Dear Ada

by Matthew Heaney
mailto:dearada@adapower.com

This issue marks the return of Dear Ada, a regular column whose purpose is to answer any and all Ada language questions, however simple or complex. Questions from those new to the language are especially encouraged. Correspondence may be directed to mailto:dearada@adapower.com.

Let us begin. Back in February, Steve (The Duck) asked the following question on comp.lang.ada:

```
I would like to create a function that returns a reference to a class-wide
object based on the external name.  Something along the lines of:

function Create_Object (External_Name : String) return Object_Access is
   Return_Tag : Ada.Tags.Tag;
begin
   Return_Tag := Ada.Tags.Internal_Tag (External_Name);
   return ??? some function of Return_Tag ???
end Create_Object;

Can this be done?
```

A function that returns a newly-created object is often referred to as a “factory function.” There is no predefined language mechanism for creating an object given just its external name, but we can build the necessary infrastructure pretty easily. The basic idea is to declare a factory function for each type in the class and use a table to bind each one to a unique name. We can implement Steve’s create subprogram by first looking up the factory associated with the requested name, and then invoking the function to create a new object.

To make all this work, we’ll need a suitable container type to use as the lookup table, and we’ll need to declare the table object somewhere. We also need to provide a way for types in the class to register their factory functions. The root of our subsystem looks like this:

```
with Ada.Tags;
package P is
   pragma Elaborate_Body;

   type T is tagged limited null record;

   type T_Class_Access is access all T'Class;
   for T_Class_Access'Storage_Size use 0;

   function Create (Key : Ada.Tags.Tag) return T_Class_Access;
private
   type Factory_Type is access function return T_Class_Access;
   procedure Register (Key : Ada.Tags.Tag; Factory : Factory_Type);
end P;
```

Here we've used the Elaborate_Body categorization pragma, so that the body of P elaborates immediately following elaboration of its spec. This ensures that the lookup table object (declared in the body of P) is fully elaborated prior to registration of the factory function for descendent types by child packages.

We have also set the size of the storage pool associated with type T_Class_Access to 0, so that no storage is associated with that access type. It is a good idea to always do this for access types (especially class-wide access types) from which no objects are actually allocated, to avoid dragging in unneeded run-time baggage.

The Create function uses type Ada.Tags.Tag as the key directly, to save clients the bother of having to convert from a tag to its external (string) name.

I have organized the type declarations such that derived types in the class are declared in child packages. Because the mechanism for supporting streaming is really an implementation detail, I have declared the register operation in the private part of the spec, so that it's only visible to children. Other techniques such as a private child function or child package would work too.

Next we need to implement the lookup table, which allows us to find the factory function pointer associated with an expanded tag value. The most suitable container type for our needs is a map, that binds one type (the "key") to some other arbitrary type (the "element"). It just so happens that the Charles library (written by, ahem, yours truly) comes with a map container optimized for keys that have type String, so we use that as the internal lookup table:

```
with Charles.Maps.Sorted.Strings.Unbounded;
pragma Elaborate_All (Charles.Maps.Sorted.Strings.Unbounded);

package body P is
   package Map_Types is
     new Charles.Maps.Sorted.Strings.Unbounded (Factory_Type);

   Map : Map_Types.Container_Type;  -- the lookup table
   …
end P;
```

Note how the Elaborate_All pragma is used on the generic map package. Anytime you instantiate a package inside another package, you need to use this pragma to ensure all the packages on which the generic package depends have also been fully elaborated.

Now that we have a map, the register operation is simple: all we have to do is insert the tag/factory pair in the map:

```
procedure Register
  (Key     : Tag;
   Factory : Factory_Type) is
begin
   Insert (Map, Key => External_Tag (Key), New_Item => Factory);
end;
```

The Create operation works by first searching for the key/element pair whose key matches the tag's external form, and then invoking the factory function element associated with that key:

```
function Create (Key : Tag) return T_Class_Access is
   I : constant Iterator_Type := Find (Map, Key => External_Tag (Key));
begin
   if I = Back (Map) then -- tag not found
      raise Tag_Error;
   end if;

   return Element (I).all;  -- invoke factory
end Create;
```

Each type in the class has a factory function for creating objects of that type. For example, let's declare a type, NT, that derives from type T:

```
package P.C is
   pragma Elaborate_Body;
   type NT is new T with null record;
   ...
end P.C;
```

We declare the factory function for NT in the body. In order to automatically register the type, we can call Register in the begin part of the body, which executes during elaboration:

```
package body P.C is
   ...
   function New_NT return T_Class_Access is
      O : constant NT_Access := new NT;
   begin
      return T_Class_Access (O);
   end;
begin
   Register (NT'Tag, Factory_Type'(New_NT'Access));
end P.C;
```

Again, package P.C is declared using Elaborate_Body categorization. Its body elaborates immediately following elaboration of its spec, which ensures that type NT has already been registered prior to use by clients that depend on P.C.

