Elaboration

The execution of declarations

Examples

Elaboration of a variable declaration sets up space for that variable.
Elaboration of a task variable creates, initializes, and runs the task.

Elaboration happens each time a subprogram is called or a block is entered.

The elaboration of Library Units (e.g. packages) is a little more complicated.

All of the Library Units in a program are considered local to a task called the Environment Task whose execution is initiated by the operating system.

The Environment Task that elaborates all of the Library Units and calls the main procedure is constructed in a step called binding that occurs between compilation and linking.

The binder is free to select the order in which the Environment Task elaborates the Library Units. The only restriction is that the order must be consistent with the semantic dependencies (*with clauses*).

When the elaboration of a Library Unit results in the use of resources from another Library Unit, there is a potential for trouble.

```
package A is
   function Fun return Integer;
end A;
```

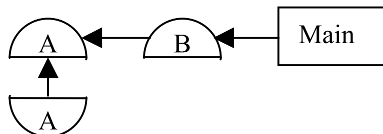
```
package body A is
   function Fun is
      return 5;
   end Fun;
end A;
```

```
with A;
package B is
   B_Con : constant Integer := A.Fun;
end B;
```

```
with B;
procedure Main is
begin
   null;
end;
```

What orders can we compile these 4 Units?

Here is a dependency diagram



One legal compilation order is

A spec
B spec
main program
A body

If the binder chooses to set up the elaboration of the units in the above order (semantically legal), we are in trouble!

The elaboration of B invokes a call to function Fun in package A whose body has not yet been elaborated.

Ada will raise the exception PROGRAM_ERROR!

However, we can determine at least one elaboration order which works for this sample code. The following elaboration order works:

A spec
A body
B spec
main program

Here is an example from the GNAT User's Manual for which there is no elaboration order that works all of the time.

```
package Unit_1 is
   function Func_1 return Integer;
end Unit_1;
```

```
with Unit_2;
with Calendar;
package body Unit_1 is
   Sqrt_1 : Float := Sqrt (0.1);
   Expression_1 : Integer :=
       Calendar.Day(Calendar.Time) rem 5;

   Q : Integer;

   function Funct_1 return Integer is
   begin
      -- use variable Sqrt_1 here
. . .
begin
   if Expression_1 = 1 then
      Q := Unit_2.Func_2;
   end if;
end Unit_1;
```

```
package Unit_2 is
   function Func_2 return Integer;
end Unit_2;
```

```
with Unit_1;
with Calendar;
package body Unit_2 is
   Sqrt_2 : Float := Sqrt (0.2);
   Expression_2 : Integer :=
       Calendar.Day(Calendar.Time) rem 4;

   Q : Integer;

   function Funct_2 return Integer is
   begin
      -- use variable Sqrt_2 here
. . .
begin
   if Expression_2 = 2 then
      Q := Unit_1.Func_1;
   end if;
end Unit_2;
```

Fortunately, this contrived example is not a common pattern. We can usually determine some elaboration order of packages that always works.

Ada was meant to be a safe language and the "Programmer Beware" approach to elaboration order is clearly not sufficient. Ada provides three lines of defense against elaboration order problems:

1. **Standard rules**

   If you *with* a unit, that unit's spec is elaborated before the unit doing the with (the client).

   A parent spec is always elaborated before a child spec.

2. **Dynamic elaboration checks**

   Checks are made at run time. If some entity is accessed before it is elaborated then the exception PROGRAM_ERROR is raised.

   A subprogram can only be called if it's body has been elaborated.

   A generic unit can only be instantiated if the body of the generic unit has been elaborated.

   Due to the execution expense, Ada does not check each global variable to see if it has been elaborated before accessing it.

   The idea is that if the subprogram body that accesses that global variable has been elaborated, than that variable has also.

   The GNAT compiler gives warnings for ***potential*** PROGRAM_ERRORs resulting from use of a ***possibly*** unelaborated resource.

3. **Elaboration control**

   Pragmas are provided for the programmer to specify the desired order of elaboration.

Java and C++ have these same elaboration problems. The Java/C++ programmers are responsible for solving the ordering problems themselves. Surprises come when a variable is accessed before it is initialized.

In his award winning study on comparing the use of Ada and Java in a train lab project, Eric Potratz discovered that elaboration problems resulted in many *static final "variables"* in his Java code not being initialized before use. (*A Practical Comparison Between Java and Ada in Implementing a Real-Time Embedded System*, ACM SIGAda 2003 Proceedings, p 71-83)

Use Of Pragmas To Control The Order Of Elaboration.

Type of Package Pragmas

There are two special types of packages that are guaranteed to have no elaboration order problems.

Pure

Has no state (no variables in package spec, no global variables in package body)
Has no tasks.
Has no protected objects
Executes no code when the package is elaborated.

