Le langage Ada : liaison avec d'autres langages

Yves Bailly

EN DEUX MOTS **L'une des exigences du langage Ada, lors de sa conception, était de pouvoir s'interfacer avec les autres principaux langages existants à l'époque. De ce point de vue, Ada est peut-être plus ouvert que d'autres langages, permettant la réutilisation de bibliothèques existantes assez facilement.**

Le standard Ada prévoit, dans son Annexe B, diverses facilités pour utiliser d'autres langages, nommément les langages C/C++, Fortran et Cobol. Ce dernier a connu son heure de gloire, dominant largement l'industrie informatique pendant de nombreuses années, mais il est aujourd'hui, disons, « un peu » passé de mode. Aussi ne le verrons-nous pas.

Par contre, le langage Fortran est toujours très utilisé dans les domaines gros consommateurs de calculs, qu'ils soient scientifiques ou industriels. Quant au langage C, il est probablement encore le plus répandu et le plus courant pour interfacer des systèmes.

Fortran

Précisons dès maintenant que votre serviteur est bien loin d'être un expert en langage Fortran. Toutefois, ces maigres connaissances permettent tout de même d'écrire une procédure (ou sous-routine, *subroutine* en anglais) permettant de résoudre le problème des Tours de Hanoï (avouez que cela vous manquait), dans un fichier `f_hanoi.f` que voici :

```
 1 subroutine Hanoi(nb, depuis, par, vers)
 2    integer, intent(in) :: nb
 3    integer, intent(in) :: depuis
 4    integer, intent(in) :: par
 5    integer, intent(in) :: vers
 6    if (nb == 1) then
 7       call Deplacer(depuis, vers)
 8    else
 9       call Hanoi(nb-1, depuis, vers, par)
10       call Hanoi(1, depuis, par, vers)
11       call Hanoi(nb-1, par, depuis, vers)
12    end if
13 end subroutine Hanoi
```

On retrouve l'algorithme classique. Remarquez l'invocation ligne 7, d'une procédure `Deplacer()` qui apparemment n'existe pas : elle sera en fait fournie par un paquetage Ada, lequel va lui-même importer la procédure `Hanoi()`. Voici la spécification du paquetage, dans `a_hanoi.ads` :

```
 1 with Interfaces.Fortran ; use Interfaces.Fortran ;
 2 package A_Hanoi is
 3
 4    procedure Hanoi(nb     : in Fortran_Integer ;
 5                    depuis : in Fortran_Integer ;
 6                    par    : in Fortran_Integer ;
 7                    vers   : in Fortran_Integer) ;
 8    pragma Import(Fortran, Hanoi, "hanoi_") ;
 9
10    procedure Deplacer(depuis: in Fortran_Integer ;
11                       vers  : in Fortran_Integer) ;
12    pragma Export(Fortran, Deplacer, "deplacer_") ;
13
14 end A_Hanoi ;
```

On importe, pour commencer, le paquetage `Interfaces.Fortran` (ligne 1), qui définit quelques types et quelques sous-programmes adaptés au Fortran. Ici, nous n'allons utiliser que le type `Fortran_Integer`, un type Ada entier compatible avec le type Fortran `integer`. Vient la déclaration de la procédure `Hanoi()`. Sa spécification est similaire à celle du programme Fortran. Seulement, elle n'aura pas d'implémentation : la ligne 8 est une directive `pragma` qui signale que cette procédure `Hanoi()` (deuxième paramètre) sera importée (`Import`) selon la convention d'appel `Fortran` (premier paramètre) depuis le nom `"hanoi_"` (troisième paramètre). Par défaut, le compilateur Fortran ajoute systématiquement un caractère de soulignement aux noms des symboles avant qu'ils ne soient traités par l'éditeur de liens, d'où sa présence dans la chaîne de caractères. Nous trouvons ensuite la procédure `Deplacer()`, cette fois-ci une « vraie » procédure Ada qui aura un corps. Mais pour qu'elle puisse être utilisée par le code Fortran, on la fait suivre d'une directive `pragma` similaire à la précédente, sauf que, cette fois, il s'agit d'exporter (`Export`) un point d'entrée. Remarquez le troisième paramètre, un caractère de soulignement a été ajouté au nom.

