# Section 6: Use of data objects

The appearance of a data object designator in a context that requires its value is termed a reference. A reference is permitted only if the data object is defined. A reference to a pointer is permitted only if the pointer is associated with a target object that is defined. A data object becomes defined with a value when the data object designator appears in certain contexts and when certain events occur (14.7).

R601 *variable* **is** *designator*

Constraint: *designator* shall not be a constant or a subobject of a constant.

R602 *designator* **is** *object-name*
**or** *array-element*
**or** *array-section*
**or** *structure-component*
**or** *substring*

R603 *logical-variable* **is** *variable*

Constraint: *logical-variable* shall be of type logical.

R604 *default-logical-variable* **is** *variable*

Constraint: *default-logical-variable* shall be of type default logical.

R605 *char-variable* **is** *variable*

Constraint: *char-variable* shall be of type character.

R606 *default-char-variable* **is** *variable*

Constraint: *default-char-variable* shall be of type default character.

R607 *int-variable* **is** *variable*

Constraint: *int-variable* shall be of type integer.

R608 *default-int-variable* **is** *variable*

Constraint: *default-int-variable* shall be of type default integer.

A literal constant is a scalar denoted by a syntactic form, which indicates its type, type parameters, and value. A named constant is a constant that has been associated with a name with the PARAMETER attribute (5.1.2.1, 5.2.9). A reference to a constant is always permitted; redefinition of a constant is never permitted.

**NOTE 6.1**

For example, given the declarations:

```
CHARACTER (10)  A, B (10)
TYPE (PERSON)  P   ! See Note 4.20
```

then A, B, B (1), B (1:5), P % AGE, and A (1:1) are all variables.

## 6.1 Scalars

A **scalar** (2.4.4) is a data entity that can be represented by a single value of the data type and that is not an array (6.2). Its value, if defined, is a single element from the set of values that characterize its data type.

**NOTE 6.2**

A scalar object of derived type has a single value that consists of values of the data types of its components (4.5.4).

A scalar has rank zero.

### 6.1.1 Substrings

A **substring** is a contiguous portion of a character string (4.4.4). The following rules define the forms of a substring:

R609 *substring* **is** *parent-string* ( *substring-range* )

R610 *parent-string* **is** *scalar-variable-name*
**or** *array-element*
**or** *scalar-structure-component*
**or** *scalar-constant*

R611 *substring-range* **is** [ *scalar-int-expr* ] : [ *scalar-int-expr* ]

Constraint: *parent-string* shall be of type character.

The first *scalar-int-expr* in *substring-range* is called the **starting point** and the second one is called the **ending point**. The length of a substring is the number of characters in the substring and is MAX ($l - f + 1$, 0), where $f$ and $l$ are the starting and ending points, respectively.

Let the characters in the parent string be numbered 1, 2, 3, ..., $n$, where $n$ is the length of the parent string. Then the characters in the substring are those from the parent string from the starting point and proceeding in sequence up to and including the ending point. Both the starting point and the ending point shall be within the range 1, 2, ..., $n$ unless the starting point exceeds the ending point, in which case the substring has length zero. If the starting point is not specified, the default value is 1. If the ending point is not specified, the default value is $n$.

If the parent is a variable, the substring is also a variable.

**NOTE 6.3**

Examples of character substrings are:

```
B(1)(1:5)           array element as parent string
P%NAME(1:1)         structure component as parent string
ID(4:9)             scalar variable name as parent string
'0123456789'(N:N)   character constant as parent string
```

### 6.1.2 Structure components

A **structure component** is part of an object of derived type; it may be referenced by an object designator. A structure component may be a scalar or an array.

R612 *data-ref* **is** *part-ref* [ *% part-ref* ] ...

R613 *part-ref* **is** *part-name* [ ( *section-subscript-list* ) ]

Constraint: In a *data-ref*, each *part-name* except the rightmost shall be of derived type.

Constraint: In a *data-ref*, each *part-name* except the leftmost shall be the name of a component of the derived-type definition of the declared type of the preceding *part-name*.

Constraint: In a *part-ref* containing a *section-subscript-list*, the number of *section-subscript*s shall equal the rank of *part-name*.

The rank of a *part-ref* of the form *part-name* is the rank of *part-name*. The rank of a *part-ref* that has a section subscript list is the number of subscript triplets and vector subscripts in the list.

Constraint: In a *data-ref*, there shall not be more than one *part-ref* with nonzero rank. A *part-name* to the right of a *part-ref* with nonzero rank shall not have the ALLOCATABLE or POINTER attribute.

The rank of a *data-ref* is the rank of the *part-ref* with nonzero rank, if any; otherwise, the rank is zero. The **base object** of a *data-ref* is the data object whose name is the leftmost part name.

A *data-ref* with more than one *part-ref* is a subobject of its base object if none of the *part-names*, except for possibly the rightmost, are pointers. If the rightmost *part-name* is the only pointer, then the *data-ref* is a subobject of its base object when used in contexts that pertain to its pointer association, but not when used in contexts where it is dereferenced to refer to its target.

**NOTE 6.4**

If X is an object of derived type with a pointer component P, then the pointer X%P is a subobject of X when considered as a pointer - that is in contexts where it is not dereferenced.

However the target of X%P is not a subobject of X. Thus, in contexts where X%P is dereferenced to refer to the target, it is not a subobject of X.

**J3 internal note**

Unresolved issue 6

We need to specify here how allocatable components interact with the term "subobject". My concept is that an ultimate component that is allocatable is always a subobject, regardless of whether it is currently allocated or not; but the contents of an allocatable component are subobjects only when the component is allocated - otherwise they don't exist and thus can't be said to be subobjects. I haven't yet worded this in standardese.

Also need to consider the relationship between subobjects and ultimate components.

Malcolm has some other ideas, which may work out better, including an intriguing idea of treating unallocated allocatable components as being more like zero-sized objects. This whole subject needs work.

R614 *structure-component* **is** *data-ref*

Constraint: In a *structure-component*, there shall be more than one *part-ref* and the rightmost *part-ref* shall be of the form *part-name*.

The type and type parameters, if any, of a structure component are those of the rightmost part name. A structure component shall be neither referenced nor defined before the declaration of the base object. A structure component is a pointer only if the rightmost part name is defined to have the POINTER attribute.

**NOTE 6.5**

Examples of structure components are:

```
SCALAR_PARENT%SCALAR_FIELD       scalar component of scalar parent
ARRAY_PARENT(J)%SCALAR_FIELD     component of array element parent
ARRAY_PARENT(1:N)%SCALAR_FIELD   component of array section parent
```

For a more elaborate example see C.3.1.

**NOTE 6.6**

The syntax rules are structured such that a *data-ref* that ends in a component name without a following subscript list is a structure component, even when other component names in the *data-ref* are followed by a subscript list. A *data-ref* that ends in a component name with a following subscript list is either an array element or an array section. A *data-ref* of nonzero rank that ends with a *substring-range* is an array section. A *data-ref* of zero rank that ends with a *substring-range* is a substring.

### 6.1.3 Type parameter inquiry

A **type parameter inquiry** is used to inquire about a type parameter of a data object. It applies to both intrinsic and derived data types.

