5 Attribute declarations and specifications

5.1 General

Every data object has a type and rank and may have type parameters and other attributes that determine the uses of the object. Collectively, these properties are the **attributes** of the object. The type of a named data object is either specified explicitly in a type declaration statement or determined implicitly by the first letter of its name (5.5). All of its attributes may be specified in a type declaration statement or individually in separate specification statements.

A function has a type and rank and may have type parameters and other attributes that determine the uses of the function. The type, rank, and type parameters are the same as those of its result variable.

A subroutine does not have a type, rank, or type parameters, but may have other attributes that determine the uses of the subroutine.

5.2 Type declaration statements

5.2.1 Syntax

R501 *type-declaration-stmt* **is** *declaration-type-spec* [[, *attr-spec*] ... ::] *entity-decl-list*

The type declaration statement specifies the type of the entities in the entity declaration list. The type and type parameters are those specified by *declaration-type-spec*, except that the character length type parameter may be overridden for an entity by the appearance of * *char-length* in its *entity-decl*.

R502 *attr-spec* **is** *access-spec*
or ALLOCATABLE
or ASYNCHRONOUS
or DIMENSION (*array-spec*)
or EXTERNAL
or INTENT (*intent-spec*)
or INTRINSIC
or *language-binding-spec*
or OPTIONAL
or PARAMETER
or POINTER
or PROTECTED
or SAVE
or TARGET
or VALUE
or VOLATILE

C501 (R501) The same *attr-spec* shall not appear more than once in a given *type-declaration-stmt*.

C502 (R501) If a *language-binding-spec* with a NAME= specifier appears, the *entity-decl-list* shall consist of a single *entity-decl*.

C503 (R501) If a *language-binding-spec* is specified, the *entity-decl-list* shall not contain any procedure names.

The type declaration statement also specifies the attributes whose keywords appear in the *attr-spec*, except that the DIMENSION attribute may be specified or overridden for an entity by the appearance of *array-spec* in its *entity-decl*.

R503 *entity-decl* **is** *object-name* [(*array-spec*)] [* *char-length*] [*initialization*]
or *function-name* [* *char-length*]

C504 (R503) If the entity is not of type character, * *char-length* shall not appear.

C505 (R501) If *initialization* appears, a double-colon separator shall appear before the *entity-decl-list*.

C506 (R503) An *initialization* shall not appear if *object-name* is a dummy argument, a function result, an object in a named common block unless the type declaration is in a block data program unit, an object in blank common, an allocatable variable, an external function, an intrinsic function, or an automatic object.

C507 (R503) An *initialization* shall appear if the entity is a named constant (5.3.11).

C508 (R503) The *function-name* shall be the name of an external function, an intrinsic function, a function dummy procedure, or a statement function.

R504 *object-name* **is** *name*

C509 (R504) The *object-name* shall be the name of a data object.

R505 *initialization* **is** = *initialization-expr*
or => *null-init*
R506 *null-init* **is** *function-reference*

C510 (R503) If => appears in *initialization*, the entity shall have the POINTER attribute. If = appears in *initialization*, the entity shall not have the POINTER attribute.

C511 (R506) The *function-reference* shall be a reference to the NULL intrinsic function with no arguments.

A name that identifies a specific intrinsic function in a scoping unit has a type as specified in 13.6. An explicit type declaration statement is not required; however, it is permitted. Specifying a type for a generic intrinsic function name in a type declaration statement is not sufficient, by itself, to remove the generic properties from that function.

An **automatic data object** is a nondummy data object with a type parameter or array bound that depends on the value of a *specification-expr* that is not an initialization expression.

NOTE 5.1

An automatic object shall not have the SAVE attribute and shall not appear in a common block.

If a type parameter in a *declaration-type-spec* or in a *char-length* in an *entity-decl* is defined by an expression that is not an initialization expression, the type parameter value is established on entry to the procedure and is not affected by any redefinition or undefinition of the variables in the expression during execution of the procedure.

