The following contributions are taken from the AdaCore “Gem of the Week” series. Gems are expert tips and insights that will help you get the most out of Ada 2005. The full collection of gems can be found at http://www.adacore.com/home/ada_answers, where you may obtain the source code and post comments as well.

Gem #1: Limited Types in Ada 2005 — Limited Aggregates

Author: Bob Duff, AdaCore

Abstract: Ada Gem #1 — Ada 2005 allows construction of limited objects via aggregates, thus gaining the advantage of the full coverage rules, which was previously available only for non limited types.

Let’s get started…

One of my favorite features of Ada is the “full coverage rules” for aggregates. For example, suppose we have a record type:

```
type Person is
   record
      Name : Unbounded_String;
      Age : Years;
   end record;
```

We can create an object of the type using an aggregate:

```
X : constant Person :=
   (Name => To_Unbounded_String (”John Doe”),
    Age => 25);
```

The full coverage rules say that every component of Person must be accounted for in the aggregate. If we later modify type Person by adding a component:

```
type Person is
   record
      Name : Unbounded_String;
      Age : Natural;
      Shoe_Size : Positive;
   end record;
```

and we forget to modify X accordingly, the compiler will remind us. Case statements also have full coverage rules, which serve a similar purpose.

Of course, we can defeat the full coverage rules by using “others” (usually for array aggregates and case statements, but occasionally useful for record aggregates):

```
X : constant Person :=
   (Name => To_Unbounded_String (”John Doe”),
    others => 25);
```

According to the Ada RM “others” here means precisely the same thing as “Age | Shoe_Size”. But that’s wrong: what “others” really means is “all the other components, including the ones we might add next week or next year”. That means you shouldn’t use “others” unless you’re pretty sure it should apply to all the cases that haven’t been invented yet.

So far, this is old news — the full coverage rules have been aiding maintenance since Ada 83. So what does this have to do with Ada 2005?

Suppose we have a limited type:

```
   type Limited_Person is limited
      record
         Self : Limited_Person_Access := Limited_Person’Unchecked_Access;
         Name : Unbounded_String;
         Age : Natural;
         Shoe_Size : Positive;
      end record;
```

This type has a self-reference; it doesn’t make sense to copy objects, because Self would end up pointing to the wrong place. Therefore, we would like to make the type limited, to prevent programmers from accidentally making copies. After all, the type is probably private, so the client programmer might not be aware of the problem. We could also solve that problem with controlled types, but controlled types are expensive, and add unnecessary complexity if not needed.

In Ada 95, aggregates were illegal for limited types. Therefore, we would be faced with a difficult choice: Make the type limited, and initialize it like this:

```
   X : Limited_Person;
   X.Name := To_Unbounded_String ("John Doe");
   X.Age := 25;
```

which has the maintenance problem the full coverage rules are supposed to prevent. Or, make the type nonlimited, and gain the benefits of aggregates, but lose the ability to prevent copies.

In Ada 2005, an aggregate is allowed to be limited; we can say:

```
   X : aliased Limited_Person :=
      (Self => null, – Wrong!

       Name => To_Unbounded_String (”John Doe”),
       Age => 25,
       Shoe_Size => 10);
   X.Self := X’Access;
```

We’ll see what to do about that “Self => **null**” in a future gem.

One very important requirement should be noted: the implementation is required to build the value of X “in place”; it cannot construct the aggregate in a temporary variable and then copy it into X, because that would violate the whole point of limited objects — you can’t copy them.

Related Source Code

Gem #2: Limited Types in Ada 2005 — <> Notation in Aggregates

Author: Bob Duff, AdaCore

Abstract: Ada Gem #2 — The <> notation may be used in aggregates to request the default value for certain components.

Let’s get started…

Last week, we noted that Ada 2005 allows aggregates for limited types. Such an aggregate must be used to initialize some object (which includes parameter passing, where we are initializing the formal parameter). Limited aggregates are “built in place” in the object being initialized.

Here’s the example:

```
type Limited_Person is limited
   record
      Self : Limited_Person_Access := Limited_Person’Unchecked_Access;
      Name : Unbounded_String;
      Age : Natural;
      Shoe_Size : Positive;
   end record;

X : aliased Limited_Person :=
   (Self => null, -- Wrong!
    Name => To_Unbounded_String (”John Doe”),
    Age => 25,
    Shoe_Size => 10);
X.Self := X’Access;
```

It seems uncomfortable to set the value of Self to the wrong value (**null**) and then correct it. It also seems annoying that we have a (correct) default value for Self, but in Ada 95, we can’t use defaults with aggregates. Ada 2005 adds a new syntax in aggregates — “<>” means “use the default value, if any”.

Here, we can say:

```
X : aliased Limited_Person :=
   (Self => <>,
    Name => To_Unbounded_String (”John Doe”),
    Age => 25,
    Shoe_Size => 10);
```

The “Self => <>” means use the default value of Limited_Person’Unchecked_Access. Since Limited_Person appears inside the type declaration, it refers to the “current instance” of the type, which in this case is X. Thus, we are setting X.Self to be X’Unchecked_Access.

Note that using “<>” in an aggregate can be dangerous, because it can leave some components uninitialized. “<>” means “use the default value”. If the type of a component is scalar, and there is no record-component default, then there is no default value.