R615 *type-param-inquiry* **is** *designator* % *type-param-name*

Constraint: The *type-param-name* shall be the name of a type parameter of the object designated by the *designator.*

A deferred type parameter of a disassociated pointer, a function procedure pointer, an unallocated variable, or a pointer with undefined association status shall not be referenced.

**J3 internal note**

Unresolved issue 140

What does it mean to "reference" a type parameter in 6.1.3? I don't think we use the term "reference" for such inquiries in any other context. The first sentence of 6.1.3 certainly doesn't use such terminology.

Also, the parsing of the sentence before this note is ambiguous; you probably intend the "deferred type parameter of" to apply to all of the elements of the list, but this is not clear, particularly since the restriction makes sense both ways. One way is covered elsewhere and has little relevance to this section, so I assume its the other way you mean to cover here.

The editor could probably fix both of these problems, but has too much other work to do.

**NOTE 6.7**

A *type-param-inquiry* has a syntax like that of a structure component reference, but it does not have the same semantics. It is not a variable and thus can never be assigned to. It may be used only as a primary in an expression.

The intrinsic type parameters can also be inquired about by using the intrinsic functions KIND and LEN.

**NOTE 6.8**

The following are examples of type parameter inquiries:

```
a%kind      !-- A is real.  Equivalent to KIND(a).
s%len       !-- S is character.  Equivalent to LEN(s).
b(10)%kind  !-- Inquiry about an array component.
p%dim       !-- P is of the derived type general_point
            !-- defined in Note 4.21.
```

## 6.2 Arrays

An **array** is a set of scalar data, all of the same type and type parameters, whose individual elements are arranged in a rectangular pattern. The scalar data that make up an array are the **array elements**.

No order of reference to the elements of an array is indicated by the appearance of the array designator, except where array element ordering (6.2.2.2) is specified.

### 6.2.1 Whole arrays

A **whole array** is a named array, which may be either a named constant (5.1.2.1, 5.2.9) or a variable; no subscript list is appended to the name.

The appearance of a whole array variable in an executable construct specifies all the elements of the array (2.4.5). An assumed-size array is permitted to appear as a whole array in an executable construct only as an actual argument in a procedure reference that does not require the shape.

The appearance of a whole array name in a nonexecutable statement specifies the entire array except for the appearance of a whole array name in an equivalence set (5.5.1.3).

### 6.2.2 Array elements and array sections

R616 *array-element* **is** *data-ref*

Constraint: In an *array-element*, every *part-ref* shall have rank zero and the last *part-ref* shall contain a *subscript-list*.

R617 *array-section* **is** *data-ref* [ ( *substring-range* ) ]

Constraint: In an *array-section*, exactly one *part-ref* shall have nonzero rank, and either the final *part-ref* shall have a *section-subscript-list* with nonzero rank or another *part-ref* shall have nonzero rank.

Constraint: In an *array-section* with a *substring-range*, the rightmost *part-name* shall be of type character.

R618 *subscript* **is** *scalar-int-expr*

R619 *section-subscript* **is** *subscript*
**or** *subscript-triplet*
**or** *vector-subscript*

R620 *subscript-triplet* **is** [ *subscript* ] : [ *subscript* ] [ : *stride* ]

R621 *stride* **is** *scalar-int-expr*

R622 *vector-subscript* **is** *int-expr*

Constraint: A *vector-subscript* shall be an integer array expression of rank one.

Constraint: The second subscript shall not be omitted from a *subscript-triplet* in the last dimension of an assumed-size array.

An array element is a scalar. An array section is an array. If a *substring-range* is present in an *array-section*, each element is the designated substring of the corresponding element of the array section.

**NOTE 6.9**

For example, with the declarations:

```
REAL A (10, 10)
CHARACTER (LEN = 10) B (5, 5, 5)
```

A (1, 2) is an array element, A (1:N:2, M) is a rank-one array section, and B (:, :, :) (2:3) is an array of shape (5, 5, 5) whose elements are substrings of length 2 of the corresponding elements of B.

An array element or an array section never has the POINTER attribute.

**NOTE 6.10**

Examples of array elements and array sections are:

```
ARRAY_A(1:N:2)%ARRAY_B(I, J)%STRING(K)(:)     array section
SCALAR_PARENT%ARRAY_FIELD(J)                  array element
SCALAR_PARENT%ARRAY_FIELD(1:N)                array section
SCALAR_PARENT%ARRAY_FIELD(1:N)%SCALAR_FIELD   array section
```

#### 6.2.2.1 Array elements

The value of a subscript in an array element shall be within the bounds for that dimension.

#### 6.2.2.2 Array element order

The elements of an array form a sequence known as the **array element order**. The position of an array element in this sequence is determined by the subscript order value of the subscript list designating the element. The subscript order value is computed from the formulas in Table 6.1.

**Table 6.1 Subscript order value**

| Rank | Subscript bounds | Subscript list | Subscript order value |
|---|---|---|---|
| 1 | $j_1{:}k_1$ | $s_1$ | $1 + (s_1 - j_1)$ |
| 2 | $j_1{:}k_1, j_2{:}k_2$ | $s_1, s_2$ | $1 + (s_1 - j_1) + (s_2 - j_2) \times d_1$ |
| 3 | $j_1{:}k_1, j_2{:}k_2, j_3{:}k_3$ | $s_1, s_2, s_3$ | $1 + (s_1 - j_1) + (s_2 - j_2) \times d_1 + (s_3 - j_3) \times d_2 \times d_1$ |
| . | . | . | . |
| . | . | . | . |
| . | . | . | . |
| 7 | $j_1{:}k_1, \ldots, j_7{:}k_7$ | $s_1, \ldots, s_7$ | $1 + (s_1 - j_1) + (s_2 - j_2) \times d_1 + (s_3 - j_3) \times d_2 \times d_1 + \ldots + (s_7 - j_7) \times d_6 \times d_5 \times \ldots \times d_1$ |

Notes for Table 6.1:
1) $d_i = \max(k_i - j_i + 1, 0)$ is the size of the $i$th dimension.
2) If the size of the array is nonzero, $j_i \le s_i \le k_i$ for all $i = 1, 2, ..., 7$.

#### 6.2.2.3 Array sections

An **array section** is an array subobject optionally followed by a substring range.

In an *array-section* having a *section-subscript-list*, each *subscript-triplet* and *vector-subscript* in the section subscript list indicates a sequence of subscripts, which may be empty. Each subscript in such a sequence shall be within the bounds for its dimension unless the sequence is empty. The array section is the set of elements from the array determined by all possible subscript lists obtainable from the single subscripts or sequences of subscripts specified by each section subscript.

In an *array-section* with no *section-subscript-list*, the rank and shape of the array is the rank and shape of the *part-ref* with nonzero rank; otherwise, the rank of the array section is the number of subscript triplets and vector subscripts in the section subscript list. The shape is the rank-one array whose *i*th element is the number of integer values in the sequence indicated by the *i*th subscript triplet or vector subscript. If any of these sequences is empty, the array section has size zero. The subscript order of the elements of an array section is that of the array data object that the array section represents.