Pour être complet, voici le corps du paquetage, dans `a_hanoi.adb` :

```
 1 with Ada.Text_IO ; use Ada.Text_IO ;
 2 package body A_Hanoi is
 3    procedure Deplacer(depuis: in Fortran_Integer ;
 4                       vers  : in Fortran_Integer) is
 5    begin
 6       Put_Line(Fortran_Integer'Image(depuis) &
 7                " --> " &
 8                Fortran_Integer'Image(vers)) ;
 9    end Deplacer ;
10 end A_Hanoi ;
```

Reste à créer un petit programme pour tester tout cela :

```
1 with A_Hanoi ;
2 procedure Hanoi is
3 begin
4    A_Hanoi.Hanoi(3, 1, 2, 3) ;
5 end Hanoi ;
```

Difficile de faire plus élémentaire : ce programme se contente d'appeler la procédure `Hanoi()` du paquetage `A_Hanoi`, le code de celle-ci étant en réalité obtenu dans le source Fortran. Pour compiler l'ensemble, deux étapes sont nécessaires : d'abord compiler le code Fortran, puis compiler et lier le programme Ada. C'est-à-dire :

```
$ gfortran --free-form -c f_hanoi.f
$ gnatmake hanoi -largs f_hanoi.o -lgfortran
gcc -c hanoi.adb
gcc -c a_hanoi.adb
gnatbind -x hanoi.ali
gnatlink hanoi.ali f_hanoi.o -lgfortran
$ ./hanoi
 1 -->  3
 1 -->  2
 3 -->  2
 1 -->  3
 2 -->  1
 2 -->  3
 1 -->  3
```

Les paramètres qui suivent `-largs` sur la ligne `gnatmake` sont passés directement à l'éditeur de liens (ici `gnatlink`). Il nous suffit donc de lier le fichier objet résultant de la première commande aux fichiers objets du programme Ada. L'exécution prouve que tout cela fonctionne.

C

Les facilités pour connecter Ada et C sont plus développées que pour Fortran. Elles sont divisées en trois paquetages :

- `Interfaces.C` contient différents types compatibles avec les types de base du C, comme `int`, `float` ou `char_array` (pour les tableaux de caractères) ;
- `Interfaces.C.Strings` fournit un ensemble d'outils dédiés aux chaînes de caractères de type `char*` ;
- Enfin, `Interfaces.C.Pointers` (générique) apporte des opérations usuelles pour la manipulation des pointeurs.

Reprenons l'exemple précédent, mais cette fois le programme principal sera implémenté en C. Il fera appel aux sous-programmes exportés par un paquetage Ada, lequel fera lui-même appel à une fonction C pour l'affichage. Voici la spécification du paquetage en question :

```
 1 with Interfaces.C;
 2 with Interfaces.C.Strings;
 3 package Hanoi_A is
 4
 5    package C renames Interfaces.C;
 6    package CS renames Interfaces.C.Strings;
 7
 8    function Chars_To_Int(c_str: in CS.chars_ptr) return C.int;
 9    pragma Export(C, Chars_To_Int, "Chars_To_Int");
10
11    procedure Afficher_Depl(c_str: in CS.chars_ptr);
12    pragma Import(C, Afficher_Depl, "Afficher_Depl");
13
14    procedure Hanoi(depuis: in C.int;
15                    via   : in C.int;
16                    vers  : in C.int;
17                    nb    : in C.int);
18    pragma Export(C, Hanoi, "Hanoi");
19
20 end Hanoi_A;
```

Les lignes 5 et 6 sont un moyen commode et usuel d'éviter de devoir saisir de grands noms de paquetages, comme `Interfaces.C.Strings` : elles ne créent pas de nouveau paquetage, simplement un synonyme pour deux paquetages existants. Ainsi, par exemple ligne 8, l'écriture `CS.chars_ptr` est équivalente à `Interfaces.C.Strings.chars_ptr`.

Ce type, justement, est destiné à être l'équivalent du type C `char*`, autrement dit un pointeur sur une chaîne de caractères. Une chaîne est elle-même représentée par le type tableau non contraint `char_array` du paquetage `Interfaces.C`, tableau dont les éléments sont d'un type `char` correspondant au type `char` du C. Le type `Interfaces.C.int` est quant à lui garanti correspondre au type C `int` sur la plate-forme considérée. D'autres types sont également proposés.

On retrouve dans cette spécification les directives `pragma Import` et `Export` que nous avons déjà rencontrées pour le Fortran, sauf que, cette fois, la convention d'appel à utiliser est celle du langage C. La chaîne donnée en dernier paramètre est, comme précédemment, le nom utilisé pour l'édition des liens. Cette chaîne est facultative, mais il est vivement recommandé de la donner systématiquement. En particulier, n'oubliez pas que le langage Ada est insensible à la casse, contrairement au langage C : par défaut, le nom exporté pour la fonction `Chars_To_Int()` serait donc `"chars_to_int"`, créant ainsi une ambiguïté dans l'écriture.