For example, if we have an aggregate of type String, like this:

```
Uninitialized_String_Const : constant String := (1..10 => <>);
```

we end up with a 10-character string all of whose characters are invalid values. Note that this is no more nor less dangerous than this:

```
Uninitialized_String_Var : String (1..10); – no initialization

Uninitialized_String_Const : constant String := Uninitialized_String_Var;
```

As always, one must be careful about uninitialized scalar objects.

Related Source Code

Ada Gems example files are distributed by AdaCore and may be used or modified for any purpose without restrictions.

Gem #3: Limited Types in Ada 2005 — Constructor Functions

Author: Bob Duff, AdaCore

Abstract: Ada Gem #3 — Constructor functions may be used to create objects of limited types. The result of such function calls is built "in place" in the actual object being created.

Let's get started…

Given that Ada 2005 allows build-in-place aggregates for limited types, the obvious next step is to allow such aggregates to be wrapped in an abstraction — namely, to return them from functions. After all, interesting types are usually private, and we need some way for clients to create and initialize objects.

```
package P is
   type T (<>) is limited private;
   function Make_T (Name : String) return T; -- constructor function

private
   type T is limited
      record
         Name : Unbounded_String;
         My_Task : Some_Task_Type;
         My_Prot : Some_Protected_Type;
      end record;
end P;

package body P is
   function Make_T (Name : String) return T is
   begin
      return (Name => To_Unbounded_String (Name), others => <>);
   end Make_T;
end P;
```

In Ada 95, constructor functions (that is, functions that create new objects and return them) are not allowed for limited types. Ada 2005 allows fully-general constructor functions. Given the above, clients can say:

```
My_T : T := Make_T (Name => "Bartholomew Cubbins");
```

As for aggregates, the result of Make_T is built in place (that is, in My_T), rather than being created and then copied into My_T. Adding another level of function call, we can do:

```
function Make_Rumplestiltskin return T is
begin
    return Make_T (Name => "Rumplestiltskin");
end Make_Rumplestiltskin;

Rumplestiltskin_Is_My_Name : constant T := Make_Rumplestiltskin;
```

It might help to understand the implementation model: In this case, Rumplestiltskin_Is_My_Name is allocated in the usual way (on the stack, presuming it is declared local to some subprogram). Its address is passed as an extra implicit parameter to Make_Rumplestiltskin, which then passes that same address on to Make_T, which then builds the aggregate in place at that address. Limited objects must never be copied! In this case, Make_T will initialize the Name component, and create the My_Task and My_Prot components, all directly in Rumplestiltskin_Is_My_Name.

Note that Rumplestiltskin_Is_My_Name is constant. In Ada 95, it is impossible to create a constant limited object, because there is no way to initialize it.

As in Ada 95, the "(<>)" on type T means that it has "unknown discriminants" from the point of view of the client. This is a trick that prevents clients from creating default-initialized objects (that is, "X : T;" is illegal). Thus clients *must* call Make_T whenever an object of type T is created, giving package P full control over initialization of objects.

Ideally, limited and nonlimited types should be just the same, except for the essential difference: you can't copy limited objects. Allowing functions and aggregates for limited types in Ada 2005 brings us very close to this goal. Some languages have a specific feature called "constructor". In Ada, a "constructor" is just a function that creates a new object. Except that in Ada 95, that only works for nonlimited types. For limited types, the only way to "construct" on declaration is via default values, which limits you to one constructor. And the only way to pass parameters to that construction is via discriminants. In Ada 2005, we can say:

```
   This_Set : Set := Empty_Set;
   That_Set : Set := Singleton_Set (Element => 42);
```

whether or not Set is limited. "This_Set : Set := Empty_Set;" seems clearer to me than:

```
   This_Set : Set;
```

which might mean "default-initialize to the empty set" or might mean "leave it uninitialized, and we'll initialize it in later".

Gem #5: Key-Based Searching In Set Containers

Author: Matthew Heaney (On2 Technologies)

Abstract: Ada Gem #5 — Sets are containers of elements. Like all containers, they are searchable, and given an element value you can find the element in the set equivalent to that value. For some applications, element equivalence is determined by just a part of the element (its “key” part), and it is often necessary to find an element given only a key value. A technique is presented here for key-based searching of elements in set containers.

Let’s get started…

This example demonstrates how to use hashed set containers.

We are given the task of implementing a game in which a user navigates among components in a maze. These map-sites comprise walls, rooms, and doors. One of the features of the game is the ability to find room objects given a room number, and we implement that feature using a set container.