##### 6.2.2.3.1 Subscript triplet

A subscript triplet designates a regular sequence of subscripts consisting of zero or more subscript values. The third expression in the subscript triplet is the increment between the subscript values and is called the **stride**. The subscripts and stride of a subscript triplet are optional. An omitted

first subscript in a subscript triplet is equivalent to a subscript whose value is the lower bound for the array and an omitted second subscript is equivalent to the upper bound. An omitted stride is equivalent to a stride of 1.

The second subscript shall not be omitted in the last dimension of an assumed-size array.

When the stride is positive, the subscripts specified by a triplet form a regularly spaced sequence of integers beginning with the first subscript and proceeding in increments of the stride to the largest such integer not greater than the second subscript; the sequence is empty if the first subscript is greater than the second.

The stride shall not be zero.

**NOTE 6.11**

For example, suppose an array is declared as A (5, 4, 3). The section A (3 : 5, 2, 1 : 2) is the array of shape (3, 2):

```
A (3, 2, 1)     A (3, 2, 2)
A (4, 2, 1)     A (4, 2, 2)
A (5, 2, 1)     A (5, 2, 2)
```

When the stride is negative, the sequence begins with the first subscript and proceeds in increments of the stride down to the smallest such integer equal to or greater than the second subscript; the sequence is empty if the second subscript is greater than the first.

**NOTE 6.12**

For example, if an array is declared B (10), the section B (9 : 1 : –2) is the array of shape (5) whose elements are B (9), B (7), B (5), B (3), and B (1), in that order.

**NOTE 6.13**

A subscript in a subscript triplet need not be within the declared bounds for that dimension if all values used in selecting the array elements are within the declared bounds.

For example, if an array is declared as B (10), the array section B (3 : 11 : 7) is the array of shape (2) consisting of the elements B (3) and B (10), in that order.

#### 6.2.2.3.2 Vector subscript

A **vector subscript** designates a sequence of subscripts corresponding to the values of the elements of the expression. Each element of the expression shall be defined. A **many-one array section** is an array section with a vector subscript having two or more elements with the same value. A many-one array section shall appear neither on the left of the equals in an assignment statement nor as an input item in a READ statement.

An array section with a vector subscript shall not be argument associated with a dummy array that is defined or redefined. An array section with a vector subscript shall not be the target in a pointer assignment statement. An array section with a vector subscript shall not be an internal file.

**NOTE 6.14**

For example, suppose Z is a two-dimensional array of shape (5, 7) and U and V are one-dimensional arrays of shape (3) and (4), respectively. Assume the values of U and V are:

```
U = (/ 1, 3, 2 /)
V = (/ 2, 1, 1, 3 /)
```

Then Z (3, V) consists of elements from the third row of Z in the order:

```
Z (3, 2)   Z (3, 1)   Z (3, 1)   Z (3, 3)
```

and Z (U, 2) consists of the column elements:

```
Z (1, 2)   Z (3, 2)   Z(2, 2)
```

and Z (U, V) consists of the elements:

```
Z (1, 2)   Z (1, 1)   Z (1, 1)   Z (1, 3)
Z (3, 2)   Z (3, 1)   Z (3, 1)   Z (3, 3)
Z (2, 2)   Z (2, 1)   Z (2, 1)   Z (2, 3)
```

Because Z (3, V) and Z (U, V) contain duplicate elements from Z, the sections Z (3, V) and Z (U, V) shall not be redefined as sections.

## 6.3 Initialization and Finalization

**J3 internal note**

Unresolved issue 190

I have not had time to do a careful job on the edits from paper 99-108r2. I have just imported large parts of it (most of section 6.3) as is, without taking much time to actually think about it. I have't even really reviewed the wording and editorial issues, much less the technical details. I'm playing typist here. (At least the rtf format imported into Frame pretty easily, making that part go relatively quickly). There is too much that is too new and too late for me to have the time. This J3 note is more to flag the section as being not as well reviewed as most than to point out specific issues. I'll just mention a few quick things that occur to me without study. (Later...I did end up doing several other J3 notes, but they still don't reflect my usual level of care).

I'm not entirely pleased with the term "finalization". Yes, the paper mentions it as a placeholder. No, I don't have a great alternative to present. "Destruction" (along with "destructor") has some attraction, even though we don't currently use the term "constructor" (because of conflicts with other terminology). I would consider the possibility of using a different term for our existing array and derived type constructors so that we could use that term here. I count only about 80 lines where the term is used, 65 of them in c4 and c7. Verifying each for a global change does not look like an overwhelming task...if someone comes up with a good candidate wording. Or we could just keep "initialization" and pair it with "destruction." To me, finalization sounds too much like making it finally ready for use instead of no longer useable. At work if I finalize a plan or a report or a program, that doesn't mean that I throw it away; it means that it is now complete and ready for use. (It also means that I've given in to a word popular with bureaucrats, but described by Strunk and White as "A pompous, ambiguous verb"). I'm concerned that this will be confusing. I have no great problems with "initialization." Later... I notice that 6.3.3.1 talks about creation and destruction of scoping units. I could see using creation/destruction or initialization/destruction of objects.

On another matter of terminology, we appear to have introduced two different meanings of the term "linkage" in simultaneous papers. One in C interop, and one here. I don't think this wise. One of them had better change.

I'm not at all sure that the business about content vs linkage is really helping in this section. I might suggest just integrating them in as extra steps in initialization. And there can be words about when some of the steps are skipped (like when pointers are disassociated, etc).

As I briefly mentioned in discussion, I consider it excessive to have 6 levels of numbered headings. I had to make a whole new paragraph type in Frame for this because there is not a single other place in the document where there are that many levels. Indeed, I tend to be somewhat dubious of many of the places where there are 5 levels, considering it a sign that perhaps the organization is questionable in such areas.

Speaking of headings, I'm a bit leary of the headings beginning with "How". I don't think "how" is really an appropriate term here. That sounds like we are talking about implementation details or something. In a quick scan, I can't find any other headings in the document that begin with "how". We certainly don't have "How DO loops are executed", etc.

I'm concerned that it may not always be completely clear when requirement are being made on programs versus processors. Most (but not all) of this section seems to be describing requirements that processors shall follow. Section 1.6 says that we explicitly say when we are specifying requirements of processors. We should do so.

I'd generally say that the whole section has an identifiably different style from much of the rest of the standard. Its a little hard to say exactly why. It is also hard to say whether this is necessarily a bad thing. (We may be working on one of the great classics of programming languages, but the standard has a long ways to go to be considered a classic of literature).

Several of the sentences pain me in their length, complexity, and resulting confusions about what modifies what, but that kind of thing takes too much time for me to fix.

**Initialization** is the process by which the status of a data object is established prior to any explicit operations on that object. **Finalization** of a data object is the processing that follows the explicit operations on that object. Every data object in a program undergoes both initialization and finalization. Data objects shall not be referenced, defined, or used in any other way before they are initialized or after they are finalized.

**NOTE 6.15**

Although every data object undergoes the conceptual processes of initialization and finalization, for most objects those processes involve no operations and thus have no affect on the object.

### 6.3.1 How initialization is performed

Initialization of an object consists of two steps, performed in order:

(1) **Preliminary initialization** establishes a default status for the object, based on its declared attributes.

(2) **Overriding initialization** may alter that status, based on explicit initialization declared for the object.