Voyons maintenant le corps du paquetage :

```
 1 with Ada.Text_IO; use Ada.Text_IO;
 2 package body Hanoi_A is
 3
 4    use C;
 5
 6    function Chars_To_Int(c_str: in CS.chars_ptr) return C.int is
 7       s: constant String := CS.Value(c_str);
 8    begin
 9       Put_Line("Chars_To_Int");
10       return C.int(Integer'Value(s));
11    end Chars_To_Int;
12
13    procedure Hanoi(depuis: in C.int;
14                    via   : in C.int;
15                    vers  : in C.int;
16                    nb    : in C.int) is
17    begin
18       if nb = 1
19       then
20          declare
21             s: constant String := C.int'Image(depuis) &
22                                   " -> " &
23                                   C.int'Image(vers);
24             c_s: CS.chars_ptr := CS.New_String(s);
25          begin
26             Afficher_Depl(c_s);
```

```
27            CS.Free(c_s);
28         end;
29      else
30         Hanoi(depuis, vers, via, nb-1);
31         Hanoi(depuis, via, vers, 1);
32         Hanoi(via, depuis, vers, nb-1);
33      end if;
34   end Hanoi;
35
36 end Hanoi_A;
```

Considérons un instant la fonction `Chars_To_Int()`, dont le rôle est de convertir une chaîne de caractère reçu du C en un entier, comme le ferait `atoi()`.

L'attribut `'Value()` permet de faire précisément cela (ligne 10), mais il attend en paramètre une chaîne Ada : nous devons donc convertir la chaîne C en chaîne Ada. Le paquetage `Interfaces.C.Strings` propose pour cela la fonction `Value()` (ligne 7), dont le paramètre est de type `chars_ptr`.

À l'inverse, la procédure `Hanoi()` va utiliser la fonction `Afficher_Depl()` obtenue du C pour effectuer l'affichage. Les lignes 21 à 23 construisent la chaîne à afficher. Mais c'est une chaîne Ada, qui doit donc être convertie en chaîne C. Là encore, `Interfaces.C.Strings` propose la fonction `New_String()` (ligne 24), qui retourne un `chars_ptr`. Celui-ci peut être transmis au code C.

Notez toutefois que cette fonction crée une nouvelle chaîne en mémoire par une allocation dynamique : lorsque vous n'en avez plus besoin, vous devez donc libérer la mémoire à l'aide de la procédure `Free()` (ligne 27).

Voici enfin le programme principal en C :

```
 1 #include <stdio.h>
 2 extern void adainit();
 3 extern void adafinal();
 4 extern int Chars_To_Int(const char* s);
 5 extern void Hanoi(int depuis, int via, int vers, int nombre);
 6 void Afficher_Depl(const char* s);
 7 int main(int argc, char* argv[])
 8 {
 9    adainit();
10    int nb = Chars_To_Int(argv[1]);
11    Hanoi(1, 2, 3, nb);
12    adafinal();
13    return 0;
14 }
15 void Afficher_Depl(const char* s)
16 {
17    printf("%s\n", s);
18 }
```

Les deux premières déclarations, lignes 2 et 3, sont nécessaires pour que l'initialisation et la terminaison de la partie Ada du programme se passent correctement. `adainit()` doit être appelée avant tout autre appel au code Ada, tandis que `adafinal()` effectue divers nettoyages lorsque celui-ci n'est plus utilisé.

Suivent les déclarations des deux sous-programmes importés depuis le paquetage Ada, le plus naturellement du monde.

Pour compiler, plusieurs étapes sont nécessaires :

```
$ gcc -c hanoi_c.c
$ gnatmake -c hanoi_a.adb
$ gnatbind -n hanoi_a.ali
$ gnatlink hanoi_a.ali hanoi_c.o -o hanoi
```

Le rôle de `gnatbind` est de vérifier la cohérence du programme et de déterminer l'ordre d'initialisation des différents paquetages. Le fichier `.ali` passé en paramètre est issu de la compilation par `gnatmake`. Enfin, l'édition des liens est effectuée par `gnatlink`.

Exécutez le programme en lui passant en paramètre le nombre de disques :

```
$ ./hanoi 3
 1 ->  3
 1 ->  2
 3 ->  2
 1 ->  3
 2 ->  1
 2 ->  3
 1 ->  3
```

Tout se passe donc bien. Le meilleur exemple de connexion entre C et Ada est sans doute la bibliothèque GtkAda [1], qui permet d'utiliser l'intégralité de la célèbre bibliothèque Gtk+ (à la base de Gimp et Gnome) en Ada. GtkAda avait déjà été évoqué dans un article de Simon Descarpentries paru dans le *Linux Magazine 66* de novembre 2004.

Maintenant commencent les difficultés...

C++

Malheureusement, la communication avec le langage C++ n'est pas normalisée. La raison en est principalement que la représentation binaire des symboles, c'est-à-dire l'encodage des noms (*name mangling*), n'est elle-même pas normalisée, contrairement aux langages C et Fortran. Ainsi, chaque compilateur fait plus ou moins ce qu'il veut dans ce domaine, selon la plate-forme.

