Writing Ada Bindings for a C Library

Victor Porton
Shay Agnon 32-29, Ashkelon, Israel; email: porton@narod.ru

1. Abstract

I share my experience of writing thick (object oriented) Ada binding of a C library. This article provides some Ada tips and tricks (especially for Ada bindings writers).

Keywords: Ada bindings, thick bindings, C.

2. Introduction

We have a C library, written in object oriented style (C structure pointers serve as objects, and C functions taking such structure pointers serve as methods). However there is no inheritance in that C library (to make things easier for us).

The particular library is *Redland RDF Libraries*, a set of libraries which parses RDF files or other RDF resources and manages them, allows to do RDF queries, etc. Don't worry if you don't know what RDF is, it is not really relevant for this article. See more info about this C library in [4] and [1] and on RDF itself in [3].

I write *thick* Ada bindings for this library. “Thick” means that the API which I create is a full fledged Ada interface. For example, it uses Ada controlled tagged types to represent objects. (It also uses derived types and some other Ada features which are not available in C.)

By Ada here I will mean Ada2012, the latest currently available Ada standard.

This is a work in progress. Please write your comments. Don’t forget to notify me through porton@narod.ru (as I am *not* subscribed to this journal).

The source code of my library is available at https://github.com/vporton/redland-bindings (currently only in `ada2012` branch).

Note that the main purpose I created Ada bindings for Redland is to use them in this project: https://en.wikiversity.org/wiki/Automatic_transformation_of_XML_namespaces

3. Little things

One thing I learned during this project, is that Ada types should have different names, they shouldn’t have the same name even if they are in different packages. This both allows to shorten the code with use directives and to increase reliability of not passing a wrong type if a use directive is indeed used.

Initially I tried to use GCC with `-fdump-ada-spec` flag to autogenerate Ada specs from C headers. But shortly I realized that it will work better if I write Convention=>C subprograms manually (mainly because I sometimes want char_array and sometimes chars_ptr for a char* argument of a C function).

4. Packages structure

I put all my API into package hierarchy RDF.*.

The package RDF itself is empty:

```
package RDF is
  pragma Pure(RDF);
end RDF;
```

I also have RDF.Auxiliary package and its subpackages for “auxiliary” things (things used by or with my bindings, but not being bindings for a particular C library function).

I will discuss some particular RDF.Auxiliary.* packages below.

5. My tagged types

As I've said above, C objects are pointers to structures. All C pointers to structures have the same format and alignment [2]. This allows to represent any pointers to C structures as pointers to Dummy_Record as defined in RDF.Auxiliary package:

```
type Dummy_Record is null record
  with Convention=>C;
```

A new Ada type (call it T_Without_Finalize for the below explained reasons) corresponding to a dynamically allocated C record is created by instantiating generic packages RDF.Auxiliary.Handled_Record or RDF.Auxiliary.Limited_Handled_Record with a Convention=>C record type (can be Dummy_Record if record layout is considered internal by the C library documentation) and a Convention=>C access to this record and deriving our type from the tagged type Base_Object in the instantiated package.

Representing C structure pointers as tagged types is not memory efficient, but here we trade efficiency for programming ease.

It would be compelling to make Limited_Handled_Record a descendant type of Handled_Record, but it is impossible in Ada because Ada.Finalization.Limited_Controlled is not a descendant of Ada.Finalization.Controlled (as it probably should be). As such I wrote two similar packages RDF.Auxiliary.Limited_Handled_Record and RDF.Auxiliary.Handled_Record which duplicate mainly the same code. This is not perfect, but neither it is very bad, as the quantity of the code of these two packages (including their bodies) is not great.

6. About finalization and related stuff

The main challenge writing object oriented bindings for a C library is *finalization*.

In the C library in consideration (as well as in many other C libraries) every object is represented as a pointer to a dynamically allocated C structure.

The corresponding Ada object can be a (tagged) record holding the pointer (aka *handle*).

Ada object representing C objects should be descendants of Ada.Finalization.Limited_Controlled or Ada.Finalization.Controlled to be properly finalized when appropriate.

But oftentimes a C function returns so called “shared handle” that is a pointer to a C struct which we should not free because it is a part of a greater C object and shall be freed (by the C library) only when that greater C object goes away.

As such I first define a tagged type T_Without_Finalize type. For this type I define such procedures as Do_Finalize and Do_Adjust which do what Finalize and Adjust should do but leave Finalize and Adjust empty, so that a shared handle is neither finalized nor copied.

