Z's space

Tuesday, September 18, 2012

AdaTutor - More Records and Types (2)

Tagged Records and Dynamic Dispatching

In Ada 95, a record may be **tagged**. This allows us later to derive a new type from the original and add fields. For example, if Day_Of_Week_Type is suitably declared along with Day_Subtype and Month_Type, Ada 95 lets us write

```
type Date is tagged record
   Day   : Day_Subtype;
   Month : Month_Type;
   Year  : Integer;
end record;

type Complete_Date is new Date with record
   Day_Of_Week : Day_Of_Week_Type;
end record;
```

Now objects of type Date have three fields: Day, Month, and Year, but objects of type Complete_Date have four fields: Day, Month, Year, and Day_Of_Week.

We can derive a new type without adding any fields by writing, for example,

```
type New_Date is new Date with null record;
```

If a tagged record is to be a private type, the declaration in the package specification contains the words **tagged private**, like this:

```
type Date is tagged private;
```

The actual structure of the record is declared in the private part of the package specification exactly as any other tagged record:

```
private
   ...
   type Date is tagged record
      Day   : Day_Subtype;
      Month : Month_Type;
      Year  : Integer;
   end record;
```

This is called a **private tagged type**.

Ada 95 also has **private extensions**. With the package specification segment shown below, the structure of type Date, with the fields Day, Month, and Year, is available outside the package. However, the fact that a Complete_Date contains a Day_Of_Week field is not:

```
   type Date is tagged record
      Day   : Day_Subtype;
      Month : Month_Type;
      Year  : Integer;
   end record;
   type Complete_Date is new Date with private;
   ...
private
   ...
```

```
   type Complete_Date is new Date with record
      Day_Of_Week : Day_Of_Week_Type;
   end record;
```

Ada 95 tagged types and Ada 95 hierarchical libraries both help support Object Oriented Design (OOD).

We can convert objects of a derived type to the parent type. For example, if we have **Cd : Complete_Date;** we can write **Date(Cd)**. This is useful in making software reusable. For example, suppose we have written a procedure to Display a Date:

```
procedure Display(D : in Date);
```

If we now wish to write a procedure to Display a Complete_Date, we could call on the procedure we've already written. If the above procedure and the associated types are defined in package P, we could write:

```
with Ada.Text_IO, P; use P;
procedure Display(Cd : in Complete_Date) is
   -- Call previous procedure to display the first 3 fields.
   Display(Date(Cd));
   -- Now display the fourth field.
   Ada.Text_IO.Put_Line(Day_Of_Week_Type'Image(Cd.Day_Of_Week));
end Display;
```

Thus the procedure to display a Date was reused when we wrote the procedure to display a Complete_Date.

Although type Date and the types derived from it are all different types, we can refer to all the types collectively as **Date'Class**. The attribute **'Class** can be applied to the name of any tagged type. If we declare an object to be of type **Date'Class**, we must initialize it so that the compiler will know the exact type of the object. The type of this object can't change at run time. For example, if we write **D : Date'Class := Date'(12, Dec, 1815);** we can't later write **D := Complete_Date'(4, Jul, 1776, Thu);**. This would change the type of the object D and make it grow larger by adding a new field, Day_Of_Week. This isn't permitted.

However, we can declare an access type to Date'Class without being specific about the exact type to be accessed:

```
type Ptr is access Date'Class;
```

Now objects of type Ptr can access objects of type Date at one time and objects of type Complete_Date at another. Also, we could build a linked list, with some of the links of type Date and some of type Complete_Date. This is called a **heterogeneous** linked list.