Bien souvent, lorsqu'un pont est jeté entre une bibliothèque C++ et Ada, cela se fait en « convertissant » le C++ en C. Autrement dit, chaque méthode de classe est remplacée par une fonction qualifiée par `extern "C"`, de façon à se ramener dans la situation de la section précédente. Cette approche présente toutefois un revers de taille : dans l'opération, on perd la sémantique objet. Les relations de dérivation entre les types disparaissent. Il devient alors assez difficile de créer de nouveaux types tout en bénéficiant du polymorphisme.

Le compilateur Gnat offre tout un ensemble de directives `pragma` adaptées au C++. Ainsi un type correspondant à une classe peut être qualifié par `pragma Cpp_Class`, une fonction par `pragma Cpp_Constructor` dans le cas d'un constructeur... Toutefois, d'une part, leur mise en œuvre est assez pénible (il faut utiliser les noms encodés, qui peuvent ressembler

à des choses comme `_ZNK7QString3midEjj` pour la simple méthode `mid()` de la classe `QString` de la bibliothèque Qt), d'autre part, ces directives ne sont pas portables sur d'autres compilateurs.

On peut, malgré tout, se débrouiller un peu, tout en conservant une certaine portabilité. L'idée consiste à créer une sorte de double passerelle, les codes Ada et C++ collaborant au travers d'une interface C. Considérons un exemple simple, deux classes dans un fichier `cpp_classes.h` :

```
#ifndef __CPP_CLASSES_H__
#define __CPP_CLASSES_H__ 1
class Base
{  public:
      Base() ;
      virtual ~Base();
      void method() const ;
      virtual void virtual_method() const ;
} ; // class Base
class Derived: public Base
{  public:
      Derived() ;
      virtual ~Derived();
      virtual void virtual_method() const ;
} ; // class Derived
#endif // __CPP_CLASSES_H__
```

Rien de bien méchant. Seule particularité, la méthode `method()` invoque la méthode virtuelle `virtual_method()`, redéfinie dans la classe dérivée `Derived`. Notre objectif est d'utiliser ces classes dans un programme Ada, tout en conservant leur relation d'héritage et la possibilité d'en dériver de nouveaux types, toujours en Ada.

Pour commencer, la manière de manipuler la mémoire n'est pas forcément identique entre un programme C/C++ et un programme Ada. Principe de base : les instances des classes C++ devront être créées dans un contexte C++. Le recours à la mémoire dynamique est presque obligatoire, le coté Ada ne stockant qu'une adresse mémoire. On va donc se créer un type Ada de base pour intégrer tout cela, dans un paquetage `Cpp_Accessors` :

```
    1 with System;
    2 with Ada.Finalization;
    3 package Cpp_Accessors is
    4    type Cpp_Ptr is new System.Address;
    5    Null_Cpp_Ptr: constant Cpp_Ptr := Cpp_Ptr(System.
Null_Address);
    6    type Ada_Ptr is new System.Address;
    7    Null_Ada_Ptr: constant Ada_Ptr := Ada_Ptr(System.
Null_Address);
    8    type Cpp_Accessor is new Ada.Finalization.Controlled with
private;
    9    not overriding
   10    procedure Create(ca : access Cpp_Accessor;
   11                     cpp: in     Cpp_Ptr);
   12    overriding
   13    procedure Finalize(ca: in out Cpp_Accessor);
   14    not overriding
   15    function Get_Cpp_Ptr(ca: in Cpp_Accessor'Class)
   16       return Cpp_Ptr;
   17 private
   18    type Cpp_Accessor is new Ada.Finalization.Controlled with
   19    record
   20       cpp: Cpp_Ptr := Null_Cpp_Ptr;
   21    end record;
   22 end Cpp_Accessors;
```

On fera ici l'hypothèse que le type `Address` du paquetage standard `System` est équivalent à un pointeur C++ comme `void*`. C'est généralement le cas, mais il paraît qu'il existe des exceptions.

Les deux premières déclarations des types `Cpp_Ptr` et `Ada_Ptr`, représentant respectivement l'adresse d'un objet C++ et l'adresse d'un objet Ada, ne sont là que pour économiser quelques touches et faire en sorte que le compilateur détecte d'éventuelles affectations hasardeuses. Le type intéressant est `Cpp_Accessor` (ligne 8), un type contrôlé afin d'en maîtriser surtout la destruction. Ce type sera chargé de « transporter » l'adresse d'un objet C++ : tous les autres types Ada correspondants à des classes C++ que nous allons déclarer dériveront de `Cpp_Accessor`. La procédure `Finalize()`, invoquée lorsqu'un objet de ce type est détruit, aura pour rôle de détruire l'objet C++ associé.