The map-site objects in our simulation are members of a common class. We use a tagged type declared as follows:

```
  type Map_Site is abstract tagged limited null record;
```

The (abstract) type has a single (abstract) operation to provide navigation to a site object:

```
  procedure Enter (Site : in out Map_Site) is abstract;
```

Next we implement concrete type Room, which derives from Map_Site. The declaration of the Room type looks like this:

```
  type Room (<>) is limited new Map_Site with private;
```

The room type is limited and indefinite (because it has “unknown discriminants”), which means that objects of the type must be explicitly initialized by calling a constructor-style function, declared as follows:

```
  not overriding
  function New_Room (Number : Positive) return not null access Room;
```

There is no way to create a room object other than to call New_Room, which is where we insert new room objects into the set used to find instances. Now we turn to the package body and the instantiation of the set package. The generic actual type is just a simple named access type (the same one we use to allocate instances):

```
   type Room_Access is not null access Room;
```

To instantiate the (hashed) set package, we need both a hash function and an equivalence function. The hash function for sets is a mapping of element value to hash value. So how do we make a hash value from a room object? The natural choice is to use the room number as the hash value for a room object, so our hash function for rooms looks like this:

```
function Hash (R : Room_Access) return Hash_Type is
begin
   return Hash_Type (R.Number);
end;
```

For the equivalence function, we can just use the predefined equality operator for type Room_Access, since set elements are access values, and access values are unique for distinct room objects.

We now have everything we need to instantiate the generic hashed set container:

```
package Room_Sets is new Ada.Containers.Hashed_Sets
  (Element_Type        => Room_Access,
   Hash                => Hash,
   Equivalent_Elements => "=");

Room_Set : Room_Sets.Set;
```

We are finally in a position to implement the Room constructor function, which looks like this:

```
function New_Room (Number : Natural)
  return not null access Room
is
  R : constant Room_Access := new Room (Number);
begin
  Room_Set.Insert (R);
  return R;
end New_Room;
```

Now that we have implemented the necessary infrastructure for creating room objects, we have to implement the function for looking up room objects using the room number as the key. Our function is declared as follows:

```
function Find_Room (Number : Natural) return access Room;
```

Now comes the interesting part. The set records all Room objects that have been created, so if a room with that number exists, it will be in the set. Our problem is that we have no immediate way to search for room objects given a room number. Remember that this is a set of rooms (not a map with room number as the key) and the search function for the instantiation looks like this:

```
function Find (Container : Set; Item : Room_Access) return Cursor;
```

The issue is that we have a room number, not a room object, so how do we look up a room object according to its number?

The solution is to take advantange of the nested generic package Generic_Keys provided by the sets, which allows you to view a set element in terms of its key. It has generic formals very similar to the set package itself, except that the operations apply to a generic formal Key_Type instead of to the element type.

One requirement for using that package is that the generic actuals for keys must deliver the same values as the generic actuals for elements. For example, the function to return the hash value of a room number (the key) must return the same value as the function that returns the hash value of the room object (the element) with that room number. So we instantiate the nested package as follows:

```
function Get_Room_Number (R : Room_Access) return Natural is
begin
   return R.Number;
end;

function Room_Number_Hash (N : Natural) return Hash_Type is
begin
   return Hash_Type (N);
end;

package Room_Number_Keys is new Room_Sets.Generic_Keys
 (Key_Type        => Natural,
  Key             => Get_Room_Number,
  Hash            => Room_Number_Hash,
  Equivalent_Keys => "=");
```

We are now ready to implement the Find_Room function. The package Room_Number_Keys provides a key-based search function, so we call that to search the set for a room object with the given room number:

```
function Find_Room (Number : Natural) return access Room is
   C : constant Room_Sets.Cursor :=
         Room_Number_Keys.Find (Room_Set, Number);
begin
   if Has_Element (C) then
      return Element (C);
   else
      return null;
   end if;
end Find_Room;
```

The search function returns a cursor, and the cursor value indicates whether the search was successful. If a room object having that room number is in the set, Has_Element returns True and so we return the Room object designated by the cursor. If that number was not found (because no room object having that number was in the set), we simply return null.

Gem #6: The Ada95 Multiple Views Idiom vs. Ada05 Interfaces

Author: Matthew Heaney (On2 Technologies)

Abstract: Ada Gem #6 — The multiple views idiom is a technique that allows you to create the effect of deriving from multiple tagged types simultaneously. It is a powerful mechanism for composing abstractions that complements interface types.

Let's get started…

One of the major changes to the Ada language is the addition of interfaces, stateless types that specify a set of operations. Like a tagged type, you can derive from an interface type, but unlike a tagged type, you can derive from multiple interfaces simultaneously.

Typically you use an interface type as a kind of specification. An abstraction can be written in terms of an interface type, which makes the abstraction completely general, in that it can be used with any type that implements that interface.

Ada95 does not have interface types, so an obvious question is, Can you create the effect of deriving from multiple interface types, but in Ada95? The answer is yes, using a technique called the "multiple views" idiom. The technique makes it possible for a type to provide different views of itself, with each view having a different type. This is very much like having multiple interfaces, but you have to build the infrastructure yourself.