I define type T with Finalize and Adjust as a derived type. T could be defined as ancestor of both T_Without_Finalize and a type which defines Finalize and Adjust. But as Ada misses inheritance from multiple tagged types, I do it with generics instead (below is a partial listing):

```
generic
  type Base is new Base_Object with private;
package Common_Handlers is
  type User_Type is new Base with null record;
  overriding procedure Finalize(Object: in out User_Type)
                          renames Do_Finalize;
  overriding procedure Adjust(Object: in out User_Type)
                        renames Do_Adjust;
  type Base_With_Finalization is new User_Type
    with null record;
end;
```

The Base generic parameter is intended to be that T_Without_Finalize type.

You see that Do_Finalize and Do_Adjust become actual handlers of finalization and adjustment.

Note that I recommend to override the subprograms Finalize_Handle and Adjust_Handle (see the source) rather than Do_Finalize and Do_Adjust themselves.

Note that values of T_Without_Finalize type may become invalid (containing dangling access values). There seems that there is no easy enough way to deal with this problem (because of the way the C library works). Just be careful when using this library not to use objects which are already destroyed.

7. User defined types

Next thing to note that I first define User_Type. This type is intended to serve among other as a base for user-defined types which may contain not only the C handle but also other fields. The type Base_With_Finalization on the other hand is meant not to be a base for types with additional fields but contain only the handle (and null record extensions).

The reason why I make distinction between User_Type and Base_With_Finalization is the following:

We define some functions like

```
function From_Filename
  (World: Raptor_World_Type_Without_Finalize'Class;
   Filename: String)
   return IOStream_Type;
```

IOStream_Type is derived from Base_With_Finalization not from User_Type directly. If we derived our User_Type from IOStream_Type then non-null record extensions would cause (by Ada rules) the necessity to redefine From_Filename function also for the derived type what is a nonsense.

We actually use User_Type (in the private part of a package) like this (for an I/O stream reading from a string):

```
type Stream_From_String(Length: size_t) is
  new IOStream_Type_User with
  record
    Str: char_array(1..Length);
  end record;
```

8. Controlling vs class-wide arguments

Controlling and class-wide arguments differ mainly in their relationship with inheritance. But as there is no inheritance in the C library which we bind, we have certain freedom to choose either.

One disadvantage of class-wide types is that such things as that is makes necessary Get_Handle(null) to be type-qualified and thus the subprogram specifications longer.

One advantage of class-wide types is that I can use (as in query_results.ads) ST‘Class where ST is a subtype with a predicate to restrict to a subtypes matching a predicate.

Example:

```
subtype URI_Term_Type_Without_Finalize is
  Term_Type_Without_Finalize
   with Dynamic_Predicate =>
     Is_URI(URI_Term_Type_Without_Finalize);
```

It is possible that in a future version of the library I will consistently replace controlling arguments with class-wide arguments. This would make it more symmetric, as all tagged arguments would be class-wide and none special controlling one.

9. Dealing with callbacks

To deal with C callbacks (particularly accepting a void* argument for additional data) in object oriented way, we need a way to convert between C void pointers and accesses to Ada tagged (even class-wide) objects. (We pass Ada tagged objects as C “user data” pointers.)

When we create a callback we need to pass an Ada object as a C pointer and a Convention=>C subprogram defined by us as the callback. The callback receives the pointer previously passed by us and in the callback code we should (if we want to stay object oriented) convert this pointer into an Ada object access.

What we need is some bijective (“back and forth”) mapping between Ada access values and C pointers.

At first I was tempted to use Ada.Unchecked_Conversion. But (despite GNAT 7.2.0 gives no warning on this) it is not in any way guaranteed to work, because the format of Ada access type and of C pointer are not necessarily the same.

Now I do conversion this way:

I convert chars_ptr to a Convention=>C access to char then this to System.Address using System.Address_To_Access_Conversions and then (also by Address_To_Access_Conversions) address to the required access to a class-wide type.

The backward conversion is analogous.

The above should work if we understand the words “back and forth” RM13.7.2(5/2) “The To_Pointer and To_Address subprograms convert back and forth between values of types Object_Pointer and Address.” as that the conversion must be bijective. (I filed a clarification request about meaning of the words “back and forth” to Ada standardization committee.)

All this is implemented in RDF.Auxiliary.Convert_Void of my library, but in my opinion this package should be added to Ada standard packages.

