Safe and Secure Software

An Invitation to **Ada 2005**

6

Safe Object Construction

Courtesy of

AdaCore
The GNAT Pro Company

John Barnes

This chapter covers a number of aspects of the control of objects. By objects here we mean both small objects in the sense of simple constants and variables of an elementary type such as Integer and big objects in the sense of Object-Oriented Programming.

Ada provides good control and flexibility in this area. This control is in many cases optional but the good programmer will use the features wherever possible and the good manager will insist upon them being used wherever possible.

Variables and constants

As we have seen we can declare a variable or a constant by writing

```
Top: Integer;                        -- a variable
Max: constant Integer := 100;        -- a constant
```

respectively. Top is a variable and we can assign new values to it whereas Max is a constant and its value cannot be changed. Note that when we declare a constant we have to give it a value since we cannot assign to it afterwards. A variable can optionally be given an initial value as well.

The advantage of using a constant is that it cannot be changed accidentally. It is not only a useful safeguard but it helps any person later reading the program and informs them of its status. An important point is that the value of a constant does not have to be static – that is computed at compile time. An example was in the program for interest rates where we declared a constant called Factor

```
function Nfv_2000 (X: Float) return Float is
  Factor: constant Float := 1.0 + X/100.0;
begin
  return 1000.0 * Factor**2 + 500.0 * Factor – 2000.0;
end Nfv_2000;
```

Each call of the function Nfv_2000 has a different value for X and so a different value for Factor. But Factor is constant throughout each individual call. Although this is a trivial example and it is clear that Factor is not changed during execution of an individual call nevertheless we should get into the habit of writing **constant** whenever possible.

Parameters of subprograms are another example of variables and constants.

Parameters may have three modes: **in**, **in out**, and **out**. If no mode is shown then it is **in** by default. All parameters of functions must be of mode **in**.

A parameter of mode **in** is a constant whose value is given by the actual parameter. Thus the parameter X of Nfv_2000 has mode **in** and so is a constant –

this means that we cannot assign to it and so are assured that its value will not change. The actual parameter can be any expression of the type concerned.

Parameters of modes **in out** and **out** are variables. The actual parameter must also be a variable. The difference concerns their initial value. A parameter of mode **in out** is a variable whose initial value is given by that of the actual parameter whereas a parameter of mode **out** has no initial value (unless the type has a default value such as **null** in the case of an access type).

Examples of all three modes occur in the procedures Push and Pop in the chapter on Safe Architecture

```
procedure Push(S: in out Stack; X: in Float);
procedure Pop(S: in out Stack; X: out Float);
```

The rules regarding actual parameters ensure that constancy is never violated. Thus we could not pass a constant such as Factor to Pop since the relevant parameter of Pop has mode **out** and this would enable Pop to change Factor.

The distinction between variables and constants also applies to access types and objects. Thus if we have

```
type Int_Ptr is access all Integer;
K: aliased Integer;
KP: Int_Ptr := K'Access;
CKP: constant Int_Ptr := K'Access;
```

then the value of KP can be changed but the value of CKP cannot. This means that CKP will always refer to K. However, although we cannot make CKP refer to any other object we can use CKP to change the value in K by

```
CKP.all := 47;                  -- change value of K to 47
```

On the other hand we might have

```
type Const_Int_Ptr is access constant Integer;
J: aliased Integer;
JP: Const_Int_Ptr := J'Access;
CJP: constant Const_Int_Ptr := J'Access;
```

where the access type itself has **constant**. This means that we cannot change the value of the object J referred to indirectly whether we use JP or CJP. Note that JP can refer to different objects from time to time but CJP cannot. Of course, the value of the object J can always be changed by a direct assignment to J.

Constant and variable views

Sometimes it is convenient to enable a client to read a variable but not to write to it. In other words to give the client a constant view of a variable. This can be done with a so-called deferred constant and the access types just described.

A deferred constant is one declared in the visible part of a package and for which we do not give an initial value. The initial value must then be given in the private part. Consider the following

```
package P is
  type Const_Int_Ptr is access constant Integer;
  The_Ptr: constant Const_Int_Ptr;                -- deferred constant
private
  The_Variable: aliased Integer;
  The_Ptr: constant Const_Int_Ptr := The_Variable'Access;
  ...
end P;
```

The client can read the value of The_Variable indirectly through the object The_Ptr of type Const_Int_Ptr by writing

```
K := The_Ptr.all;          -- indirect read of The_Variable
```

But since the access type Const_Int_Ptr is declared as **access constant** the value of the object referred to by The_Ptr cannot be changed by writing

```
The_Ptr.all := K;          -- illegal, cannot change The_Variable indirectly
```

However, any subprogram declared in the package P can access The_Variable directly and so write to it. This technique is particularly useful with tables where the table is computed dynamically but we do not want the client to be able to change it.