#### 6.3.1.1 Preliminary initialization

The effect of preliminary initialization depends on whether it is the content of the object that is being initialized or the status of its linkage to content. Initialization of object content is performed if one of the following is true:

**J3 internal note**

Unresolved issue 182

Why the "effect" of initialization? Seems to me that we are talking about different initializations happening, not just the initialization having different effects. SImillar comment about finalization later.

(1) The object has neither the ALLOCATABLE nor the POINTER attribute.

(2) The object has either the ALLOCATABLE or the POINTER attribute and is being allocated in an ALLOCATE statement.

Initialization of object linkage is performed if the object has either the ALLOCATABLE or the POINTER attribute and is not being allocated in an ALLOCATE statement.

##### 6.3.1.1.1 Preliminary initialization of object content

Preliminary initialization of object content is performed on each element of the entity (treating a scalar entity as a single element), based on the type of the entity:

(1) If the data type is one of the intrinsic types INTEGER, REAL, COMPLEX, or LOGICAL, the element is undefined.

(2) If the data type is the intrinsic type CHARACTER, each character position in the element is undefined.

(3) If the data type is a derived type, the following steps are performed:

(a) If the data type is an extended type, initialization is performed (recursively) for the parent component subobject of the element.

**J3 internal note**

Unresolved issue 178

It is editorially unclear whether there really is a parent component or only a notation that looks like such a component exists. (This is unresolved issue 19.) Depending on which way this is resolved, this text may need to be changed to say something like the "components from the parent type" instead of the parent component and to allow for their default initialization.

(b) Initialization is performed (recursively) for the explicitly declared component subobjects of the element, in the order those components were declared. Default initialization (4.5) for a component is treated like explicit initialization for the recursive initialization of the component subobject.

**NOTE 6.16**

The preliminary initialization of an object of derived type includes both the preliminary and the overriding initialization of its component subobjects.

#### 6.3.1.1.2 Preliminary initialization of object linkage

If an object has the ALLOCATABLE attribute, preliminary initialization of its linkage affects only its allocation status (6.4.1.2). The allocation status becomes not currently allocated.

If an object has the POINTER attribute, preliminary initialization of its linkage affects only its pointer association status (14.6.2.1). The pointer association status becomes undefined.

### 6.3.1.2 Override initialization

The effect of override initialization also depends on whether initialization of object content or object linkage is being performed (6.3.1.1). Additionally, it depends on which of the following cases is applicable:

**J3 internal note**

Unresolved issue 179

I can't follow this. It seems organizationally confused. Is this actually trying to tell me what happens in each of the 3 cases listed, or is it just introductory material trying to say that these are three things that will be influences in ways to be described below. My first inclination was that you were going to tell me what happens in each of the 3 cases. That's what I'd normally expect when seeing something like this. But it seemed woefully incomplete. Most notably item (1) (which doesn't even say that nothing happens - its just empty), but the other cases also left me feeling like thay hadn't really said what happens.

If its just to say that these three cases will be different in ways described below, then it should *JUST* say that. Instead, it looks like it tries to say that, plus mix in a little bit (but not all) of material about each case. I find this a very confusing presentation style.

(1) There is no explicit initialization (5.1) for the object.

(2) Explicit initialization for the object occurs in the type declaration statement for the object. Such explicit initialization always applies to the entire object.

This case is also considered to apply if the initialization being performed is the recursive initialization of a component subobject, as described in 6.3.1.1, and a default initialization is specified in the component declaration statement for that component. Such an initialization applies to the entire component subobject.

(3) Explicit initialization for the object occurs in at least one DATA statement (5.2.13). Each explicit initialization in a DATA statement applies to a scalar object, possibly a

subobject of a named object. There may be subobjects of the object for which no explicit initialization is provided.

#### 6.3.1.2.1 Override initialization of object content

The effect of override initialization of object content further depends on whether the type of the object is one for which an initial procedure (4.5.1.5) has been specified.

##### 6.3.1.2.1.1 Override by assignment

If the type has no initial procedures, override initialization is handled in accordance with the rules of intrinsic assignment.

(1) If there is no explicit initialization, the status of the object remains as established by preliminary initialization.

(2) If the object is explicitly initialized in a type declaration statement or component definition statement, that initialization is assigned to the object, as a whole, in accordance with the rules of intrinsic assignment.

(3) If the object is explicitly initialized in DATA statements, the identified scalar objects (possibly elements or component subobjects of the object) are assigned their corresponding initial values in accordance with the rules of intrinsic assignment. Any portion of the object not identified in a DATA statement retains its status as established by preliminary initialization.

##### 6.3.1.2.1.2 Override by initial procedure

If the type has at least one initial procedure, execution of initial procedures replaces intrinsic assignment in override initialization.

(1) If there is no explicit initialization, override initialization is performed by invoking an initial procedure for this type with an argument list that consists solely of the object being initialized, or, if no such procedure exists, by the object retaining its status as established by preliminary initialization.

(2) If the object is explicitly initialized in a type declaration statement or component definition statement, the initialization shall consist of a single defined structure constructor (4.5.6.2) for the type. Override initialization is performed by invoking an initial procedure for the type with an argument list consisting of the object itself followed by the argument list from the structure constructor.

**J3 internal note**

Unresolved issue 180

We seem to be simultaneously talking about initialization as a process and as a piece of syntax. It probably happens throughout this section, but it first really hit me in the above para, where we say that "initialization consists of a single defined structure constructor", while we have been talking about the process of initialization. The first sentence of 6.3 says "Initialization is the process...". I don't think a structure constructor is a process.

(3) If the object is explicitly initialized in DATA statements, those initializations shall not be for component subobjects of the object. Override initialization is performed separately for each element of the object. Elements for which an initialization is provided are processed as in (2). Elements for which no initialization is provided are processed as in (1).

#### 6.3.1.2.2 Override initialization of object linkage

If an object has the ALLOCATABLE attribute, explicit initialization of its linkage is prohibited. The status of the object remains as established by preliminary initialization.

If an object has the POINTER attribute, the only permitted initialization is to NULL( ). If such an explicit initialization is present, the pointer association status of the object becomes disassociated. Otherwise, the status of the object remains as established by preliminary initialization (i.e. undefined).

**NOTE 6.17**

Since any explicit initialization for a pointer applies to its linkage, not its content, elements allocated through a pointer are always treated as having no explicit initialization for purposes of determining the nature of their override initialization.

**J3 internal note**

Unresolved issue 181

I don't understand note 6.17 in 6.3.1.2.2. Admit I didn't spend very long trying to puzzle it out. My first confusion was the word "elements". I'm not sure what kind of elements these are. Array elements are the only elements that occur to me off-hand, but I'm mostly just lost here. Perhaps if I could figure out what elements of what we were talking about, the rest might be more clear. We could then go on to explain the "allocated through a pointer" part. Am I allocating through a pointer when I allocate a%b%c%d where a%b is a pointer?

Um, say. I see there is more of this "element" stuff later. Are we trying to introduce a synonym for component without actually saying so? Or do we really mean array element (and if so, is a scalar considered to be an element of itself)? Or is it something more subtle?

