GNAT / C++ Low-Level Interface

Cyrille Comar, GNAT project (comar@cs.nyu.edu)

1. Introduction

This document presents a design for a low-level interface to C++ classes. The basic idea is to map a C++ class to a tagged type with specific characteristic in order to be compatible with the C++ semantics. Such a tagged type can in turn be used as the root of a hierarchy of regular tagged types. Inheritance and dispatching can then cross the language boundary from C++ to Ada and, why not, from Ada to C++. The obvious advantage of such an interface is to open the gigantic world of C++ libraries to Ada users.

The main goal of this proposal is to define the minimum set of pragmas and semantic processing that have to be added to the GNAT compiler for making such an interface not only possible but also fully functionnal. Our goal is not to make the process of interfacing C++ code as easy as possible to the user. In particular, in our model, we assume that the interfacer has some knowledge about the internals of the C++ compiler. The kind of interface we provide is more likely to be used by third-party tools specialized in interface binding generation than by direct users.

Every C++ compiler has a different implementation scheme and it is not possible to design a completely compiler independant interface, nevertheless we have triedis to avoid as much as possible to base the interface on the behavior of a particular C++ compiler and to isolate most compiler dependancies in a frun-time file that is possible to change without rebuilding a complete compiler. We also want to be able to partially interface a huge c++ library, that is to say,interface just the classes we are planning to use without needing to interface all their ancestors. Our last goal is to make it possible to interface a library using multiple inheritance even if such a concept is not available ion the Ada side..Some general assumptions are made on the way the C++ compiler works. For instance, we suppose that dispatching is implemented by means of a hidden pointer buried in any object pointing to a table containing the address of virtual members: the Vtable.

All the information GNAT needs to know about how a particular C++ compiler works is grouped in the package Interfaces.CPP. The body of this package can be customized in order to interface to a new compiler..Customizing this package requires precise information about the layout of the Vtable, and a scheme for implementing the membership test.

Interfacing a C++ library requires to know the layout of C++ class instances, including the position og thr Vtable pointer which appears explicitly in the equivalent Ada object. The user needs also to be able to provide the mangled names of member functions that are interfaced to.

2. Pragmas supporting the interface

2.1. CPP_Class

```
pragma CPP_Class (Entity => Typ);
```

meaning:

Typ is the Ada9x version of a c++ class. If typ is defined as a tagge type then it must contain one or more explicit vtable pointer. If typ is a non tagged record then it cannot contain a vtable pointer. Those types are considered limited from the Ada-side (except for initialization, see

CPP_Constructors for details) because '=' is not provided and assignment has a different semantics in the two languages. If necessary, a c++ user-defined assignement can be imported and used as a regular operation.

checks:

- typ is a type mark designating a record or a private type
- if Underlying_Type of 'typ' is tagged then at least one of its field is of type Interfaces.CPP.Vtable_Ptr
- if Underlying_Type of 'typ' is not tagged then no field has that type

Comment:

If the c++ class has no vtable pointer it cannot be associated to a tagged type, because it is not easy to solve the problem of where to put the vtable pointer in derived types. We could change this rule by specifying that if a cpp_class has no Vtable pointer, then its descendants must be cpp_class too. That is to say you can begin to derive regular tagged types only when a vtable pointer is present.

2.2. CPP_Vtable

```
pragma CPP_Vtable (
   Entity      => Typ,
   Vtable_Ptr  => Field_Name,
   Entry_Count => Static_Number);
```

meaning:

- allows dynamic dispatching trough the vtable pointer Field_Name
- the first CPP_Vtable pragma defines the vtable to be extended in further derivations
- Static_Number specify the number of entries in the Vtable. It is used to extend the the vtable during derivations

checks :

- Typ must be tagged and `CPP_Class`
- Field_Name must be of type `Interfaces.CPP.Vtable_Ptr`

comment:

if no `pragma CPP_Vtable` is defined on a type, the first component of type Interfaces.CPP.Vtable_Ptr is considered the only vtable pointer its Entry_Count is the number of primitive operations of the type.

