resultat = Variable_2 / Variable_1;
}
```

On aimerait avoir à la sortie de ce programme la valeur 100,9 dans la variable Résultat. Le code de gauche ne passe à la compilation que si l'on convertit explicitement Variable_2 en Float. Le code de droite passe sans problème. A l'exécution cependant, le code de gauche calcule bien 100,9 alors que le code de droite trouve un résultat égal à 100. Motif ? Priorité opératoire. Le compilateur C++ considère d'abord la division entre un entier et un flottant, et en déduit un entier, puis il affecte cet entier dans une variable flottante en effectuant une nouvelle conversion.

De nos jours, de telles erreurs sont souvent détectées à la compilation par des messages d'alerte (quoique gcc ne râle pas dans ce cas même avec le flag –Wall). Il n'en reste pas moins que certaines erreurs peuvent se glisser à cause de ce genre de subtilités, et que la proposition de Ada permet dans une certaine mesure de diminuer les bugs qui leur sont liés.

Remarquez aussi que pour donner une valeur initiale à la première variable, nous sommes obligés d'écrire un littéral 10.0, alors que 10 suffit dans le cas C++. Il s'agit encore d'une restriction de typage, le littéral 10.0 est de type flottant alors que 10 est de type entier.

Le typage est aussi contrôlé dynamiquement, c'est-à-dire que les valeurs données aux variables doivent être dans leur ensemble de définition. Il existe par exemple un type Positive, dont les valeurs sont comprises entre 1 et un grand entier. Affecter la valeur 0 dans une variable de ce type provoquera une erreur à l'exécution (et, s'il s'agit d'un littéral, un warning sera même visible à la compilation).

Ce type de test est particulièrement utile pour détecter les erreurs de dépassement au plus tôt. On pourrait s'inquiéter de la lenteur du code généré si, à chaque affectation, à chaque opération, il fallait tester les bornes des valeurs. Heureusement, les compilateurs sont aujourd'hui capables de limiter ces tests au strict nécessaire et on peut admettre que la vitesse de l'exécutable n'est pas atteinte de façon notable. Quoi qu'il en soit, certains compilateurs permettent de retirer les tests en phase de production, ce qui peut avoir un sens dans le cas de code très bien testés et critiques.

5.2. Construction de nouveaux types

Lorsque l'on prend l'habitude d'utiliser le typage fort, on se rend compte qu'il est souvent très intéressant de poser soi même des contraintes de conversions afin d'éviter des erreurs d'inattention. On peut par exemple vouloir interdire des affectations entre des variables contenant des minutes et des variables contenant des heures. Ada permet même de définir des ensembles de définition, c'est-à-dire des bornes pour les types. Ainsi, on peut écrire :

```
procedure Test is
    type Heure is range 0 .. 23;
    type Minute is range 0 .. 59;

    H : Heure  := 12;
    M : Minute := 56;
begin
   M := H;
   H := 24;
end Test;
```

La première ligne de code qui suit le **begin** ne passera pas à la compilation. Heure et Minute étant des types distincts, l'affectation de l'un dans l'autre est impossible sans conversion implicite. La ligne suivante sera acceptée, se contentant de générer un message d'alerte. Mais à l'exécution du code, une exception sera levée : il est en effet impossible d'affecter la valeur 24 à un type dont l'intervalle de définition va de 0 à 23. Notons aussi qu'il existe en Ada des entiers modulo et leur arithmétique.

Comme en C++, Ada propose une syntaxe pour la déclaration d'énumération. En voici un exemple :

```
type Jour is
  (Lundi,
   Mardi,
   Mercredi,
   Jeudi,
   Vendredi,
   Samedi,
   Dimanche);
```

```
enum Jour
{
    Lundi,
    Mardi,
    Mercredi,
    Jeudi,
    Vendredi,
    Dimanche
}
```

Il existe une syntaxe un peu complexe pour associer des valeurs aux entrées d'une énumération, alors qu'en C++, un signe d'égalité suffit. Ada préfère mettre l'accent sur le caractère abstrait de ces valeurs, qui ne sont censées avoir aucun lien avec les entiers. Il existe cependant une relation d'ordre entre elles, ainsi que la notion de premier et de dernier, nous verrons par la suite comment les utiliser.

On peut également déclarer un type comme étant dérivé d'un autre type. Le nouveau type peut alors être associé à de nouvelles fonctions et à de nouvelles contraintes. Voici un exemple d'utilisation du type jours :

```
type Jour_Ouvrable is new Jour range Lundi .. Vendredi;
type Week_End is new Jour range Samedi .. Dimanche;
```

Etant donné qu'il s'agit à chaque fois de nouveaux types, les conversions implicites entre deux valeurs sont interdites. Il faut alors utiliser des opérateurs de conversion explicites.

Ada propose également la possibilité de créer des sous types. Un sous type hérite de toutes les caractéristiques de son type parent, et peut éventuellement ajouter un certain nombre de fonctionnalités et peut contraindre l'intervalle. Par exemple, voici comment on pourrait implémenter le type unsigned int en Ada :

```
subtype Unsigned_Int is Integer range 0 .. Integer'Last;
```

Qui signifie littéralement que Unsigned_Int est un entier compris entre 0 et la dernière valeur du type Integer. On rencontre une seconde fois cette syntaxe étrange, avec une apostrophe qui suit un type. On éclaircira ce point dans le chapitre suivant. Notons que Ada définit déjà ce sous type sous le nom de Natural.

5.3. Attributs

Le langage définit un certain nombre de propriétés pour chaque type : les bornes de début et de fin pour les types énumératifs et numériques, la façon de traduire une valeur de ce type en une chaîne de caractères, et inversement. Toutes ces propriétés sont appelées attributs du type, et sont accessibles en faisant suivre le nom du type d'une apostrophe.

Il existe un certain nombre d'attributs prédéfinis pour chaque type, nous ne vous en donnons ici qu'un bref aperçu.

L'attribut Image et sa réciproque Value permet de transformer une valeur d'à peu près n'importe quel type non composite en une chaîne de caractères et inversement. Nous pouvons par exemple écrire :

```
declare
   A : Integer := 99;
begin
   Put_Line (Integer'Image (A));
   A = Integer'Value ("98");
end;
```

Il existe aussi un certain nombre d'attributs qui ne sont valables que pour les types énumératifs et entiers : il s'agit des attributs Val, Pos. Val renvoie la valeur d'un énumératif en fonction de sa position, Pos renvoie sa position dans l'énumération. Ainsi, pour avoir le code ascii (plus précisément, ISO-8858-1) d'un caractère, on peut écrire :

```
declare
   Carac      : Character := 'a';
   Code_Ascii : Integer;
begin
   Code_Ascii := Character'Pos (Carac);
end;
```

Les attributs First et Last n'ont pas de paramètres, ils renvoient respectivement la valeur du premier et du dernier élément du type scalaire.

Les attributs Succ et Pred renvoient respectivement l'élément suivant et précédent d'un type primitif. Dans le cas de valeurs flottantes, cela peut être surprenant (un réel n'a pas de « suivant » au sens mathématique), mais étant donné que les réels sont discrétisés en machine, on peut « inventer » des opérations précédent et suivant, représentant le réel le plus proche représentable en machine. On peut par exemple obtenir le jour suivant de la manière qui suit :

