AI-00303 Removal of Library-Level Requirement for Interrupt Handler Objects

!standard C.3.1 (08) 04-12-02 AI95-00303/03

!standard C.3.1 (13)
!class binding interpretation 02-07-10
!status Amendment 200Y 04-12-02
!status ARG Approved 10-0-0 04-11-20
!status work item 02-07-10
!status received 02-06-21
!qualifier Error
!priority Low
!difficulty Medium
!subject Removal of library-level requirement for interrupt handler objects

!summary

There is no restriction on the location of objects of protected types containing Interrupt_Handler pragmas.

!question

The legality rule C.3.1 (8) says:

The Interrupt_Handler pragma is only allowed immediately within a protected_definition. The corresponding protected_type_declaration shall be a library level declaration. In addition, any object_declaration of such a type shall be a library level declaration.

However, this is a contract model violation. Consider:

```
package P is
   protected type Handler is
      procedure My_Handler;
      pragma Interrupt_Handler (My_Handler);
   ...
   end Handler;

   generic
     type L is limited private;
   package G is
     procedure Something;
     ...
   end G;
end P;

package body P is
   protected body Handler is ..
   end Handler;

   package body G is
     O1 : L;
     procedure Something is
       O2 : L;
     begin
       ...
     end Something;
```

```
   end G;

end P;

with P;
procedure Main is
   package HG is P.G (P.Handler); --???
begin
   ...
end Main;
```

Clearly, we want C.3.1 (8) to apply to O1 and O2. However, 12.3(11) says that legality rules are not enforced in the private part and body of instances. Thus, no check is made here.

For HG, we want both O2 and O1 to be illegal. However, if we moved the instantiation of HG to the library level, only O2 is illegal. Clearly the check depends both on the actual parameter and the location of the instantiation.

When and how is this check done? (We don't need the check.)

!recommendation

(See summary.)

!wording

Delete the last sentence of C.3.1 (8).

!discussion

The usual solution to contract model problems in generic bodies is to use an "assume-the-worst" rule. However, such a rule in this case would prevent declaring any objects of a limited private type. Such an incompatibility would be unacceptable, especially for such a rare case.

Another possible solution would be to prevent such protected types from matching generic limited private types. This seems very inconsistent with the rest of the language.

A third solution would be a run-time check in instance bodies. Such a check would be very similar to the accessibility check (see 3.10.2(28-29)). In most implementations, this check could be made at compile-time, but implementations using some form of generic sharing may have to make the check at runtime. But such a check would potentially have runtime overhead, and certainly would add complexity to compilers.

A better question is whether we need the rule at all. What is it trying to prevent?

* It prevents the need to support non-library level interrupt handlers. But pragma Attach_Handler already supports those. Besides, any attempt to attach a non-library level handler is going to fail the accessibility check when taking 'Access.

* It eliminates the need to define what happens when interrupt handlers are finalized. Yes, but we've already defined that behavior.

* It prevents trouble if an ordinary call is made to an interrupt handler routine. OK, but implementations are already given the permission to restrict ordinary calls on such routines (that's the purpose that the AARM gives for C.3.1 (17)).

Thus, it appears that there is no language need to enforce this rule. So, we drop the rule.

This has no impact on implementations at all, as they can continue to enforce the old rule as an implementation-defined restriction (as defined in C.3.1 (17)). Such restrictions can violate the contract model if the implementer wishes, while the language must not do that. If there is no problem with a nested object with an Interrupt_Handler pragma, the implementation can allow that, too. (Of course, such an object cannot actually handle interrupts because its handlers would fail the accessibility check.)

Removing this rule has no effect on the Ravenscar profile, as it already restricts all nested protected objects.

!corrigendum C.3.1 (8)

@drepl
The Interrupt_Handler pragma is only allowed immediately within a @fa<protected_definition>. The corresponding @fa<protected_type_declaration> shall be a library level declaration. In addition, any @fa<object_declaration> of such a type shall be a library level declaration.
@dby
The Interrupt_Handler pragma is only allowed immediately within a @fa<protected_definition>. The corresponding @fa<protected_type_declaration> shall be a library level declaration.