Let's suppose that we declare, in the specification for a package P, type Date (a tagged record), type Complete_Date derived from Date with an extra field, and two overloaded procedures Display, one to display a Date and one to display a Complete_Date. Suppose that the specification for P also includes the definition for type Ptr:

```
type Ptr is access Date'Class;
```

Now suppose that our program, which **with**s and **use**s P, declares an array A of two elements of type Ptr. Let A(1) access a Date and A(2) access a Complete_Date. The program to follow shows how the system will decide at **run time** which of two overloaded versions of Display to call, depending on the type of A(I).all:

```
package P is
   ...
   type Date is tagged record ...
   type Complete_Date is new Date with record ...
   type Ptr is access Date'Class;
   procedure Display(D : in Date);
   procedure Display(D : in Complete_Date);
end P;

with P; use P;
procedure Dynamic_Dispatching_Demo is
   A : array(1 .. 2) of Ptr;
begin
   A(1) := new Date'(12, Dec, 1815);
   A(2) := new Complete_Date'(4, Jul, 1776, Thu);
```

AdaTutor - Advanced Topics (7)
AdaTutor - Advanced Topics (6)
AdaTutor - Advanced Topics (5)
AdaTutor - Advanced Topics (4)
Ref: Change Your Forgotten Windows Password with t...
AdaTutor - Advanced Topics (3)
AdaTutor - Advanced Topics (2)
AdaTutor - Advanced Topics
AdaTutor - More Records and Types (4)
AdaTutor - More Records and Types (3)
AdaTutor - More Records and Types (2)
AdaTutor - More Records and Types
AdaTutor - Outside Assignment 6
AdaTutor - Generics and Tasking (2)
AdaTutor - Generics and Tasking
AdaTutor - Outside Assignment 5
AdaTutor - Exceptions, Ada.Text_IO (2)
AdaTutor - Exceptions, Ada.Text_IO
AdaTutor - Access Types, User Defined Types, and D...
AdaTutor - Access Types, User Defined Types, and D...
AdaTutor - Subprograms and Packages (3)
AdaTutor - Subprograms and Packages (2)

► August (26)
► July (6)
► June (11)

Labels

Ada AdaTutor tutorial Slackware Programming GtkAda GCC Linux C GNAT X11 computer git i3 nginx GPS Linus Torvalds OpenSSL Slackware64 TeX html5 Android CA GNU I18N LaTeX MacBook OpenVMS Patrick Volkerding UDP bash boot manager gnutls guacamole http headers rEFInd redshift regression slackbuilds.org slackpkg texia

Follow by Email

Email address... Submit

Followers

Followers (3)

Follow

About Me

Qunying

View my complete profile

```
   for I in A'Range loop
      -- Decides at run time which Display to call.
      Display(A(I).all);
   end loop;
end Dynamic_Dispatching_Demo;
```

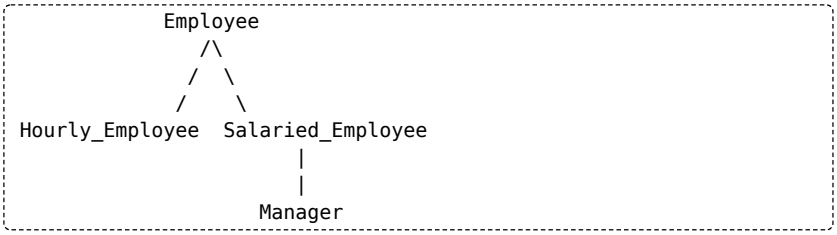
This is called **dynamic dispatching** or **tagged type dispatching**, and is available only in Ada 95, not in Ada 83, because Ada 83 doesn't have tagged records.

In Ada 95, the attributes **'Class** and **'Tag** can be used in tests. To illustrate this, we need a slightly more complicated example:

```
type Employee is tagged record
   Name : String(1 .. 5);
end record;

type Hourly_Employee is new Employee with record
   Hours_Per_Week : Integer;
end record;

type Salaried_Employee is new Employee with record
   Days_Vacation : Integer;
end record;

type Manager is new Salaried_Employee with record
   Num_Supervised : Integer;
end record;
```

Here we created a hierarchy of tagged types as follows:

```
type Employee ...
type Hourly_Employee is new Employee ...
type Salaried_Employee is new Employee ...
type Manager is new Salaried_Employee ...
```



Note that **Employee'Class** includes all four types above, while **Salaried_Employee'Class** includes only Salaried_Employee and Manager. The attribute **'Class** means the type named and all the types below it in the hierarchy.

