Date:	1 February 1999
To:	J3
From:	Van Snyder
Subject:	Unresolved issues 72-76 and 78-81 concerning explicitly typed allocations
References:	98-208r2

1 Edits

Edits refer to 99-007. Page and line numbers are displayed in the margin. Absent other instructions, a page and line number or line number range implies all of the indicated text is to be replaced by immediately following text, while a page and line number followed by + indicates that immediately following text is to be inserted after the indicated line. Remarks for the editor are noted in the margin, or appear between [and] in the text.

[Editor: In the same paragraph.] New in Fortran 2000 is the ability to declare that structure components and objects that are not arrays have the ALLOCATABLE attribute; they therefore are automatically deallocated under the same conditions as allocatable arrays were automatically deallocated in Fortran 95. Also new in Fortran 2000 is the capability to specify nonkind type parameters, and the specific types of polymorphic objects, during allocation. xvi:21+

[Editor: Delete from "A deferred" to the end of the text of unresolved issue 81. The sentence on 31:1-2 is moved to 100:13+.] 31:1-8

[Editor: Replace "a pointer or allocatable array" by "allocatable or a pointer"] 38:28

[Editor: Replace "allocatable arrays or pointers" by "pointers or allocatable variables"] 38:36

[Editor: Replace "array" by "variable"] 45:37

[Editor: Replace "arrays" by "variables" twice.] 49:4,6

Constraint: If an asterisk is used as a *type-param-value* in the declaration of an entity, the entity shall be a dummy argument. 53:23-26

Constraint: If a colon is used as a *type-param-value* in the declaration of an entity, the entity shall have the ALLOCATABLE or POINTER attribute.

If a colon is specified for the *type-param-value* of a nonkind type parameter, the type parameter is a **deferred type parameter**. The value of a deferred type parameter of an object may be specified when the object is allocated (6.4.1), assumed from an actual argument when the object is a dummy argument associated with an actual argument, or assumed from a *target* during execution of a pointer assignment statement (7.5.2) in which the object appears as a *pointer-object*. 53:35-36

If a component of a derived type is allocatable, the corresponding constructor expression shall either be a reference to the intrinsic function NULL() with no arguments, an allocatable variable, or shall evaluate to an object of the same type; if the component is an allocatable array the expression shall evaluate to an array. If the expression is a reference to the intrinsic function NULL(), the corresponding component of the constructor has a status of not currently allocated. If the expression is an allocatable variable, the corresponding component of the constructor has the same allocation status as that allocatable variable; if it is allocated it has the same value, and if it is furthermore an array it has the same shape. With any other expression the corresponding component of the constructor has an allocation status of currently allocated, and has the same value as the expression; if it is an array it has the same shape as the expression. 55:19-32

When the constructor is an actual argument, the allocation status of the allocatable component is available through the associated dummy argument. Note 4.49

If a derived type has an ultimate component that is allocatable, its constructor shall not appear as a *data-stmt-constant* in a DATA statement (5.3.13), as an *initialization-expr* in an *entity-decl* (5.1), or as an *initialization-expr* in a *component-initialization* (4.5.1.2).

[Editor: Delete (if you agree this paper fixes issue 72).] 63:1-33

[Editor: Replace "array" by "variable" twice.] 63:43,44

[Editor: Replace "array" by "variable" twice.] 64:31,32

[Replace "array" by "variable"] 64:37

[Editor: Replace "A" by "In a *type-declaration-stmt*, a".] 65:38

[Editor: Replace "specification expression" by "*type-declaration-stmt*".] 65:40

[I agree that *char-len-param-value* should be replaced by *type-param-value*.] 67:32-38

[Editor: delete – some is moved to be after the next section, and some is moved to sections on ALLOCATABLE and POINTER attributes.] 73:30-74:18

[Editor: Change section number to 5.1.2.4.3.] 74:19

[Editor: Insert new section 5.1.2.4.4 and 5.1.2.4.5. Change the index item for deferred-shape array to refer to the next two sections.] 75:9+

5.1.2.4.4 Allocatable arrays

An **allocatable array** is an array that has the ALLOCATABLE attribute. It has a specified rank. Its bounds, and hence its shape, may be specified explicitly, or deferred until space is allocated by execution of an ALLOCATE statement (6.4.1). The bounds shall be declared in a type declaration statement, a component definition statement, a DIMENSION statement (5.3.5), a TARGET statement (5.3.8), or an ALLOCATABLE statement (5.3.6). If the shape of an allocatable array is deferred, that is, the allocatable array is a **deferred-shape array**, the bounds shall be declared by a *deferred-shape-spec-list*. Otherwise the bounds shall be declared by an *explicit-shape-spec-list*.