Examples of *execution of code* (shown shaded) during elaboration of a unit in a non-pure package.

```
with A;
package body C is
   Con : constant Integer :=  A.Fun;
   Var : Integer;                -- Package has state so can't be pure
begin
   Var := 4 * Con;
end C;
```

A unit that is *pure* guarantees that no call to any subprogram in the unit can result in an elaboration problem. This means that the compiler does not need to worry about the point of elaboration of such units, and in particular, does not need to check any calls to any subprograms in this unit.

Preelaborable

Same as Pure, except may have state (may have package/global variables). Package body C above would be Preelaborable if it did not have the execution during elaboration of the unit.

A unit that is preelaborable also guarantees that no call to any subprogram in the unit can result in an elaboration problem. This means that the compiler does not need to worry about the point of elaboration of such units, and in particular, does not need to check any calls to any subprograms in this unit.

Standard Rules for Elaboration Order

The binder will always elaborate all the Pure packages before all the Preelaborable packages and all the Preelaborable packages before the remaining packages.

However, the compiler does not determine whether a package is pure or preelaborable.

We must tell it through pragmas in the package spec

```
package DAC is
pragma Pure(DAC);
```

```
package List is
pragma Preelaborate(List);
```

Elaboration Order Pragmas

A second group of pragmas gives us an even more direct control over the order of elaboration.

pragma Elaborate_Body

> This pragma requires that the body of a unit be elaborated immediately after its spec. We can use this pragma to solve the elaboration order problem given on page 1 of this handout.

```
package A is
pragma Elaborate_Body;          -- Elaborate the body right after the spec
   function Fun return integer;
end A;
```

```
with A;
package B is
   B_Con : constant Integer := A.Fun;
end B;
```

> Recall that the standard rules require the spec of unit A to be elaborated before the with'ing unit (B in this example) Given the pragma in A, we also know that the body of A will be elaborated before B, so that calls to A are safe and do not need a check.

Note that, unlike pragma Pure and pragma Preelaborate, the use of Elaborate_Body does not guarantee that the program is free of elaboration problems, because it may not be possible to satisfy the requested elaboration order. Let's go back to the example on page 2 with Unit_1 and Unit_2. If a programmer marks Unit_1 as Elaborate_Body, and not Unit_2, then the order of elaboration will be:

> Spec of Unit_2
> Spec of Unit_1
> Body of Unit_1
> Body of Unit_2

Now that means that the call to Func_1 in Unit_2 need not be checked, it must be safe. But the call to Func_2 in Unit_1 may still fail if Expression_1 is equal to 1, and the programmer must still take responsibility for this not being the case.

If all units carry a pragma Elaborate_Body, then all problems are eliminated, except for calls entirely within a body, which are in any case fully under programmer control. However, using the pragma everywhere is not always possible. In particular, for our Unit_1/Unit_2 example, if we marked both of them as having pragma Elaborate_Body, then clearly there would be no possible elaboration order.

The previous three pragmas (Pure, Preelaborate and Elaborate_Body) allow a server to guarantee safe use by clients, and clearly this is the preferable approach. Consequently a good rule in Ada 95 is to mark units as Pure or Preelaborate if possible, and if this is not possible, mark them all as Elaborate_Body if possible.

But as we have seen, there are situation where none of these three pragmas can be used. So Ada also provides methods for clients to control the order of elaboration of the servers on which they depend:

pragma Elaborate (unit)

This pragma is placed in the context clause, after a with statement, and it requires that the body of the named unit be elaborated before the unit in which the pragma occurs. The idea is to use this pragma if the current unit calls at elaboration time, directly or indirectly, some subprogram in the named unit.

pragma Elaborate_All (unit)

This is a stronger version of the Elaborate pragma. Consider the following example:

```
Unit A with's unit B and calls B.Func in elaboration code
Unit B with's unit C, and B.Func calls C.Func
```

Now if we put a pragma Elaborate (B) in unit A, this ensures that the body of B is elaborated before the call, but not the body of C, so the call to C.Func could still cause PROGRAM_ERROR to be raised. The effect of a pragma Elaborate_All is stronger, it requires not only that the body of the named unit be elaborated before the unit doing the with, but also the bodies of all units that the named unit uses, following with links transitively. For example, if we put a pragma Elaborate_All (B) in unit A, then it requires not only that the body of B be elaborated before A, but also the body of C, because B with's C.

Elaboration Usage Rule

We are now in a position to give a usage rule in Ada 95 for avoiding elaboration problems, at least if dynamic dispatching and access to subprogram values are not used. This document does not cover those cases.

The rule is simple.

> If a unit has elaboration code that can directly or indirectly make a call to a subprogram in a with'ed unit, or instantiate a generic unit in a with'ed unit, then if the with'ed unit does not have pragma Pure, Preelaborate, or Elaborate_Body, then the client should have an Elaborate_All for the with'ed unit.