2.3. Import

```
pragma Import (
  Convention    => CPP,
  Entity        => Subprogram_or_Variable,
  External_Name => Whatever,
  Link_Name     => "manually mangled name");
```

meaning:

- same as `Pragma_Import (C, ...)`

comment:

- the mangling has to be done manually through the Link_Name. (a future high-level interface should be able to mangle the External_Name automatically)
- It is possible to avoid finding out the mangled names of function members when they are only used in dispatching calls because the external name is not used in this case.

2.4. CPP_Virtual

```
pragma CPP_Virtual
  Entity     => Subprogram,
  Vtable_Ptr => Field_Name,
  Position   => Static_Number)
```

checks:

- Check that Field_Name is of type Interfaces.CPP.table_Ptr
- Check that the same offset is not used twice (directly or because of defaults)

meaning:

- provide necessary information for dispatching. `Component` is the vtable pointer component in the dispatching type of `Subprogram`. `Offset` is the position in the vtable of this subprogram.

comment:

- `Vtable_Ptr` and `Offset` optional. By default `Vtable_Ptr` is the component declared in the first pragma `CPP_Vtable`. and `Offset` is the position corresponding to the order of declaration of primitive operations.
- A non primitive operation must not have a CPP_Class parameter otherwise it would be taken for a virtual function with no pragma CPP_Virtual, it must instead have parameter of type typ'Class.
- A primitive operation with a CPP_Class parameter must have a pragma CPP_Virtual

2.5. CPP_Constructor

```
pragma CPP_Constructor (
  Entity    => Fname,
  Link_Name => "manually mangled name");
```

meaning:

- this pragma specifies a c++ constructor. Those constructors appear as regular functions to the Ada user althrough they an only be used to initialize objects. The initialization expression of objects whose type is CPP_Class must be a constructor call. If no init expression the default constructor is called.

checks:

- Fname must be the name of a function
- the function must have one of the following profiles:

```
function Fname                    return Typ'Class;   -- default constructor
function Fname (Ref: Typ'Class)   return Typ'Class;   -- copy constructor
function Fname (<parameters>)     return Typ'Class;   -- other constructor
```

- these functions can only be used as initialization expression of an object declaration

comments:

- the above rules make it impossible to create a CPP object in absence of constructors which seems a rather nice consequence.

2.6. CPP_Destructor

```
pragma CPP_Destructor  (Entity => Pname);
```

meaning:

- this pragma specifies a c++ destructor. This destructor is called automatically upon scope exit

checks:

- Pname must be the name of a procedure with the following profile:

```
procedure Pname (Obj : in out Typ);
```

3. An Example

In this example we have 3 C++ classes: Origin, A and B deriving one from the other. We want to interface A and B (we don't need Origin). From the Ada side we define 3 tagged types A, B and C. The first 2 corresponds to the C++ classes A and B and the third is a regular Ada derivation. We illustrate partial binding (origin is not interfaced as well as some of the primitives), dispatching and overriding across languages.

3.1. The C++ class library :

```
class Origin {
 public:
  int o_value;
};

class A : public Origin {
 public:
  void non_virtual (void);
  virtual void overridden (void);
  virtual void not_overridden (void);
  A();
  int   a_value;
};

class B : public A {
 public:
```

```
  virtual void overridden (void);
  B();
  int b_value;
};
```