5.2.2 Initialization

The appearance of *initialization* in an *entity-decl* for an entity without the PARAMETER attribute specifies that the entity is a variable with **explicit initialization**. Explicit initialization alternatively may be specified in a DATA statement unless the variable is of a derived type for which default initialization is specified. If *initialization* is =*initialization-expr*, the variable is initially defined with the value specified by the *initialization-expr*; if necessary, the value is converted according to the rules of intrinsic assignment (7.4.1.3) to a value that agrees in type, type parameters, and shape with the variable. A variable, or part of a variable, shall not be explicitly initialized more than once in a program. If the variable is an array, it shall have its shape specified in either the type declaration statement or a previous attribute specification statement in the same scoping unit.

If *initialization* is =>*null-init*, the variable shall be a pointer, and its initial association status is disassociated.

Explicit initialization of a variable that is not in a common block implies the SAVE attribute, which may be confirmed by explicit specification.

5.2.3 Examples of type declaration statements

NOTE 5.2

```
REAL A (10)
LOGICAL, DIMENSION (5, 5) :: MASK1, MASK2
COMPLEX :: CUBE_ROOT = (-0.5, 0.866)
INTEGER, PARAMETER :: SHORT = SELECTED_INT_KIND (4)
INTEGER (SHORT) K     !  Range at least -9999 to 9999.
REAL (KIND (0.0D0)) A
REAL (KIND = 2) B
COMPLEX (KIND = KIND (0.0D0)) :: C
CHARACTER (LEN = 10, KIND = 2)  A
CHARACTER B, C *20
TYPE (PERSON) :: CHAIRMAN
TYPE(NODE), POINTER :: HEAD => NULL ( )
TYPE (humongous_matrix (k=8, d=1000)) :: mat
```

(The last line above uses a type definition from Note 4.25.)

5.3 Attributes

5.3.1 Constraints

An attribute may be explicitly specified by an *attr-spec* in a type declaration statement or by an attribute specification statement (5.4). The following constraints apply to attributes.

C512 An entity shall not be explicitly given any attribute more than once in a scoping unit.

C513 An *array-spec* for a function result that does not have the ALLOCATABLE or POINTER attribute shall be an *explicit-shape-spec-list*.

C514 The ALLOCATABLE, POINTER, or OPTIONAL attribute shall not be specified for a dummy argument of a procedure that has a *proc-language-binding-spec*.

5.3.2 Accessibility attribute

The **accessibility attribute** specifies the accessibility of an entity via a particular identifier.

R507	*access-spec*	**is**	PUBLIC
		or	PRIVATE

C515 (R507) An *access-spec* shall appear only in the *specification-part* of a module.

Identifiers that are specified in a module or accessible in that module by use association have either the PUBLIC or PRIVATE attribute. Identifiers for which an *access-spec* is not explicitly specified in that module have the default accessibility attribute for that module. The default accessibility attribute for a module is PUBLIC unless it has been changed by a PRIVATE statement (5.4.1). Only identifiers that have the PUBLIC attribute in that module are available to be accessed from that module by use association.

NOTE 5.3

In order for an identifier to be accessed by use association, it must have the PUBLIC attribute in the module from which it is accessed. It can nonetheless have the PRIVATE attribute in a module in which it is accessed by use association, and therefore not be available for use association from a module where it is PRIVATE.

NOTE 5.4

An example of an accessibility specification is:

```
REAL, PRIVATE :: X, Y, Z
```

5.3.3 ALLOCATABLE attribute

An entity with the **ALLOCATABLE attribute** is a variable for which space is allocated by an ALLOCATE statement (6.3.1) or by an intrinsic assignment statement (7.4.1.3).

5.3.4 ASYNCHRONOUS attribute

An entity with the **ASYNCHRONOUS attribute** is a variable that may be subject to asynchronous input/output.

The base object of a variable shall have the ASYNCHRONOUS attribute in a scoping unit if

(1) the variable appears in an executable statement or specification expression in that scoping unit and

(2) any statement of the scoping unit is executed while the variable is a pending I/O storage sequence affector (9.5.1.4).

Use of a variable in an asynchronous input/output statement can imply the ASYNCHRONOUS attribute; see subclause (9.5.1.4).

An object may have the ASYNCHRONOUS attribute in a particular scoping unit without necessarily having it in other scoping units (11.2.1, 16.4.1.3). If an object has the ASYNCHRONOUS attribute, then all of its subobjects also have the ASYNCHRONOUS attribute.