To illustrate the difference between interfaces and multiple views, we'll first design a simple abstraction for persistence in terms of an interface type, and then redesign it using the multiple views technique. An interface for persistence would look something like this:

```
package Persistence_Types2 is
   type Persistence_Type is limited interface;

   procedure Write
     (Persistence : in Persistence_Type;
      Stream      : not null access Root_Stream_Type'Class) is
abstract;
   … -- Read not shown here
end Persistence_Types2;
```

Any type that derives from Persistence_Type is saying that it can be saved out to (and loaded in from) some persistent medium. Deriving from the interface type is easy:

```
package P2 is
   type T is limited new Persistence_Type with null record;

   overriding
   procedure Write
     (Persistence : in T;
      Stream      : not null access Root_Stream_Type'Class);
   …
end P2;
```

Now all the application has to do is provide a persistence mechanism, that accepts an object that supports the persistence interface. Here's our abstraction for doing that:

```
package Persistence_IO2 is
   … -- Initialization details omitted here
   procedure Save (Persistence : in Persistence_Type'Class);
end Persistence_IO2;
```

This provides the infrastructure that allows any type (here, type T) that derives from Persistence_Type to be written to the persistence medium. A file-based implementation might look something like this:

```
package body Persistence_IO2 is
   File : Ada.Streams.Stream_IO.File_Type;
   …
   procedure Save
     (Persistence : in Persistence_Type'Class)
   is
   begin
      Persistence.Write (Stream (File));
   end Save;
end Persistence_IO2;
```

Now we did all this using an Ada05 interface type, but nothing we did strictly requires an interface type. You can achieve the same effect using a tagged type, which is how it would have been done in Ada95. The "interface" is just an abstract tagged null record:

```
package Persistence_Types is
   type Persistence_Type is abstract tagged limited null record;

   procedure Write
     (Persistence : in Persistence_Type;
      Stream      : access Root_Stream_Type'Class) is abstract;
   … -- Read omitted here
end Persistence_Types;
```

Here the Persistence_Type is a tagged type instead of an interface type, but it's otherwise the same. Even the Persistence_IO abstraction is the same as before. What is different is how the type is used to support persistence in some other type. That other type will provide a "persistence view" of itself. We wish for the type to support multiple views (just as it would were it implementing multiple interfaces), so it's not a simple matter of directly deriving from persistence type, since Ada does not support derivation from more than one tagged type (this is true of both Ada95 and Ada05).

What we will do to implement a persistence view is to create an intermediary type that derives from Persistence_Type, but with an access discriminant that designates the parent (the type that supports the view). This allows operations of the intermediary type to bind to the parent type, since the discriminant provides access to the parent's state. To see how this all works, let us first show what the public part of the parent type looks like:

```
package P is
   type T is limited private;

   type Persistence_Class_Access is
     access all Persistence_Type'Class;

   function Persistence (Object : access T)
     return Persistence_Class_Access;
   … others views would be declared here
private
   … -- see text below
end P;
```

Type T supports persistence by providing a Persistence function that returns an access value designating an instance of Persistence_Type. The "persistence view" of type T is obtained by invoking this accessor function. This mechanism can be extended any number of times, by providing multiple accessor functions that each return distinct views.

The Persistence function returns an access value that actually designates a component of record T. The type of the component is the intermediary type we mentioned earlier, that derives from Persistence_Type, declared like this:

```
private
   type Persistence_View (Object : access T) is
     new Persistence_Type with null record;  -- no state req'd here

   procedure Write
     (Persistence : in Persistence_View;
      Stream      : access Root_Stream_Type'Class);
```

Note that the Persistence_View type is a null extension (it doesn't require any state of its own), but with an access discriminant that designates the parent type T. This allows the implementation of operation Write to see the representation of the parent type, since the Persistence parameter has an access discriminant that designates the T instance, and so the operation has access to all of the state that needs to be written to persistent storage.

The parent type T is implemented by declaring an aliased component of the intermediary type:

```
   type T is limited record
      Persistence : aliased Persistence_View (T'Access);
      … -- rest of state here
   end record;
```

The Persistence component is aliased since function Persistence returns an access value designating that component:

```
function Persistence (Object : access T)
  return Persistence_Class_Access
is
begin
   return Object.Persistence'Access;
end;
```

The other difference between interface types and the multiple views idiom is how a client would actually invoke the operation to perform the persistence operation. In the interface case it's simple: type T derives from Persistence_Type directly, so instances of type T can be passed to any operation whose type is Persistence_Type'Class. The Persistence_IO package has just such an operation, so the call would look like this:

```
procedure Test_Persistence2 is
   Object : T;
begin
   …
   Persistence_IO2.Save (Object);
end;
```

You have to do a little more work in the multiple views case, since the conversion from T to Persistence_Type isn't automatic. Here we need to explicitly invoke the Persistence function to "convert" from type T to Persistence_Type'Class. The accessor function has an access parameter and so we must declare the instance of type T as aliased. The call looks like this:

```
procedure Test_Persistence is
   Object : aliased T;
begin
   …
   Persistence_IO.Save (Persistence (Object'Access).all);
end;
```

That's all there is to it. The multiple views idiom is actually a very powerful mechanism for composing abstractions. I have characterized the technique as an Ada95 idiom, but note that even in Ada05 it is occasionally useful, such as when you need to mix a tagged type into an existing hierarchy. The most common example is needing to add controlledness to a leaf type in a class, because the root type doesn't itself derive from controlled.

Related Source Code

Ada Gems example files are distributed by AdaCore and may be used or modified for any purpose without restrictions.

Gem #7: The Beauty of Numeric Literals in Ada

Author: Franco Gasperoni (AdaCore)

Abstract: Ada Gem #7 — Did you know that the marvels of Ada extend to integer and real numbers (numeric literals really). Read on to learn about this Ada Gem.