To demonstrate that the Create operation really does allocate objects of the correct type, we can declare some objects and then print the expanded name:

```
declare
   O1 : constant T_Class_Access := Create (T'Tag);
   O2 : constant T_Class_Access := Create (NT'Tag);
begin
   Put ("O1="); Put (Expanded_Name (O1'Tag)); New_Line;
   Put ("O2="); Put (Expanded_Name (O2'Tag)); New_Line;
end;
```

When I run the program, it displays the following output:

```
O1=P.T
O2=P.C.NT
```

This confirms that the objects have the correct tags.

One of the features missing from the Ada95 streams facility is the ability to input an object whose type is limited. For example:

```
declare
   O : T'Class := T'Class'Input (Stream);
begin
```

This won't compile because assignment isn't available for limited types.

The default input stream function works by first reading the external tag out of the stream, and then, knowing the type, invoking a type-specific input operation. We can adapt our schema above to add support for streaming limited types, allowing us to create a constructor like:

```
declare
   O : constant T_Class_Access := Input (Stream);
begin
```

First we have to stream it out. The root package provides a class-wide output operation, that writes the external tag into the stream, and then writes the actual object:

```
procedure Output
  (Stream : access Root_Stream_Type'Class;
   Item   : in     T'Class) is

   I : constant Iterator_Type := Find (Map, Key => External_Tag (Item'Tag));
begin
   if I = Back (Map) then
      raise Tag_Error;
   end if;

   String'Output (Stream, Key (I));

   Write (Stream, Item);
end Output;
```

Although not strictly necessary for output, we check to make sure that the tag is in the map prior to manipulating the state of the stream. This is to ensure that input won't fail because the factory hasn't been registered. In general it's better to learn that there's a problem sooner rather than later.

Write is a primitive operation that handles the details of writing the object into the stream. Since Item above has type T'Class, and Write is primitive for the type, the Write call dispatches according to Item's tag. This is a very simple example of the "template method" pattern.

To illustrate how to actually stream out an object, I gave type T some state (here, an integer component), so its Write operation looks like this:

```
procedure Write
  (Stream : access Root_Stream_Type'Class;
   Item   : in     T) is
begin
   Integer'Write (Stream, Item.I);
end;
```

That takes care of writing. To implement reading, we first read the external tag out of the stream, and then use that as the key to look up the factory function for that type:

```
function Input (Stream : access Root_Stream_Type'Class)
   return T_Class_Access is

   Key : constant String := String'Input (Stream);

   I : constant Iterator_Type := Find (Map, Key);
begin
   if I = Back (Map) then  -- not found
      raise Tag_Error;
   end if;

   return Element (I) (Stream);  --invoke factory
end Input;
```

The factory function now accepts a stream object as a parameter, so that once the type has been identified (by reading its external tag), then the factory is called to finish the job of reading the object out of the stream.

The actual factory function is implemented by constructing an object of the type with a "default" value, and then calling Read as a post-initialization step to give the object its "stream" value. For type NT, it looks like this:

```
function New_NT (Stream : access Root_Stream_Type'Class)
   return T_Class_Access is

   O : constant NT_Access := new NT;
begin
   Read (Stream, O.all);
   return T_Class_Access (O);
end;
```

The Read operation is the analog of Write, consuming the stream elements that belong to this object. Because there are no discriminants for this type, we can immediately allocate an object and then read its state from the stream. If NT had discriminants, we would have to first stream in the discriminant values, then use them to create the object, and then stream in the remainder of its state.

In this example type NT extends its parent type T with a float component F:

```
procedure Read
  (Stream : access Root_Stream_Type'Class;
   Item   :    out NT) is
begin
   Read (Stream, T (Item));
   Float'Read (Stream, Item.F);
end;
```

The Read for type NT is implemented by calling the Read for type T – that way it only has to worry about streaming in the components of the extension (here, F). Note how Item is converted to type T: this is called a “view conversion.” It doesn’t construct a new object; rather, it creates a “T view” of the same object, so that we can call T’s version of Read.

To test this I created a file using Stream_IO, and then streamed out an instance of each type:

```
declare
   O1 : T;
   O2 : NT;
begin
   O1.I := 42;

   O2.I := 1776;
   O2.F := 3.141592654;

   Output (Stream (File), O1);
   Output (Stream (File), O2);
end;
```

I closed the file and then reopened it, and streamed in the two objects, like this:

```
declare
   X1 : constant T_Class_Access := Input (Stream (File));
   X2 : constant T_Class_Access := Input (Stream (File));
begin
   Put ("X1: "); Print (X1.all); New_Line;
   Put ("X2: "); Print (X2.all); New_Line;
end;
```

When I ran this program, the output was:

```
X1: Name=Q.T I=42
X2: Name=Q.C.NT I=1776 F=3.14159
```

The output and input operations thus behave as expected.

As you can see, the factory function pattern is very general, and here it allows us to effectively extend the language, to support streaming of limited types. Factory functions are very nearly ubiquitous, and so every programmer should be familiar with them.