R518 *deferred-shape-spec* **is** :

The rank of an allocatable array with deferred shape is the number of colons in the *deferred-shape-spec-list*. The rank of an allocatable array with explicit shape is the number of *explicit-shape-specs* in the *explicit-shape-spec-list*.

No part of an unallocated allocatable array shall be referenced or defined. The array may, however, appear as an argument to an intrinsic inquiry function that is inquiring about argument presence, allocation status, or a property of the type or type parameters. The size, bounds, and shape of an unallocated allocatable array with deferred shape are undefined. An unallocated allocatable array with explicit shape may appear as an argument to an intrinsic inquiry function that is inquiring about the size, bounds or shape.

The lower and upper bounds of each dimension of an allocatable array with deferred shape are those specified in the ALLOCATE statement (6.4.1) when space for the array is allocated.

5.1.2.4.5 Array pointers

An **array pointer** is an array that has the POINTER attribute. It has a specified rank. Its bounds, and hence its shape, are specified when space is allocated by execution of an ALLOCATE statement (6.4.1), or determined when it is associated with a target by execution of a pointer assignment statement (7.5.2). The bounds shall be declared by a *deferred-shape-*

spec-list in a type declaration statement, a component definition statement, a DIMENSION statement (5.3.5), a TARGET statement (5.3.8), or a POINTER statement (5.3.7). A pointer array has deferred shape. An array pointer is a deferred-shape array.

The rank of an array pointer is the number of colons in the *deferred-shape-spec-list*.

No part of a disassociated array pointer shall be referenced or defined. It may, however, appear as an argument to an intrinsic inquiry function that is inquiring about argument presence, association status, or a property of the type or type parameters. The size, bounds, and shape of a disassociated array pointer are undefined.

The lower and upper bounds of each dimension of an array pointer may be specified in three ways:

(1) Both lower and upper bounds are specified in an ALLOCATE statement (6.4.1) when the array pointer is allocated, or

(2) The lower bounds are specified in a pointer assignment statement (7.5.2) and the upper bounds are calculated from the extent of the *target* as explained in 7.5.2.

(2) Both lower and upper bounds are assumed from the *target* in a pointer assignment statement (7.5.2) in which the array pointer appears as the *pointer-object* but no *bounds-spec-list* appears.

[Editor: Add a sentence at the beginning of the paragraph:] 75:28

The pointer attribute may be specified in a type declaration statement, a component definition statement, or a POINTER statement (5.3.7).

[Editor: Replace the sentence "If the pointer...." by "If the pointer is an array it shall be declared as specified in [new section] 5.1.2.4.5."] 75:30:31

The **ALLOCATABLE attribute** may be specified for a variable in a type declaration statement, a component definition statement, or an ALLOCATABLE statement (5.3.6). 76:14-16

[Editor: set "procedure pointer" in bold face, and add it to the index.] 78:31

Constraint: If a *proc-entity* is an external function, *proc-interface* is present, and the result type is not character, the type parameters of the result type shall be deferred or specified by initialization expressions. If the result type is character, the length parameter shall be deferred, specified by an initialization expression, or an asterisk. 79:2+

[Editor: Delete unresolved issue 16. The first paragraph is covered by 246:47-48, which applies to abstract interfaces, and the second is fixed below (I hope). I couldn't find where there is a definition for what is meant by "the procedure's characteristics shall be consistent with those specified in the procedure definition" at 246:47-48, but when I do find it I presume there will be work to be done there to account for deferred parameters.] 79:11-27

If *proc-interface* is present and consists of *abstract-interface-name*, it specifies an explicit specific interface (12.3.2.1) for the declared procedures or procedure pointers. 79:33+

If *proc-interface* is present and consists of *restricted-type-spec*, it specifies that the declared procedures or procedure pointers are functions having the specified result type, but does not specify any other characteristics. If a type is specified for an external function, its function definition (12.5.2.1) shall specify the same result type and type parameters.

If *proc-interface* is absent, the procedure declaration statement does not specify whether the declared procedures or procedure pointers are subroutines or functions.

Deferred parameters of function result types have no values; they simply indicate that those parameters of the function result will be determined by the function, when it is invoked.

R531 *allocatable-stmt* **is** ALLOCATABLE [::] ■ 82:14-19
■ *object-name* [(*allocatable-shape-spec-list*)] ■
■ [, *object-name* [(*allocatable-shape-spec-list*)]]...