The named access type is not really necessary since we can equally write

```
package P is
  The_Ptr: constant access constant Integer;       -- deferred constant
private
  The_Variable: aliased Integer;
  The_Ptr: constant access constant Integer := The_Variable'Access;
  ...
end P;
```

Note the double use of **constant** in the declaration of The_Ptr. The first says that The_Ptr is itself a constant. The second says that it cannot be used to change the value of the object that it refers to.

Limited types

The types we have met so far (Integer, Float, Date, Circle and so on) have various operations. Some are predefined, such as the equality operation to compare two values (with =) and some also have user-defined operations, such as Area in the case of the type Circle. The operation of assignment is also available for all the types mentioned so far.

Sometimes assignment is undesirable. There are two main reasons why this might be the case

- the type might represent some resource such as an access right and copying could imply a violation of security,
- the type might be implemented as a linked data structure and copying would simply copy the head of the structure and not all of it.

We can prevent assignment by declaring the type as **limited**. A good illustration of the second problem occurs if we implement the stack using a linked list. We might have

```
package Linked_Stacks is
  type Stack is limited private;
  procedure Clear(S: out Stack);
  procedure Push(S: in out Stack; X: in Float);
  procedure Pop(S: in out Stack; X: out Float);

private
  type Cell is
    record
      Next: access Cell;
      Value: Float;
    end record;

  type Stack is access all Cell;
end Stacks;
```

The body might be

```
package body Stacks is

  procedure Clear(S: out Stack) is
  begin
    S := null;
  end Clear;

  procedure Push(S: in out Stack; X: in Float) is
  begin
    S := new Cell'(S, X);
  end Push;
```

```
   procedure Pop(S: in out Stack; X: out Float) is
   begin
      X := S.Value;
      S := Stack(S.Next);
   end Pop;

end Stacks;
```

This uses the normal linked list style of implementation. Note that the type Stack is declared as limited private so that assignment of a stack as in

```
This_One, That_One: Stack;
...
This_One := That_One;          -- illegal, type Stack is limited
```

is prohibited. If assignment had been permitted then all that would have happened is that This_One would end up pointing to the start of the list defining the value of That_One. Calling Pop on This_One would simply move it down the chain representing That_One. This sort of problem is known as aliasing – we would have two ways of referring to the same entity and that is often very unwise.

In this example there is no problem with declaring a stack, it is automatically initialized to be null which represents an empty stack. However, sometimes we need to create an object with a specific initial value (necessary if it is a constant). We cannot do this by assigning in a general way as in

```
type T is limited ...
...
X: constant T := Y;        -- illegal, cannot copy value in variable Y
```

because this involves copying which is forbidden since the type is limited.

Two techniques are possible. One involves aggregates and the other uses functions. We will consider aggregates first. Suppose the type represents some sort of key with components giving the date of issue and the internal code number such as

```
type Key is limited
   record
      Issued: Date;
      Code: Integer;
   end record;
```

The type is limited so that keys cannot be copied. (They are a bit visible but we will come to that in a moment.) But we can write

```
K: Key := (Today, 27);
```

since, in the case of a limited type, this does not copy the value defined by the aggregate as a whole but rather the individual components are given the values Today and 27. In other words the value for K is built *in situ*.

It would be more realistic to make the type private and then of course we could not use an aggregate because the components would not be individually visible. Instead we can use a constructor function. Consider

```
package Key_Stuff is
  type Key is limited private;
  function Make_Key( ... ) return Key;
  ...
private
  type Key is limited
    record
      Issued: Date;
      Code: Integer;
    end record;
end Key_Stuff;

package body Key_Stuff is

  function Make_Key( ... ) return Key is
  begin
    return New_Key: Key do
      New_Key.Issued := Today;
      New_Key.Code := ... ;
    end return;
  end Make_Key;
  ...
end Key_Stuff;
```

The external client (for whom the type is private) can now write

```
My_Key: Key := Make_Key( ... );            -- no copying involved
```

where we assume that the parameters of Make_Key are used to compute the internal secret code.

It is worth carefully examining the function Make_Key. It has an extended return statement which starts by declaring the return object New_Key. When the result type is limited (as here) the return object is actually built in the final destination of the result of the call (such as the object My_Key). This is similar to the way in which the components of the aggregate were actually built *in situ* in the earlier example. So again no copying is involved.

