Cours d'Ada 95
pour le programmeur C++

V 0.2 (12/05/2004)
(actuellement en cours de rédaction, document non terminé)
Auteur : Quentin Ochem (simboy@users.sf.net)

Note sur le présent document

Ce document a été rédigé pour permettre aux programmeurs connaissant le C++ et souhaitant rejoindre le projet G3C (http://www.sf.net/projects/g3c) d'apprendre rapidement le langage Ada. De solides connaissances en C++ sont requises pour sa compréhension. Seul un tour d'horizon du langage est proposé ici. Pour plus de détails, vous pouvez vous référer aux liens donnés dans le chapitre titré « references ».

Toutes les remarques ou suggestions sont largement appréciées. Vous pouvez les envoyer à cette adresse : simboy@users.sf.net.

1. 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 ce langage ont coutume de prétendre qu'ils les implémentent mieux, nous nous contenterons de dire qu'elle 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 systématiques de warnings et de normes de codage. Mais personne n'est à l'abri d'une erreur d'inattention non contrôlée.

Le langage Ada vient 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 peu 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).

2. Structure des fichiers

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, un fichier, c'est toujours une unité de compilation, séparée en deux parties : la spécification (.ads) et le corps (.adb). Il y a deux types d'unités de compilation : les paquetages et les procédures. 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. Cette façon de protéger des données est assez différente de la façon 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 :

```
use Nom_Paquetage;
```

```
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 paquetage, 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 de fichier de spécification.

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 :

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

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

```
#include <iostream>

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

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

3. Syntaxe générale

3.1. Déclarations

Comme en C++, l'indication de fin de ligne est le point virgule. Un bloc ne se déclare pas avec deux accolades, mais avec 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'opérateur d'affectation est le :=. Il n'y a pas d'incrémentation raccourcie (++). Les types de base sont Integer, Boolean Float et Character pour int, bool, float et char.

Prenons par exemple ce code qui réalise la déclaration puis l'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 .

3.2. Conditions

Une condition en Ada s'énnonce **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** marque la fin de la condition. Les opérateurs de comparaisons 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 opérateur **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 type 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 tests soient réalisés. 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 suivant, 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 résous la plupart des problèmes qui avaient poussés 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 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;
  default:
    cout << "Autre chose" << endl;
}
```

3.3. 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 premier **loop** est éventuellement précédé d'une instructor **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 est non 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 entières. 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 pour un parcours à l'envers. 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 parcoure 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 qu'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.

4. Types

4.1. Typage fort

L'une des plus notables caractéristiques d'Ada, et l'une des moins appréciés par les nouveaux programmeurs, c'est la quasi absence de conversions implicites, et la nécessité de convertir à la main. 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 ce 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 (quoi que gcc ne râle pas dans ce cas même avec le flags –Wall). Il n'en reste pas moins que certaines erreurs peuvent se glisser à cause de ce genre de subtilités, et que la proposition d'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 litéral 10.0, alors que 10 suffit dans le cas C++. Il s'agit encore d'une restriction de typage, 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.

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 au maximum 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.

4.2. Construction de nouveaux types

Lorsque l'on prend l'habitude d'utiliser le typage fort, on se rend compte qu'il est souvent très intéressent de se 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 .. 60;

    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, générera éventuellement un message d'alerte. Mais à l'exécution du code, une erreur 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.

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

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

```
enum Jours
{
    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 préfères mettre l'accent sur le caractère abstrait de ces valeurs, qui ne sont sensées 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.

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

4.3. Attributs

On peut identifier un certain nombre de propriétés pour chaque type : les bornes de début et de fin pour les types énumératifs et entiers, 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, First, Last, Succ et Pred. Val renvoie la valeur d'un entier énumératif en fonction de sa position, Pos renvoie sa position dans l'énumération. Ainsi, pour avoir le code ascii 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 ne prennent pas de paramètres, ils renvoient respectivement la valeur du premier et du dernier élément de l'énumération.

Les attributs Succ et Pred renvoient respectivement l'élément suivant et précédent d'une énumération. 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'addition.

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 en annexe).

4.4. 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 valeurs proposées sont appropriées. 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'index 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 < 100 ; ++i)
{
    Tableau [I] = Carcou;
    Carcou++;
}
```

On remarque un certain nombre de choses. Tout d'abord, en Ada, il faut typer les indexes, et en donner le sous ensemble qui définit les bornes. Ici, l'index 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 index, 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 index et celle du plus grand.

A la manière du C, Ada n'a pas de construction particulière pour les chaînes de caractère. La classe String du C++ n'a pas vraiment d'équivalent direct (voir éventuellement le paquetage Ada.Unbounded_Strings 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 caractère, 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 aurait mieux valut 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'élément importe ici. Les numéros d'indexes 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'exression **others =>** value donne une valeur par défaut à toutes les cases du tableau non définies. Ainsi, les exemples suivants sont parfaitement valides :

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

   Tableau_1 : Type_Tableau (1 .. 100) := (others => 0);
   Tableau_2 : Type_Tableau := (1, 2, 3, 4, 5, 6, 7, 8, 9);
begin
   Tableau_2 := (1, 2, 3, others => 10);
end;
```

4.5. Les pointeurs

5. Procédures et fonctions

5.1. 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 à l'extérieur de la procédure, 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 modifié. 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 et 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);


   (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 définissions sont dans un .adb. Comme en C++, une fonction non déclarée ne peut pas être utilisée. Ada supporte également complètement 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 parenthèse ouvrantes / fermantes alors qu'il n'y a rien à écrire dans un code en Ada.

Ada ne donne pas de détails quand 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 intéressent cependant d'avoir dans l'idée que la plupart des compilateurs implémentent le mode par référence.

Vous remarquerez que les paramètres 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.

5.2. 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;
```

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 à part entière, 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);
```

6. Paquetages

6.1. 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 dans 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.

6.2. Paquetages enfants

7. Classes

7.1. 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 une assez proche d'une structure comme en C : il n'y a pas de notion d'héritage. Pour simplifier nos exemples, nous utiliseront 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 on l'air de fonction 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);
}
```

C'est particulièrement utile lorsque l'on souhaite déclarer une fonction primitive d'un type qui n'est pas le premier paramètre. On peut ainsi surdéfinir tous les opérateurs (que nous étudierons plus tard).

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.

7.2. 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 public, 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).

7.3. 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++.

8. Généricité

9. Exceptions

9.1. 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 particuler, le type **exception**. En standard, il est même impossible d'y associer un texte (ce problème est résolu 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. 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'erreur. :

```
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**.

9.2. Exceptions étendues

10. Temps réel

11. Références

Ce petit guide est largement incomplet. Il suffit si vous voulez commencer à travailler en Ada, mais de très nombreux détails et caractéristiques n'y sont pas abordés. Si vous souhaitez aller plus loin, voici deux références :

Le cours en ligne de Daniel Feneuille (de l'I.U.T. d'informatique d'Aix en Provence) à l'URL : http://libre.act-europe.fr/french_courses/

« Programmer en Ada 95 » de John Barnes aux éditions Vuibert