Now let's declare an array of four elements that access objects of types in Employee'Class:

```
type P is access Employee'Class;
A : array(1 .. 4) of P;
```

Finally, let's let each element of our array access a different type in our hierarchy:

```
A(1) := new Employee'(Name => "Alice");
A(2) := new Hourly_Employee'(Name => "Bob  ",
                             Hours_Per_Week => 40);
A(3) := new Salaried_Employee'(Name => "Carol",
                               Days_Vacation => 10);
A(4) := new Manager'(Name => "Dave ", Days_Vacation => 5,
                     Num_Supervised => 3);
```

Then the test **A(1).all'Tag = A(2).all'Tag** will be False, because **A(1).all** and **A(2).all** have different types. The test **A(4).all in Salaried_Employee'Class** will be True, because type Manager is in Salaried_Employee'Class. Finally, the test **A(4).all in Hourly_Employee'Class** will be False, because Hourly_Employee'Class contains only type Hourly_Employee, not type Manager.

The test for tag equality or inequality (e.g., **A(1).all'Tag = A(2).all'Tag**) requires that we **with** and **use** Ada.Tags, a package that comes with Ada 95.

In Ada 95, the formal parameters ("dummy arguments") of a subprogram may also be of a class-wide type. For example, in a package that has defined both of our Display procedures and the associated types, we can write:

```
procedure Show(X : in Date'Class) is
begin
  Display(X);  -- Decides at run time which Display to call.
end Show;
```

It this case, the exact type of X is not known until run time, and can vary from one call of Show to the next. Therefore, the statement **Display(X);** will decide at run time which version of Display to call.

Related to this are **access parameters** in Ada 95. Instead of writing

```
procedure Show(X : in Date'Class);
```

we can write

```
procedure Show(X : access Date'Class);
```

In this case, Show is no longer called with an object belonging to Date'Class (that is, a Date or a Complete_Date), but rather with an object of any access type that accesses objects belonging to Date'Class. For example, we could now call Show with an object of the type Ptr we defined earlier:

```
D : Ptr := new Complete_Date(4, Jul, 1776, Thu);
Show(D);
```

Show again decides at run time which version of Display to call, but now Show must dereference the access value passed to it:

```
procedure Show(X : access Date'Class) is
begin
   Display(X.all);  -- Decides at run time which Display to call.
end Show;
```

The **accessibility check** which we discussed earlier when we talked about access types has to be performed at run time in the case of access parameters. The Ada 95 compiler automatically generates code to do this. For access types accessing named objects, discussed earlier, the accessibility check is performed at compile time. In both cases the purpose of the accessibility check is to insure that no object of an access type can ever access an object that no longer exists.

Here's our sample tagged type dispatching program, rewritten to make use of access parameters:

```
package P is
   ...
   type Date is tagged record ...
   type Complete_Date is new Date with record ...
   type Ptr is access Date'Class;
   procedure Display(D : in Date);
   procedure Display(D : in Complete_Date);
   -- Body calls Display(X.all);
   procedure Show(X : access Date'Class);
end P;

with P; use P;
procedure Dynamic_Dispatching_Demo is
   A : array(1 .. 2) of Ptr;
begin
   A(1) := new Date'(12, Dec, 1815);
   A(2) := new Complete_Date'(4, Jul, 1776, Thu);
   for I in A'Range loop
      Show(A(I));
   end loop;
end Dynamic_Dispatching_Demo;
```

Again, the **for** loop first displays a Date and then displays a Complete_Date.

< prev next >

Posted by Qunying at 21:54
Labels: Ada, AdaTutor, dynamic dispatching, record, tagged record, tutorial

No comments:

Post a Comment

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)