À ce type, on fait correspondre une classe C++, chargée cette fois de transporter l'adresse d'un objet Ada :

```
    1 #ifndef __ADA_ACCESSOR_H__
    2 #define __ADA_ACCESSOR_H__ 1
    3 class Ada_Accessor
    4 { public:
    5       Ada_Accessor(void* ada_ptr);
    6       virtual ~Ada_Accessor();
    7       void* Get_Ada_Ptr() const;
    8    private:
    9       Ada_Accessor(const Ada_Accessor&);
   10       Ada_Accessor& operator=(const Ada_Accessor&);
   11       void* _ada_ptr;
   12 }; // class Ada_Accessor
   13 extern "C" void Delete_Cpp(Ada_Accessor* aa);
   14 #endif // __ADA_ACCESSOR_H__
```

Remarquez la présence d'un destructeur virtuel (ligne 6) : sa présence est extrêmement importante pour la suite. La fonction `Delete_Cpp()`, ligne 13, contient la simple instruction `delete aa` (après vérification de la non-nullité du pointeur). Qualifiée `extern "C"`, elle pourra donc être invoquée depuis Ada :

```
    1 package body Cpp_Accessors is
    2    procedure Delete_Cpp(w: in Cpp_Ptr) ;
    3    pragma Import(C, Delete_Cpp, "Delete_Cpp") ;
...
   12    overriding
   13    procedure Finalize(ca: in out Cpp_Accessor) is
   14    begin
   15       Delete_Cpp(Get_Cpp_Ptr(ca));
   16    end Finalize;
...
   25 end Cpp_Accessors;
```

Ainsi la destruction d'un objet Ada dérivant de `Cpp_Accessor` entraînera automatiquement la destruction de l'objet C++ associé. Passons maintenant aux types Ada miroirs de nos classes C++ :

```
 1 with Cpp_Accessors;
 2 use  Cpp_Accessors;
 3 package Ada_Classes is
 4    type Base is new Cpp_Accessor with null record;
 5    not overriding
 6    procedure Create(b: access Base);
 7    not overriding
 8    procedure Virtual_Method(b: in Base);
 9    not overriding
10    procedure Method(b: in Base'Class);
11    --
12    type Derived is new Base with null record;
13    overriding
14    procedure Create(b: access Derived);
15    overriding
16    procedure Virtual_Method(b: in Derived);
17 end Ada_Classes;
```

Ces types sont créés à l'image des classes C++, dont elles reprennent les noms, jusqu'aux méthodes. Le type `Base` dérive de `Cpp_Accessor` comme annoncé, le type `Derived` (ligne 12) dérive de `Base` - reflet de la situation en C++. Pour établir le lien entre les deux langages, nous allons devoir créer une paire de classes et quelques fonctions, déclarées dans l'en-tête `binding.h` :

```
 1 #ifndef __BINDING_H__
 2 #define __BINDING_H__ 1
 3 #include "cpp_classes.h"
 4 #include "ada_accessor.h"
 5 class Base_Accessor: public Base, public Ada_Accessor
 6 {
 7    public:
 8       Base_Accessor(void* ada_ptr);
 9       virtual void virtual_method() const;
10 }; // class Base_Accessor
11 extern "C"
12 {
13    Base_Accessor* Base_Base(void* ada_ptr);
14    void Base_Method(Base_Accessor* ba);
15    void Base_Virtual_Method(Base_Accessor* ba);
16    void Dispatch_Base_Virtual_Method(void* ada_ptr);
17 }
18 class Derived_Accessor: public Derived, public Ada_Accessor
19 {
20    public:
21       Derived_Accessor(void* ada_ptr);
22       virtual void virtual_method() const;
23 }; // class Derived_Accessor
24 extern "C"
25 {
26    Derived_Accessor* Derived_Derived(void* ada_ptr);
27    void Derived_Virtual_Method(Derived_Accessor* ba);
28    void Dispatch_Derived_Virtual_Method(void* ada_ptr);
29 }
30 #endif // __BINDING_H__
```

À partir de chacune des classes que nous voudrons utiliser en Ada, on dérive une nouvelle classe fondée également sur `Ada_Accessor`, transporteur d'adresses d'objets Ada vue plus haut (lignes 5 à 10 et 18 à 23). Ainsi, par exemple, `Base_Accessor` dérive simultanément de `Base` et `Ada_Accessor`. Ces nouvelles classes surdéfinissent les méthodes virtuelles pour lesquelles elles vont constituer une interface. Par ailleurs, des fonctions C sont prévues pour accéder aux méthodes des classes (lignes 13 à 15 et 26-27). Les fonctions dont le nom commence par `Dispatch_` (lignes 16 et 28) ne sont en fait que des points d'accès à des fonctions exportées en Ada. Maintenant accrochez-vous, nous allons examiner simultanément le corps du paquetage `Ada_Classes` et l'implémentation associée à l'en-tête précédent (Tab. 1).