NOTE 5.5

The ASYNCHRONOUS attribute specifies the variables that might be associated with a pending input/output storage sequence (the actual memory locations on which asynchronous input/output is being performed) while the scoping unit is in execution. This information could be used by the compiler to disable certain code motion optimizations.

The ASYNCHRONOUS attribute is similar to the VOLATILE attribute. It is intended to facilitate

NOTE 5.5 (cont.)

traditional code motion optimizations in the presence of asynchronous input/output.

5.3.5 BIND attribute for data entities

The BIND attribute for a variable or common block specifies that it is capable of interoperating with a C variable that has external linkage (15.3).

R508 *language-binding-spec* **is** BIND (C [, NAME = *scalar-char-initialization-expr*])

C516 An entity with the BIND attribute shall be a common block, variable, or procedure.

C517 A variable with the BIND attribute shall be declared in the specification part of a module.

C518 A variable with the BIND attribute shall be interoperable (15.2).

C519 Each variable of a common block with the BIND attribute shall be interoperable.

C520 (R508) The *scalar-char-initialization-expr* shall be of default character kind. If the value of the *scalar-char-initialization-expr* after discarding leading and trailing blanks has nonzero length, it shall be valid as an identifier on the companion processor.

NOTE 5.6

The C International Standard provides a facility for creating C identifiers whose characters are not restricted to the C basic character set. Such a C identifier is referred to as a universal character name (6.4.3 of the C International Standard). The name of such a C identifier might include characters that are not part of the representation method used by the processor for type default character. If so, the C entity cannot be referenced from Fortran.

The BIND attribute for a variable or common block implies the SAVE attribute, which may be confirmed by explicit specification.

5.3.6 DIMENSION attribute

5.3.6.1 General

The **DIMENSION attribute** specifies that an entity is an array. The rank or rank and shape is specified by its *array-spec*.

R509 *array-spec* **is** *explicit-shape-spec-list*
or *assumed-shape-spec-list*
or *deferred-shape-spec-list*
or *assumed-size-spec*

C521 (R509) The maximum rank is seven.

NOTE 5.7

Examples of DIMENSION attribute specifications are:

```
SUBROUTINE EX (N, A, B)
   REAL, DIMENSION (N, 10) :: W          ! Automatic explicit-shape array
   REAL A (:), B (0:)                    ! Assumed-shape arrays
   REAL, POINTER :: D (:, :)             ! Array pointer
   REAL, DIMENSION (:), POINTER :: P     ! Array pointer
```

NOTE 5.7 (cont.)

```
REAL, ALLOCATABLE, DIMENSION (:) :: E ! Allocatable array
```

5.3.6.2 Explicit-shape array

An **explicit-shape array** is a named array that is declared with an *explicit-shape-spec-list*. This specifies explicit values for the bounds in each dimension of the array.

R510	*explicit-shape-spec*	**is**	[*lower-bound* :] *upper-bound*
R511	*lower-bound*	**is**	*specification-expr*
R512	*upper-bound*	**is**	*specification-expr*

C522 (R510) An *explicit-shape-spec* whose bounds are not initialization expressions shall appear only in a subprogram or interface body.

If an explicit-shape array has bounds that are not initialization expressions, the bounds, and hence shape, are determined at entry to the procedure by evaluating the bounds expressions. The bounds of such an array are unaffected by the redefinition or undefinition of any variable during execution of the procedure.

The values of each *lower-bound* and *upper-bound* determine the bounds of the array along a particular dimension and hence the extent of the array in that dimension. The value of a lower bound or an upper bound may be positive, negative, or zero. The subscript range of the array in that dimension is the set of integer values between and including the lower and upper bounds, provided the upper bound is not less than the lower bound. If the upper bound is less than the lower bound, the range is empty, the extent in that dimension is zero, and the array is of zero size. If the *lower-bound* is omitted, the default value is 1. The rank is equal to the number of *explicit-shape-spec*s.

5.3.6.3 Assumed-shape array

An **assumed-shape array** is a nonpointer dummy argument array that takes its shape from the associated actual argument array.

R513	*assumed-shape-spec*	**is**	[*lower-bound*] :

The rank is equal to the number of colons in the *assumed-shape-spec-list*.