### 6.3.2 How finalization is performed

As with initialization, the effect of finalization depends on whether finalization of object content or object linkage is being performed. Finalization of object content is performed for objects not having the ALLOCATABLE or POINTER attribute and for objects having one of those attributes and being deallocated in a DEALLOCATE statement. Otherwise, finalization of object linkage is performed for objects having the ALLOCATABLE or POINTER attribute.

#### 6.3.2.1 Finalization of object content

If the object is of intrinsic type, finalization of the object involves no further operations.

If the object is of derived type, finalization consists of the following steps, performed in order:

(1) If the type has at least one final procedure, a final procedure is invoked with an argument list that consists solely of the object being finalized.

A standard-conforming program is required to provide for the finalization of each object that has been initialized and whose finalization would include the execution of a final procedure.

**J3 internal note**

Unresolved issue 183

I'm sure glad there is a note explaining this (6.3.2.1(1). I'd be happier if I thought that the normative text supported what the note said. I know of no normative definition of "provide for" and I sure can't deduce that it means what the note says.

And as I read it (admitedly very poorly), this says that we can't use structure constructors for any type that has a final procedure, because the structure constructor creates an anonymous object that we have no way of "providing for the finalization of". Simillar problems with other anonymous objects. Maybe I just misread it. Or maybe it doesn't really say, but rather assumes that it will just be interpreted to mean "do the right thing."

And I'm left in the air when this goes out of its way to note that there might be multiple final procedures and then just tells me that an unspecified one of them will be invoked. I suppose there must be more on this somewhere. Can we at least have a hint or an xref?

I'm also quite surprised to see that we feel it necessary to say that this requirement applies only to standard-conforming programs. I guess I thought that's what the whole document was about. Does the standard have any requirements that apply to non-standard-conforming programs?

**NOTE 6.18**

For most objects this requirement is fulfilled automatically. The effect of this requirement on a program is that it must deallocate any object it allocates through a POINTER if the type is one with a final procedure.

It is not expected that processors will enforce this requirement. Rather, it allows processors, when faced with a program that violates this requirement, to process it as it wishes. Reasonable handling might be to allow the program to "leak memory" and not finalize the objects so leaked or to perform "garbage collection" to recover such objects so they can be finalized and deallocated.

(2) Each element of the object is processed as follows:

(a) Finalization is performed (recursively) for the explicitly declared component subobjects of the element, in the reverse of the order those components were declared.

(b) If the data type is an extended type, finalization is performed (recursively) for the parent component subobject of the element.

**J3 internal note**

Unresolved issue 184

See note in section 6.3.1.1.1 about parent component subobject.

### 6.3.2.2 Finalization of object linkage

If an object has the ALLOCATABLE attribute, finalization of its linkage consists of checking whether its allocation status is currently allocated and, if so, finalizing its content and deallocating the object.

If an object has the POINTER attribute, finalization of the object involves no further operations.

**J3 internal note**

Unresolved issue 185

Further than what? Is it simply that finalization of the linkage involves no operations, or is it really that it involves no further operations in addition to some other ones that I haven't figured out? And why did we switch from talking about finalization of the linkage to finalization of the object?

### 6.3.3 When initialization and finalization are performed

Initialization and finalization are performed collectively for the data objects declared in a scoping unit. Additional initialization and finalization may result from the execution of specific statements.

#### 6.3.3.1 Initialization and finalization of scoping units

Scoping unit initialization and finalization are associated respectively with the creation and destruction of instances of the main program, subprograms, and modules.

An instance of the main program is created immediately before it is executed and destroyed when a STOP statement or the END PROGRAM statement is executed.

When a procedure defined by a subprogram is invoked from an instance of a subprogram or the main program, an instance of that subprogram is created. That instance is said to **derive** from the instance that invoked the procedure. The derived instance is destroyed when its execution completes.

If a subprogram or main program directly or indirectly references a module, each instance of the subprogram or main program accesses entities from an instance of the module determined as follows: If the instance of a subprogram derives, directly or indirectly, from an instance of a scoping unit that directly or indirectly references the same module, it accesses entities from the same instance of the module as the instance from which it derives. Otherwise, a new instance of the module is created; this new instance of the module is the one accessed by the instance of the subprogram or main program; it is not destroyed until after the instance of the subprogram or main program is destroyed.

If a subprogram or main program contains data objects in a named common block or directly or indirectly references a module containing such data objects, each instance of the subprogram or main program accesses an instance of the storage sequence for that common block determined as follows. If the instance of a subprogram derives, directly or indirectly, from an instance of a scoping unit that contains that common block or directly or indirectly references a module containing the common block, it accesses the same instance of the storage sequence as the instance from which it derives. Otherwise, a new instance of the storage sequence is created; this new instance of the storage sequence is the one accessed by the instance of the subprogram or main program; it is not destroyed until after the instance of the subprogram or main program is destroyed.

A processor is permitted, at its discretion, to merge instances of modules and common block storage sequences that are not required to exist at the same time, eliminating the destruction (and associated finalization) of the earlier instance and the creation (and associated initialization) of the later instance.

**NOTE 6.19**

Taken to the extreme, if a processor uses a strategy in which an instance of a subprogram or the main program always invokes procedures one at a time, none of the instances of modules and common block storage sequences would be required to exist at the same time, and the processor would be permitted to merge all of the instances of a given module or common block storage sequence into a single instance. Other strategies might allow merging to result in one instance per hardware processor on a multiprocessor system.

If a subprogram is nested in another scoping unit and thus has access to entities in that scoping unit by host association, an instance of the subprogram accesses those entities from the same

instance of the host scoping unit from which the subprogram itself was accessed in order to be in invoked.

**J3 internal note**

Unresolved issue 186

Do the words "accessed in order to be" add anything here? As best as I can tell, they add nothing but complication.

Scoping unit initialization and finalization can be further divided into the initialization and finalization of objects that are distinct in separate instances of the scoping unit and the initialization and finalization of objects that are shared among all the instances of the scoping unit in the program.

#### 6.3.3.1.1 Instance initialization and finalization of scoping units

Instance initialization and finalization of a scoping unit applies to all objects local to the scoping unit that are not accessed by use association or host association and that are not dummy arguments and do not have either the SAVE or the PARAMETER attribute. Instance initialization of a scoping unit also applies to dummy arguments with the INTENT(OUT) attribute, but instance finalization does not.

**NOTE 6.20**

6.3.3.2 provides for the corresponding actual argument to be finalized immediately before invoking the procedure, so fresh initialization is appropriate. The dummy argument is not finalized during execution of the procedure, but the corresponding actual argument is eventually finalized, so everything still “balances”.

Instance initialization of a scoping unit occurs after the instance initialization of any module it references, and instance finalization occurs before the instance finalization of any referenced module.

Except during the execution of initial and final procedures, instance initialization of a nested scoping unit occurs after instance initialization of the host scoping unit and instance finalization occurs before instance finalization of the host. During execution of initial and final procedures, the instance of the host scoping unit may be only partially initialized or finalized and the program is restricted from using those objects which are not yet initialized or already finalized.

Objects in common are initialized before objects not in common and finalized after. Objects in common are not initialized if the storage sequence is not newly created. Objects in common are not finalized if the storage sequence is not about to be destroyed.

Objects in common are initialized in order of their first declaration and finalized in the reverse of that order.