!ACATS test

Eliminate any existing test cases for objects from BXC3002.

!appendix

At the Vienna ARG meeting, an objection was raised to having a requirement that an object declaration for a time handler be library-level. The objection was that this was a contract model violation.

Randy pointed out that the language already includes such a rule in C.3.1 (8). The ARG decided that this rule needs to be fixed, with a run-time rule similar to that used for accessibility.

See the minutes of that meeting for more information.

****

From: Randy Brukardt
Sent: Wednesday, July 10, 2002 7:49 PM

One of the assignments I was given at the recent ARG meeting as to write an AI about C.3.1(8), as Steve Baird pointed out that it was a contract model violation. (This came up in the context of an unrelated proposal.)

I've attached my version of that AI. I'd like to point out the problem isn't so much the contract model violation, but rather that the standard rule for legality rules says that they are not enforced in a generic private part or body. That means that this problem (admittedly minor) will not be detected at all, which could cause obscure problems if it is encountered in practice.

Comments on the AI are welcome. (Note: It hasn't been posted on the web site yet.)

****

From: Jean-Pierre Rosen

Sent: Thursday, July 11, 2002 5:00 AM

I assume that this check happens at the time the object declaration is elaborated. Might be worse mentioning though...

****

From: Robert A. Duff
Sent: Monday, October 14, 2002 5:25 PM

I was directed at the ARG meeting to write up AI-303 in a much more general fashion. The AI asks about a particular case (a protected type with pragma Interrupt_Handler being passed as a generic actual parameter). The solution proposed at the meeting was to add pragma No_Deeper_Instances (on a generic, meaning instantiations can't be at a deeper accessibility level than the generic), and a pragma No_Deeper_Objects (on a type, meaning objects of the type cannot be at a deeper accessibility level than the type).

I can understand the point of No_Deeper_Instances on a generic. It might allow some things in generic bodies that might not otherwise be allowed.

However, I don't understand the point of No_Deeper_Objects on a type. When would a programmer put this on a type? Sure, there are some language-defined types that have this special property. (Like the CPU timer types in AI 297.) But why have a pragma? Why not just say that these types are special? And why not fix the generic contract model by saying they can't be passed to generics? (Nor can types derived from them, nor types containing them as components.) It seems like these types are special because they have a special relationship with the run-time system -- I see no general purpose here.

Furthermore, what would the generic thing allow? I thought at first that it would allow type extensions in generic bodies, but AARM-3.9.1(4.c-4.k) disabused me of that notion (something to do with abstract procedures).

I really can't get too excited about types derived from interrupt-handler types, or record types with components that are interrupt-handler types, and passing those to generics. Or CPU timer types used likewise.

Someone, please remind me why pragmas No_Deeper_... are of any use to Ada programmers.

****

From: Tucker Taft
Sent: Monday, October 14, 2002 8:17 PM

Both of these pragmas would change the rule about accessibility level in the corresponding declarative region, and perhaps eliminate various run-time checks on accessibility levels. This should allow the use of 'access rather than 'unchecked_access when initializing self-referencing components, as an example, or when passing the "current instance" to initializing functions, etc. Presumably 'access on parameters of a limited tagged no-deeper-objects type could also be allowed.

It would also seem that no-deeper-object types might only be passable to no-deeper-instance generics. One could presumably allow the pragma on a formal type to indicate that the actual *could* be a no-deeper-object type.

In general, I am always suspicious if we find something useful twice in the standard (e.g. no-deeper-object timer handlers and no-deeper-object interrupt handlers), but decide it would never be of use to a "typical" end user.

That's like Java deciding it is useful to have the type "boolean" with special literals true and false, but concluding that enumeration types in general aren't useful.

Also, given how many Restrictions have been defined that talk about no nested this and no local that, I suspect that in real-time and safety-critical programming, ensuring that all instances of a type/generic are at library level might be quite valuable.

****

From: Alan Burns
Sent: Tuesday, October 15, 2002 2:55 AM

> Someone, please remind me why pragmas No_Deeper_... are of any use to Ada programmers.