```
Jour_Courant := Jours'Succ (Jour_Courant);
```

Cela permet, entre autres, d'éviter d'avoir systématiquement à redéfinir l'opérateur d'incrémentation.

Il existe de très nombreux autres attributs, il est possible que nous en rencontrions d'autres dans la présentation du langage, mais pour une liste exhaustive, vous pouvez vous référer soit au manuel de référence du langage, soit à l'excellent ouvrage de John Barnes (voir le chapitre « références »).

5.4. Tableaux et chaînes de caractères

Contrairement au C++, un tableau en Ada n'est pas utilisable comme un pointeur vers une adresse mémoire dont on connaît plus ou moins la taille. Un tableau, c'est une structure à part entière, qui possède un certain nombre d'attributs accessibles, qui permettent, entre autres, d'accéder à ses bornes. Les dépassements de tableaux sont impossibles en Ada, le programme contrôle toujours que les indices proposés sont appropriés. D'autre part, la borne inférieure d'un tableau n'est pas nécessairement 0. Le programmeur peut choisir de faire démarrer son tableau à l'indice qui l'arrange le plus.

Voilà un premier exemple qui déclare un tableau de 26 éléments caractères, et qui en initialise les valeurs de 'a' à 'z':

```
declare
   Tableau : array (Integer range 1 .. 26)
      of Character;
   Carcou : Character := 'a';
begin
   for I in Tableau'Range loop
      Tableau (I) := Carcou;
      Carcou := Character'Succ (Carcou);
   end loop;
end;
```

```
char Tableau [26];
char Carcou = 'a';

for (int I = 0 ; I < 25 ; ++i)
{
    Tableau [I] = Carcou;
    Carcou++;
}
```

On remarque un certain nombre de choses. Tout d'abord, en Ada, il faut typer les indices, et en donner le sous ensemble qui définit les bornes. Ici, l'indice est de type Integer, et va de 1 à 26, mais nous aurions pu utiliser n'importe quel type énumératif (par exemple le type Jours défini plus haut). Ensuite, les tableaux possèdent un attribut particulier : Range. Il s'agit de l'un des rares attributs à s'appliquer sur une variable, et non sur un type. Il sert ici à donner les bornes de parcours de la boucle for. On ne risque donc pas de se tromper d'un indice, comme cela arrive en C++. On peut également utiliser les attributs First et Last pour un tableau, qui renvoie respectivement la valeur du plus petit indice et celle du plus grand.

A la manière du C, Ada n'a pas de construction particulière pour les chaînes de caractères. La classe String du C++ n'a pas vraiment d'équivalent direct (voir éventuellement le paquetage Ada. Strings.Unbounded pour plus de détails). Une chaîne de caractères est, en C++, simplement un tableau de caractères. La variable que nous avons déclarée plus haut est donc homogène à une chaîne de caractères, même si on ne peut pas réaliser d'opération avec une telle chaîne. En effet, même si les types sont proches, le type de la variable Tableau n'est pas nommé. Pour pouvoir réaliser de telles opérations, il aurait fallu écrire :

```
Tableau : String (1 .. 26);
```

A noter qu'une chaîne est bornée par sa taille, pas par un caractère '\0' comme en C++. De manière générale, il faut mieux éviter de créer des types de tableaux anonymes. A cause du typage fort, ils ne peuvent plus être utilisés directement avec d'autres tableaux. Il faudrait mieux écrire :

```
type Tab_Carac is array (Integer range <>) of Character;
Tableau : Tab_Carac (1 .. 26);
```

Remarquez l'utilisation du diamant (<>) à la déclaration du type de tableau. En Ada, on dirait qu'il s'agit d'une boîte. Cela signifie que le tableau n'a pas de valeurs d'indice contraintes. C'est pour cette raison que l'on est obligé de les contraindre à la déclaration de la variable.

En C++, l'opérateur d'affectation ne réalise pas la copie des valeurs d'un tableau dans un autre tableau, mais la copie de l'adresse du premier tableau dans le second. En Ada, par contre, il s'agit d'une vraie copie. On pose donc une contrainte : le nombre d'éléments de la source doit être le même que celui de la destination. Comme c'est rarement le cas, on a tendance dans la pratique à affecter des tranches de tableaux. Ada permet ainsi une syntaxe de ce style :

```
declare
   Tableau_1 : Tab_Carac (1 .. 100);
   Tableau_2 : Tab_Carac (1 .. 10);
begin
   Tableau_1 (1 .. 5) := Tableau_2 (1 .. 5);
   Tableau_1 (26 .. 30) := Tableau_2 (6 .. 10);
end;
```

Remarquez que seul le nombre d'éléments importe ici. Les numéros d'indices n'ont pas besoin de se correspondre, ils « glissent » automatiquement.

Il existe une syntaxe en C++ qui permet d'initialiser directement tous les éléments d'un tableau. Ada propose le même type de syntaxe, et va beaucoup plus loin. Elle isole la notion de littéral tableau, ici appelé agrégat. Un littéral tableau est une suite de valeurs séparées par des virgules, délimitée par un jeu de parenthèses. Le nombre d'éléments est ainsi connu et, lors d'une initialisation, il n'est donc plus nécessaire de définir les bornes. Lorsque les bornes sont connues, l'utilisation de l'expression **others** => value donne une valeur par défaut à tous les éléments du tableau non définis. Ainsi, les exemples suivants sont parfaitement valides :

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

   Tableau_1 : Type_Tableau (-2 .. 43) := (others => 0);
   Tableau_2 : Type_Tableau := (1, 2, 3, 4, 5, 6, 7, 8, 9);
begin
   Tableau_2 := (1, 2, 3, others => 10);
end;
```

5.5. Les pointeurs

La notion de pointeur est sans doute l'une des caractéristiques du langage Ada les moins faciles à admettre pour le programmeur C++. Il est, en effet, impossible de les utiliser à la volée. Une variable pointeur est nécessairement typée par un type de pointeur, et non par un type d'objet. Par exemple, pour faire quelque chose d'aussi simple qu'un pointeur sur entier, voici ce que les deux langages proposent :

```
type MonPointeur is access Integer;
Var : MonPointeur;
```

```
int * Var;
```

Les concepteurs de Ada ont tenté, par ce biais, de sécuriser l'utilisation des pointeurs. A l'usage cependant, on prend l'habitude de déclarer ces types de pointeur dans une portée globale, et on gagne peu ou pas par rapport à une syntaxe plus directe, à tel point que la nouvelle norme, attendue pour 2005, semble apporter des flexibilités sur ce point.

Notez qu'en Ada, on ne parle pas de pointeur mais de type accès. Ceci dit, le mot pointeur est tellement usité que nous l'emploierons par abus de langage dans tout ce document.