Constraint: If the DIMENSION attribute for an *object-name* is specified elsewhere in the scoping unit, the *array-spec* shall be an *allocatable-shape-spec-list*.

R531A *allocatable-shape-spec-list* **is** *explicit-shape-spec-list*
or *deferred-shape-spec-list*

This statement specifies the ALLOCATABLE attribute (5.1.2.9) for a list of objects.

[Change "array" to "variable" twice.] 84:40,44

[Change "array" to "variable" twice.] 89:41,42

[Change "array" to "variable" twice.] 90:31,32

[Change "array" to "variable" twice.] 93:1,2

[Change "arrays" to "variables"] 97:26

A deferred type parameter of a disassociated pointer, a function procedure pointer, or an unallocated variable shall not be referenced. 100:13+

[Change "arrays" to "variables" twice.] 104:12,17

R623 *allocate-stmt* **is** ALLOCATE [*type-spec* ■ 104:18-19
■ [, DIMENSION (*allocate-shape-spec-list*)] ::] ■
■ *allocation-list* [, STAT = *stat-variable*])

Constraint: If an *allocate-object* is a deferred-shape array, the number of *allocate-shape-specs* shall be the same as the rank of the deferred-shape array. 104:28-29

Constraint: If an *allocate-object* is not a deferred-shape array, an *allocate-shape-spec-list* shall not be specified.

Constraint: If a DIMENSION specification is present, no *allocate-object* shall have an *allocate-shape-spec-list*.

[Editor: change "and" to "or a type that is".] 104:31

[Editor: Move to 104:29+, so that array bounds constraints are contiguous, and type specifier and type parameter constraints are contiguous.] 104:33-34

Constraint: If an *allocate-object* has a deferred type parameter, *type-spec* shall appear. 104:34+

Constraint: A *type-param-value* in a *type-spec* in an ALLOCATE statement shall not be a colon.

Constraint: In a *type-spec* in an ALLOCATE statement in which an *allocate-object* is a dummy argument that has an assumed type parameter, the *type-param-value* of every assumed type parameter of that dummy argument shall be an asterisk. No other *type-param-value* in a *type-spec* in an ALLOCATE statement shall be an asterisk.

[Editor: Delete (if you agree this paper fixes lines 105:3-5 of issue 74; lines 6-10 are updated and moved downward).] 105:1-10

When an ALLOCATE statement having a *type-spec* is executed, type parameters are specified by *type-param-values* in the *type-spec* in the ALLOCATE statement. Exactly one value shall be specified for every type parameter. If the value specified for a nondeferred type parameter is not the same as the value specified in the object's declaration, an error condition exists. 105:17-40

If a *type-param-value* in a *type-spec* in an ALLOCATE statement is an asterisk it denotes the current value of that assumed type parameter.

Parameter values are assigned to type parameters for derived types as specified in 4.5.5. For intrinsic types, parameter values are assigned to type parameters, with the correspondence determined by position or by name as defined in 5.1. A parameter value is not required to be of the same integer kind as the corresponding type parameter.

Unresolved issue 74
It was agreed in the specifications for explicitly typed allocations that values need be specified only for deferred type parameters. This paper proposes that every type parameter shall be specified if any type parameter is specified, because there is a constraint on *derived-type-spec* that this can't happen, and the necessary edits in 4.5.5 to allow specifying a subset of the parameters are difficult to construct. Is it worth the trouble to allow eliding nondeferred type parameters in an allocate statement? Insert J3 internal note

---

If a DIMENSION specification is present, its *allocate-shape-spec-list* applies to every *allocate-object* in the *allocation-list*. 106:1-17

[Editor: Starting with "If..."] If an *allocate-shape-spec* list is specified for an array that does not have deferred shape, and the bounds specified in the *allocate-shape-spec* list are not the same as those specified in the declaration of the object, an error condition exists. 106:30-32

[Editor: Replace "array" by "variable" every time it appears in this range (and replace "an" by "a" as necessary). Yes, I checked them all.] 107:11-43

[Editor: Replace "array" by "variable" twice.] 109:13,16

[Editor: Replace "array" by "variable" every place it appears in this range (and replace "an" by "a" as necessary). Yes, I checked them all.] 109:37-110:38

[Editor: Remove the word "array."] 119:41

[Editor: Add "with deferred shape" after "array"] 119:42

[Editor: Remove the word "array" twice.] 138:38-39

Do we need to be careful about using the word "shape" at 138:42? That is, does "shape" allow scalars? J3 note

