Rationale

For many abstractions, it is possible to create a "better" implementation if one can validly make assumptions about the state of the abstraction at the time its operations are executed. For example, it may be possible to code a faster and more compact operation if one can assume that some aspects of the implementation have been set up prior to the execution of an operation, or it may be possible to provide amore feature-rich implementation if one can assume that certain information is being maintained in the representation "as you go". In some cases, a correct implementation of the abstraction may be possible only if such assumptions can be made.

In order for such an assumption to be valid, two things must be true: The operations must implemented such that assumed property is preserved by those operations, and something must establish that property initially.

In some abstractions, there is a natural first operation where this establishing can take place. In others, forcing the existence of such explicit initialization is unnatural and likely to result in use because people either forget to do the initialization or inadvertently perform the initialization more than once. For this reason, abstraction implementors would like a means for performing the necessary initialization automatically, without any explicit action by the user of the abstraction.

Fortran 95 provided such an automatic mechanism in default initialization. Most abstractions requiring the establishment of initial conditions can do so with default initialization, but the limitations of this form are such that there are many that cannot. For example, pointers cannot be initialized this way to a non-null state. More subtly, there is no direct way to perform initialization that affects more than the components of the representation. It would often be possible to work around this limitation by using default initialization to establish a recognizable partially initialized state and adding code to each operation to recognize this state and complete the initialization, but the time and space costs of distributing the initialization this way are frequently unacceptably high.

Thus, we find there is a residual need to be able to provide arbitrary code that is to be automatically executed on the representation of an abstraction before all other operations. Because of the precedent in C++, we have used the word "constructor" to describe this capability in our requirements. Unfortunately, we have used "constructor" to mean something different in Fortran, so I suggest that we instead call this an initializer.

Similarly, although for different reasons, there is a need for finalizers (called "destructors" in C++). One class of reasons involves the recovery of resources. Fortran 95 and the Data Type Enhancements TR already provide for some automatic resource recovery in the automatic deallocation of allocatable arrays and allocatable components. Resource recovery that might require user code includes memory deallocation based on reference counts, release of I/O units, or the deactivation of automatic display through a graphical interface. A second class of reasons involves deferred or buffered operations. Often operations can be completed more efficiently if they are batched together. For this reason, execution of operations may be deferred in hope of combining them with later operations. I/O buffering is a classic example of this strategy. At the point we know that there will be no later operations, the deferred operations must be completed.

As with initial operations, there will sometimes be an appropriate explicit final operation, but in many other cases such an explicit operation would be unnatural and error prone. Unlike initial operations, there is no way to simulate general finalizers using the existing features of the language.

Specifications: The Big Picture

Procedures may be linked syntactically with derived types to make them initializers or finalizers for data objects of that type. Once this link is established it cannot be overridden or suppressed by any kind of manipulations of procedure accessibility. It should be possible to write initializers that receive entire objects, so that zero-sized objects may be initialized or elements in different positions in an array may receive different automatic initializations, or to write initializers that are applied on an element by element basis.

Initializers are invoked automatically as part of the initialization process that occurs after a data object is allocated and before it is used. (By allocation, we mean the process by which all variables obtain storage, not just those that obtain storage through an ALLOCATE statement.) Finalizers are invoked automatically as part of the finalization process that occurs after all use of a data object and before it is deallocated.

(Unless altered by this specification, the initialization process takes place at those points where derived type objects receive default initialization or allocatable components in a derived type are initialized to not being allocated, and the finalization process takes place at those points where allocatable components are automatically deallocated.)

The initialization process consists of the following steps (in order):

- If the type extends a parent type, the initialization process is applied (recursively) to the parent type.
- The initialization process is applied to the declared components in the derived type except those that are pointers or allocatable.
- Default initialization is applied and allocatable components are given their default state.
- If present, the initializer is invoked, or explicit initialization is applied.

Note: I believe that the above correctly reflects the way explicit initializations and multiple levels of default initialization currently interact.

The finalization process consists of the following steps (in order):

- If present, the finalizer is invoked.
- Allocatable components are deallocated. (This will trigger the finalization process for those components.)
- The finalization process is applied (recursively) to the declared components.
- If the type extends a parent type, finalization is applied to the parent type.

