ISO/IEC JTC1/SC22/WG5-N1040

Revision of paper 94-202r2 from /oof to X3J3

Subject: **Text for X3J3/009 re: Allocatable Components in Structures (B3)**
References: WG5-N930, Resolutions of the Berchtesgaden WG5 Meeting, B9
WG5-N931, Requirements for Allowing Allocatable derived-type Components

Revision history: WG5-N1040, revised to resolve received comments
94-202r2 remove allocatable entities from COMMON, do not specify a storage unit;
clarify/simplify constructors (revised post-meeting per meeting 129 comments)
94-202r1 provide restriction for entity-decls, add constructors
94-202 original presentation, May 1994 (meeting 129)

Requirement Title: B9/B3 Allocatable derived-type components

Status: X3J3 consideration in progress

Technical Description:

Currently, the ALLOCATABLE attribute is limited to local named array data entities. One cannot have components of derived-type objects, nor as dummy arguments or function return values. In addition, one cannot have ALLOCATABLE CHARACTER's. There is a requirement from WG5 that these restrictions be removed, that ALLOCATABLE be extended and regularized.

In addition, if an ALLOCATABLE object is still allocated when it goes out of scope, its allocation status becomes undefined. There is a desire that the deallocation be provided "automatically" by the processor. That is addressed in a separate paper (X3J3/94-211).

The current paper provides a major step towards ALLOCATABLE arrays as full, first-class entities. It provides for them to appear as components of derived types, as per WG5-N931, "Requirements for Allowing Allocatable Derived-type Components".

Still outstanding from the general request:

- ALLOCATABLE dummy arguments and function return values
- ALLOCATABLE strings
- ALLOCATABLE derived types (needs Parameterized Derived Types)

While these are mostly fairly straightforward, a large number of edits appear to be required. Due to the shortage of time, we recommend that this development be undertaken for Fortran 2000.

Per direction from X3J3, this proposal does not provide for allowing ALLOCATABLE objects, nor derived type objects with ALLOCATABLE components in COMMON or EQUIVALENCE.

Motivation:

ALLOCATABLE provides functionality which shares much with that of the POINTER features. Indeed, the underlying implementation is probably nearly identical, with one important difference. The address pointer implicit in ALLOCATABLE objects is not accessible to the user. ALLOCATABLE objects, therefore, cannot be aliased with other objects as a result of pointer assignments. For this reason, they behave much better than pointers with respect to optimization.

Given the benefits of ALLOCATABLE, it appears advisable to extend the facility to handle derived type components as well as normal variables. The extension introduces little additional complexity into either the language definition or implementation; in fact, one can argue that the generalization serves to remove an arbitrary restriction from the language.

The duties of the implementor follow directly from current practice. If a local object has the ALLOCATABLE attribute, the implementation must arrange to set the object to non-allocated status at the time that the object is created. This duty is now extended to components of derived type objects. In addition, if a derived type object with an ALLOCATABLE component is itself created via an ALLOCATE statement, the component must be set to non-allocated status as part of the creation.

There exists some complication with regards to constructors. There is no "null" value for an allocatable object, so the constructor must specify an actual value set for an allocatable component. (This will generally be an array constructor with an implied DO to provide the right number of elements.)
(Note that, even though "assignment semantics" apply, this value cannot be given as a scalar, as there is no size given in the type definition.)

The value of the constructor will include the allocatable component in an "allocated" state with the indicated contents. Note that the rules of assignment for allocatable arrays involve copying the contents, not the implicit pointer. Thus, if the constructor is used in an assignment statement, the allocatable components of the target variable must already be allocated.

Lastly, because allocation cannot be done prior to execution, allocatable components cannot be initialized in DATA statements or with = initialization.

(Note that some of these conditions may be changed slightly if we approve the proposal for constructors with keyword and optional fields.)

Detailed EDITS:

{{note: edits are marked with a brief comment about the particular subject area being addressed. In some cases, a subject area is relevant but requires no edits; in such cases, a null edit (plus comment) is given. }}

[throughout] {general nomenclature}

{no problem. The Standard references "allocatable array"'s throughout. There is no problem, in general, with this terminology applying equitably to either data entities or to components; so no pervasive change is required.}

Subclause 4.4.1

[33:20] {definition of component-attr-spec}

after: "R427 *component-attr-spec* **is** POINTER"

add new line:

" **or** ALLOCATABLE"

[33:26]

add a new constraint following the third constraint of R427:

"Constraint: POINTER and ALLOCATABLE must not both appear in the same *component-def-stmt*."

[33:26+] {{WG5: Since allocatable components prohibit appearance in storage-associated contexts, do not allow sequence types to contain allocatable components.}}

add a new constraint following the above:

"Constraint: The ALLOCATABLE attribute must not be specified if SEQUENCE is present in the *derived-type-def*."

[33:31-32] {WG5(reworded):dimension info}

replace the first constraint of R429 with

"Constraint: Each *component-array-spec* must be an *explicit-shape-spec-list* unless the POINTER attribute or the ALLOCATABLE attribute is specified."

[33:33] {WG5(reworded):dimension info}

in the second constraint of R429,

change: "POINTER attribute" to "POINTER attribute or the ALLOCATABLE attribute"

[33:38+] {default initialization, per 94-138}

change: "The POINTER attribute must not appear"

to: "Neither the POINTER nor ALLOCATABLE attribute must appear".