Ouf ! Le reste est très similaire. Revenons un instant sur le mécanisme en place et tous ses appels indirects. Supposons l'existence d'une instance Ada du type `Base`. Appelons-la `b_ada`. À celle-ci correspond une instance dynamique de `Base_Accessor`, que nous appellerons `b_cpp` (bien qu'il n'existe pas de variable coté C++). La figure 1 montre le déroulement des appels lorsque le code Ada invoque l'opération `Method()` sur `b_ada` (Tab. 2).

Vérifions tout cela sur un petit exemple. Créons un paquetage `Ext` dérivant un nouveau type à partir de `Base` :

Dérivation d'un type Ada à partir d'un type C++

```
with Ada_Classes; use Ada_Classes;
package Exts is
   type Ext_Base is new Base with null record;
   overriding
   procedure Virtual_Method(e: Ext_Base);
...
end Exts;
```

```
with Ada.Text_IO; use Ada.Text_IO;
package body Exts is
   overriding
procedure Virtual_Method(e: Ext_Base) is
   begin
      Put_Line("(Ada) Ext_Base.Virtual_Method");
   end Virtual_Method;
...
end Exts;
```

Tableau 3

Remarquez qu'il n'y a plus trace de toute la lourde mécanique précédente : tout cela n'est que du pur Ada. Voici le programme d'exemple :

```
 1 with Ada.Text_IO; use Ada.Text_IO;
 2 with Ada_Classes; use Ada_Classes;
 3 with Exts; use Exts;
 4 procedure Test is
 5    p: access Base'Class;
 6 begin
 7    Put_Line("...Pointeur sur Base...");
 8    p := new Base;
 9    p.Create;
10    p.Method;
11    Put_Line("...Pointeur sur Ext_Base...");
12    p := new Ext_Base;
13    p.Create;
14    p.Method;
15 end Test;
```

Établissement de la liaison entre le code Ada et le code C++

ada_classes.adb

```
 1 package body Ada_Classes is
 2    function Base_Base(ap: in Ada_Ptr)
 3       return Cpp_Ptr;
 4    pragma Import(C,
 5                  Base_Base,
 6                  "Base_Base");
 7    procedure Base_Method(cpp: in Cpp_Ptr);
 8    pragma Import(C,
 9                  Base_Method,
10                  "Base_Method");
11    procedure Base_Virtual_Method(cpp: in Cpp_Ptr);
12    pragma Import(C,
13                  Base_Virtual_Method,
14                  "Base_Virtual_Method");
```

binding.cpp

```
 1 #include <iostream>
 2 #include "binding.h"
 3 extern "C"
 4 {
 5    Base_Accessor* Base_Base(void* ada_ptr)
 6    {
 7       return new Base_Accessor(ada_ptr);
 8    }
 9    void Base_Method(Base_Accessor* ba)
10    {
11       if ( ba != 0 )
12          ba->method();
13    }
14    void Base_Virtual_Method(Base_Accessor* ba)
15    {
16       if ( ba != 0 )
17          ba->Base::virtual_method();
18    }
19 }
```

Les fonctions C sont importées depuis le code C++ par le code Ada. La première a pour rôle de créer une instance de `Base_Accessor`, en stockant l'adresse d'un objet Ada correspondant. L'adresse retournée sera stockée du coté Ada.

Les deux autres fonctions ne sont guère que des relais vers les méthodes de la classe à utiliser. Remarquez tout de même le cas de la méthode virtuelle : ce n'est pas celle (surdéfinie) de `Base_Accessor` qui est invoquée, mais celle de sa classe ancêtre `Base`.

```
23    not overriding
24    procedure Create(b: access Base) is
25       cpp_base: constant Cpp_Ptr := Base_Base(Ada_Ptr(b.
all'Address));
26    begin
27       Create(Cpp_Accessor(b.all)'Access, cpp_base);
28    end Create;
```

```
20 Base_Accessor::Base_Accessor(void* ada_ptr):
21    Base(),
22    Ada_Accessor(ada_ptr)
23 {
24 }
```

La première opération primitive du type Ada `Base` est la création d'une instance. Celle-ci est obtenue par la fonction C `Base_Base()`, qui ne fait que créer une instance (dynamique) de `Base_Accessor`. Le pointeur sur l'objet Ada est obtenu à partir du paramètre de la procédure, à l'aide de l'attribut `'Address`, puis passé à la fonction qui le transmet au constructeur.