The net outcome is that Ada provides a way of creating initial values for objects declared by clients and yet prevents the client from making copies. The limited type mechanism gives the provider of resources such as the keys considerable control over their use.

Controlled types

Ada provides a further mechanism for the safe management of objects through the use of controlled types. This enables us to write special code to be executed when

1) an object is created and,
2) when it ceases to exist and,
3) when it is copied if it is of a nonlimited type.

The mechanism is based on types called Controlled and Limited_Controlled declared in a predefined package thus

```
package Ada.Finalization is
  type Controlled is abstract tagged private;
  procedure Initialize(Object: in out Controlled) is null;
  procedure Adjust(Object: in out Controlled) is null;
  procedure Finalize(Object: in out Controlled) is null;

  type Limited_Controlled is abstract tagged limited private;
  procedure Initialize(Object: in out Limited_Controlled) is null;
  procedure Finalize(Object: in out Limited_Controlled) is null;
private
  ...
end Ada.Finalization;
```

The central idea (for a nonlimited type) is that the user declares a type which is derived from Controlled and then provides overriding declarations of the three procedures Initialize, Adjust and Finalize. These procedures are called when an object is created, when it is copied, and when it ceases to exist, respectively. Note carefully that these calls are inserted automatically by the system and the programmer does not have to write explicit calls. The same mechanism applies to a limited type which has to be derived from Limited_Controlled but there is no procedure Adjust since copying is not permitted. These operations are typically used to provide complex initializations, deep copying of linked structures, storage reclamation at the end of the lifetime of an object, and other housekeeping activities that are specific to the type.

As an example, suppose we reconsider the stack and decide that we want to use the linked mechanism (so there is effectively no upper bound to the capacity of the stack) but wish to allow copying one stack to another. We can write

```
package Linked_Stacks is
  type Stack is private;
  procedure Clear(S: out Stack);
  procedure Push(S: in out Stack; X: in Float);
  procedure Pop(S: in out Stack; X: out Float);
private
  type Cell is
    record
      Next: access Cell;
      Value: Float;
    end record;
  type Stack is new Controlled with
    record
      Header: access Cell;
    end record;
  overriding
  procedure Adjust(S: in out Stack);
end Linked_Stacks;
```

The type Stack is now just private. The full type shows that it is actually a tagged type derived from the type Controlled and has a component Header which effectively is the stack in the previous formulation. In other words we have introduced a wrapper. Note that the user cannot see that the type is controlled and tagged. Since we want to make assignment work properly we have to override the procedure Adjust. Note also that we have supplied the overriding indicator so that the compiler can double check that Adjust does indeed have the correct parameters.

The package body might be

```
package body Linked_Stacks is
  procedure Clear(S: out Stack) is
  begin
    S := (Controlled with Header => null);
  end Clear;
  procedure Push(S: in out Stack; X: in Float) is
  begin
    S.Header := new Cell'(S.Header, X);
  end Push;
  procedure Pop(S: in out Stack; X: out Float) is
  begin
    X := S.Header.Value;
```

```
     S.Header := S.Header.Next;
   end Pop;

   function Clone(L: access Cell) return access Cell is
   begin
     if L = null then
       return null;
     else
       return new Cell'(Clone(L.Next), L.Value);
     end if;
   end Clone;

   procedure Adjust(S: in out Stack) is
   begin
     S.Header := Clone(S.Header);
   end Adjust;

end Linked_Stacks;
```

Assignment will now work properly. Suppose we write

```
This_One, That_One: Stack;
...
This_One := That_One;          -- calls Adjust automatically
```

The raw assignment of That_One to This_One copies just the record containing the component Header. The procedure Adjust is then called automatically with This_One as parameter. Adjust calls the recursive function Clone which actually makes the copy. This process is often called a deep copy. The result is that This_One and That_One now contain the same elements but are otherwise disjoint structures.

Another notable point is that the procedure Clear sets the parameter S to a record whose header component is null; the structure is known as an extension aggregate. The first part of the extension aggregate just gives the name of the parent type (or the value of an object of that type) and the part after **with** gives the values of the additional components, if any. The procedures Pop and Push are straightforward.

The reader might wonder about reclamation of unused storage when Pop removes an item and also when Clear sets a stack to empty. This will be discussed in the next chapter when we consider memory management in general.

Note that Initialize and Finalize are not overridden and thus inherit the null procedure of the type Controlled. So nothing special happens when a stack is declared – this is correct since we just get a record whose Header is null by default and nothing else is required. Also nothing happens when an object of type Stack ceases to exist on exit from a procedure and so on – this again raises the issue of the reclamation of storage and will be addressed in the next chapter.