Une fois le pointeur créé, on peut créer un objet par l'allocateur new comme en C++, et déréférencer en utilisant le suffixe **.all**.

```
   Var := new Integer;
   Var.all := 5;
```

```
Var = new int;
*Var = 5;
```

Ada permet en outre une affectation directe de l'objet pointé lors de sa création, par le biais d'une qualification. On pourra donc écrire de la même manière :

```
Var := new Integer'(5);
```

Un autre point particulièrement désagréable, mais cette fois plus facile à défendre, concerne les droits d'affectation. Dans sa forme basique, il est interdit d'affecter à un pointeur l'adresse d'un objet sur la pile. Pour le rendre possible, il faut qualifier le dit objet d'**aliased**, entendez de référençable. Il faut également que le pointeur soit généralisé, donc autorisé à pointer dans n'importe quelle zone mémoire, ce qui s'obtient en le qualifiant par **all**. La référence, quant à elle, s'obtient par l'attribut Access. Par exemple :

```
   type MonPointeur is access all Integer;

   VarInt : aliased Integer;
   VarPtr : MonPointeur;
begin
   VarPtr := VarInt'Access;
```

```
int VarInt;
int * VarPtr;

VarPtr = &VarInt;
```

D'autres règles se superposent encore à celles-ci. Par exemple, il est interdit d'affecter l'adresse d'une variable à un pointeur déclaré dans une portée plus grande, pour éviter que le dit pointeur puisse se retrouver à l'extérieur de la portée en pointant toujours sur un objet qui n'existe plus. Dans de très rares cas, il est absolument nécessaire d'outrepasser ces règles. Aussi Ada propose t'il des moyens de forcer ces opérations, moyens que nous n'étudierons pas ici.

6. Procédures et fonctions

6.1. Forme générale

Alors qu'en C++, une procédure n'est rien d'autre qu'une fonction avec un type de retour void, en Ada, la notion de procédure est clairement séparée de la notion de fonction. Outre le fait qu'une procédure ne peut pas renvoyer de valeur, une fonction n'a pas le droit d'avoir de paramètre résultat, l'équivalent des paramètres références non constants en C++.

Il existe quatre types de modes de passages pour les paramètres en Ada : le mode **in**, qui est le mode par défaut quand rien n'est spécifié, le mode **out** qui modifie la valeur de la variable au niveau de l'appelant, le mode **in out** qui récupère la valeur en entrée et la modifie en sortie (l'équivalent de la référence non constante en C++), et le mode **access** qui demande un pointeur dont l'adresse ne sera pas modifiée. Les fonctions ne peuvent avoir que des paramètres **in** ou **access**, alors qu'il n'y a aucune contrainte pour les paramètres d'une procédure. Voici tout de suite deux exemple de déclaration puis de définition de sous programmes :

```
procedure Proc
  (Var1 : Integer;
   Var2 : out Integer;
   Var3 : in out Integer);

function Func (Var : Integer) return Integer;

procedure Proc
  (Var1 : Integer;
   Var2 : out Integer;
   Var3 : in out Integer)
is
begin
   Var2 := Func (Var1);
   Var3 := Var3 + 1;
end Proc;

function Func (Var : Integer) return Integer
is
begin
   return Var + 1;
end Func;
```

```
void Proc
   (int Var1,
    int & Var2,
    int & Var3);

int Func (int Var);

void Proc
   (int Var1,
    int & Var2,
    int & Var3)
{
    Var2 = Func (Var1);
    Var3 = Var3 + 1;
}

int Func (int Var)
{
    return Var + 1;
}
```

Il n'est pas absolument nécessaire de déclarer le profil d'un sous programme avant de l'implémenter, mais il s'agit de l'usage le plus fréquent, et les compilateurs qui incluent des options de style refusent en général une fonction non spécifiée. Typiquement, la déclaration de la fonction est dans le .ads (sauf pour les fonctions très spécifiques et très internes) et les définitions sont dans un .adb. Comme en C++, une fonction non déclarée ne peut pas être utilisée. Ada prend également complètement en charge les problèmes de récursivité. Un petit détail, à l'appel d'un sous programme sans paramètres, C++ demande quand même l'écriture du jeu de parenthèses ouvrantes / fermantes alors qu'il n'y a rien à écrire dans un code en Ada.

Ada ne donne pas de détails quant au mode de passage des paramètres, c'est-à-dire que l'on ne doit pas savoir s'ils le sont par valeur ou par référence. Il peut être intéressant cependant d'avoir dans l'idée que la plupart des compilateurs implémentent le mode par référence.

Vous remarquerez que les paramètres formels sont séparés par des points virgules. C'est lié au fait que, en Ada, la virgule est utilisée pour lister des paramètres du même type. Les règles de déclaration sont

en fait exactement les mêmes que pour les déclarations de variables, y compris pour les valeurs par défaut.

6.2. Surnommage (ou surdéfinition)

Ada permet bien évidemment de surdéfinir les sous programmes, avec plus de performance que le C++ : grâce au typage fort, il est en effet possible de surdéfinir des fonctions ayant le même profil pour les paramètres mais un type de retour différent. Ainsi il est parfaitement possible de discriminer l'appel de ces deux fonctions :

```
function Valeur (Str : String) return Integer;
function Valeur (Str : String) return Float;
```

Cette discrimination est rendue possible par le fait qu'en Ada, il est interdit d'appeler une fonction sans en récupérer la valeur de retour. Comme il est possible de déterminer le type de la variable qui va recevoir ce résultat, il est possible de choisir la bonne fonction, sauf dans les mêmes cas d'ambiguïtés que les paramètres classiques.

Ada propose également la classique surdéfinition des opérateurs, de la même façon qu'elle est permise en C++. L'unique particularité de cette surdéfinition est que l'affectation (:=) n'est pas considéré comme un opérateur, il n'est pas possible de le surdéfinir. Pour Ada, un opérateur est une fonction, qui a un nom compris entre deux guillemets. Par exemple :

```
function "=" (Left : Jour; Right : Integer)
   return Boolean;
```

```
bool operator "=" (Jour Left, int Right);
```

7. Paquetages

7.1. Protection des déclarations

Comme nous l'avons vu dans le premier chapitre, le paquetage est l'unité de librairie de langage Ada, comme l'est le jeu de fichier .h / .cxx pour le C++. Il y a trois parties à un paquetage, réparties sur deux fichiers : une partie de spécification publique, une partie de spécification privée, et un corps. Les spécifications sont décrites dans le fichier .ads, la partie privée est séparée de la partie publique par le mot clef private. Voici la structure globale d'un fichier ads, puis d'un fichier adb :

```
package Nom_Paquetage is
-- spécifications publiques
private
-- spécifications privées
end Nom_Paquetage;
```

```
package body Nom_Paquetage is
-- corps
end Nom_Paquetage;
```

Le séparateur private permet de cacher l'implémentation d'un type particulier. Alors qu'en C++, il n'est possible de cacher que certaines parties d'une classe, en Ada, on peut aller jusqu'à cacher toute la définition d'un type. Ainsi, la syntaxe suivante est possible :

```
   type Type_1 is private;
   type Type_2 is private;
private
   type Type_1 is new Integer range 1 .. 1000;
   type Type_2 is array (Integer range 1 .. 1000) of Integer;
```

Concernant les procédures, et les sous programmes, celles déclarées au dessus du **private** seront accessibles de l'extérieur, les autres ne le seront pas. La partie corps (implémentation) a évidemment accès à la partie privée.