**J3 internal note**

Unresolved issue 187

What does "first declaration" mean? There is an ancient Dick Weaver interp request (number 96) that we never did manage to resolve, which asks for a definition of the term "declaration." Since we are a bit hazy about what declaration means, I'm even less sure about what the first one is.

I tended to feel that part of the problem was that Fortran doesn't actually have a well-defined concept of declaration. There are just a whole bunch of things that can contribute information. And after you see the whole scoping unit, you guess what you have from the information you have collected. (As in: hmm...isn't given a dimension attribute, but is used with a parens after it - must be a function). Sort of a mess, but that's the hand we were dealt.

My general ill-defined notion of declaration in Fortran was that the declaration of an object was "something like" the collection of all its attributes, including those specified implicitly and explicitly. I didn't think you could have multiple declarations of an object. Instead I thought you might have multiple specifications that contributed to a single declaration. Guess that just shows my ignorance? (Quite possibly true).

Assuming that each of the statements that specifies an attribute of the object is a declaration of the object (I guess the intent must be something like that), is an implicit statement a declaration of all implicitly typed objects (my guess is "no"? Is the dummy argument list a declaration of the dummy arguments (my guess is "yes")? Is the first appearance of an implicitly typed variable its declaration (my guess is "yes").

This needs major work if you want to use that terminology. My recommendation is to punt and say something much easier to make concrete.

Objects not in common are initialized in order of their first declaration and finalized in the reverse of that order.

A processor is permitted, as an exception to the above orderings, to initialize function result variables and INTENT(OUT) dummy arguments before the point indicated by that order and to finalize function result variables after the point indicated.

**NOTE 6.21**

This exception allows a processor the option to put the initialization and finalization of a function result and/or the initialization of INTENT(OUT) dummy arguments in the caller of a procedure rather than the procedure itself.

**NOTE 6.22**

The purpose of these orderings is to establish which objects can and cannot be used in initial and final procedures. Initialization and finalization may be performed in other orders as long as the same effect is achieved. For example, it is generally possible to perform all initialization not involving initial procedures before that which does.

#### 6.3.3.1.2 Program initialization and finalization of scoping units

Program initialization and finalization of a scoping unit applies to all objects local to the scoping unit that are not accessed by use association or host association and that have either the SAVE or the PARAMETER attribute.

Program initialization of a scoping unit occurs before all instance initialization of that scoping unit, and program finalization occurs after all instance finalization. If there is no instance initialization or finalization of a scoping unit (e.g., because a procedure is never called or a module is unreferenced), it is processor dependent whether the program initialization and finalization for that scoping unit occurs.

**J3 internal note**

This allows the processor the choice of either unconditionally performing program initialization for all scoping units at the beginning of the program or of deferring program initialization of a scoping unit to the beginning of the first instance initialization for that scoping unit.

Program initialization of a scoping unit occurs after the program initialization of any module it references, and program finalization occurs before the program finalization of any referenced module.

Except during the execution of initial and final procedures, program initialization of a nested scoping unit occurs after program initialization of the host scoping unit and program finalization occurs after program finalization of the host. During execution of initial and final procedures, the program of the host scoping unit may be only partially initialized or finalized and the program is restricted from using those objects that are not yet initialized or already finalized.

Objects in common are initialized before objects not in common and finalized after. Objects in common are not initialized if the storage sequence is not newly created. Objects in common are not finalized if the storage sequence is not about to be destroyed.

Objects in common are initialized in order of their first declaration and finalized in the reverse of that order.

Objects not in common are initialized in order of their first declaration and finalized in the reverse of that order.

#### 6.3.3.2 Initialization and finalization resulting from statement execution

Execution of an ALLOCATE statement to allocate storage for a pointer or allocatable object causes initialization of the object content. The pointer or allocatable object linkage is not initialized by this process.

Execution of a DEALLOCATE statement to deallocate storage for a pointer or allocatable object causes finalization of the object content before it is deallocated. The pointer or allocatable object linkage is not finalized by this process.

When an expression result is passed to a procedure, it is placed in an anonymous data object that is then associated with the dummy argument. Such anonymous data objects are initialized and finalized by the processor. Initialization occurs as part of the process that defines the expression result. Function result initialization is performed by the initialization of the function result variable. Defined structure constructor initialization is performed by the initial procedure (4.5.1.5) that implements that constructor. Finalization occurs at the discretion of the processor when it no longer needs that expression result. Exactly when this occurs may depend on whether a processor reuses a computed expression result or recomputes it each time it is needed.

An object or subobject, supplied as an actual argument associated with a dummy argument with the INTENT(OUT) attribute, undergoes finalization reflecting the attributes of the dummy argument immediately before the procedure is invoked. This finalization “balances” the initialization of the dummy argument during the execution of the procedure.

**J3 internal note**

Unresolved issue 188

It bothers me when we feel the need to use quote marks around English words in normative text. The quote marks mean that the word is not being used with its actual meaning. Perhaps we should say what we mean instead of making the reader guess - particularly in the normative text.

**NOTE 6.23**

The attributes of the dummy argument control the nature of the finalization of the actual argument. For example, if an ALLOCATABLE object is associated with an INTENT(OUT), ALLOCATABLE dummy argument, its finalization reflects the ALLOCATABLE attribute and includes the deallocation of the object. However, if the dummy argument does not have the ALLOCATABLE attribute, the finalization applies only to the elements of the object and the object is not deallocated.

**J3 internal note**

Unresolved issue 189

It has been pointed out that the above is not the behavior called for in the TR, but that the TR was internally inconsistent in its specification of the handling of ALLOCATABLE objects and of objects of derived type with ALLOCATABLE components under these conditions, so we believe that this is the correct behavior to specify in the standard. We need to add something to section 1 to note the change from the TR.

**NOTE 6.24**

In all cases where this finalization is non-trivial, the interface is required to be explicit, so the processor may at its discretion perform the finalization in the caller and the initialization in the procedure to minimize the amount of information that must be communicated to the procedure.

## 6.4 Dynamic association

Dynamic control over the creation, association, and deallocation of pointer targets is provided by the ALLOCATE, NULLIFY, and DEALLOCATE statements and pointer assignment. ALLOCATE (6.4.1) creates targets for pointers; pointer assignment (7.5.2) associates pointers with existing targets; NULLIFY (6.4.2) disassociates pointers from targets, and DEALLOCATE (6.4.3) deallocates targets. Dynamic association applies to scalars and arrays of any type.

The ALLOCATE and DEALLOCATE statements also are used to create and deallocate variables with the ALLOCATABLE attribute.

**NOTE 6.25**

For detailed remarks regarding pointers and dynamic association see C.3.2.

### 6.4.1 ALLOCATE statement

The **ALLOCATE statement** dynamically creates pointer targets and allocatable variables.

R623 *allocate-stmt* **is** ALLOCATE ( [ *type-spec* :: ] *allocation-list* [, *alloc-opt-list* ] )

R624 *alloc-opt* **is** STAT = *stat-variable*
**or** ERRMSG = *errmsg-variable*

R625 *stat-variable* **is** *scalar-int-variable*

R626 *errmsg-variable* **is** *scalar-default-char-variable*

R627 *allocation* **is** *allocate-object* [ ( *allocate-shape-spec-list* ) ]

R628 *allocate-object* **is** *variable-name*
**or** *structure-component*

R629 *allocate-shape-spec* **is** [ *allocate-lower-bound* : ] *allocate-upper-bound*

R630 *allocate-lower-bound* **is** *scalar-int-expr*

R631 *allocate-upper-bound* **is** *scalar-int-expr*

Constraint: Each *allocate-object* shall be a nonprocedure pointer or an allocatable variable.