The extent of a dimension of an assumed-shape array dummy argument is the extent of the corresponding dimension of the associated actual argument array. If the lower bound value is d and the extent of the corresponding dimension of the associated actual argument array is s, then the value of the upper bound is $s + d - 1$. If *lower-bound* appears it specifies the lower bound; otherwise the lower bound is 1.

5.3.6.4 Deferred-shape array

A **deferred-shape array** is an allocatable array or an array pointer.

An **allocatable array** is an array that has the ALLOCATABLE attribute and a specified rank, but its bounds, and hence shape, are determined by allocation or argument association.

An **array pointer** is an array with the POINTER attribute and a specified rank. Its bounds, and hence shape, are determined when it is associated with a target.

R514	*deferred-shape-spec*	**is**	:

C523 An array that has the POINTER or ALLOCATABLE attribute shall have an *array-spec* that is a *deferred-shape-spec-list*.

The rank is equal to the number of colons in the *deferred-shape-spec-list*.

The size, bounds, and shape of an unallocated allocatable array or a disassociated array pointer are undefined. No part of such an array shall be referenced or defined; however, the array may appear as an argument to an intrinsic inquiry function as specified in 13.1.

The bounds of each dimension of an allocatable array are those specified when the array is allocated.

The bounds of each dimension of an array pointer may be specified in two ways:

(1) in an ALLOCATE statement (6.3.1) when the target is allocated;
(2) by pointer assignment (7.4.2).

The bounds of the array pointer or allocatable array are unaffected by any subsequent redefinition or undefinition of variables on which the bounds' expressions depend.

5.3.6.5 Assumed-size array

An **assumed-size array** is a dummy argument array whose size is assumed from that of an associated actual argument. The rank and extents may differ for the actual and dummy arrays; only the size of the actual array is assumed by the dummy array. An assumed-size array is declared with an *assumed-size-spec*.

R515 *assumed-size-spec* **is** [*explicit-shape-spec-list* ,] [*lower-bound* :] *

C524 An *assumed-size-spec* shall not appear except as the declaration of the array bounds of a dummy data argument.

C525 An assumed-size array with INTENT (OUT) shall not be of a type for which default initialization is specified.

The size of an assumed-size array is determined as follows.

(1) If the actual argument associated with the assumed-size dummy array is an array of any type other than default character, the size is that of the actual array.
(2) If the actual argument associated with the assumed-size dummy array is an array element of any type other than default character with a subscript order value of r (6.2.2.2) in an array of size x, the size of the dummy array is $x - r + 1$.
(3) If the actual argument is a default character array, default character array element, or a default character array element substring (6.1.1), and if it begins at character storage unit t of an array with c character storage units, the size of the dummy array is MAX (INT $((c - t + 1)/e)$, 0), where e is the length of an element in the dummy character array.
(4) If the actual argument is of type default character and is a scalar that is not an array element or array element substring designator, the size of the dummy array is MAX (INT (l/e), 0), where e is the length of an element in the dummy character array and l is the length of the actual argument.

The rank is equal to one plus the number of *explicit-shape-spec*s.

An assumed-size array has no upper bound in its last dimension and therefore has no extent in its last dimension and no shape. An assumed-size array name shall not be written as a whole array reference except as an actual argument in a procedure reference for which the shape is not required.

If an *explicit-shape-spec-list* appears, it specifies the bounds of the first rank -1 dimensions. If *lower-bound* appears it specifies the lower bound of the last dimension; otherwise that lower bound is 1. An assumed-size array may be subscripted or sectioned (6.2.2.3). The upper bound shall not be omitted from a subscript triplet in the last dimension.

If an assumed-size array has bounds that are not initialization expressions, the bounds are determined at entry to the procedure. The bounds of such an array are unaffected by the redefinition or undefinition of any variable during execution of the procedure.

5.3.7 EXTERNAL attribute

The **EXTERNAL attribute** specifies that an entity is an external procedure, dummy procedure, procedure pointer, or block data subprogram.

C526 An entity shall not have both the EXTERNAL attribute and the INTRINSIC attribute.

In an external subprogram, the EXTERNAL attribute shall not be specified for a procedure defined by the subprogram.