7.2. Paquetages hiérarchiques

Ada propose ce que l'on pourrait appeler un « héritage » de paquetages. Il est en effet possible de déclarer une unité comme étant « enfant » d'une autre, de la manière suivante :

```
package Pere.Fils is
-- corps
end Pere.Fils;
```

Ici, Pere.Fils est enfant de Pere. La partie publique de Pere.Fils a donc accès à la partie publique de Pere. Quant à la partie privée du fils (et c'est là que réside l'intérêt des paquetages enfants) elle a accès

à la partie privée du père. Par contre, les deux corps de paquetages n'ont aucun lien de visibilité l'un avec l'autre. Cette particularité de Ada est à mettre en parallèle avec le système d'héritage de classes, et participe au développement d'applications de façon incrémentale. Un certain nombre de compilateurs (dont GNAT) ont d'ailleurs pris la liberté d'ajouter leurs propres extensions sous la forme d'enfants de paquetages standards !

En résumé, on retrouve dans les architectures de paquetages un certain nombre de propriétés attendues traditionnellement dans les architectures de classes : un fils permet l'extension des fonctionnalités du père sans changer les applications n'utilisant que la ressource père.

8. Classes

8.1. Le type record

Il n'y a pas en Ada de distinction entre ce qui est struct ou class. Tout est de type enregistrement (**record**). A la base, un type record est assez proche d'une structure comme en C : il n'y a pas de notion d'héritage. Pour simplifier nos exemples, nous utiliserons le mot clef struct pour les objets sans héritage et class pour les objets avec héritage (même s'il s'agit en C++ quasiment de la même chose). Voici un exemple de déclaration :

```
type Mon_Type is record
   Champ_1 : Integer;
   Champ_2 : Character;
end record;
```

```
struct Mon_Type
{
  public:
    int Champ_1;
    char Champ_2;
}
```

Il est bien évidemment possible de cacher l'implémentation du type record à l'utilisateur. Comme nous l'avons rapidement évoqué dans le chapitre 2, ce n'est pas au niveau du type que l'on peut faire ce masquage, mais au niveau du paquetage. On n'a pas le choix : soit on montre tout, soit on cache tout. On pourrait donc avoir à l'intérieur d'un paquetage :

```
   type Mon_Type is private;

private

   type Mon_Type is record
      Champ_1 : Integer;
      Champ_2 : Character;
   end record;
```

```
struct Mon_Type
{
  private:
    int Champ_1;
    char Champ_2;
}
```

Autre différence : en Ada, les primitives d'un type ont l'air de fonctions normales : elles ne sont pas « contenues » dans le type comme en C++. Ainsi, si l'on déclare une fonction publique et une fonction privée, on aura :

```
   type Mon_Type is private;

   procedure Procedure_1 (Obj : Mon_Type);

private

   type Mon_Type is record
      Champ_1 : Integer;
      Champ_2 : Character;
   end record;

   procedure Procedure_2 (Obj : Mon_Type);
```

```
struct Mon_Type
{
  public:
    void Procedure_1 (void);

  private:
    int Champ_1;
    char Champ_2;

    void Procedure_2 (void);
}
```

Contrairement en C++, donc, le paramètre sur lequel porte la liaison dynamique n'est pas forcément le premier. Il est donc possible de surnommer de la même manière les opérateurs qui étaient parfois problématiques en C++, par exemple l'opérateur + que l'on veut « membre » pour MaClasse + Int comme pour Int + MaClasse.

Les types record en Ada n'ont pas de constructeur. Par contre, on peut associer des valeurs par défaut aux différents champs à la déclaration :

```
type Mon_Type is record
   Champ_1 : Integer   := 0;
   Champ_2 : Character := 'a';
```

```
end record;
```

Lorsque l'on a accès à l'implémentation du type, il est également possible d'utiliser les agrégats comme avec les tableaux, on utilisera ainsi soit l'une des deux syntaxes suivantes :

```
Var_1 : Mon_Type := (89, 'c');
Var_2 : Mon_Type := (Champ_2 => 'e', Champ_1 => 67);
```

Comme pour les procédures, lorsque l'on utilise la notation par nom, l'ordre dans lequel les valeurs sont données n'a pas d'importance.

8.2. Dérivation et liaison dynamique

En Ada, une classe est un type record étiqueté (**tagged**), en ce sens qu'il possède un champ supplémentaire caché, une étiquette, qui permet de résoudre les cas de liaison dynamique. Voici un exemple contenant une classe et une dérivation de cette classe, avec les données cachées à l'utilisateur :

```
   type Mon_Type is tagged private;

   procedure Procedure_1 (Obj : Mon_Type);
   procedure Procedure_2
     (Obj : Mon_Type'Class);

   type Nouveau is new Mon_Type with private;

   procedure Procedure_1 (Obj : Nouveau);

private

   type Mon_Type is tagged record
      Champ_1 : Integer;
   end record;

   type Nouveau is new Mon_Type with record
      Champ_2 : Character;
   end record;
```

```
class Mon_Type
{
  public:
    virtual void Procedure_1 (void);
    void Procedure_2 (void);

  private:
    int Champ_1;
}

class Nouveau : public Mon_Type
{
  public:
    virtual void Procedure_1 (void);

  private:
    char Champ_2;
}
```

On note un certain nombre de choses : tout d'abord, la notion de dérivation publique, privée ou protégée n'existe pas en Ada. Si on veut cacher une dérivation, il suffit ne pas la préciser dans la partie publique, mais alors vu de l'extérieur il n'y aura pas de lien de parenté. On note aussi que les primitives sont par défaut « virtuelles ». Pour qu'une primitive ne soit pas virtuelle, il faut faire suivre le type de la variable de l'attribut 'Class. On notera aussi que la dérivation multiple est interdite en Ada.

Concernant la liaison dynamique, les règles diffèrent. Alors que C++ l'applique uniquement pour les variables références ou pointeurs, en Ada, la liaison dynamique est toujours résolue. C'est-à-dire que les deux codes suivants ne donnent pas le même résultat :

```
procedure Bug (A : Mon_Type) is
begin
   Procedure_1 (A);
end Bug;
```

```
void (Mon_Type A)
{
    A.Procedure_1 ();
}
```

Si A est de type « Nouveau », le code de gauche appellera le procedure_1 de nouveau et celui de droite le procedure_1 de Mon_Type. Pour simplifier, on peut admettre qu'Ada passe toujours ces paramètres dans le mode « référence » du C++ (même si rien n'est imposé dans l'implémentation à ce propos).

Il arrive de vouloir créer une fonction ou un procédure qui ne profite pas des lois de liaison dynamique. Plus généralement, lorsque l'on déclare une procédure, un seul de ses paramètres peut profiter de la liaison dynamique (en fait ils peuvent être plusieurs, si ils sont de même type, mais nous ne parlerons pas de ce cas ici). Ces paramètres « non primitifs » sont dits à échelle de classe, et suffixés de l'attribut 'Class. Ainsi, la procédure Procedure_2 de l'exemple donné ci-dessus n'effectue pas de liaison dynamique sur son paramètre Obj.