By following this rule a client is assured that calls can be made without risk of an exception.

If we do not follow the rule, our program will be in one of four states:

No order exists

No order of elaboration exists which follows the rules, taking into account any Elaborate, Elaborate_All, or Elaborate_Body pragmas. In this case, an Ada 95 compiler must diagnose the situation at bind time, and refuse to build an executable program. We mentioned an example of this state earlier (page 5) where we said that adding Elaborate_Body pragmas to both Unit_1 and Unit_2 gave no possible elaboration order.

One or more orders exist, all incorrect

One or more acceptable elaboration orders exists, and all of them generate an elaboration order problem. In this case, the binder can build an executable program, but PROGRAM_ERROR will be raised when the program is run.

Several orders exist, some right, some incorrect

One or more acceptable elaboration orders exists, and some of them work, and some do not. The programmer has not controlled the order of elaboration, so the binder may or may not pick one of the correct orders, and the program may or may not raise an exception when it is run. This is the worst case, because it means that the program may fail when moved to another compiler, or even another version of the same compiler.

One or more orders exists, all correct

One ore more acceptable elaboration orders exist, and all of them work. In this case the program runs successfully. This state of affairs can be guaranteed by following the rule we gave above, but may be true even if the rule is not followed.

An additional advantage of following our Elaborate_All rule is that the program continues to stay in the ideal (all orders OK) state even if maintenance changes some bodies of some subprograms. Conversely, if a program that does not follow this rule happens to be safe at some point, this state of affairs may deteriorate silently as a result of maintenance changes.

Default Behavior with GNAT - Ensuring Safety

The GNAT compiler/binder/linker was set up to be as safe as possible. If it even thinks that the order of elaboration has a potential problem, it won't generate an executable.

The default behavior in GNAT ensures elaboration safety. In its default mode GNAT implements the rule we previously described as the right approach. Let's restate it:

> If a unit has elaboration code that can directly or indirectly make a call to a subprogram in a with'ed unit, or instantiate a generic unit in a with'ed unit, then if the with'ed unit does not have pragma Pure, Preelaborate, or Elaborate_Body, then the client should have an Elaborate_All for the with'ed unit.

In default mode GNAT traces all calls that are potentially made from elaboration code, and puts in any missing implicit Elaborate_All pragmas. The advantage of this approach is that no elaboration problems are possible if the binder can find an elaboration order that is consistent with these implicit Elaborate_All pragmas. The disadvantage of this approach is that no such order may exist.

If the binder does not generate any diagnostics, then it means that it has found an elaboration order that is guaranteed to be safe.

There are often times where it is impossible to follow our rule. The most common problems in our train projects come with circular package dependencies:

Package body Trains withs package Turnouts.
Package body Turnouts withs package Trains.

Note that there is no problem compiling this circular dependency.

GNAT's default behavior with regards to elaboration prevents us from producing an executable!

What to do if the Default GNAT Elaboration Behavior Fails

Fix the program

The most desirable option from the point of view of long-term maintenance is to rearrange the program so that the elaboration problems are avoided.

Make packages Pure or Preelaborable if possible.
Use pragma Elaborate Body when possible.
Use pragma Elaborate on all generic packages.

One useful technique is to place the initialization code (elaboration code) into separate child packages. Another is to move some of the initialization code to explicitly called initialization subprograms, where the program controls the order of initialization explicitly. (This is how Eric Potratz solved his Java elaboration problems).

Perform dynamic checks

If the compilations are done using the -gnatE (dynamic elaboration check) switch, then GNAT behaves in a quite different manner. Dynamic checks are generated for all calls that could possibly result in raising an exception. With this switch, the compiler does not generate implicit Elaborate_All pragmas. The behavior then is exactly as specified in the Ada 95 Reference Manual. The binder will generate an executable program that may or may not raise PROGRAM_ERROR, and then it is the programmer's job to ensure that it does not raise an exception. Note that it is important to compile all units with the switch, it cannot be used selectively.

Suppress checks

The drawback of dynamic checks is that they generate a significant overhead at run time, both in space and time. If you are absolutely sure that your program cannot raise any elaboration exceptions, then you can use the -f switch for the gnatbind step, or -bargs -f if you are using gnatmake. This switch tells the binder to ignore any *implicit* Elaborate_All pragmas that were generated by the compiler, and suppresses any circularity messages that they cause. The binder will not ignore any *explicit* elaborate pragmas added by the programmer.

The resulting executable will work properly if there are no elaboration problems, but if there are some order of elaboration problems they will not be detected, and unexpected results may occur.

Most of the information in this document was taken from the GNAT User's Guide. See this guide for additional details.