How to do this in practice? The best way to explain is an example (for a user-defined I/O Stream which calls our function Do_Write_Bytes when “write” message is sent to it):

```
package My_Conv is
  new RDF.Auxiliary.Convert_Void
    (Handled_IOStream_Type_User'Class);
function raptor_iostream_write_bytes_impl
  (context: chars_ptr; ptr: chars_ptr; size, nmemb: size_t)
   return int
  with Convention=>C;
function raptor_iostream_write_bytes_impl
  (context: chars_ptr; ptr: chars_ptr; size, nmemb: size_t)
   return int is
begin
  declare
     Result: constant int := Do_Write_Bytes
(My_Conv.To_Access (context).all, ptr, size, nmemb);
  begin
     return Result;
  end;
exception
  when others =>
     return -1;
end;
```

10. Storage pools for memory allocation

I tried to define storage pools for C allocation/deallocation functions such as raptor_alloc_memory() and raptor_free_memory(), but it appeared to be impossible by the following reason:

System.Storage_Pools receives Alignment argument which is an integer multiple of the alignment of the allocated type. This alignment may be greater than the alignment raptor_alloc_memory() warrants (Dummy_Record’Alignment) and so lead to undefined behavior.

I have sent a proposal to the standardization committee to make the programmer able to restrict the maximum alignment.

Because using allocators appeared to be impossible, I did it instead this way (for Locator_Handle which is a pointer to Locator_Type record):

```
package Locator_Conv is
  new RDF.Auxiliary.Convert_Void(Locator_Type_Record);

function Copy_Locator (Handle: Locator_Handle)
  return Locator_Handle
is
  Size: constant size_t :=
size_t((Locator_Type'Max_Size_In_Storage_Elements *
Storage_Unit + (char'Size-1)) / char'Size);
  Result2: constant chars_ptr :=
    RDF.Raptor.Memory.raptor_alloc_memory(Size);
  Result: constant Locator_Handle :=
    Locator_Handle(Locator_Conv.To_Access(Result2));
begin
  Result.all := Handle.all;
  Result.URI := raptor_uri_copy(Handle.URI);
  Result.File :=
RDF.Raptor.Memory.Copy_C_String(Handle.File);
  return Result;
end;
```

Note that (Locator_Type'Max_Size_In_Storage_Elements * Storage_Unit + (char'Size-1)) / char'Size is the ceiling of floating point division of Locator_Type'Max_Size_In_Storage_Elements * Storage_Unit by char'Size (but without using floating point). Using ceiling warrants that the allocated space is at least as big as required space.

Here I allocate with raptor_alloc_memory() function the amount of memory which is max size needed (apparently not to overwrite nearby memory) for a record pointed by Locator_Type (ARM specifies this max size only for memory returned by an allocator, but I am pretty sure that in any reasonable implementation of Ada the same amount of memory will work well if it is allocated by raptor_alloc_memory() function instead and the nearby memory thus won’t be overwritten).

11. More little things

Ada standard misses a function converting a C string (with possible NULs) described by a chars_ptr and and its length in characters into an Ada String.

I define function Value_With_Possible_NULs which does this in terms of Interfaces.C.Pointers. Note that the pointer defined in suitably instantiated Interfaces.C.Pointers is correctly converted from/to chars_ptr with Ada.Unchecked_Conversion.

The Ada standard To_C with Trim_Nul=>False is broken: RM B.3(51) “If Append_Nul is False and Item'Length is 0, then To_C propagates Constraint_Error.” Said in another way the Standard means: “This does not work with empty strings.” So I wrote a wrapper My_To_C_Without_Nul around it.

I would write a lot more advice how to write Ada bindings for a C library, but you can just follow my source, which can serve as an example.

I “encode” values of C strings (which can be NULL) as an Ada indefinite holder holding a String. If the string is NULL, the holder is empty. However often it is enough to transform an empty Ada string into NULL C string (this can work only if we don’t differentiate between empty and null strings).

12. References

[1] Bootstrapping RDF applications with Redland. David Beckett, https://www.dajobe.org/papers/xtech2005/

[2] ISO/IEC 9899:2011 section 6.2.5 paragraph 28.

[3] Resource Description Framework (RDF). RDF Working Group, https://www.w3.org/RDF/

[4] The Design and Implementation of the Redland RDF Application Framework. David Beckett, 2001, http://www10.org/cdrom/papers/490/