Let's get started…

On an unusually hot end-of-summer day some years ago, I stepped into the Courant Institute of Mathematical Sciences. It was my first day of class at New York University and my first day with Ada. The heat was pounding and the classroom air-conditioning unit was being repaired.

The breeze that day came from something simple and elegant: numeric literals in Ada. I was fortunate that my programming languages class started with something so cool. The first thing that struck me is the ability to use underscores to separate groups of digits. I always found

```
  3.14159_26535_89793_23846_26433_83279_50288_41971_69399_37510
```

more readable and less error prone to type than

```
      3.14159265358979323846264338327950288419716939937510
```

what do you think? (I wish underscores were allowed when typing one's credit card number on the internet.)

This is just a cool beginning. Let's talk about based literals. I never understood the rationale behind the strange and unintuitive method to specify based literals in the C programming language where you have only 3 possible bases: 8, 10, and 16 (why no base 2?). Furthermore, requiring that numbers in base 8 be preceded by a zero feels like a bad joke on us programmers (what values do 0210 and 210 represent in C?)

To my pleasant surprise that humid end-of-summer day, I learnt that Ada allows any base from 2 to 16 and that we can write the decimal number 136 in any one of the following notations

```
        2#1000_1000#     8#210#     10#136#     16#88#
```

Coming from a microcontroller background where my I/O devices were memory mapped I liked the ability to write:

```
        Lights_On  : constant := 2#1000_1000#;
        Lights_Off : constant := 2#0111_0111#;
```

and have the ability to turn on/off the lights as follows:

```
   Output_Devices := Output_Devices  or   Lights_On;
   Output_Devices := Output_Devices  and  Lights_Off;
```

Of course we can also use records with representation clauses to do the above, which is even more elegant, and I leave that to a future gem (any volunteers out there?).

Back to based literals. The notion of base in Ada allows for exponents. That is particularly pleasant. For instance we can write:

```
     Kilobinary  : constant := 2#1#e+10;
     Megabinary  : constant := 2#1#e+20;
     Gigabinary  : constant := 2#1#e+30;
     Terabinary  : constant := 2#1#e+40;
     Petabinary  : constant := 2#1#e+50;
     Exabinary   : constant := 2#1#e+60;
     Zettabinary : constant := 2#1#e+70;
     Yottabinary : constant := 2#1#e+80;
```

In based literals the exponent, like the base, uses the regular decimal notation and specifies the power of the base that the based literal should be multiplied with to obtain the final value. For instance `2#1#e+10 = 1 x` 2^{10} `= 1_024` (in base 10), whereas `16#F#e+2 = 15 x` 16^{2}`= 15 x 256 = 3_840` (in base 10).

Based numbers apply equally well to real literals. We can for instance write:

```
    One_Third : constant := 3#0.1#;  --  same as 1.0/3
```

Whether we write `3#0.1#` or `1.0/3`, or even `3#1.0#e-1`, Ada allows us to specify exactly rational numbers for which decimal literals cannot be written.

This brings us to the last nice feature of Ada for this gem. As Bob Duff would put it: Ada has an open-ended set of integer and real types.

As a result, numeric literals in Ada do not carry with them their type as in C. The actual type of the literal is determined from the context. This is particularly helpful in avoiding overflows, underflows, and loss of precision (think about 32l in C, which is very different from 321).

And this is not all: all constant computations done at compile time are done in infinite precision be they integer or real. This allows us to write constants with whatever size and precision without having to worry about overflow or underflow. We can for instance write:

```
           Zero : constant := 1.0 - 3.0 * One_Third;
```

and be guaranteed that constant Zero has indeed value zero. This is very different from writing:

```
One_Third_Approx : constant := 0.33333333333333333333333333333;
```

```
Zero_Approx      : constant := 1.0 - 3.0 * One_Third_Approx;
```

where `Zero_Approx` is really `1.0e-29` (and that will show up in your numerical computations.) The above is quite handy when we want to write fractions without any loss of precision. Along these same lines we can write:

```
   Big_Sum : constant := 1           +
                         Kilobinary  +
                         Megabinary  +
                         Gigabinary  +
                         Terabinary  +
                         Petabinary  +
                         Exabinary   +
                         Zettabinary;

   Result : constant := (Yottabinary - 1) / (Kilobinary - 1);
   Nil : constant := Result - Big_Sum;
```

and be guaranteed that Nil is equal to zero.

But I am getting carried away by the elegance of Ada numeric literals and almost forgot about the date of our next gem which will be on the `2#10_10#` of September (sorry I couldn’t resist :) .

Have a happy summer fellow developers.

Related Source Code

Ada Gems example files are distributed by AdaCore and may be used or modified for any purpose without restrictions.

Gem #8: Factory Functions

Author: Matthew Heaney (On2 Technologies)

Abstract: Ada Gem #8 — A factory function is a technique for constructing an object from one class given only an object in some other class. The technique is used to implement assignment for objects having a class-wide type, such that tag-mismatch exceptions cannot occur.

Let's get started…

Suppose we have a generic package that declares a stack class. The root of the hierarchy would be as follows:

```
generic
  type Element_Type is private;
package Stacks is
  type Stack is abstract tagged null record;

  procedure Push
   (Container : in out Stack;
    Item      : in     Element_Type) is abstract;
 …
end Stacks;
```

Assume there are various concrete types in the class, say an unbounded stack (that automatically grows as necessary) and a bounded stack (implemented as a fixed-size array).

