Section 6: Use of data objects

The appearance of a data object name or subobject 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 name or subobject designator appears in certain contexts and when certain events occur (14.7).

R601 *variable* **is** *scalar-variable-name*
or *array-variable-name*
or *subobject*

Constraint: *array-variable-name* shall be the name of a data object that is an array.

Constraint: *array-variable-name* shall not have the PARAMETER attribute.

Constraint: *scalar-variable-name* shall not have the PARAMETER attribute.

Constraint: *subobject* shall not be a subobject designator (for example, a substring) whose parent is a constant.

R602 *subobject* **is** *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.

Pointers and allocatable arrays shall not be defined in circumstances explained in 5.1.2.4.3. Dummy arguments or variables associated with dummy arguments shall not be defined in circumstances explained in 12.4.1.1, 12.4.1.5, 12.4.1.6, and 12.5.2.1.

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 (4.4.3).

A scalar has rank zero.

6.1.1 Substrings

A **substring** is a contiguous portion of a character string (4.3.2.1). 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 a subobject 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 type of the preceding *part-name*.

Constraint: In a *part-ref* containing a *section-subscript-list*, the number of *section-subscripts* 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 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 **parent object** of a *data-ref* is the data object whose name is the leftmost part name.

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 parent object. A structure component has the INTENT, TARGET, or PARAMETER attribute if the parent object has the attribute. A structure component is a pointer only if the rightmost part name is defined to have the POINTER attribute.

NOTE 6.4

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 6.5

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 non-zero 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.

6.2 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 name or 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

R615 *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*.

R616 *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.

R617 *subscript* **is** *scalar-int-expr*

R618 *section-subscript* **is** *subscript*
or *subscript-triplet*
or *vector-subscript*

R619 *subscript-triplet* **is** [*subscript*] : [*subscript*] [: *stride*]

R620 *stride* **is** *scalar-int-expr*

R621 *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 6.6

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 has the INTENT, TARGET, or PARAMETER attribute if its parent has the attribute, but it never has the POINTER attribute.

NOTE 6.7

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	$1 + (s_1 - j_1)$ $+ (s \ \ - j_2) \times d_1$ $+ (s_3 - j_3) \times d_2 \times d_1$
.	.	.	.
.	.	.	.
.	.	.	.
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 $i = 1, 2, \ldots, 7$.

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 6.8

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 6.9

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 6.10

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 6.11

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 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 (6.3.1) creates targets for pointers; pointer assignment (7.5.2) associates pointers with existing targets; NULLIFY (6.3.2) disassociates pointers from targets, and DEALLOCATE (6.3.3) deallocates targets. Dynamic association applies to scalars and arrays of any type.

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

NOTE 6.12

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

6.3.1 ALLOCATE statement

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

R622	*allocate-stmt*	**is**	ALLOCATE (*allocation-list* [, STAT = *stat-variable*])
R623	*stat-variable*	**is**	*scalar-int-variable*
R624	*allocation*	**is**	*allocate-object* [(*allocate-shape-spec-list*)]
R625	*allocate-object*	**is**	*variable-name*
		or	*structure-component*
R626	*allocate-shape-spec*	**is**	[*allocate-lower-bound* :] *allocate-upper-bound*
R627	*allocate-lower-bound*	**is**	*scalar-int-expr*
R628	*allocate-upper-bound*	**is**	*scalar-int-expr*

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

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

An *allocate-object*, or a subobject of an *allocate-object*, shall not appear in a bound in the same ALLOCATE statement. The *stat-variable* shall not appear in a bound in the same ALLOCATE statement.

NOTE 6.13

An example of an ALLOCATE statement is:

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

The *stat-variable* shall not be allocated within the ALLOCATE statement in which it appears; nor shall it depend on the value, bounds, allocation status, or association status of any *allocate-object* or subobject of an *allocate-object* allocated in the same statement.

At the time 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.

NOTE 6.14

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.

6.3.1.1 Allocation of allocatable arrays

An allocatable array that has been allocated by an ALLOCATE statement and has not been subsequently deallocated (6.3.3) is **currently allocated** and is definable. Allocating a currently allocated allocatable array causes an error condition in the ALLOCATE statement. At the beginning of execution of a program, allocatable arrays have the allocation status of not currently allocated and are not definable. The ALLOCATED intrinsic function (13.14.9) may be used to determine whether an allocatable array is currently allocated.

6.3.1.2 Allocation status

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

(1) Not currently allocated. An allocatable array 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 an array.

(2) Currently allocated. An allocatable array 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 an array.

An allocatable array with the SAVE attribute has an initial status of not currently allocated. If the array is allocated, its status changes to currently allocated. The status remains currently allocated until the array is deallocated.

An allocatable array that does not have the SAVE attribute, that is a local variable of a procedure, and that is not accessed by use or host association, has a status of not currently allocated at the beginning of each invocation of the procedure. The status may change during execution of the procedure. If the array has a status of currently allocated when the procedure is exited by execution of a RETURN or END statement, it is deallocated.

An allocatable array that does not have the SAVE attribute and that is accessed by use association has an initial status of not currently allocated. The status may change during execution of the program. If the array has an allocation status of currently allocated when execution of a RETURN or END statement results in no scoping unit having access to the module, it is processor dependent whether the allocation status remains currently allocated or the array is deallocated.

NOTE 6.15

The following example illustrates the effects of SAVE on allocation status.

```
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
```

6.3.1.3 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 (13.14.13) 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.

6.3.2 NULLIFY statement

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

R629	*nullify-stmt*	**is**	NULLIFY (*pointer-object-list*)
R630	*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*, or subobject of another *pointer-object*, in the same NULLIFY statement.

6.3.3 DEALLOCATE statement

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

R631	*deallocate-stmt*	**is**	DEALLOCATE (*allocate-object-list* [, STAT = *stat-variable*])

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

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

The *stat-variable* shall not be deallocated within the same DEALLOCATE statement; nor shall it depend on the value, bounds, allocation status, or association status of any *allocate-object* or subobject of an *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 6.16

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.

NOTE 6.17

An example of a DEALLOCATE statement is:

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

6.3.3.1 Deallocation of allocatable arrays

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

When the execution of a procedure is terminated by execution of a RETURN or END statement, an allocatable array that is a local variable of the procedure retains its allocation and definition status if

(1) It has the SAVE attribute,

(2) It is accessed by use association, if the module defining the array is also accessed by another scoping unit that is currently in execution, or

(3) It is accessed by host association.

When the execution of a procedure is terminated by execution of a RETURN or END statement, an allocatable array that is a local variable of the procedure and is not included in the above categories has allocation status as follows:

(1) If it is accessed by use association, its allocation status is processor dependent.

(2) Otherwise, it is deallocated (as if by a DEALLOCATE statement).

NOTE 6.18

The ALLOCATED intrinsic function may be used to determine whether an array is still currently allocated or has been deallocated.

NOTE 6.19

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.

6.3.3.2 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 array, 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.

When the execution of a procedure is terminated by execution of a RETURN or END statement, the pointer association status of a pointer declared or accessed in the subprogram that defines the procedure becomes undefined unless it is one of the following:

(1) A pointer with the SAVE attribute,

(2) A pointer in blank common,

(3) A pointer in a named common block that appears in at least one other scoping unit that is currently in execution,

(4) A pointer declared in the scoping unit of a module if the module also is accessed by another scoping unit that is currently in execution,

(5) A pointer accessed by host association, or

(6) A pointer that is the return value of a function declared to have the POINTER attribute.

When a pointer target becomes undefined by execution of a RETURN or END statement, the pointer association status (14.6.2.1) becomes undefined.