The basic idea of this feature is simple — between the time memory is "allocated" for a variable and the first explicit operation on that variable, the processor automatically generates a call to the initial procedure for that variable (if there is one for its type), and between the last explicit operation and the "deallocation" of the variable, a call to the final procedure is generated.

At the basic level, the only syntax is that which identifies a procedure as being the initial or final procedure for a type. For that syntax, we have chosen to use type-bound procedure notation with "special" names. The usual semantics of type-bound procedures implies that initial and final procedures are applicable wherever the type itself is. Note that, as in the example below, a type need not be extensible ("object-oriented") to have type-bound procedures.

```
MODULE I_F_demo
  TYPE I_F_type
    INTEGER :: id
   CONTAINS
    PROCEDURE :: (INITIAL) => demo_init
    PROCEDURE :: (FINAL) => demo_final
  END TYPE I_F_type
  INTEGER :: last_id = 0, pop = 0
 CONTAINS
  SUBROUTINE demo_init(the_var)
    TYPE(I_F_type),INTENT(INOUT) :: the_var
    the_var%id = last_id + 1; last_id = the_var%id; pop = pop + 1
  END SUBROUTINE demo_init
  SUBROUTINE demo_final(the_var)
    TYPE(I_F_type),INTENT(INOUT) :: the_var
    pop = pop - 1
  END SUBROUTINE demo_final
  SUBROUTINE info(tag,the_element)
    CHARACTER(*),INTENT(IN) :: tag
    TYPE(I_F_type),INTENT(IN) :: the_element
    PRINT *,tag,' is element#',the_element%id,'/',last_id,' -- ', &
    & pop-1,' others exist.'
  END SUBROUTINE info
END MODULE I_F_demo

PROGRAM demo
  USE I_F_demo
  TYPE(I_F_type) :: var
  TYPE(I_F_type),POINTER :: ptr
  CALL info('var',var)
  CALL inner; CALL info('var',var)
  ALLOCATE(ptr); CALL info('var',var); CALL info('ptr',ptr)
  CALL inner; CALL info('var',var)
  DEALLOCATE(ptr); CALL info('var',var)
 CONTAINS
  SUBROUTINE inner
    TYPE(I_F_type) :: var2
    CALL info('var',var); CALL info('var2',var2)
  END SUBROUTINE inner
END PROGRAM demo
```

The output of this example should be something like the following:

```
var is element#1/1 -- 0 others exist.
var is element#1/2 -- 1 others exist.
var2 is element#2/2 -- 1 others exist.
var is element#1/2 -- 0 others exist.
var is element#1/3 -- 1 others exist.
ptr is element#3/3 -- 1 others exist.
var is element#1/4 -- 2 others exist.
var2 is element#4/4 -- 2 others exist.
var is element#1/4 -- 1 others exist.
var is element#1/4 -- 0 others exist.
```

This output trace reflects

+ the creation of var in demo,

+ the creation of var2 in the first call to inner,

– the destruction of var2 in the first call to inner,

+ the allocation of ptr,

+ the creation of var2 in the second call to inner,

– the destruction of var2 in the second call to inner, and

– the deallocation of ptr.

[The destruction of var in demo also took place and generated one final call to demo_final, but since no output was generated afterwards, we don't really see its effects.]

The bulk of the specification of this feature describes <u>when</u> these procedures and <u>in what order</u>:

- Most of these simply make explicit ordering requirements that should be "intuitively obvious".
- A few provide ordering options that may make implementation more convenient.
- A few provide institute fairly arbitrary ordering decisions to give users a basis for portable programming in esoteric situation.

<u>Components</u>

Since, as a general rule, initial and final procedures tend to perform operations on the components of the type, it follows that if a component is itself of a derived type with initial and final procedures, the initial procedure for the component must be called before the initial procedure for the variable which contains it, and the final procedure for the component is called after the final procedure for the variable which contains it.

A similar order applies to types that extend another type, as the type being extended is effectively a component within the extension type.

Allocatable and Pointer

When you ALLOCATE a variable, the initial procedure is called for its elements. When you DEALLOCATE a variable, the final procedure is called. [This corresponds to when a compiler generates code to do default initialization and automatic deallocation of allocatable components.] Note that the automatic deallocation of an allocatable variables and components triggers final procedures just as explicit DEALLOCATE statements do.

If you ALLOCATE a variable of a type that has a final procedure, your program is required to deallocate it. [If the variable is allocatable, this should happen automatically, but if it is a pointer, this places a requirement on the logic of your program.]

“Ordinary” nonSAVEd

In an executable scoping unit, the initial routines for “ordinary” nonSAVEd variables are called “in the prolog” and final routines “in the epilog”. [Again, this corresponds to when a compiler generates code to do default initialization and automatic deallocation of allocatable components.]

“Ordinary” SAVEd

The time of initialization and finalization for “static” variables has intentionally been only partially specified to allow various implementation strategies, including the following:

1. In a processor with sufficient support in the linker, a list of static initializations to perform can be accumulated prior to execution, so the main program prolog can do those calls.

2. If the linker doesn’t provide the support for building such a list, then on each invocation of an executable scoping unit, the processor can check a flag to determine whether that is the first invocation of the scoping unit, and, if so, do the initialization of the static variables.

Other strategies are possible (e.g., starting asynchronous execution of the initialization in the main program prolog and waiting for it to complete in the executable scoping unit prolog), but the above two are the most likely strategies. In either case, the likely strategy for finalization is to build a list of finalizations to perform during initialization and then process the list immediately prior to the implicit closing of I/O units on program termination. A compiler would be permitted to do the analysis to determine if a scoping unit could not possibly be referenced again and, if so, to begin the finalization early, so it seems unlikely that any compiler will go to the trouble of doing so.

Modules

In general, modules (or specific instances of modules) are initialized before the scoping units that reference them and finalized after. It is expected that this will typically be achieved in one of two ways:

1. With the necessary linker support to build lists of static initialization routines, this list can be properly sorted before being processed from the main program prolog.

2. Otherwise, the prolog for all executable scoping units can test flags to determine whether the modules being referenced have been initialized, yet. (It could then also set a flag to cause the matching finalization to occur when exiting that instance of the executable scoping unit.)

F90 and F95 describe modules as having multiple instances, but go to significant lengths to allow implementations which reuse a single static instance. The rules for initial and final procedures attempt to be consistent, but if an implementation has a reason to support multiple instances of a module (e.g., parallel execution on a multi-processor system), it might be appropriate to use approach 1 for the shared “static” variables and approach 2 for the variables specific to particular instances.

Within a Scoping Unit

An arbitrary ordering rule has be applied which, in effect, says that initial and final procedures for a variable may reference variables declared before that variable in the scoping unit, but not those declared after.

Ω