If an external procedure or dummy procedure is used as an actual argument or is the target of a procedure pointer assignment, it shall be declared to have the EXTERNAL attribute.

A procedure that has both the EXTERNAL and POINTER attributes is a procedure pointer.

5.3.8 INTENT attribute

The **INTENT attribute** specifies the intended use of a dummy argument. An INTENT (IN) dummy argument is suitable for receiving data from the invoking scoping unit, an INTENT (OUT) dummy argument is suitable for returning data to the invoking scoping unit, and an INTENT (INOUT) dummy argument is suitable for use both to receive data from and to return data to the invoking scoping unit.

R516	*intent-spec*	**is**	IN
		or	OUT
		or	INOUT

C527 An entity with the INTENT attribute shall be a dummy data object or a dummy procedure pointer.

C528 (R516) A nonpointer object with the INTENT (IN) attribute shall not appear in a variable definition context (16.5.7).

C529 A pointer with the INTENT (IN) attribute shall not appear in a pointer association context (16.5.8).

The INTENT (IN) attribute for a nonpointer dummy argument specifies that it shall neither be defined nor become undefined during the execution of the procedure. The INTENT (IN) attribute for a pointer dummy argument specifies that during the execution of the procedure its association shall not be changed except that it may become undefined if the target is deallocated other than through the pointer (16.4.2.1.3).

The INTENT (OUT) attribute for a nonpointer dummy argument specifies that the dummy argument becomes undefined on invocation of the procedure, except for any subcomponents that are default-initialized (4.5.4.5). Any actual argument that becomes associated with such a dummy argument shall be definable. The INTENT (OUT) attribute for a pointer dummy argument specifies that on invocation of the procedure the pointer association status of the dummy argument becomes undefined. Any actual argument associated with such a pointer dummy shall be a pointer variable.

NOTE 5.8

If the actual argument is finalizable it will be finalized before undefinition and default initialization of the dummy argument (4.5.6).

The INTENT (INOUT) attribute for a nonpointer dummy argument specifies that any actual argument associated with the dummy argument shall be definable. The INTENT (INOUT) attribute for a pointer dummy argument specifies that any actual argument associated with the dummy argument shall be a pointer variable.

NOTE 5.9

The INTENT attribute for an allocatable dummy argument applies to both the allocation status and the definition status. An actual argument associated with an INTENT(OUT) allocatable dummy argument is deallocated on procedure invocation (6.3.3.1).

If no INTENT attribute is specified for a dummy argument, its use is subject to the limitations of the associated actual argument (12.4.1.2, 12.4.1.3, 12.4.1.4).

NOTE 5.10

An example of INTENT specification is:

```
SUBROUTINE MOVE (FROM, TO)
   USE PERSON_MODULE
   TYPE (PERSON), INTENT (IN) :: FROM
   TYPE (PERSON), INTENT (OUT) :: TO
```

If an object has an INTENT attribute, then all of its subobjects have the same INTENT attribute.

NOTE 5.11

If a dummy argument is a derived-type object with a pointer component, then the pointer as a pointer is a subobject of the dummy argument, but the target of the pointer is not. Therefore, the restrictions on subobjects of the dummy object apply to the pointer in contexts where it is used as a pointer, but not in contexts where it is dereferenced to indicate its target. For example, if X is a dummy argument of derived type with an integer pointer component P, and X has INTENT(IN), then the statement

```
X%P => NEW_TARGET
```

is prohibited, but

```
X%P = 0
```

is allowed (provided that X%P is associated with a definable target).

Similarly, the INTENT restrictions on pointer dummy arguments apply only to the association of the dummy argument; they do not restrict the operations allowed on its target.

NOTE 5.12

Argument intent specifications serve several purposes in addition to documenting the intended use of dummy arguments. A processor can check whether an INTENT (IN) dummy argument is used in a way that could redefine it. A slightly more sophisticated processor could check to see whether an INTENT (OUT) dummy argument could possibly be referenced before it is defined. If the procedure's interface is explicit, the processor can also verify that actual arguments corresponding to INTENT (OUT) or INTENT (INOUT) dummy arguments are definable. A more sophisticated processor could use this information to optimize the translation of the referencing scoping unit by taking advantage of the fact that actual arguments corresponding to INTENT (IN) dummy arguments will not be changed and that any prior value of an actual argument corresponding to

NOTE 5.12 (cont.)

an INTENT (OUT) dummy argument will not be referenced and could thus be discarded.

