Safe and Secure Software

An Invitation to **Ada 2005**

8

Safe Startup

Courtesy of AdaCore
The GNAT Pro Company

John Barnes

We can carefully write a program so that it behaves properly when running, but it is all to no avail if it will not start properly.

The motor car that will not start is no good even if when going it behaves like a Rolls-Royce.

In the case of a computer program, the key things are to ensure that data is initialized properly and this often means to ensure that its various components are initialized in the correct order.

Elaboration

A program typically consists of a number of library packages P, Q, R and so on, plus a main subprogram M. The general idea is that when the program is started the various packages are elaborated, after which the main subprogram is called. The elaboration of a package consists of the creation of the various entities declared at the top level in the package – but not entities declared within subprograms in the package because these are created when the subprograms are called.

Thus consider again the package Stack in the chapter on Safe Architecture. In outline it was

```
package Stack is
   procedure Clear;
   procedure Push(X: Float);
   function Pop return Float;
end Stack;

package body Stack is

   Max: constant := 100;
   Top: Integer range 0 .. Max := 0;
   A: array (1 .. Max) of Float;

   ... -- procedures Clear and Push and function Pop

end Stack;
```

The elaboration of the specification of the package does nothing in this case because there are no objects declared in it. The elaboration of the body of the package notionally causes the space for the integer Top and the array A to be set aside. In this particular case the size of the array is known before the program executes because it is given by the constant Max which happens to have a static value and so the storage can be effectively set aside even before the program is loaded.

But Max need not have had a static value – it might have been given the result of some function call thus

```
Max: constant := Some_Function;
Top: Integer range 0 .. Max := 0;
A: array (1 .. Max) of Float;
```

and then the space required for A would be computed as part of the elaboration of the package body. If we had been careless and declared Max as a variable and forgotten to give it an initial value thus

```
Max: Integer;
Top: Integer range 0 .. Max := 0;
A: array (1 .. Max) of Float;
```

then the size of the array would be given by the value that Max happened to have. If Max were negative then the attempt to declare the array would raise Constraint_Error and if Max were too large than it might raise Storage_Error.

It should also be noted that we gave an initial value of zero to the variable Top so that the user did not have to call the procedure Clear before calling Push or Pop.

Alternatively we can give the package body an explicit initialization part so that it becomes

```
package body Stack is

  Max: constant := 100;
  Top: Integer range 0 .. Max;
  A: array (1 .. Max) of Float;

  ... -- procedures Clear and Push and function Pop

begin                              -- initialization part
  Top := 0;
end Stack;
```

The initialization part can contain any statements at all. It is executed as part of the elaboration of the package body and so before any of the subprograms in the package can be called by code outside the package.

Readers might feel that it is surely always best to give all variables an initial value anyway just in case. In the example given here the value zero is indeed a sensible initial value and corresponds to a call of Clear. In some situations there is no obvious initial value and giving a value just in case is not always wise because it can actually obscure real errors. We will come back to this briefly when we discuss SPARK in the final chapter.

In the case of numeric variables, the consequences of using a value that has not been set are not disastrous. But the consequence of using an access value or some other implicit address which has not been set could be. In the case of access types in Ada these either have a default value of **null** or must be initialized as we have seen.

A related kind of potential error concerns "access before elaboration". This means attempting to use something before it has been properly elaborated. Consider

```
package P is
  function F return Integer;
  X: Integer := F;                    -- raises Program_Error
end;
```

where the body of F is of course in the body of the package P. We cannot successfully call F to give an initial value to X before the body has been elaborated. So in this case the exception Program_Error is raised. The same sort of error in C could have unpredictable effects.

Elaboration pragmas

Within a single compilation unit the rule is that declarations are elaborated in the order in which they appear in the text.

In the case of a program linked from several different units, a unit is always elaborated after all those on which it depends. Thus a body is elaborated after the corresponding specification, the specification of a child is elaborated after the specification of its parent and any unit is elaborated after the specifications of all those mentioned in a (nonlimited) with clause.

However, this only partially dictates the order and is sometimes not enough to ensure the correct behavior of the program. We can extend the example above as follows

```
package P is
  function F return Integer;
end P;

package body P is
  function F return Integer is ...
end P;

with P;
package Q is
  X: Integer := P.F;
end;
```

It is important that the body of P has been elaborated before the specification of Q is elaborated because this elaboration requires that the body of F itself (and everything on which this body might in turn depend) be already elaborated. But the above rules do not ensure this and Program_Error might be raised at runtime.

We can force the required order of elaboration by inserting a pragma in the context clause for Q thus

```
with P;
pragma Elaborate_All (P);
package Q is
  X: Integer := P.F;
end;
```

Note that the All in Elaborate_All indicates the transitive nature of the pragma. Its effect is that at runtime the elaboration code for package P (and all the packages on which it depends) will be executed before the elaboration code for Q.

There is also a pragma Elaborate_Body which can be given with a specification and indicates that its body must be elaborated immediately after the specification.

Dynamic loading

A related topic concerns dynamic loading. Some languages are designed to create a single coherent program that is fully assembled before being run. Ada, C and Pascal are like that. The operating system may swap lumps of the program in and out of memory using paging algorithms but that is an implementation detail.

Other languages are designed to be much more dynamic and enable new code to be compiled, loaded and executed while the program is running. Cobol and Java are like that.

An approach used with programs written in languages such as C is to use dynamic linked libraries (DLLs) whereby an indirect call is used to invoke the new code. But this is not safe since there is no checking that the parameters of the new code match those of the old calling sequence.

One approach that can be used with Ada is to use the dispatching mechanism as the hook to dynamic linking. The point about dispatching is that it enables existing compiled code containing a class (such as Geometry.Object'Class) to call operations (such as Area) of further types (such as Pentagon, Hexagon and so on) without the central code having to be recompiled. This was briefly

mentioned in the chapter on Safe Object-Oriented Programming. Moreover the mechanism is completely type safe.

A good example of how dynamic linking can be added within this framework is given in [2].

North American Headquarters
104 Fifth Avenue, 15th floor
New York, NY 10011-6901, USA
tel +1 212 620 7300
fax +1 212 807 0162
sales@adacore.com
www.adacore.com

European Headquarters
46 rue d'Amsterdam
75009 Paris, France
tel +33 1 49 70 67 16
fax +33 1 49 70 05 52
sales@adacore.com
www.adacore.com