À l'issu de la ligne (Ada) 25, on a donc une instance de `Base_Accessor` contenant l'adresse d'un objet Ada. Le pointeur retourné par la fonction `Base_Base()` est ensuite communiqué à la procédure `Create()` du type Ada de base `Cpp_Accessor` : à l'issu de la ligne 25, l'instance Ada de `Base` contient donc l'adresse de l'instance C++ de `Base_Accessor`. Chacune des deux instances « connaît » donc son alter ego. Passons sur la procédure Ada `Method()`. Elle ne fait qu'invoquer la fonction C `Base_Method()` en lui transmettant l'adresse préalablement stockée.

```
15    procedure Dispatch_Base_Virtual_Method(b: access Base'Class);
16    pragma Export(C,
17                  Dispatch_Base_Virtual_Method,
18                  "Dispatch_Base_Virtual_Method");
19    procedure Dispatch_Base_Virtual_Method(b: access Base'Class) is
20    begin
21       b.Virtual_Method;
22    end Dispatch_Base_Virtual_Method;
```

```
25 void Base_Accessor::virtual_method() const
26 {
27    std::cout << "(Cpp) Base_Accessor[" << this
28          << "]::virtual_method()\n";
29    Dispatch_Base_Virtual_Method(Get_Ada_Ptr());
30 }
```

Voyons maintenant le cas de la méthode virtuelle. Du coté Ada, on trouve la déclaration et l'implémentation de la procédure `Dispatch_Base_Virtual_Method()`. Son paramètre est un type classe (au sens Ada du terme, revoyez éventuellement l'article consacré à la programmation orientée objet en Ada pour plus de détails) : l'invocation de `Virtual_Method()` ligne 21 est donc un appel polymorphe, le code effectivement exécuté dépendra du type réel du paramètre au moment de l'exécution.

Par ailleurs, le mode de passage `access` est compatible (dans notre cas) avec un pointeur C : vous l'aurez compris, cette procédure sera invoquée du coté C++ avec en paramètre un pointeur sur un objet Ada, précédemment communiqué à `Base_Base()`. Cette procédure, déclarée dans l'en-tête ligne 16, est utilisée par la méthode virtuelle surdéfinie dans `Base_Accessor`. Lorsque cette méthode est invoquée, par exemple par une autre méthode de `Base`, l'appel est transmis au code Ada qui effectue un appel polymorphe (*dispatching*) de l'opération primitive `Virtual_Method()`.

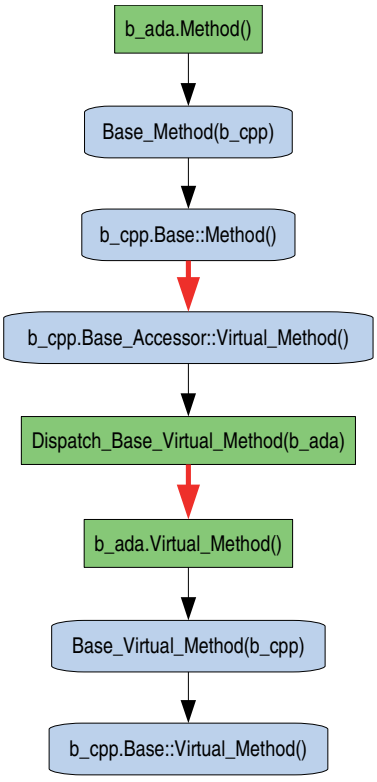
```
29    not overriding
30    procedure Virtual_Method(b: in Base) is
31    begin
32       Base_Virtual_Method(Get_Cpp_Ptr(b));
33    end Virtual_Method;
```

```
14    void Base_Virtual_Method(Base_Accessor* ba)
15    {
16       if ( ba != 0 )
17          ba->Base::virtual_method();
18    }
```

Cette opération primitive, dans son état « par défaut », invoque la fonction `Base_Virtual_Method()`, qui renvoit elle-même à la méthode virtuelle de la classe `Base`, non pas de la classe `Base_Accessor`, comme cela est indiqué explicitement ligne 17.

Tableau 1

Échanges polymorphes entre Ada et C++



Les portions de code en Ada sont sur fond vert, tandis que le code C++ est sur fond bleu. Les flèches rouges signalent les appels de fonction polymorphes.

Pour mémoire, la méthode `method()` de la classe (C++) `Base` invoque la méthode virtuelle `virtual_method()` de cette même classe. Il semble donc logique, voire rassurant, que dans notre exemple le fait d'appeler l'opération `Method()` sur le type Ada `Base` aboutisse finalement à la méthode `virtual_method()` de la classe C++ `Base`. Alors, pourquoi tout ce chemin ?

Rappelez-vous que nous cherchons à conserver la sémantique objet lors du passage en Ada. Cela signifie que si on dérive un type `Ext_Base` à partir de `Base`, et que l'on redéfinisse l'opération `Virtual_Method()`, il serait souhaitable que cela soit l'opération redéfinie qui soit appelée – et ce, que l'appel vienne du code Ada ou du code C++ initial. Le long cheminement mis en place a en quelque sorte pour effet de mettre en adéquation le polymorphisme C++ avec le polymorphisme Ada.

Dans la pratique, cela signifie que si on invoque l'opération `Method()` sur le type dérivé `Ext_Base()`, on aboutira bien dans l'opération `Virtual_Method()` surdéfinie, bien que l'on soit passé par le code C++.