In the proposal on timing objects it is necessary for these to be only declared at library level. I have this as a rule. Meeting felt that the pragma would be clearer way of doing this. In general it is for objects that should not disappear in an inner scope.

****

From: Robert A. Duff
Sent: Tuesday, October 15, 2002 9:02 AM

> Both of these pragmas would change the rule about accessibility level in the corresponding declarative region, and perhaps eliminate various run-time checks on accessibility levels. This should allow the use of 'access rather than 'unchecked_access when initializing self-referencing components, as an example, or when passing the "current instance" to initializing functions, etc. Presumably 'access on parameters of a limited tagged no-deeper-objects type could also be allowed.

OK, this answers at least part of my question about what rules can be relaxed, but it doesn't answer my more important question, which is:

Why would an Ada programmer want a type all of whose objects must be declared at the same level as the type?

I understand that if the programmer *did* want that, then the above benefits acrue (you can be sure self-referential pointers don't dangle, and so forth). But where is the compelling example showing that this "no deeper" property was wanted in the first place?

> It would also seem that no-deeper-object types might only be passable to no-deeper-instance generics. One could presumably allow the pragma on a formal type to indicate that the actual *could* be a no-deeper-object type.

OK.

> In general, I am always suspicious if we find something useful twice in the standard (e.g. no-deeper-object timer handlers and no-deeper-object interrupt handlers), but decide it would never be of use to a "typical" end user.

Yes, you and I share the same philosophy here. The amount of special "magic" reserved to the language designer should be minimized.

(I learned this very early on in my programming career. I wrote a program in Pascal that printed lots of error messages using the built-in WriteIn procedure (which allows all kinds of magic, like different types of parameters), and then when I wanted to do some special processing of the error messages before printing, I found that it was impossible to split in a compatible Writeln-like procedure.)

But the two cases we're talking about (timer objects and interrupt handlers) are magical anyway -- they are closely tied to the run-time system. IMHO, that weakens the above philosophical argument.

> That's like Java deciding it is useful to have the type "boolean" with special literals true and false, but concluding that enumeration types in general aren't useful.

Yeah, I'll buy that analogy, although I'll bet use of Boolean is slightly more common in Ada than use of interrupt handlers. ;-)

> Also, given how many Restrictions have been defined that talk about no nested this and no local that, I suspect that in real-time and safety-critical programming, ensuring that all instances of a type/generic are at library level might be quite valuable.

Again -- it's all built-in stuff. I'm looking for at least one example of where a programmer would want the pragma in normal code (not tied to the implementation).

****

From: Robert A. Duff
Sent: Tuesday, October 15, 2002 9:10 AM

> In the proposal on timing objects it is necessary for these to be only declared at library level. I have this as a rule. Meeting felt that the pragma would be clearer way of doing this.

Yes, I understand that, but the timing objects are part of the language. They are known to the run-time system. I'm more looking for how an Ada programmer (other than a compiler vendor!) would use pragmas No_Deeper_...

>... In general it is for objects that should not disappear in an inner scope.

To partly answer my own question, if the programmer is putting objects on a global queue of some sort, making them all library level ensures they don't disappear while still on the queue. (In fact this is what timer objects do, and I can believe that regular programmers might want to do the same sort of thing.)

However, I find that argument rather weak. I've written such code, and I use finalization to make sure objects are removed from the global data structure just before disappearing. In fact, that would be the clean way to do timer objects. The only reason we don't do it that way for timers is efficiency (and perhaps predictability of efficiency) – we don't want the overhead of finalization in those kinds of real-time systems.

So is No_Deeper really restricted to those cases where objects are registered with a global data structure, AND the programmer finds finalization to be too costly?

I'm not totally *against* the idea of the No_Deeper pragma, but if I thought it was "really important" I might do my homework sooner. ;-)

I must admit that the idea of searching the RM for all the accessibility-level-related rules fills me with fear and loathing. ;-) Note that some such rules are not clearly labeled as level-related (e.g., you can't do thus-and-such in a generic body).