8.3. Classes abstraites

Comme en C++, Ada propose la création de classes virtuelles pures, aussi appelées classes abstraites. Alors qu'en C++, une classe est abstraite à partir du moment où elle contient des fonctions virtuelles pures, il faut en Ada spécifier que la classe est abstraite, et il n'est pas nécessaire d'écrire de telles fonctions. Voici un exemple de déclaration de classe abstraite :

```
   type Mon_Type is abstract tagged private;

   procedure Procedure_1 (Obj : Mon_Type)
      is abstract;

private

   type Mon_Type is abstract tagged record
      Champ_1 : Integer;
      Champ_2 : Character;
   end record;
```

```
class Mon_Type
{
  public:
    void Procedure_1 (void) = 0;

  private:
    int Champ_1;
    char Champ_2;
}
```

Ce type est ensuite simplement dérivable comme n'importe quel autre type étiqueté. Si le type dérivé n'est pas abstrait, alors toutes les fonctions abstraites doivent impérativement être implémentées, comme en C++.

9. Généricité

9.1. Modèle général

Tout comme les protections de code, la généricité ne s'applique pas sur les mêmes objets en C++ et en Ada. Un template C++ peut décrire généraliser une classe ou une fonction. En Ada, il est possible de créer une procédure générique, un paquetage générique, mais pas une classe générique. Nous verrons par la suite que les paquetages génériques offrent des possibilités d'organisation de code assez agréables.

Commençons par ce qui se ressemble : une fonction générique. Celle-ci réalise une opération d'addition entre deux types.

```
generic
   MonType is (<>);
function Add (A, B : MonType)
   return MonType is
begin
   return A + B;
end;
```

```
template <class MonType>
MonType Add (MonType A, MonType B)
{
   return A + B;
}
```

Remarquez la déclaration du type avec (<>). Il s'agit de la déclaration de paramètre générique discret, qui peut être instancié uniquement à l'aide d'un type dérivé d'entier ou d'une énumération. Nous verrons dans le chapitre suivant quels sont les différents types de paramètres que l'on peut exprimer.

Sans rentrer davantage dans les commentaires, voyons maintenant comment utiliser ces objets. Une chose assez désagréable en Ada, pour qui est habitué au C++, est qu'il est impossible d'utiliser une entité générique directement. Il faut nécessairement créer une instanciation de l'entité avant utilisation. Par exemple, pour faire un appel à la fonction ci-dessus, il faudra procéder ainsi :

```
   function AddInt is new Add (Integer);
   Res : Integer;
begin
   Res := AddInt (3, 4);
```

```
int Res;

Res := Add <int> (3, 4);
```

Bien évidemment, une fois instanciée, la nouvelle fonction peut être appelée autant de fois que nécessaire.

9.2. Paramètres de généricité

Ada offre une véritable richesse dans l'expression des paramètres de généricité, richesse qu'il est impossible de modéliser en C++. Mais cette richesse est nécessaire pour rentrer dans le cadre de contraintes proposées par le langage. Voyons par exemple une fonction qui renvoie le maximum de deux valeurs, avec comme paramètre générique l'expression Ada la plus vague, avec le mot clé **private** :

```
generic
   MonType is private;
   with function ">" (A, B : MonType)
      return Boolean;
function Max (A, B : MonType);

function Max (A, B : MonType) is
   return MonType is
begin
   if (A > B) then
```

```
template <class MonType>
MonType Max (MonType A, MonType B)
{
   if (A > B) return A;
   else return B;
}
```

```
      return A;
   else
      return B;
   end if;
end;
```

On a eu besoin, en Ada, de déclarer en paramètre non seulement le type, mais en plus la fonction utilisée à l'intérieur, ici l'opérateur de supériorité. Notez au passage que pour discriminer l'unité générique de ses paramètres, on préfixe le nom de la fonction du mot clef **with**. Nous y reviendrons. Le C++, lui, détecte *a posteriori* l'utilisation de l'opérateur, et provoquera une erreur de compilation si, par exemple, nous utilisons la fonction avec une classe qui ne le possède pas.

En Ada, on aurait pu contraindre le type à être discret, comme dans l'exemple précédent. On a alors l'autorisation d'usage de tous les attributs relatifs aux types discrets, ainsi que de toutes les fonctions et opérateurs associés. C'est la raison pour laquelle l'opérateur « + » de l'exemple du paragraphe 9.1 n'est pas déclaré comme paramètre générique.

Remarquez aussi que dans le cas générique, spécification et implémentation sont toujours séparées, et que l'implémentation ne rappelle pas les paramètres de généricité.

D'autre part, comme en C++, il est possible de passer n'importe quel objet en paramètre de généricité.

Voici un aperçu non exhaustif des contraintes que l'on peut mettre sur ces types, et de leur signification :

```
type T is limited private; -- T est un type limité.
type T is private; -- T est un type caché, mais on peut effectuer des affectations.
type T (<>) is private; -- T est un type caché non contraint.
type T is tagged private; -- T est un type étiqueté, que l'on peut donc dériver.
type T1 is new T2; -- T1 est un type dérivé de T2, non étiqueté.
type T1 is new T2 with private; -- T1 est un type dérivé de T2, étiqueté.
type T is (<>); -- T est discret.
type T is range <>; -- T est un entier.
type T is digits <>; -- T est un flottant.
type T1 is access T2; -- T1 est type pointeur sur T2.
```

En règle générale, on peut retrouver ici à peu près toutes les formes de déclarations vues ailleurs.

Comme on l'a vu plus haut, il est possible de mettre en paramètres de généricité paquetages, fonctions et procédures, à condition cependant de les préfixer du mot clé **with**, ceci afin de ne pas les confondre avec l'unité générique qui commencera par le mot **package**, **function** ou **procedure** sans le **with**. Les fonctions et procédures peuvent être décrits à l'aide des types génériques décrits plus haut. L'intérêt d'un paquetage en paramètre générique est que l'on peut le contraindre à être lui-même dérivé d'un autre paquetage générique, par exemple :

```
with package P is new GenPkg;
```

Nous allons voir le problème des paquetages génériques dans le chapitre suivant.

9.3. Paquetages génériques

De la même manière que l'on rend une fonction générique, il est possible de rendre un paquetage générique en préfixant la spécification du mot clé **generic**, par exemple :

```
generic
   type MonType is (<>);
package MonPaquetage is
   VarGlob : MonType;
end MonPaquetage;
```

On instanciera ce paquetage à l'aide de l'opérateur **new**. Voici un exemple de deux instantiations consécutives :

```
package P1 is new MonPaquetage (Integer);
package P2 is new MonPaquetage (Integer);
```

On comprend ici la nécessité de faire une instanciation explicite. En C++, deux unités instanciées par le même jeu de paramètres partagent le même jeu de variables globales (en particulier les variables statiques de classe). Ici, P1 et P2 sont instanciés de la même façon, mais possèdent pourtant deux variables VarGlob distinctes.

Notons également que, lorsqu'un paquetage est générique, ses enfants le sont nécessairement, même s'ils n'ajoutent pas de paramètres de généricité. D'autre part, comme pour les fonctions et procédures, le corps du paquetage n'a pas à être préfixé des paramètres de généricité.