In a given scoping unit, the initialization process for all variables having the SAVE attribute and not having the ALLOCATABLE or POINTER attribute is performed before the initialization process for any variable not having the SAVE attribute. Note: This still allows the initialization of SAVEd variables to occur at any time from the beginning of execution to the beginning of execution of the prolog for the scoping unit in which it appears. Reversed orderings hold for the finalization process.

The initialization process for all variables in a module having the SAVE attribute and not having the ALLOCATABLE or POINTER attribute is performed before the initialization process for any variable in a scoping unit referencing that module. The initialization process for all variables in a module not having the SAVE, ALLOCATABLE, or POINTER attribute is performed before the initialization process for any variable not having the SAVE attribute in a scoping unit referencing that module. Similar orderings hold between a host scoping unit and scoping units contained within it. Reversed orderings hold for the finalization process.

The finalization processes resulting from normal program termination take place before the automatic closing of files.

---- Begin optional inclusion ----

If two variables not having the ALLOCATABLE or POINTER attribute are explicitly declared in the same scoping unit and either both have the SAVE attribute or both do not have the SAVE attribute, the one whose name appears first in the declaration constructs of the scoping unit will be initialized first. Reversed orderings hold for the finalization process.

---- End optional inclusion ----

Note that the various ordering constraints determine only a partial ordering. Subject to those limitations, the ordering of these operations is processor dependent.

Specifications: Details

A program that ALLOCATEs a pointer variable of a type with a finalizer shall also DEALLOCATE it. (This DEALLOCATE might be in the finalizer of some other variable.) Note: Since this makes a program that does not DEALLOCATE not standard conforming,this allows a processor, as an extension, to "garbage collect" such "dangling" objects and finalize them at program termination (or, in some cases, before), but it would not require a processor to do so.

Variables of a type with an initializer or finalizer shall not appear in COMMON. Note: With sufficient restrictions, it might be possible to allow such variables in COMMON. Subgroup did not wish to take the time now to work out what restrictions would be necessary.

Variables of a type with an initializer shall not be explicitly initialized. Similarly, components of such a type shall not have an overriding default initialization. Note: Subgroup intends to explore the idea of alternative initializers to allow some form of these initializations.

For INTENT(OUT) dummy arguments, the associated actual argument goes through the finalization process before the dummy argument goes through the initialization process. Note that this has bearing on the integration of the Enhanced Data Types TR.

As an exception to the earlier ordering rules, the initialization of a function result variable or of an INTENT(OUT) may (but need not) occur before other initializations implied by the execution of that procedure. Similarly, the finalization of a function result variable or of an INTENT(OUT) may occur after other finalizations implied by the execution of that procedure. Note: It is expected that typically the finalization of the function result will not occur until the result has been used in the context of the expression in which the function reference appeared.

In cases where the nonSAVEd variables in a module would be finalized and later reinitialized, the processor may suppress both the finalization and initialization and retain the status of all the

nonSAVEd variables in the module. Note that the first initialization and last finalization cannot be suppressed.

Although normal termination, including at least the execution of a "plain" STOP statement, results in the execution of finalizers, there should be some way to signal an abnormal termination where execution of the remaining finalizers should not be attempted.

---- Begin alternative A ----

In scoping units where a type is defined or in which it is accessible entirely by host association, initializers and finalizers for that type are not automatically invoked (but they may still be explicitly invoked.

---- Begin alternative B ----

In scoping units where a type is defined or in which it is accessible entirely by host association, an additional attribute should be available to suppress automatic invocation of initializers and finalizers for some objects of that type. Note that the dummy argument of an initializer should not be INTENT(OUT) unless this attribute is also specified, lest an infinite recursion result.

---- Begin alternative C ----

Note that the dummy argument of an initializer should not be INTENT(OUT), lest an infinite recursion result.

---- End alternatives ----

Related Work

Subgroup needs to look further at the issues of these types appearing in COMMON and explicit initialization.

Subgroup needs to identify the nature of the abnormal termination signal.

Subgroup needs to look at the implication of initializers and finalizers with polymorphic dummy arguments.

There may be a need for additional features (e.g., the one proposed in 97-210) to fully meet the original WG5 requirement in this area.

There is a perceived need for initializers and finalizers for entire modules (and, to a lesser extent, other scoping units). It may be possible to simulate this feature using object initializers and finalizers, but if it is not, additional work may be necessary in this area.

Ω