On peut donc bien parler de « double passerelle » : les fonctions `extern "C"` permettent à Ada d'utiliser le code C++, tandis que le C++ s'appuie sur les procédures `Dispatch_*` pour la mise en œuvre du polymorphisme.

Tableau 2

La variable `p` est de type accès sur le type classe `Base` : c'est un peu comme si en C++ on déclarait un pointeur sur la classe `Base`. Cette variable peut donc être initialisée dynamiquement à n'importe quel type dérivant de `Base`.

Pour commencer, on lui affecte un objet de type `Base` (ligne 8), dûment créé, sur lequel on invoque l'opération (non virtuelle) `Method()`. On l'a vu, celle-ci doit normalement aboutir à la méthode virtuelle `virtual_method()` de la classe C++ `Base`.

Puis, on lui affecte un objet de type `Ext_Base`, dérivant de `Base` et redéfinissant la méthode virtuelle en Ada. Voici ce que cela donne :

```
...Pointeur sur Base...
(Cpp) Base[0x8070838]::virtual_method()
...Pointeur sur Ext_Base...
(Ada) Ext_Base.Virtual_Method
```

C'est bien la méthode redéfinie qui est invoquée, sans qu'il soit besoin d'ajouter de nouveau code C++.

Nous pouvons donc dériver de nouveaux types à partir des types importés du C++, tout en concernant le mécanisme du polymorphisme.

Compilation

Pour compiler un tel programme, le plus simple est de passer par un fichier décrivant le projet, une sorte de `Makefile` qui sera soumis à l'utilitaire `gprmake` fourni avec le compilateur Gnat. Voici un modèle, qui va compiler tous les fichiers C++ (d'extensions `.cpp`) et Ada du répertoire courant, puis assembler tous les fichiers objets obtenus lors de l'édition des liens :

```
 1 project Cpp is
 2    for Languages use ("C++", "Ada");
 3    for Main use ("test");
 4    package Naming is
 5       for Specification_Suffix ("C++") use ".h";
 6       for Implementation_Suffix ("C++") use ".cpp";
 7    end Naming;
 8    package Compiler is
 9       for Default_Switches("C++")
10          use("-Wall",
11              "-pedantic");
12       for Default_Switches("Ada")
13          use ("-Wall",
14               "-gnatwa",
15               "-gnatVa",
16               "-s");
17    end Compiler;
18    package Builder is
19    end Builder;
20 end Cpp;
```

Le nom du fichier doit être celui du projet (ici `Cpp`, ligne 1) et avoir l'extension `.gpr`. La compilation se fait simplement avec :

```
$ gprmake -P cpp.gpr
```

Ce qui fournira l'exécutable `test`. Consultez la documentation de Gnat pour plus de détails concernant le format de ces fichiers projets.

Perspectives

La technique évoquée plus haut ne résout pas tous les problèmes. Elle demande naturellement à être approfondie et renforcée, mais elle laisse entrevoir la possibilité de réaliser des passerelles en Ada vers de grandes bibliothèques C++, comme par exemple la bibliothèque graphique Qt4. Voici une bien modeste démonstration de ce qui n'est encore qu'un projet en devenir :

Et le code associé :

```
with Qt.Core.QStrings;
with Qt.Gui.QApplications;
with Qt.Gui.QWidgets.QLabels;
procedure Test_Qt is
   qapp: aliased Qt.Gui.QApplications.QApplication;
   ql  : aliased Qt.Gui.QWidgets.QLabels.QLabel;
   qs  : aliased Qt.Core.QStrings.QString;
begin
   Qt.Gui.QApplications.Create(qapp'Access);
   Qt.Core.QStrings.Create(
      qs'Access,
      "Un <i>binding</i> <b>Ada</b> pour <b>Qt4</b>");
   Qt.Gui.QWidgets.QLabels.Create(ql'Access, qs);
   Qt.Gui.QWidgets.Show(ql);
   Qt.Gui.QApplications.Exec;
end Test_Qt;
```

S'il existe des volontaires pour un coup de main, ce sera avec le plus grand plaisir !

Conclusion

Les possibilités offertes par l'ouverture du langage Ada sur d'autres langages, notamment le langage C, sont sans limites. Il est ainsi possible d'associer la puissance et la robustesse du langage Ada à la richesse et la diversité des bibliothèques existantes, sans devoir systématiquement réinventer la roue. La prochaine fois, nous nous pencherons sur les fonctionnalités offertes par le langage pour l'exécution de tâches en parallèle, ouvrant la voie à la réalisation de programmes multitâches sans avoir recours à des bibliothèques spécialisées.

Yves Bailly,

RÉFÉRENCES

- [1] GtkAda : https://libre2.adacore.com/GtkAda/main.html
- [2] Codes sources de l'article : http://www.kafka-fr.net/articles/ada/sources_13.tar.bz2