[33:39] {sequence types}

{{WG5: No changes needed. A sequence type cannot have allocatable components.}}

[34:2] {when one needs to have "::" in a component definition}

change: "or both"

to "the ALLOCATABLE attribute, or any combination thereof"

[35:24+] {examples}

add new paragraphs:

"A derived type may have a component that is allocatable. For example:

```
TYPE STACK
      INTEGER :: index
      INTEGER, ALLOCATABLE, DIMENSION(:) :: contents
END TYPE STACK
```

For each object of type `STACK`, the shape of the component `CONTENTS` is determined by execution of an `ALLOCATE` statement."

Subclause 4.4.4
[37:28] {constructors}{{WG5: Interaction with 94-009.006 in last sentence of this addition}}
Add, at the end of this subclause, new paragraphs:
"Where a component in the derived type is an allocatable array, the corresponding constructor expression must evaluate to an array. The value of the constructor will have a component that is allocated (i.e., has an allocation status of allocated), and with contents as given by the constructor expression. Note that the allocation status of the allocatable component is available to the user program if the constructor is associated with a dummy argument, but generally not for other uses. Note, also, that when the constructor value is used in assignment, the corresponding component of the target must already be allocated.

Where a derived type contains an allocatable component at any level of component selection, the constructor must not appear as a *data-stmt-constant* in a DATA statement (5.2.9), as an *initialization-expr* in an *entity-decl* (5.1), or as an *initialization-expr* in a *component-initialization* (4.4.1)."

Subclause 5.1
[40:5] {=initialization}
after: "an allocatable array,"
add: "a derived-type object with allocatable components,"

Subclause 5.1.2.4.3
[46:8] {allocatable arrays are no longer necessarily named}
in the first line of the second paragraph, change: "a named array" to "an array"

[46:13] {where & how you can declare ALLOCATABLEs}
after: "in a type declaration statement"
add: "or a component definition statement"

Subclause 5.1.2.9
[48:24] {storage units, sequence}
{WG5: This change has been deleted as unnecessary since derived type objects containing allocatable components cannot appear in COMMON or EQUIVALENCE}.

Subclause 5.2.9
[52:34] {DATA statement restrictions}
change: "or an allocatable array"
to: "an allocatable array, or a derived-type object containing an allocatable component at any level of component selection"

Subclause 5.4
[56:11] {WG5: do not allow allocatable components in NAMELIST}
change: ", or an allocatable array"
to: ", an allocatable array, or a derived-type object containing an allocatable component at any level of component selection"

Subclause 5.5.1
[57:1] {WG5: do not allow allocatable components in EQUIVALENCE}
after: "an allocatable array,"
add: "a derived-type object containing an allocatable component at any level of component selection,"

Subclause 5.5.2
[58:30] {WG5 (wording change): do not allow allocatable components in COMMON}
after: "an allocatable array,"
add: "a derived-type object containing an allocatable component at any level of component selection,"

Subclause 5.5.2.3
[59:42+] {Common Association}
{no edit needed; not allowed in COMMON}

Subclause 6.3.1
[67:16] {allocate statement}
{ok - because POINTER components were allowed}

Subclause 6.3.1.1
[68:1+] {allocation of allocatable arrays - initial status}
{ok, existing text appears to be sufficient}

[68:7+] {{WG5: Specify initial state of allocatable components after ALLOCATE}}
Add new paragraph at the end of this subclause:
"If an object of derived type is created by an ALLOCATE statement, any allocatable components (at any level of component selection, while the parent components do not possess the POINTER or ALLOCATABLE attributes) have an allocation status of not currently allocated."

Subclause 6.3.3.1
[69:8] {deallocation}
add footnote: "Allocatable array components of derived type objects with the SAVE attribute also retain their allocation status."
{{note: similar issue for 69:29, for pointers }}

Subclause 7.1.6.1
[77:40] {{remove derived-type constructors with allocatable array components from initialization exprs}}
after: "(3) A structure constructor where each component is an initialization expression"
add: "and no component has the ALLOCATABLE attribute"

Subclause 9.4.2
[124:16] {we do not allow derived-types with pointer components in i/o statements, neither should we we allow ones with allocatable components}
after: "If a derived type ultimately contains a pointer component"
add: "or contains an allocatable component at any level of component selection"

add to Rationale Section:

5.1.2.9 ALLOCATABLE attribute

ALLOCATABLE arrays provide a degree of flexibility intermediate between that of automatic arrays and pointers. The bounds of an automatic array can be calculated on non-constant values, but only at scope entry. Often, this is not sufficiently powerful; nor does it provide for SAVE arrays whose size is not given by a constant expression. Pointers do provide full control, with the time and size of allocation given by execution of the ALLOCATE statement. Because of the pointer assignment statement, pointers are a little less closely controlled than ALLOCATABLE objects. More than one pointer may be associated with a given object, a condition known as "aliasing" which can negatively impact optimizing compilers. Pointers also allow memory leaks if not properly deallocated.

Array variables, and components of derived types, can be ALLOCATABLE where it is desired to calculate their size as a function of some value determined during execution.

Initialization of allocatable arrays is not permitted for SAVE variables, because such initialization may take place prior to execution, yet the ALLOCATE action is normally only defined during execution.