Now suppose we want to assign one stack to another, irrespective of the specific stack type, something like this:

```
procedure Op (T : in out Stack'Class; S : Stack'Class) is
begin
   T := S;  -- raises exception if tags don't match
   …
end;
```

This compiles, but isn't very robust, since if the tag of the target stack doesn't match the tag of the source stack, then an exception will occur. Our goal here is to figure out how to assign stack objects (whose type is class-wide) in a manner such that the assignment is guaranteed to work without raising a tag-mismatch exception.

One way to do this is to make an assignment-style operation that is primitive for the type, so that it will dispatch according to the type of the target stack. If the type of the source stack is class-wide, then there can't be a tag mismatch (and hence no exception) since there's only one controlling parameter.

(Note that you could do it the other way too, by dispatching on the tag of the source stack. You could even make the operation class-wide, so that it doesn't need to dispatch at all. The idea is to avoid passing more than a single controlled operand.)

The assign operation would be declared like this:

```
   procedure Assign
     (Target : in out Stack;
      Source : Stack'Class) is abstract;
```

which would allow us to rewrite the above assignment statement as:

```
  procedure Op (T : in out Stack'Class; S : Stack'Class) is
  begin
     T.Assign (S);  -- dispatches according T's tag
     …
  end;
```

Each type in the class will have to override Assign. As an example, let's follow the steps the necessary to implement the operation for the bounded stack type. Its spec would look like this:

```
   generic
   package Stacks.Bounded_G is
     type Stack (Capacity : Natural) is
       new Stacks.Stack with private;

     procedure Assign
       (Target : in out Stack;
        Source : Stacks.Stack'Class);
     …
   private
     type Stack (Capacity : Natural) is
       new Stacks.Stack with
     record
        Elements  : Element_Array (1 .. Capacity);
        Top_Index : Natural := 0;
     end record;
   end Stacks.Bounded_G;
```

This is just a canonical implementation of a bounded container form that uses a discriminant to control how much storage for the object is allocated. The interesting part is implementing the Assign operation, since we need some way to iterate over items in the source stack. Here's a skeleton of the implementation:

```
   procedure Assign
     (Target : in out Stack;  -- bounded form
      Source : Stacks.Stack'Class)
   is
     …
   begin
      …
```

```
    for I in reverse 1 .. Source.Length loop
        Target.Elements (I) := <get curr elem of source>
        <move to next elem of source>
     end loop;
     …
  end Assign;
```

Note carefully that, assuming we visit items of the source stack in top-to-bottom order, it's not a simple matter of pushing items onto the target stack, since if we did that the items would end up in reverse order. That's the reason why we populate the target stack array in reverse, starting from largest index (the top of the stack) and working backwards (towards the bottom of the stack).

The question is, how do you iterate over the source stack? Assume that each specific type in the stack class has its own iterator type, matched to that stacks's particular representation (similar to how the containers in the standard library are implemented). The issue is that the type of the source stack formal parameter is class-wide. How do we get an iterator for the source stack actual parameter, if its specific type is not known (not known statically, that is)?

The answer is, just ask the stack for one! A tagged type has dispatching operations, some of which can be functions, so here we just need a dispatching function to return an iterator object. The idiom of dispatching on an object whose type is in one class, to return an object whose type is in another class, is called a "factory function" or "dispatching constructor."

An operation can only be primitive for one tagged type, so if the operation dispatches on the stack parameter then the function return type must be class-wide. We now introduce type Cursor, the root of the stack iterator hierarchy, and amend the stack class with a factory function for cursors:

```
  type Cursor is abstract tagged null record;  -- the iterator

  function Top_Cursor  -- the factory function
    (Container : not null access constant Stack)
    return Cursor'Class is abstract;

  … -- primitive ops for the Cursor class
```

Each type in the stack class will override Top_Cursor, to return a cursor that can be used to visit the items in that stack object. We can now complete our implementation of the Assign operation for bounded stacks as follows:

```
   procedure Assign
     (Target : in out Stack;
      Source : Stacks.Stack'Class)
   is
      C : Stacks.Cursor'Class := Source.Top_Cursor;  -- dispatches

   begin
      Target.Clear;

      for I in reverse 1 .. Source.Length loop
         Target.Elements (I) := C.Element;  -- dispatches
         C.Next;  -- dispatches
      end loop;

      Target.Top_Index := Source.Length;
   end Assign;
```

The Source parameter has a class-wide type, which means the call to Top_Cursor dispatches (since Top_Cursor is primitive for the type). This is exactly what we want, since different stack types will have different representations, and will therefore require different kinds of cursors. The cursor object (here, C) returned by the factory function is itself class-wide, which means that cursor operations also dispatch. The function call C.Element returns the element of Source at the current position of the cursor, and C.Next advances the cursor to the next position (towards the bottom of the stack).

Related Source Code

Ada Gems example files are distributed by AdaCore and may be used or modified for any purpose without restrictions.