10. Exceptions

10.1. Exceptions standard

La notion d'exception est très pauvre en Ada, comparé à ce qu'elle est en C++. Une exception, c'est une variable d'un type particulier, le type **exception**. En standard 83, il est même impossible d'y associer un texte (ce problème est résolu depuis la norme 95 à l'aide du paquetage Ada.Exceptions, décrit dans le chapitre suivant). Cela dit, il apparaît à l'usage que la structure fournie par Ada est largement suffisante dans la plupart des cas. A noter que de très nombreuses exceptions prédéfinies sont levées par le système lors des divers contrôles (par exemple, dépassements de bornes) ce qui permet de mettre en place une politique de tolérance aux erreurs. Une exception se déclare comme une variable, de la façon qui suit :

```
Mon_Exception : exception;
```

Il est ensuite possible de la lever à l'aide du mot clef « **raise** » (au lieu du throw C++). On remarque que les exceptions sont des variables en Ada, alors qu'il s'agissait de types en C++. Il n'est donc pas nécessaire de créer de nouvelles instances en Ada.

```
raise Mon_Exception;
```

```
throw Mon_Exception;
```

Comme en C++, on récupère les exceptions de tout un bloc. Cependant, il n'est pas nécessaire d'ajouter une directive de type « try » au début d'un bloc, seul l'équivalent du « catch » suffit en bout de bloc. On test ensuite la valeur de l'exception à l'aide de commandes **when**, un peu à la manière d'une instruction case. Pour tester n'importe quel type d'exception, on utilise la syntaxe **when others** équivalente au catch (…). Voici un exemple de récupération d'erreurs :

```
begin
   Appel_De_Procedure;
exception
   when Exception_1 =>
      Put_Line ("Erreur 1");
   when Exception_2 =>
      Put_Line ("Erreur 2");
   when others =>
      Put_Line ("Erreur inconnue");
end;
```

```
try
{
    Appel_De_Procedure
}
catch (Exception_1)
{
    cout << "Erreur 1" << endl;
}
catch (Exception_2)
{
    cout << "Erreur 2" << endl;
}
catch (...)
{
    cout << "Erreur inconnue" << endl;
}
```

Comme en C++, il est possible de lever une exception dans un traitant d'exception. Pour relever l'exception courante, encore une fois comme en C++, il suffit d'écrire l'instruction de levée d'exception sans argument, ici **raise**.

10.2. Exceptions étendues (Ada 95)

Heureusement, la forme réduite des exceptions de la norme 83 a été rapidement étendue, de façon à intégrer l'un des aspects les plus utilisés : le message d'erreur. Lever une exception étendue, c'est en fait lever une exception simple en y ajoutant de l'information texte, par une fonction issue du paquetage Ada.Exceptions. Voici un exemple :

```
with Ada.Exceptions; use Ada.Exceptions;

-- code

Mon_Exception : exception;

-- code

Raise_Exception (Mon_Exception'Identity, « Mon message »);
```

Pour identifier une exception, il faut en faire suivre le nom de l'attribut Identiy. Il sera ensuite possible de récupérer les informations de l'instruction dans un bloc de traitement, de la façon suivante :

```
exception
  when Recu : Mon_Exception =>
    Put_Line (Exception_Name (Recu)); -- Affiche le nom de l'exception.
    Put_Line (Exception_Message (Recu)); -- Affiche le message donné avec l'exception.
    Put_Line (Exception_Information (Recu)); -- Affichage du nom et du message
end;
```

11. Programmation concurrente et temps réel

11.1. Avertissement

Il n'est pas forcément pertinent de faire une comparaison des deux langages sur le point de la programmation concurrente. Alors que, en C++, il est nécessaire de passer par une librairie, en Ada, le langage fournit des structures propres.

Afin de vous éviter d'aller fouiller dans la librairie de C++ pour comprendre les exemples qui suivent, nous avons préféré utiliser des primitives simplifiées, que nous introduirons au fur et à mesure. L'important est de comprendre ce qui différencie ces deux langages en termes de structure. Il s'agit cependant de comparaisons fonctionnelles. Pour le compilateur, en effet, posséder des structures de langage donne une connaissance sémantique qui autorise un certain nombre d'optimisations. La question de l'efficacité du code n'a donc pas à être posée ici.

11.2. Tâches

Les tâches sont à rapprocher de la notion de thread. Elles permettent de créer des processus concurrents qui partagent le même espace d'adressage. Pour simplifier, nous admettrons que nous avons en C++ les fonctions suivantes :

```
class Thread
{
   void run () = 0; // Code à exécuter
   void start (); // démarre le thread
}
```

Voici maintenant un petit exemple simple, qui lance deux tâches en parallèle affichant chacune les vingt six lettres de l'alphabet.

```
procedure Main is
   task body MyTask is
   begin
      for I in 'A' .. 'Z' loop
         Put_Line (I);
      end loop;
   end;
begin
   for I in 'A' .. 'Z' loop
      Put_Line (I);
   end loop;
end;
```

```
class MyThread : public Thread
{
   public:

   void run ()
   {
      for (char i = 'A'; i <= 'Z'; i++)
      {
         cout << i << endl;
      }
   }
}

void main (void)
{
   MyThread MyTask;
   MyTask.start ();

   for (char i = 'A'; i <= 'Z'; i++)
   {
      cout << i << endl;
   }

}
```

En Ada, donc, une tâche se déclare un peu comme une procédure, à ceci près qu'il faut utiliser le mot **task** pour l'identifier. Elle doit être déclarée à l'intérieur d'un bloc déclaratif (dans un sous-

programme, une autre tâche, un bloc **declare**…). Plusieurs tâches peuvent d'ailleurs être déclarées à l'intérieur du même bloc. Elle démarre automatiquement sur le **begin** du bloc en question, et la fin de son exécution est systématiquement attendue au niveau du **end** du même bloc. Toutes les tâches déclarées directement dans un bloc commencent donc en même temps.

Il est également possible de créer des « types de tâches » que l'on peut ensuite instancier autant de fois que l'on veut. Une tâche est alors instanciée dans un bloc, démarre au **begin** de ce bloc et se finit sur le **end**. Il est également possible de paramétrer ces types de tâches. Voici un exemple, fondé sur l'exemple précédent, qui compte affiche dix fois une portion de l'alphabet contenue entre une lettre donnée en paramètre et Z.

```
task type MyTask (First : Character);

task body MyTask (First : Character) is
   for I in First .. 'Z' loop
      Put_Line (I);
   end loop;
end MyTask;

procedure Main is
   Tab : array (0 .. 9) of MyTask ('G');
begin
   null;
end;
```

```
class MyThread : public Thread
{
   public:

   char First;

   void run ()
   {
      for (char I = First; I <= 'Z'; I++)
      {
         cout << I << endl;
      }
   }
}

void main (void)
{
   MyThread Tab [10];

   for (int I = 0; I < 9; I++)
   {
      Tab [I].First = 'G';
      Tab [I].start ();
   }
}
```

La dernière façon de créer des tâches, et non la moindre, est de les construire non plus statiquement mais dynamiquement, par pointeur. La tâche démarre alors au moment de son instanciation, et se termine n'importe quand. Très utilisé pour faire des démons par exemple, ou des traitements en tâche de fond. Voici un exemple, partant du principe que les déclarations données ci-dessus existent :

```
type Ptr_Task is access MyTask;

procedure Main is
   Daemon : Ptr_Task;
begin
   Daemon = new MyTask ('G');
end;
```

```
void main (void)
{
   MyThread * Daemon = new MyThread ('G');
}
```

11.3. Rendez-vous

La notion de rendez-vous est difficile à modéliser en C++, sans le support d'un mécanisme complexe. Nous n'allons que le survoler ici, tant ce système est riche mais également complexe et caractéristique de Ada.