---

[Editor: Remove the word "array"] 139:13

[Editor: Add "with deferred shape" at the end of the sentence.] 140:9

Constraint: If the *pointer-object* is a function procedure pointer, the *pointer-object* and *target* shall have deferred corresponding type parameters. 140:18-19+

[Editor: Delete. It's covered by the constraint at 140:16-17.] 140:24-26

If *pointer-object* is a data object or function procedure pointer, all nondeferred type parameters of *pointer-object* shall have the same values as corresponding type parameters of *target*. If *pointer-object* is a data object that has deferred type parameters, the values of those parameters are assumed from the values of corresponding parameters of *target*. The corresponding parameters of *target* can be deferred, assumed, or explicitly declared. 141:11-19

Deferred parameters of function procedure pointers have no values; instead, they indicate that those parameters of the function result will be determined by the function, when it is invoked. Remember that any entity with deferred type parameters, including a function result, is required to have the ALLOCATABLE or POINTER attribute. Note $7.46\frac{1}{2}$

Remove unresolved issue 78. There is no constraint that all of the type parameters agree, only that all kind type parameters agree; kind type parameters cannot be deferred. The paragraph at 141:11-13 *does* discuss dynamic type parameters, by saying that values of deferred type parameters of *pointer-object* are assumed by *target*.
Verifying that the values of nondeferred nonkind type parameters of *pointer-object* are the same as corresponding parameters of *target* requires a run-time check, regardless whether nondeferred parameters of *pointer-object* correspond to deferred or nondeferred parameters of *target*. The most obvious case is when they're both assumed. If the check fails there's no way to catch it. This is in the same category as array bounds checking. A subscript out-of-bounds is an indication that the program is not standard-conforming; so is a mismatch of *pointer-object* and *target* type parameters. I know lots of people want to use square brackets for array constructors, but I have another suggestion, at least within pointer assignment statements: Let me put [STAT=*variable* and/or ERRMSG=*default-char-variable*] at the end of the statement, so that I can catch mismatched type parameters. J3 note (not an edit)

[Editor: Replace "array" by "variable"] 185:14

[Editor: Remove the word "array"] 185:35

[Editor: Replace "an allocatable array" by "allocatable"] 244:14

[Editor: Replace "a pointer or an allocatable array" by "allocatable or a pointer" twice.] 244:26-27

(b) A dummy argument that is allocatable, an assumed shape array, a pointer, or a target, 245:21-23

The constraint at line 23 is not needed, because of the constraint at 53:25-26. J3 note

Except for the case of the character length parameter of an actual argument of type default character associated with a dummy argument that is not assumed shape, the type parameters of an actual argument that correspond to nondeferred nonassumed type parameters of the associated dummy argument shall have the same values as corresponding type parameters of the associated dummy argument. 254:29-31

In the most generally allowed case, verifying that the value of a nondeferred nonassumed type parameter of a dummy argument is the same as the corresponding type parameter of the associated actual argument requires a run-time check. If the check fails, there's no way to trap it. This is in the same category as array bounds checking. A subscript out-of-bounds is an indication that the program is not standard-conforming; so is a mismatch of actual and dummy argument type parameters.
Should this remark be in appendix C, or is it enough just to have it transiently for ourselves? J3 note (not an edit)

If a dummy argument that does not have INTENT(OUT) has deferred or assumed type parameters, the initial values of those parameters are assumed from the values of the corresponding type parameters of the associated actual argument.

If the dummy argument that does not have INTENT(IN) has deferred type parameters (and is therefore allocatable or a pointer), it may be re-allocated using type parameter values different from the original deferred parameter values of the associated actual argument. If it is a pointer, it may acquire type parameter values different from the original deferred parameter values of the associated actual argument by pointer assignment. Note 12.19$\frac{1}{2}$

An actual argument shall have deferred the same type parameters as the corresponding dummy argument.