Constraint: If any *allocate-object* in the statement has a deferred type parameter, *type-spec* shall appear.

Constraint: If a *type-spec* appears, it shall specify a type that is the same as the declared type of each *allocate-object* that is not polymorphic (5.1.1.8) and is an extension type (4.5.3) of the declared type of each *allocate-object* that is polymorphic.

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

Constraint: A *type-param-value* shall be an asterisk if and only if each *allocate-object* is a dummy argument for which the corresponding type parameter is assumed.

Constraint: An *allocate-shape-spec-list* shall be specified for an object if and only if the object is an array.

Constraint: The number of *allocate-shape-spec*s in an *allocate-shape-spec-list* shall be the same as the rank of the *allocate-object*.

Constraint: No *alloc-opt* shall appear more than once in a given *alloc-opt-list*.

An *allocate-object* shall not depend on the value, bounds, allocation status, or association status of any other *allocate-object* in the same ALLOCATE statement.

The optional *type-spec* specifies the dynamic type and type parameters of the objects to be allocated. If a *type-spec* is specified, allocation of a polymorphic object allocates an object with the specified dynamic type; otherwise it allocates an object with a dynamic type the same as its declared type.

When an ALLOCATE statement having a *type-spec* is executed, any *type-param-values* in the *type-spec* specify the type parameters. If the value specified for a type parameter differs from a corresponding nondeferred value specified in the declaration of any of the *allocate-object*s then an error condition occurs.

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.

A bound or type parameter value shall not depend on *stat-variable*, *errmsg-variable*, or on the value, bounds, allocation status, or association status of any *allocate-object* in the same ALLOCATE statement.

Neither *stat-variable* nor *errmsg-variable* shall be allocated within the ALLOCATE statement in which it appears; nor shall they depend on the value, bounds, allocation status, or association status of any *allocate-object* in the same ALLOCATE statement.

**NOTE 6.26**

An example of an ALLOCATE statement is:

```
ALLOCATE (X (N), B (-3 : M, 0:9), STAT = IERR_ALLOC)
```

When an ALLOCATE statement is executed for an array, the values of the lower bound and upper bound expressions determine the bounds of the array. Subsequent redefinition or undefinition of any entities in the bound expressions do not affect the array bounds. If the lower bound is omitted, the default value is 1. If the upper bound is less than the lower bound, the extent in that dimension is zero and the array has zero size. If an *allocate-shape-spec-list* is specified for an array that does not have deferred shape, the bounds specified in the *allocate-shape-spec-list* shall be the same as those specified in the declaration of the object.

**NOTE 6.27**

An *allocate-object* may be of type character with zero character length.

If the STAT= specifier is present, successful execution of the ALLOCATE statement causes the *stat-variable* to become defined with a value of zero. If an error condition occurs during the

execution of the ALLOCATE statement, the *stat-variable* becomes defined with a processor-dependent positive integer value and each *allocate-object* will have a processor-dependent status; each *allocate-object* that was successfully allocated shall be currently allocated or be associated, each *allocate-object* that was not successfully allocated shall retain its previous allocation status or pointer association status.

If an error condition occurs during execution of an ALLOCATE statement that does not contain the STAT= specifier, execution of the program is terminated.

The ERRMSG= specifier is described in 6.4.1.4.

#### 6.4.1.1 Allocation of allocatable variables

An allocatable variable that has been allocated by an ALLOCATE statement and has not been subsequently deallocated (6.4.3) is **currently allocated** and is definable. Allocating a currently allocated allocatable variable causes an error condition in the ALLOCATE statement. At the beginning of execution of a program, allocatable variables have the allocation status of not currently allocated and are not definable. The ALLOCATED intrinsic function (13.16.9) may be used to determine whether an allocatable variable is currently allocated.

When an object of derived type is created by an ALLOCATE statement, any allocatable ultimate components have an allocation status of not currently allocated.

#### 6.4.1.2 Allocation status

The allocation status of an allocatable entity is one of the following at any time during the execution of a program:

(1) Not currently allocated. An allocatable variable with this status shall not be referenced or defined; it may be allocated with the ALLOCATE statement. Deallocating it causes an error condition in the DEALLOCATE statement. The ALLOCATED intrinsic returns .FALSE. for such a variable.

(2) Currently allocated. An allocatable variable with this status may be referenced, defined, or deallocated; allocating it causes an error condition in the ALLOCATE statement. The ALLOCATED intrinsic returns .TRUE. for such a variable.

**J3 internal note**

Unresolved isssue 196

Paper 99-108r1 deleted about 2/3 of the normative text in 6.4.1.2 (allocation status), with the comment that "This should be covered in 6.3." I did the requested deletion, but I'm more interested in whether the material *IS* all covered in 6.3 than in whether it should be. I'm not sure, but it was a pretty big deletion, so I want to flag it si that we might have some hope of finding where the material went if it isn't all covered.

I'm particularly unsure whether 6.3 describes how dummy allocatable arguments or components inherit the allocation status of the actual (except for INTENT(OUT) dummies). And I don't off-hand see, even in the old text, where we explain that the allocation status of the dummy gets copied back to the actual on return. Perhaps that's somewhere else (like chapter 12). So its possible that this is all ok, but I wanted to flag it for verification, particularly considering how non-commital the paper was on the question. If the authors of the paper didn't check whether it was actually covered (as opposed to should be), then someone else should so check.

Same concern about the deletion of most of 6.4.3.1 (deallocation of allocatable variables). Someone should check this. Likewise in 6.4.3.2, though the material deleted there wasn't as extensive and looks to me more likely to be ok.

**NOTE 6.28**

The following example illustrates the effects of SAVE on allocation status. The effect of terminating execution of a procedure on an allocatable object accessed by use association is affected by the presence or absence of the SAVE attribute and by the processor-dependent merging of instance of modules (6.3.3).

```
MODULE MOD1
TYPE INITIALIZED_TYPE
   INTEGER :: I = 1  ! Default initialization
END TYPE INITIALIZED_TYPE
SAVE :: SAVED1, SAVED2
INTEGER :: SAVED1, UNSAVED1
TYPE(INITIALIZED_TYPE) :: SAVED2, UNSAVED2
ALLOCATABLE :: SAVED1(:), SAVED2(:), UNSAVED1(:), UNSAVED2(:)
END MODULE MOD1

PROGRAM MAIN
CALL SUB1   ! The values returned by the ALLOCATED intrinsic calls
            ! in the PRINT statement are:
            ! .FALSE., .FALSE., .FALSE., and .FALSE.
            ! Module MOD1 is used, and its variables are allocated.
            ! After return from the subroutine, whether the variables
            ! which were not specified with the SAVE attribute
            ! retain their allocation status is processor dependent.

CALL SUB1   ! The values returned by the first two ALLOCATED intrinsic
            ! calls in the PRINT statement are:
            ! .TRUE., .TRUE.
            ! The values returned by the second two ALLOCATED
            ! intrinsic calls in the PRINT statement are
            ! processor dependent and each could be either
            ! .TRUE. or .FALSE.

CONTAINS
   SUBROUTINE SUB1
   USE MOD1    ! Brings in saved and not saved variables.
   PRINT *, ALLOCATED(SAVED1), ALLOCATED(SAVED2), &
            ALLOCATED(UNSAVED1), ALLOCATED(UNSAVED2)
   IF (.NOT. ALLOCATED(SAVED1)) ALLOCATE(SAVED1(10))
   IF (.NOT. ALLOCATED(SAVED2)) ALLOCATE(SAVED2(10))
   IF (.NOT. ALLOCATED(UNSAVED1)) ALLOCATE(UNSAVED1(10))
   IF (.NOT. ALLOCATED(UNSAVED2)) ALLOCATE(UNSAVED2(10))
   END SUBROUTINE SUB1
END PROGRAM MAIN
```