Un rendez-vous, c'est un point de synchronisation dans le programme, pendant lequel deux tâches peuvent faire un échange. Considérons les quelques lignes suivantes :

```
procedure Main is
   task Suite is
```

```
      entry Go;
   end Suite;

   task body Suite is
   begin
      accept Go do
      end Go;

      Put_Line ("Suite");
   end;
begin
   Put_Line ("Debut");
   Suite.Go;
end;
```

Une nouvelle entité apparaît ici, introduite par le mot clé **entry**. Il s'agit d'un point de synchronisation, ou point de rendez-vous. Dans le modèle Ada, les deux suites d'instructions, la procédure Main et la tâche Suite, commencent en même temps. La première instruction que reçoit Suite est une instruction **accept**, qui a pour effet d'attendre une occurrence de rendez-vous. L'entrée peut d'ailleurs être paramétrée, il faut alors rentrer les paramètres avant le **do**. Entre le **do** et le **end**, un code peut être écrit (par exemple en fonction des dits paramètres).

En résumé, ici, Suite et Main démarrent. Main affiche « Début », Suite attend un Go, Main envoie un Go à Suite et Suite affiche « Suite ». Les deux flux d'instructions ont ainsi synchronisé leurs sorties à l'écran.

11.4. Rendez-vous multiple

Il est également possible de créer des points de rendez-vous multiples, grâce à l'instruction **select**. Prenons un exemple simple, le classique lecteur consommateur. On imagine ici une tâche à qui l'on peut demander soit d'ajouter, soit de retrancher un entier, soit d'en récupérer la valeur.

```
task Compteur is
   entry Lire (Val : out Integer);
   entry Incrementer;
   entry Decrementer;
end task;

task body Compteur is
   Cpt : Integer := 0;
begin
   loop
      select
         accept Incrementer do
            Cpt := Cpt + 1;
         end;
      or
         accept Decrementer do
            Cpt := Cpt - 1;
         end;
      or
         accept Lire (Val out Integer) do
            Val := Cpt;
         end;
      or
         delay 1.0 * Minute;
         exit;
      end select;
   end loop;
end Compteur;
```

Ici, lorsque le flot d'instructions arrive au select, il se met à attendre de façon concurrente quatre évènements, qui sont trois entrées différentes et un délai. Si le délai d'une minute d'attente est dépassé, la tâche ira lire l'instruction à la suite du **delay**, sortira de la boucle à cause du **exit**, et se terminera. Dans les

autres cas, on exécutera le code demandé. A noter qu'un système d'atomicité (on pourrait parler abusivement de sémaphores) est automatiquement mis en place. Ainsi, nous avons la garantie qu'aucun des quatre blocs ne sera exécuté en parallèle.

Ceci dit, l'intérêt de cette structure n'est pas seulement l'atomicité des instructions (il existe d'autres formes plus simples pour cela) mais plutôt la possibilité de réagir à des évènements asynchrones.

11.5. Objets protégés

Le concept d'atomicité des opérations est modélisé en Ada par la notion d'objet protégé. Là encore, difficile de proposer une traduction « littérale » du C++, sans introduire un grand nombre de fonctions imprécises. Un objet protégé, c'est un objet dont les méthodes (fonctions et procédures) ne peuvent êtres appelées que en exclusion mutuelle (moniteur Hoare). On peut donc aller beaucoup plus loin que la notion de simple sémaphore. En effet, que fait t-on lorsqu'une fonction protégée appelle une autre fonction du même objet protégé (donc sur les mêmes sémaphores) ? Un programmeur peu averti aura alors construit son premier deadlock, s'il travaille dans un langage type C++. Ada, grâce à sa connaissance du code, sera en mesure de comprendre de ne placer la protection que dans le premier appel. Et ce n'est qu'un exemple simpliste.

On peut reprendre l'exemple, donné plus haut avec la structure select, pour proposer un autre modèle :

```
protected type Compteur is
   procedure Lire (Val : out Integer);
   procedure Incrementer;
   procedure Decrementer;
private
   Cpt : Integer := 0;
end Compteur;

protected body Compteur is

   procedure Lire (Val : out Comtpeur) is
   begin
      Var = Cpt;
   end Lire;

   procedure Incrementer is
   begin
      Cpt = Cpt + 1;
   end Incrementer;

   procedure Decrementer is
   begin
      Cpt = Cpt + 1;
   end Decrementer;

end Compteur;
```

On pourrait s'inquiéter des deux manières de faire une chose à priori complètement identiques. Il est vrai que dans certains cas, on hésite entre une tâche et un type protégé. Ceci dit, dans les cas plus complexes, l'un et l'autre trouvent rapidement leur intérêt, et le plus souvent, se complètent. Rappelons que la tâche a une propriété que l'objet protégé n'a pas : elle s'exécute en continu.

Dans son écriture, un objet protégé se comporte un peu comme un paquetage, avec une partie private. L'analogie s'arrête là, impossible, par exemple, de déclarer des types à l'intérieur. Il est d'ailleurs souvent dommage de ne pas pouvoir cacher l'instantiation de certains types dans cette partie private, Ada nous interdit de déclarer un type à l'intérieur d'un autre type.

Voici maintenant un exemple d'utilisation de cet objet :

```
   Var : Compteur;
   Val : Integer;
begin
   Var.Incrementer;
   Var.Decrementer;
   Var.Lire (Val);
```

Bien entendu, ici, faire un objet protégé n'a pas d'intérêt. Il faut imaginer qu'il y a d'autres tâches dans la nature capable d'appeler ces différentes primitives n'importe quand.

Il est intéressent de noter que l'on retrouve ici la notation pointée, chère aux utilisateurs des langages à objets dérivés du C++. Cette notation pointée devrait d'ailleurs être généralisée à tous les objets avec la nouvelle norme en 2005.

Un mot encore, sur ces objets protégés. Comme les tâches, ils possèdent des entrées, déclarées avec le mot **entry**. Ces dernières ont cependant un sens un peu différent. Une entrée est une procédure qui se bloque sur une condition. Nous allons par exemple ajouter une entrée qui attend que le compteur vaille zéro pour se déclencher :

```
protected type Compteur is
  -- Code
   entry AttendreZero;
private
   Cpt : Integer;
end Compteur;

protected body Compteur is

   -- code

   entry AttendreZero when Cpt = 0 is
   begin
      Put_Line ("Le compteur vaut zéro !");
   end AttendreZero;

end Compteur;
```

Une entrée s'appelle comme une procédure normale, à ceci près qu'elle attendra la clause à droite du **when** pour s'exécuter. Ainsi, tout processus appelant AttendreZero ne pourra continuer son flux d'instruction que lorsque Cpt vaudra 0, sauf si l'appel de cette entrée est soumis à une clause **select**.