Gem #9: Classwide Operations, Iterators, and Generic Algorithms

Author: Matthew Heaney (On2 Technologies)

Abstract: Ada Gem #9 — A generic algorithm manipulates container elements in a way that is completely general, working with any container (including arrays) that provides a way to iterate over its elements. In this example we systematically alter an operation for copying containers within a common class, converting it to a generic algorithm that ultimately works for any kind of container.

Let's get started…

In the last gem, we used a stack class to demonstrate factory functions (to construct iterator objects), and implemented an assignment operation that dispatched on the type of the target stack. We mentioned in passing that that operation could be implemented by dispatching on the source stack, so let's show how to do that.

We reorder the parameters so that the Source stack is first in the parameter list (so that it's the "distinguished receiver" of a prefix-style call), and change its type from classwide to specific. We also change the name of the operation from Assign to Copy, per convention. The new declaration is as follows:

```
procedure Copy
  (Source : Stack;
   Target : in out Stack'Class) is abstract;
```

In the earlier example, we had to populate the target stack in reverse, so that the elements would be in the correct order. We were able to do that because the operation was implemented by the specific type, and hence it had direct access to the representation of the (target) stack. Here the target type is classwide, so the only way to populate it is in forward order, using Push. That means we'll have to iterate over the source stack in reverse, so that the items are properly ordered in the target.

The bounded stack type is implemented as an array, so implementing Copy is easy because the bottom of the stack begins at the beginning of the array:

```
procedure Copy
  (Source : Stack;  -- bounded stack (array-based)
   Target : in out Stacks.Stack'Class)
is
begin
   Target.Clear;

   for I in 1 .. Source.Top_Index loop    -- from bottom to top
      Target.Push (Source.Elements (I));  -- Elements is the array
   end loop;
end Copy;
```

We also said in the earlier gem that the operation need not be primitive for the type. If we change the source stack's type to classwide, then the operation itself becomes classwide:

```
procedure Copy2  -- classwide op, not primitive
  (Source : Stack'Class;
   Target : in out Stack'Class);
```

If we make the type of the source stack classwide, then we'll need a different way to iterate over items of the source stack in reverse order, since we don't have access to its representation anymore.

To do that we'll amend the cursor type to include some additional operations. First we'll add a new factory function, to construct a cursor object that (initially) designates the element at the bottom of the stack:

```
function Bottom_Cursor
  (Container : not null access constant Stack)
  return Cursor'Class is abstract;
```

We'll also need an operation to move the cursor to the element that precedes the current item:

```
procedure Previous (Position : in out Cursor) is abstract;
```

That gives us everything we need to turn Copy into a classwide operation, so that it only needs to be implemented once:

```
procedure Copy2
  (Source : Stack'Class;
   Target : in out Stack'Class)
is
   C : Cursor'Class := Bottom_Cursor (Source'Access);

begin
   Target.Clear;

   while C.Has_Element loop
      Target.Push (C.Element);
      C.Previous;
   end loop;
end Copy2;
```

Note that we declared the classwide stack operation in the root package (see stacks.ads), but it could have just as easily been declared as a generic child procedure:

```
generic
procedure Stacks.Generic_Copy3
  (Source : Stack'Class;
   Target : in out Stack'Class);
```

Actually, we could move the operation out of the package hierarchy entirely:

```
with Stacks;
generic
   with package Stack_Types is new Stacks (<>);
   use Stack_Types;
procedure Generic_Stack_Copy4
  (Source : Stack'Class;
   Target : in out Stack'Class);
```

We can generalize this even more, such that the copy algorithm works for any kind of stack:

```
generic
   type Stack_Type (<>) is limited private;
   type Cursor_Type (<>) is private;
   type Element_Type (<>) is private;

   with function Bottom_Cursor
     (Stack : Stack_Type)
     return Cursor_Type is <>;
   with procedure Push
     (Stack : in out Stack_Type;
      Item  : Element_Type) is <>;
   with procedure Previous
     (Cursor : in out Cursor_Type) is <>;
   …
procedure Generic_Stack_Copy5
  (Source : Stack_Type;
   Target : in out Stack_Type);
```

This illustrates the difference between the dynamic polymorphism of tagged types and the static polymorphism of generics. There is no need for a stack class anymore (having a dedicated copy operation that works only for types in that class), since the generic algorithm works for any stack. (This is exactly how the standard container library is designed. Container types are tagged, but they are not members of a common class.)

Instantiating this operation on our stack type is easy, since the names of the generic actual operations match the names of the generic formal operations, so we don't need to specify them explicitly (since the generic formals are marked as accepting a <> default):

```
procedure Test_Copy5 (S : Stack) is
   procedure Copy5 is
      new Generic_Stack_Copy5
     (Stack,
      Cursor,
      Integer); -- default everything else

   T : Stack (S.Length);

begin
   Copy5 (Source => S, Target => T);
end;
```

But why stop there? We can write a generic copy algorithm for any kind of container. We just need to generalize iteration a little, to mean "visit these items in the way that makes sense for this source container," and generalizing insertion, to mean "add this element in the way that makes sense for this target container." The declaration would be:

```
generic
   type Container_Type (<>) is limited private;
   type Cursor_Type (<>) is private;
   type Element_Type (<>) is private;

   with function First
     (Container : Container_Type)
     return Cursor_Type is <>;
   with procedure Insert
     (Container : in out Container_Type;
      Item      : Element_Type) is <>;
   with procedure Advance
     (Cursor : in out Cursor_Type) is <>;

procedure Generic_Copy6
  (Source : Container_Type;
   Target : in out Container_Type);
```