3.2. The Ada interface

```
with Interfaces.CPP;
use  Interfaces.CPP;
package ex6_if is


   package A_Class is
      --
      --   Translation of C++ class A
      --

      type A is tagged
         record
            O_Value : Integer;
            A_Value : Integer;
            Vptr : Interfaces.CPP.Vtable_Ptr;
         end record;

      pragma CPP_Class (Entity => A);
      pragma CPP_Vtable (Entity => A, Vtable_Ptr => Vptr, Entry_Count => 2);

      --   Member Functions

      procedure Non_Virtual (This : in A'Class);
      pragma Import (CPP, Non_Virtual, "non_virtual", "_non_virtual__1A");

      procedure Overridden (This : in A);
      pragma CPP_Virtual (Entity      => Overridden,      --  long form
                          Vtable_Ptr  => Vptr,
                          Entry_Count => 1);
      pragma Import (CPP, Overridden, "", "_overridden__1A");

      procedure Not_Overridden (This : in A);
      pragma CPP_Virtual (Not_Overridden);                -- short form
      pragma Import (CPP, Not_Overridden, "", "_not_overridden__1A");

      function Constructor return A'Class;
      pragma CPP_Constructor (Entity => Constructor);
      pragma Import (CPP, Constructor, "A", "___1A");

   end A_Class;

   package B_Class is

      type B is new A_Class.A with
         record
            B_Value : Integer;
         end record;

      pragma CPP_Class (Entity => B);
```

```
      function Constructor return B'Class;
      pragma CPP_Constructor (Entity => Constructor);
      pragma Import (CPP, Constructor, "B", "___1B");

      procedure Overridden (This : in B);
      pragma CPP_Virtual (Overridden, Vptr, 1);
      pragma Import (CPP, Overridden, "overridden", "_overridden__1B");

   end B_Class;

   package Ada_Extension is

      type C is new B_Class.B with
         record
            C_Value : Integer := 3030;
         end record;

      --  no more pragma CPP_Class, CPP_Vtable; or CPP_Virtual:
      --  this is a regular Ada tagged type

      procedure Overridden (This : in C);

      procedure Not_Overridden (This : in C);
      pragma CPP_Virtual (Not_Overridden, Vptr, 2);
      pragma Import (CPP, Not_Overridden, "", "_not_overridden__1A");
   end Ada_Extension;
end Ex6_If;
```

3.3. an Ada main program

```
with ex6_if;
procedure ex6_main is

   use ex6_if;
   use ex6_if.A_Class;
   use ex6_if.B_Class;

   A_Obj : A_Class.A;
   B_Obj : B_Class.B;
   C_Obj : Ada_Extension.C;

   procedure Dispatch (Obj : A_Class.A'Class) is
   begin
      Overridden (Obj);
      Not_Overridden (Obj);
   end Dispatch;

begin
   Dispatch (A_Obj);
   Dispatch (B_Obj);
   Dispatch (C_Obj);
end ex6_main;
```

3.4. The bodies

```
with Gnat.IO; use Gnat.IO;
package body Ex6_If is
   package body Ada_Extension is

      procedure Overridden (This : in C) is
      begin
         Put ("in Ex6_If.Ada_Extension.Overridden, a_value = ");
         Put (This.A_Value);
         Put (", b_value = ");
         Put (This.B_Value);
         Put (", c_value = ");
         Put (This.C_Value);
         New_Line;
      end Overridden;

   end Ada_Extension;
end Ex6_If;

#include "ex6.h"
#include <fstream.h>

void A::non_virtual (void)
{
   cout << "in A::non_virtual, a_value = " << a_value << "\n";
}

void A::overridden(void)
{
   cout << "in A::overridden, a_value = " << a_value << "\n";
}

void A::not_overridden(void)
{
   cout << "in A::not_overridden, a_value = " << a_value << "\n";
}

A::A(void)
{
   a_value = 1010;
   cout << "in A::A, a_value = " << a_value << "\n";
}

void B::overridden (void)
{
   cout << "in B::overridden, a_value = " << a_value << "b_value = " << b_value
<< "\n";
}
B::B(void)
{
```

```
   b_value = 2020;
   cout << "in B::B, a_value = " << a_value << "b_value = " << b_value << "\n";
}
```

3.5. compilation and Execution

```
lang8{comar}84: /usr/local/gnu/bin/make ex6_main
gcc -c -g   ex6_main.adb
gcc -c -g   ex6_if.adb
/usr/local/bin/gcc -c -g ex6.C
gnatbl  ex6_main.ali -o ex6_main ex6.o -lg++

lang8{comar}80: ex6_main
in A::A, a_value = 1010
in A::A, a_value = 1010
in B::B, a_value = 1010b_value = 2020
in A::A, a_value = 1010
in B::B, a_value = 1010b_value = 2020
in A::overridden, a_value = 1010
in A::not_overridden, a_value = 1010
in B::overridden, a_value = 1010b_value = 2020
in A::not_overridden, a_value = 1010
in Ex6_If.Ada_Extension.Overridden, a_value = 1010, b_value = 2020, c_value = 3030
in A::not_overridden, a_value = 1010
```