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 automatic initial operation.

Similarly, although for different reasons, there is a need for automatic final operations (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 automatic final routines using the existing features of the language.

Preliminary Specification

As we have already alluded, Fortran 95 and the Data Type Enhancements TR already have several limited automatic initial and final operations: The initialization of allocatable arrays, allocatable components, and components with default initialization all require a kind of compiler-generated initial operation. Similarly,

the automatic deallocation of allocatable arrays and allocatable components require a kind of compiler-generated final operation. Thus, our preliminary specification is that user-supplied initial operations are executed under the same circumstances that allocatable components and components with default initialization are initialized, and user-supplied final operations are executed under the same circumstance that allocatable components are automatically deallocated.

Additional Specification: Ordering

If a type has both a user-supplied initial operation and allocatable components or components with default initialization, the "system" initialization takes place prior to the execution of the initial operation. Conversely, "system" deallocation of allocatable components takes place after any final operation for the same type.

If a type with a user-supplied initial operation has components of types that also have user-supplied initial operations, the component initial operations are executed before the containing initial operation. Conversely, component final operations are executed after containing final operations.

Similarly, base type initial operations are performed before extension type initial operations, and base type final operations are performed after extension type final operations.

The interaction of explicit initialization and initial operations raises interesting issues. I believe that explicit partial initialization (i.e., explicit initialization of components) should be prohibited, because there is no "right" ordering between such initialization and initial operations. (If the initialization is done before the initial operation, the initial operation is likely to obliterate its effects. If the initialization is done after, it may invalidate the conditions established by the initial operation.) Explicit initialization of complete elements may be reasonable, because the necessary conditions can be established in the construction of the initialization value, but if that value is then to be placed in the element by intrinsic assignment rather than defined assignment, then the initial operation for element should be suppressed (because its effects would be immediately overridden in an uncontrolled way. Thus, the initial operation on the initialization value would effectively be the initial operation for that element. To keep initial and final operations properly paired, the final operation for that initialization value should probably also be suppressed. There may still be problems getting things right when the initialization value is a symbolic constant rather than an immediately constructed or computed value, so this area may require further thought.

Because of the possibility that initial and final operations reference entities outside of the components of the representation, there is a potential need to control the relative order of the execution of the initial and final operations of entities declared in the same scoping unit. One possible method is to say that initial operations are performed in declaration order and final operations in reverse declaration order.

Additional Specification: Procedure Interfaces

The fact that default initialization and initial operations are applied to INTENT(OUT) dummy arguments, creates a possible anomaly. To counter this problem, I suggest that final operations and automatic deallocation of allocatable components be applied to corresponding actual arguments just prior to procedure invocation. In order for the compiler to know that it needs to perform these operations, an explicit interface should be required in this case. (The allocatable components part of this may be a "hole" in the Data Type Enhancements pDTR that we should note during the first ballot.)

It should be possible for function result variables initial operations to precede invocation of the function and for the final operation to be deferred until after the function value has been used in the expression in which it appears.

Additional Specification: Other

I would recommend that all outstanding final operations be executed if a STOP statement is executed.

It may be desirable that automatic initial and final operations not be performed in the scoping unit in which the type is defined. This is somewhat inconsistent with default initialization and the handling of allocatable components, but it allows for the creation of alternative initialization operations without paying the cost of an unnecessary default initialization.

Related Issues: "Static" Storage

It is expected that in most implementations, default initialization and the initialization of allocatable components for variable with the SAVE attribute will be done in the linker/loader rather than by execution of code at run time. Similarly, the automatic deallocation of allocatable arrays and components with the SAVE attribute is not strictly necessary, since execution is about to complete, anyway. Thus, the existing implementation models for these features do not readily extend to applying user-supplied initial and final operations to variables with the SAVE attribute. Instead, there will either have to be additional support in the linker for identifying initial operations to be performed at the start of execution, or the compiler will have to use flags to recognize the first entry into each scoping unit so the initial operations of the SAVEd variables in that scoping unit can be performed. (A single flag per scoping unit will suffice for all the SAVEd variable initial operations in that scoping unit.) A "system" initial operation for these variables can be used to build up a list of final operations to be performed at program termination.

A similar issue is the handling of nonSAVEd variables in COMMON and modules. Although it is possible for there to be multiple instances (serially) of a common block or a module, we have been careful to allow processors to reuse a single instance, performing the initialization just once and deferring automatic deallocation to the end (when it doesn't really need to be done). With the introduction of user-supplied initial and final operations we must either perpetuate this situation even more explicitly, or "bite the bullet" and require the processor to treat the logically separate instances as truly distinct. (If we follow the latter approach, I would suggest we also clean up all the special cases we have created for modules and common blocks.) I lean toward the "bite the bullet" approach, but am quite willing to take guidance from the committee on this issue.

Looking Forward to Syntax

There are two probable alternatives approaches to the syntax: The first is to use yet another special generic ID to "name" such initial and final operations. The second is to put something in the type definition that identifies its initial and final operations. (Note that in a parameterized derived type with KIND parameters, there may be multiple initial operations and multiple final operations, one for each supported combination of KIND parameters.)

The key issue is that once an initial or final operation has been associated with a type, it should not be possible to use visibility rules to prevent that operation from being performed for a particular entity. This tends to come naturally with the second syntactic approach. It would require special rules for the first approach. Note that this is similar to the property we want for defined assignment, so it is likely that we will want to use the same syntactic/semantic approach to these operations as we use to "fix" defined assignment.

Related Features

If you read the original WG5 requirement, you will find that it applies only to entities allocated in an ALLOCATE statement and that it allows the provision of "extra" arguments to ALLOCATE that would be

passed through to the "constructor". I believe that limiting this feature to entities created with an ALLOCATE statement would have been unacceptable to many constituencies, including those interested in object-oriented programming. However, we are left with the problem of addressing the functionality of those "extra" arguments. I suggest two ways: First, now that we have allocatable dummy arguments, it is possible to write procedures to one's allocating. Such procedures can have as many extra arguments as one needs. Second, the examples in the requirement could have been well handled by derived type parameters. If these values are to be specified in an ALLOCATE statement, we must once again revive the idea of extending the ALLOCATE statement to non-KIND derived type parameters. Because that issue can be explored separately, I will do so, but please bear in mind that I will be doing so not on its own merit, but as a means of completing this requirement.

It should be noted that in some cases, there are initial operations that apply to an entire type rather than to individual objects of that type. For example, one may need to initialize some kind of dynamic dispatch table or a storage management scheme for the abstraction. I am currently leaning towards providing this capability by providing for initial and final operations associated with a module rather than specific instance of a type. Such operations for the module containing the definition of a type would serve as the initial and final operations for that type. With some combinations of decisions on the issues in this document, it would be possible to achieve the affects of such operations by declaring an extra type and an object of that type at the beginning of the module, using the initial and final operations for that object as the module initial and final operations. For other combinations, an additional feature may be required. I will defer further work on this issue until we make those decisions and know whether we need an additional feature.

Ω