We can instantiate this using our stack type, but note that the generic actuals no longer match the generic formals, so we need to specify them explicitly:

```
procedure Test_Copy6 (S : Stack) is
   procedure Copy6 is
      new Generic_Copy6
     (Stack,
      Cursor,
      Integer,
      First => Bottom_Cursor,
      Insert => Push,
      Advance => Previous);

   T : Stack (S.Length);

begin
   Copy6 (Source => S, Target => T);
end;
```

One assumption we've made here is that the source and target containers have the same type. Suppose we would like to copy the items in a stack to, say, an array. One approach would be to introduce another generic formal container type (a "source container" type that is distinct from the "target container" type), but there's another way. Consider the implementation of the copy algorithm:

```
procedure Generic_Copy6
  (Source : Container_Type;
   Target : in out Container_Type)
is
   C : Cursor_Type := First (Source);
begin
   Clear (Target);
   while Has_Element (C) loop
      Insert (Target, Element (C));
      Advance (C);
   end loop;
end Generic_Copy6;
```

Notice that the only thing we do with the source container is to use it to construct a cursor. If we pass in the cursor directly, that eliminates any mention of the source stack, which in turn allows the source and target containers to be different types. Our algorithm now becomes:

```
  generic
     type Container_Type (<>) is limited private;
     type Cursor_Type (<>) is private;
     type Element_Type (<>) is private;
     …
  procedure Generic_Copy7
    (Source : Cursor_Type;
     Target : in out Container_Type);
```

We can now copy from an integer stack to an array like this:

```
  procedure Copy_From_Stack_To_Array (S : in out Stack) is
    T : Integer_Array (1 .. S.Length);
    I : Positive := T'First;

    procedure Insert
      (Container : in out Integer_Array;
       Item      : Integer)
    is
    begin
       Container (I) := Item;
       I := I + 1;
    end;

    procedure Copy7 is
       new Generic_Copy7
      (Integer_Array,
       Cursor,
       Integer,
       Advance => Next);

 begin
    Copy7 (Source => S.Top_Cursor, Target => T);
 end Copy_From_Stack_To_Array;
```

The target “container” is just an array. The only special thing we need to do is synthesize an insertion operation, to pass as the generic actual. We can also use the same algorithm to go the other way, from an array to a stack:

```
procedure Copy_From_Array_To_Stack (S : Integer_Array) is
  T : Stack (S'Length);

  function Has_Element (I : Natural) return Boolean is
  begin
     return I > 0;
  end;

  function Element (I : Natural) return Integer is
  begin
     return S (I);
  end;

  procedure Advance (I : in out Natural) is
  begin
     I := I - 1;
  end;

  procedure Copy7 is
     new Generic_Copy7
    (Stack,
     Natural,
     Integer,
     Insert => Push);

begin
  Copy7 (Source => S'Last, Target => T);
end Copy_From_Array_To_Stack;
```

Now the source container is an array, and the "cursor" is just the array index (an integer subtype). We have the familiar problem of ensuring that the target stack is populated in the correct order. As before, we simply iterate over the array in reverse, by passing the index S'Last as the initial cursor value, and then "advancing" the cursor by decrementing the index value.

The algorithm can be generalized further still. In this final version, we eliminate the generic formal element type. That means we'll need to modify the generic formal Insert operation, by passing the source cursor as a parameter instead of the source element. The declaration of the generic algorithm now becomes:

```
generic
   type Container_Type (<>) is limited private;
   type Cursor_Type (<>) is private;

   with procedure Insert
     (Target : in out Container_Type;
      Source : Cursor_Type) is <>;
   …
procedure Generic_Copy8
  (Source : Cursor_Type;
   Target : in out Container_Type);
```

The algorithm is now agnostic about the mapping from cursor to element (since it doesn't even know about elements), which is more flexible, since it allows the client to choose whatever mechanism is the most efficient. To use the new algorithm, all we need to do is make a slight change to the generic actual Insert procedure, as follows:

```
  procedure Copy_From_Stack_To_Array (S : in out Stack) is
    T : Integer_Array (1 .. S.Length);
    I : Positive := T'First;

    procedure Insert
      (Target : in out Integer_Array;
       Source : Cursor)  -- now a cursor instead of an element
    is
    begin
       Target (I) := Element (Source);
       I := I + 1;
    end;

    procedure Copy8 is
       new Generic_Copy8
      (Integer_Array,
       Cursor,
       Advance => Next);

 begin
    Copy8 (Source => S.Top_Cursor, Target => T);
 end Copy_From_Stack_To_Array;
```

The basic idea is that a generic algorithm can be used over a wide range of containers (including array types). A cursor provides access to the elements in a container, but as we've seen, once you have a cursor then the container itself sort of disappears. From the point of view of a generic algorithm, a container is merely a sequence of items.

Related Source Code

Ada Gems example files are distributed by AdaCore and may be used or modified for any purpose without restrictions.