**J3 internal note**

Unresolved issue 197

For something that is in a note to help explain thing, the second sentence of note 6.28 in 6.4.1.2 is awfully hard to read. Its hard to follow what "on an allocatable object" is supposed to modify; presumably neither "procedure", "execution", nor "terminating", but all the way back to "effect". And it seems a bit much that the effect is affected by something. All in all, this sentence reads more like the kind of standardese that we sometimes use notes to explain, rather than like an explanation that makes it all clear.

### 6.4.1.3 Allocation of pointer targets

Following successful execution of an ALLOCATE statement for a pointer, the pointer is associated with the target and may be used to reference or define the target. Allocation of a pointer creates an object that implicitly has the TARGET attribute. Additional pointers may become associated with

the pointer target or a part of the pointer target by pointer assignment. It is not an error to allocate a pointer that is currently associated with a target. In this case, a new pointer target is created as required by the attributes of the pointer and any array bounds specified in the ALLOCATE statement. The pointer is then associated with this new target. Any previous association of the pointer with a target is broken. If the previous target had been created by allocation, it becomes inaccessible unless it can still be referred to by other pointers that are currently associated with it. The ASSOCIATED intrinsic function (13.16.13) may be used to determine whether a pointer is currently associated.

At the beginning of execution of a function whose result is a pointer, the association status of the result pointer is undefined. Before such a function returns, it shall either associate a target with this pointer or cause the association status of this pointer to become defined as disassociated.

#### 6.4.1.4 ERRMSG= specifier

If an error condition occurs during execution of an ALLOCATE or DEALLOCATE statement, the processor shall assign an explanatory message to *errmsg-variable*. If no such condition occurs, the processor shall not change the value of *errmsg-variable*.

### 6.4.2 NULLIFY statement

The **NULLIFY statement** causes pointers to be disassociated.

R632 *nullify-stmt* **is** NULLIFY ( *pointer-object-list* )

R633 *pointer-object* **is** *variable-name*
**or** *structure-component*

Constraint: Each *pointer-object* shall have the POINTER attribute.

A *pointer-object* shall not depend on the value, bounds, or association status of another *pointer-object* in the same NULLIFY statement.

**NOTE 6.29**

When a NULLIFY statement is applied to a polymorphic pointer (5.1.1.8), its dynamic type becomes the declared type.

### 6.4.3 DEALLOCATE statement

The **DEALLOCATE statement** causes allocatable variables to be deallocated and it causes pointer targets to be deallocated and the pointers to be disassociated.

R634 *deallocate-stmt* **is** DEALLOCATE ( *allocate-object-list* [ , *alloc-opt-list* ] )

Constraint: Each *allocate-object* shall be a nonprocedure pointer or an allocatable variable.

An *allocate-object* shall not depend on the value, bounds, allocation status, or association status of another *allocate-object* in the same DEALLOCATE statement; nor shall it depend on the value of the *stat-variable* or *errmsg-variable* in the same DEALLOCATE statement.

Neither *stat-variable* nor *errmsg-variable* shall be deallocated within the same DEALLOCATE statement; nor shall they depend on the value, bounds, allocation status, or association status of any *allocate-object* in the same DEALLOCATE statement.

If the STAT= specifier is present, successful execution of the DEALLOCATE statement causes the *stat-variable* to become defined with a value of zero. If an error condition occurs during the execution of the DEALLOCATE statement, the *stat-variable* becomes defined with a processor-dependent positive integer value and each *allocate-object* that was successfully deallocated shall be

not currently allocated or shall be disassociated. Each *allocate-object* that was not successfully deallocated shall retain its previous allocation status or pointer association status.

**NOTE 6.30**

The status of objects that were not successfully deallocated can be individually checked with the ALLOCATED or ASSOCIATED intrinsic functions.

If an error condition occurs during execution of a DEALLOCATE statement that does not contain the STAT= specifier, execution of the program is terminated.

The ERRMSG= specifier is described in 6.4.1.4.

**NOTE 6.31**

An example of a DEALLOCATE statement is:

```
DEALLOCATE (X, B)
```

### 6.4.3.1 Deallocation of allocatable variables

Deallocating an allocatable variable that is not currently allocated causes an error condition in the DEALLOCATE statement. An allocatable variable with the TARGET attribute shall not be deallocated through an associated pointer. Deallocating an allocatable variable with the TARGET attribute causes the pointer association status of any pointer associated with it to become undefined.

**NOTE 6.32**

The ALLOCATED intrinsic function may be used to determine whether a variable is still currently allocated or has been deallocated. Processor merging of instances of modules (6.3.3.1) can make it processor-dependent whether an allocatable object accessed by use association is deallocated when execution of a procedure is terminated.

**J3 internal note**

Unresolved issue 198

Note 6.32 in 6.4.3.1 says that "Processor merging...can make it processor dependent...". What does this mean? Is it or is it not processor dependent? If the "processor merging" can make it processor dependent, then I think you just said it is processor dependent whether or not something is processor dependent, which makes no sense. If we are trying to say that some cases are processor dependent, but others aren't, then perhaps it would be best to say so explicitly.

**NOTE 6.33**

In the following example:

```
SUBROUTINE PROCESS
  REAL, ALLOCATABLE :: TEMP(:)
  REAL, ALLOCATABLE, SAVE :: X(:)
  ...
END SUBROUTINE PROCESS
```

on return from subroutine PROCESS, the allocation status of X is preserved because X has the SAVE attribute. TEMP does not have the SAVE attribute, so it will be deallocated. On the next invocation of PROCESS, TEMP will have an allocation status of not currently allocated.

### 6.4.3.2 Deallocation of pointer targets

If a pointer appears in a DEALLOCATE statement, its association status shall be defined. Deallocating a pointer that is disassociated or whose target was not created by an ALLOCATE statement causes an error condition in the DEALLOCATE statement. If a pointer is currently associated with an allocatable entity, the pointer shall not be deallocated.

A pointer that is not currently associated with the whole of an allocated target object shall not be deallocated. If a pointer is currently associated with a portion (2.4.3.1) of a target object that is independent of any other portion of the target object, it shall not be deallocated. Deallocating a pointer target causes the pointer association status of any other pointer that is associated with the target or a portion of the target to become undefined.