[Editor: Delete] 255:15-31

The above text makes it clear that deferred parameters *can* be changed if the dummy argument doesn't have INTENT(IN). The intent *is* that actual and dummy arguments shall have deferred the same type parameters – even in the INTENT(IN) case. If you really want to use deferred parameters just to get at the values of the actual argument's parameters, you should use assumed parameters instead. This is not precisely symmetrical to the case of pointer assignment because when you do a pointer assignment, and then later re-allocate either the *pointer-object* or the *target*, the other one isn't affected. This is not true in the case of actual and dummy arguments – if you re-allocate the dummy argument, the associated actual argument gets re-allocated, so it better have deferred at least all of the deferred parameters of the dummy argument. Specifying concrete values for parameters of the dummy that correspond to deferred parameters of the actual is just an opportunity for them to disagree. If you want to enforce a specific value for a parameter of the dummy that corresponds to a deferred parameter of the actual, defer the dummy's parameter, and check its value explicitly. If you want to set a deferred parameter of the actual argument to a specific value during allocation, defer the corresponding parameter of the dummy and put the desired value in the allocate statement, not the type declaration for the dummy argument.
Should this remark be in appendix C, or is it enough just to have it transiently for ourselves? J3 note (not an edit)

[Editor: Replace "an allocatable array" by "allocatable" twice.] 257:18

[Editor: "non-deferred" is used here, but "nondeferred" and "nonkind" are used elsewhere. Which do you prefer?] 257:19

Type parameters of the result value of an actual argument that is associated with a dummy function procedure or dummy function procedure pointer shall agree in the following ways: 258:31+

(1) Type parameters of the result value of the actual argument that correspond to nondeferred nonassumed parameters of the result value of the dummy function procedure or dummy function procedure pointer shall have the same values.

(2) The result values of the actual argument and the dummy function procedure or dummy function procedure pointer shall have deferred the same type parameters.

[Editor: Remove the first sentence.] 267:32-33

[Editor: Add a new section] 278:15+

13.9 Allocation status inquiry functions

The inquiry function ALLOCATED tests whether an allocatable variable is currently allocated.

[Editor: Delete this line. Moved to 283:9+.] 282:31

[Editor: Add a new section] 283:9+

13.12.20 Allocation status inquiry functions

ALLOCATED (ARRAY)	Array allocation status
ALLOCATED (A)	Scalar variable allocation status

[Editor: Add a new section] 287:39+

13.15.10 ALLOCATED (A)

Description. Indicate whether or not an allocatable scalar is allocated.

Class. Inquiry function.

Argument. A shall be an allocatable scalar.

Result Characteristics. Default logical scalar.

Result Value. The result has the value true if A is currently allocated and has the value false if A is not currently allocated.

ARRAY	may be of any type. It shall not be a scalar. If it is an unallocated allocatable array, it shall have explicit shape.	306:12-13

[Editor: Add the sentence "If it is an unallocated allocatable variable, it shall not have deferred length."] 306:35

[Editor: Replace "array" by "object"] 318:15

[Editor: Remove the first "array" and replace the second one by "object"] 318:23

Argument. SOURCE may be of any type. It may be array valued or a scalar. If it is an unallocated allocatable array, it shall have explicit shape. It shall not be an assumed size array. 326:31-33

ARRAY	may be of any type. It shall not be a scalar. If it is an unallocated allocatable array, it shall have explicit shape. If ARRAY is an assumed size array, DIM shall be present with a value less than the rank of ARRAY.	327:37-40
ARRAY	may be of any type. It shall not be a scalar. If it is an unallocated allocatable array, it shall have explicit shape. If ARRAY is an assumed size array, DIM shall be present with a value less than the rank of ARRAY.	332:34-37

(5) An *object-name* in an *allocate-stmt*; 343:38

($5\frac{1}{2}$) An *array-name* in a *dimension-stmt*;

[Editor: Replace "array" by "object"] 353:4

allocatable variable (5.1.2.4.3): A *variable* having the ALLOCATABLE attribute. It may be *referenced* or *defined* only when it has space allocated. If it is an array, it has a *shape* only when it has space allocated. An allocatable variable may be a *named* variable or a *structure component*. 385:12-14

[Editor: Replace "array" by "variable"] 387:31

[Editor: Replace "a pointer or allocatable array" by "allocatable or a pointer"] 388:4

C.11.1.4 Automatic arrays and allocatable variables (5.1, 5.1.2.4.3) 436:1-10

A major advance for writing modular software is the presence of automatic arrays, created on entry to a subprogram and destroyed on return, and allocatable variables, including arrays whose rank is fixed but whose actual size and lifetime is fully under the programmer's control through explicit ALLOCATE and DEALLOCATE statements. The declarations

```
SUBROUTINE X (N, A, B)
...
REAL WORK (N, N); REAL, ALLOCATABLE :: HEAP (:, :)
```

specify an automatic array WORK and an allocatable array HEAP. Note that a stack is an adequate storage mechanism for the implementation of automatic arrays, but a heap will be needed for allocatable variables.