****

From: Robert Dewar
Sent: Tuesday, October 15, 2002 9:18 AM

<<I understand that if the programmer *did* want that, then the above benefits acrue (you can be sure self-referential pointers don't dangle, and so forth). But where is the compelling example showing that this "no deeper" property was wanted in the first place?>>

Good question! This again seems in the category of "neat clever ideas" which do not emanate from real application program needs.

I am very suspicious of neat and clever :-)

****

From: Tucker Taft
Sent: Tuesday, October 15, 2002 9:51 AM

It seems no surprise to me that both cases needing no_nested_objects relate to systems-level programming. I do believe these are for systems-level/real-time/safety-critical programming. The pragmas (or at least the no_nested_objects one) could be in the systems-programming annex, as far as I am concerned.

Interrupt handlers and timer handlers are "magic" relative to the standard Ada run-time system, but many situations require run-time-like code to be application specific, with "handlers" of various sorts that don't map directly to hardware interrupts, but rather application-level "software" interrupts. I remember that DEC's operating systems often used the concept of software interrupts.

Saying that the answer is finalization is probably missing the point for small, safety-critical systems, exactly where the guarantees provided by this kind of pragma would be most useful.

Note also that there really isn't anything terribly magic about the CPU timer proposal now, since it doesn't involve a new time type. It is a package or two that could be written by anyone, so long as we give them this pragma to prevent dangling references (and they can figure out some approach to do the mapping from Ada tasks to RTOS threads).

I have definitely seen handler-like things defined in Ada-based embedded-system application-specific “infrastructure" code before.

****

From: Michael F. Yoder
Sent: Tuesday, October 15, 2002 1:10 PM

>However, I don't understand the point of No_Deeper_Objects on a type. When would a programmer put this on a type?

If a programmer wants a global data structure that incorporates all instances of the type--for example, a doubly linked list, or a hash table. And moreover, she doesn't want to pay the price of

finalization overhead. This rule prevents putting an instance on the stack and then popping the frame, or allocating from a collection that then gets popped from the stack.

Another case is where the global data structure supports deletion only with difficulty. (If using a vendor-supplied package, deletion might even be unavailable.) The programmer decides it's better to have the library-level restriction than to have to support deleting the objects (from the global data structure) when returning from subprograms.

I suppose the canonical case is where the programmer says "Yes, I could supply a valid Finalize subprogram, Ada being a Turing-complete language and all, but I really don't want to."

>Sure, there are some
>language-defined types that have this special property. (Like the CPUtimer types in AI 297.) But why have a pragma? Why not just say that these types are special? And why not fix the generic contract model by saying they can't be passed to generics?

I dislike rules of the form "This is a duck. Nevertheless it is forbidden to walk or quack." I also strongly shun unnecessarily adding magical things available to implementers but not users.

> (Nor can types derived from them, nor types containing them as components.) It seems like these types are special because they have a special relationship with the run-time system -- I see no general purpose here.

I don't think this is so. This isn't inherently special to the run-time system, it's rather that the run-time system happens to want to do a thing requiring the restriction, namely, the objects need to be tied together in some kind of global relation.

****

From: Robert A. Duff
Sent: Tuesday, October 15, 2002 1:59 PM

> If a programmer wants a global data structure that incorporates all instances of the type--for example, a doubly linked list, or a hash table. And moreover, she doesn't want to pay the price of finalization overhead.

This reminded me that I have at least one such case in the program I'm currently working on. There is a general "statistics printing" package. The idea is that some packages in the system want to print some statistics from time to time. For example, there are various storage-pool packages that can print out information about how many bytes have been allocated and deallocated so far, the high-water mark of allocation, and so forth.

Any such package "registers" itself by declaring (in the package body) a type derived from the Statistics_Object type, and overriding the Print primitive, and declaring one object of the type. The Initialize operation of Statistics_Object links all objects into a single singly-linked list. Whenever statistics are desired, one can call Print_Statistics, which walks down the list and calls all the Print ops.

The point of all this is that the caller of Print_Statistics knows *when* it might be interesting to print stuff out (namely, in the main loop, but only if the user has requested this verbosity), but it does not need to know about all the packages that have some interesting statistics to print.