L'utilisation des tâches en Ada peut avoir beaucoup d'intérêt dans le cadre d'enseignements relatifs au temps réel et à la programmation concurrente. Un certain nombre de concepts y sont décrits plus explicitement que dans les langages de programmation classique. A ce sujet, vous trouverez des exemples plus complets dans l'article cité en référence « enseigner Ada ».

12. Les petits bonheurs de Ada

12.1. Introduction

Ce chapitre regroupe quelques bijoux du langage Ada. Il s'agit de concepts particulièrement avancés, voir exotiques, qu'il n'est absolument pas nécessaire de connaître pour programmer. Pire, ils se fondent sur des concepts qui n'ont pas été étudiés dans ce document, et vous ne pourrez donc vraisemblablement pas les utiliser directement. Mais il est bon de savoir que de telles merveilles existent pour les étudier plus en profondeur lorsque le besoin s'en fait sentir.

12.2. Gestion des données bit à bit

Bien que travaillant à très haut niveau, Ada propose un certain nombre de moyens de contraindre le compilateur à utiliser en machine une représentation particulière. Ces contraintes peuvent se révéler particulièrement intéressantes lorsque l'on souhaite utiliser des fractions de mots machines pour mémoriser des valeurs. Admettons par exemple que nous souhaitions mémoriser et manipuler un type d'octet par leurs poids forts et leur poids faibles. Ada nous permet d'écrire la chose suivante :

```
   subtype Caractere_Hexa is Integer range 0 .. 15;

   type Octet is record
      Poids_Forts, Poids_Faibles : Caractere_Hexa;
   end record;

   for Octet use record
      Poids_Forts at 0 range 0 .. 3;
      Poids_Faibles at 0 range 4 .. 7;
   end record;

   O : Octet;

begin

   O.Poids_Forts := 10;
   O.Poids_Faibles := 12;
```

On commence par contraindre un sous type d'entier par les valeurs possibles. Le type d'enregistrement est ensuite déclaré simplement, suivi d'une clause de représentation introduite par la structure **for … use**, suivi de la liste de tous les champs du type d'enregistrement. Pour chaque champ, on spécifie ensuite le numéro de l'octet sur lequel on place la valeur (ici, il n'y en a qu'un, l'octet zéro) suivi de l'intervalle des bits de cet octet que l'on souhaite utiliser.

Les opérations utilisées par la suite, en particulier les affectations, fonctionnent exactement de la même manière qu'en l'absence des clauses de représentations. L'étendue des possibilités et de la simplicité d'une telle syntaxe est laissée à votre appréciation.

12.3. Sérialisation d'enregistrements

L'opération de sérialisation consiste à transférer un objet dans un flux (une socket, un fichier…). En réception, on recrée chaque objet transféré, ce qui implique que l'on sache de quel type d'objet il s'agit. Les problèmes arrivent lorsque l'on envoie un objet dont on ne connaît que le type de base. Par exemple, de TA dérivent TB1 et TB2, et la procédure en réception sait qu'elle va recevoir des objets

dérivés de TA. Comment savoir s'il faut créer des TB1 ou TB2 ? Passe encore s'il n'y a que deux types, on peut envoyer un code discriminant en entête (un entier par exemple, qui vaut 1 pour TB1 et 2 pour TB2) mais comment faire lorsque l'on ne maîtrise pas la dérivation, et que l'utilisateur peut à loisir ajouter des types dérivés ?

Le C++ propose d'identifier chaque classe de manière unique par une chaîne de caractères, obtenue via l'opérateur typeof. Mais cette proposition ne résout qu'une partie du problème : en réception, il faut savoir faire la correspondance entre cet identificateur et le constructeur de la classe en question, ce qui présuppose donc un traitement manuel pour chaque classe susceptible de passer dans ce flux.

En Ada, il est possible de construire un objet directement à partir d'un flux. L'exemple suivant fait appel à un certain nombre de notions non étudiées dans ce document, mais devrait cependant être compréhensible :

```
type TA is private;
type TB1 is new TA with private;
type TB2 is new TA with private;

type PtrTA is access all TA'Class;

-- [...]

procedure Lire (Stream : access Root_Stream_Type'Class; Obj : out PtrTA) is
begin
   Obj := new TA(PtrTA'Class'Input (Stream));
end Lire;

procedure Ecrire (Stream : access Root_Stream_Type'Class; Obj : in PtrTA) is
begin
   PtrTA'Class'Output (Stream, Obj.all);
end Ecrire;
```

Vous l'aurez compris, le type de flux de C++ est Root_Stream_Type, et une des façons d'y écrire est d'utiliser les attributs Input et Output du type de la variable. Ici, on va même plus loin. On ne se contente pas d'utiliser les attributs de l'objet, mais de la classe entière (TA'Class). En écrivant par exemple « PtrTA'Class'Output (Stream, Obj.all) » on annonce que l'on n'écrit pas un objet de type TA mais un objet de la classe de TA. Le compilateur est alors capable de rajouter les informations nécessaires dans le flux pour que, en réception, on connaisse le bon objet à récupérer et qu'on soit capable de le recréer. Ici donc, on peut indifféremment appeler Ecrire avec des objets de type TA, TB1 ou TB2 (ou tout objet appartenant à la classe de TA), l'écriture appellera la bonne fonction de sérialisation, et la lecture créera le bon objet. En toute simplicité.

Conclusion

Ada n'a été ici que très simplement survolé. La partie sans doute la plus intéressante du langage, concernant les processus concurrents, n'est traitée que par une poignée de pages qui peinent à vous sensibiliser sur des structures particulièrement efficaces. Ceci dit, vous devriez être à même maintenant de programmer vos premiers codes en Ada, voir de rejoindre un projet utilisant déjà ce langage. Il y a encore à voir, mais vous connaissez les bases.

Si votre curiosité a été éveillée, le but de ce document aura été atteint. Merci de l'avoir lu jusqu'au bout, merci d'avoir accepté, au travers de ces quelques chapitres, de ranger vos a prioris sur ce langage méconnu. Si vous avez envie d'aller plus loin, n'hésitez pas à consulter les ouvrages proposés en référence.

13. Références

13.1. Cours et ressources en ligne

Le cours en ligne de D. Feneuille (http://libre.act-europe.fr/french_courses/)
Ada, C, C++ and Java vs The Steelman (www.adahome.com/History/Steelman/steeltab.htm)
Enseigner Ada (http://d.feneuille.free.fr/enseignerada.htm)
C++ ??: A Critique of C++ (http://burks.brighton.ac.uk/burks/pcinfo/progdocs/cppcrit/)

13.2. Cours en librairie

LA référence, « Programmer en Ada 95 » de John Barnes aux éditions Vuibert (ISBN 2-84180-973-0)
« Programmation séquentielle avec Ada95 » aux éditions presses romanes (ISBN 2-88074-404-0)
« Programmation concurrente et temps réel avec Ada95 » aux éditions presses romanes (2-88074-408-3)

13.3. Outils

Le compilateur GNAT, l'environnement de développement GPS et autres merveilles sur http://libre.act-europe.fr/

13.4. Divers

L'association Ada-France (http://www.ada-france.org/)
Formation privée Adalog (http://www.adalog.org/)
Formation publique CNAM (http://deptinfo.cnam.fr/Enseignement/CycleA/APA/)