There is a comment saying that you have to declare all Statistics_Objects at library level, or else you'll get dangling pointers. And if we want to relax that restriction, we'll need to implement a Finalize on Statistics_Object that removes the object from the list.

It wasn't that I wanted to avoid the overhead of Finalize. It was just that I didn't want to bother implementing the "remove from list" operation, since it is never needed. Note that I would need a doubly-linked list to implement Finalize, since this is a multi-tasking system.

So if pragma No_Deeper_Objects existed, I could get compile-time checking for this abstraction, as it is, without the capability of nested statistics-printing objects. This is "merely" a debug/trace kind of facility -- but that doesn't make it any easier to debug dangling pointers.

Anyway, I now agree that the capability of AI-303 is useful. But I still claim that it's not the highest-priority thing -- after all, if clients obey the comment in the statistics package, all is well. E.g., I claim that AI-287, allowing aggregates of limited type is *far* more important. (To use the same example, there are *hundreds* of cases where AI-287 would make my program safer, whereas there are just a few cases where AI-303 would do so. We're talking about a program of around 50,000 lines of code, expected to grow to 3 times that by the time it's done.)

****

From: Michael F. Yoder
Sent: Tuesday, October 15, 2002 2:30 PM

I'm not terribly excited about the feature either, but I have a strong preference for removing a wart over adding one, and the fact there's some real uses too is a nice bonus. (The existing wart being the special rule for interrupt-handler types.) I recognize that the priority level of this item is entirely inherited from CPU timer types.

****

From: Robert I. Eachus
Sent: Tuesday, October 15, 2002 4:22 PM

Actually, I am coming to the conclusion that pragma No_Deeper_Objects is of general use. At first I thought it belonged in the real-time Annex, but I am now realizing that the pragma, in addition to its legality effects, eliminates a lot of run-time checking. This can either be in the form of user written or compiler generated code.

Let me give the example that prompted this. I have a set of packages that creates a user-extensible message type, and manages the message passing between tasks. To make the message type both user extensible and robust, I have to use controlled types. There is a compiler generated overhead for this, plus there is a lot of code that has to manage objects that may disappear at any time. It was Bob's mention of doubly-linked lists that triggered the memory. (And the problem of making sure that finalizing a list of messages worked. The data operations had to be done in first to last order, but the object finalizations were last to first. The final code looked elegant, but was almost impossible to understand.)

Most uses of the package only declare a few specific objects, and often the content, if any, of the message, is an enumeration type or an access value. (The real message is the destination and the return "address" of the sender.) For those that need more messages or big messages I can provide an other (generic) package that manages a user-sized pool of messages with a pointer (excuse my C) to the "real" contents. Even with both packages, the overall simplicity of the code should be much better. And using a protected type instead of a controlled type and a protected type to manage concurrency has to be a win in performance.

>E.g., I claim that AI-287, allowing aggregates of limited type is *far* more important. (To use the same example, there are *hundreds* of cases where AI-287 would make my program safer, whereas there are just a few cases where AI-303 would do so. We're talking about a program of around 50,000 lines of code, expected to grow to 3 times that by the time it's done.)

See above. I think the big win has to come during design. And I am not even sure that I need the ARG to add the pragma, since it should, if generally useful, find its way into all compilers. What I really need is for users to understand the meaning of the pragma. So maybe the answer is to put it in the systems programming Annex after all. It will get the documentation into the RM.

****

From: Ted Baker
Sent: Wednesday, October 16, 2002 6:41 AM

> ... I also strongly shun unnecessarily adding magical things available to implementers but not users.

Yes. I hope everyone will keep this in mind. I've found that an embedded systems application builder ends up needing all the basic tools that a language implementers needs. It seems to me that one of the main ways Ada 83 improves over Ada 95 is in moving away from the "closed system" viewpoint, in providing a more complete set of tools for user-defined extension using the language. I hope the next revision of the language will continue in that direction.

****

From: Tucker Taft
Sent: Wednesday, October 16, 2002 9:45 AM

>... one of the main ways Ada 83 improves over Ada 95...
>   ^^^^^^   ^^^^^^

I hope that wasn't a Freudian slip!