INTENT (OUT) means that the value of the argument after invoking the procedure is entirely the result of executing that procedure. If there is any possibility that an argument should retain its current value rather than being redefined, INTENT (INOUT) should be used rather than INTENT (OUT), even if there is no explicit reference to the value of the dummy argument. Because an INTENT(OUT) variable is considered undefined on entry to the procedure, any default initialization specified for its type will be applied.

INTENT (INOUT) is not equivalent to omitting the INTENT attribute. The actual argument corresponding to an INTENT (INOUT) dummy argument is always required to be definable, while an argument corresponding to a dummy argument without an INTENT attribute need be definable only if the dummy argument is actually redefined.

5.3.9 INTRINSIC attribute

The **INTRINSIC attribute** specifies that the entity is an intrinsic procedure. It may be a generic procedure (13.5), a specific procedure (13.6), or both.

If the specific name of an intrinsic procedure (13.6) is used as an actual argument, the name shall be explicitly specified to have the INTRINSIC attribute. An intrinsic procedure whose specific name is marked with a bullet (•) in 13.6 shall not be used as an actual argument.

C530 If the name of a generic intrinsic procedure is explicitly declared to have the INTRINSIC attribute, and it is also the generic name of one or more generic interfaces (12.3.2.1) accessible in the same scoping unit, the procedures in the interfaces and the specific intrinsic procedures shall all be functions or all be subroutines, and the characteristics of the specific intrinsic procedures and the procedures in the interfaces shall differ as specified in 16.2.3.

5.3.10 OPTIONAL attribute

The **OPTIONAL attribute** specifies that the dummy argument need not be associated with an actual argument in a reference to the procedure (12.4.1.6). The PRESENT intrinsic function can be used to determine whether an optional dummy argument is associated with an actual argument.

C531 An entity with the OPTIONAL attribute shall be a dummy argument.

5.3.11 PARAMETER attribute

The **PARAMETER attribute** specifies that an entity is a named constant. The entity has the value specified by its *initialization-expr*, converted, if necessary, to the type, type parameters and shape of the entity.

C532 An entity with the PARAMETER attribute shall not be a variable or a procedure.

A named constant shall not be referenced unless it has been defined previously in the same statement, defined in a prior statement, or made accessible by use or host association.

NOTE 5.13

Examples of declarations with a PARAMETER attribute are:

```
REAL, PARAMETER :: ONE = 1.0, Y = 4.1 / 3.0
INTEGER, DIMENSION (3), PARAMETER :: ORDER = (/ 1, 2, 3 /)
```

NOTE 5.13 (cont.)

```
TYPE(NODE), PARAMETER :: DEFAULT = NODE(0, NULL ( ))
```

5.3.12 POINTER attribute

Entities with the **POINTER attribute** can be associated with different data objects or procedures during execution of a program. A pointer is either a data pointer or a procedure pointer. Procedure pointers are described in 12.3.2.3.

C533 An entity with the POINTER attribute shall not have the ALLOCATABLE, INTRINSIC, or TARGET attribute.

C534 A procedure with the POINTER attribute shall have the EXTERNAL attribute.

A data pointer shall not be referenced unless it is pointer associated with a target object that is defined. A data pointer shall not be defined unless it is pointer associated with a target object that is definable.

If a data pointer is associated, the values of its deferred type parameters are the same as the values of the corresponding type parameters of its target.

A procedure pointer shall not be referenced unless it is pointer associated with a target procedure.

NOTE 5.14

Examples of POINTER attribute specifications are:

```
TYPE (NODE), POINTER :: CURRENT, TAIL
REAL, DIMENSION (:, :), POINTER :: IN, OUT, SWAP
```

For a more elaborate example see C.2.1.

5.3.13 PROTECTED attribute

The **PROTECTED attribute** imposes limitations on the usage of module entities.

C535 The PROTECTED attribute shall be specified only in the specification part of a module.

C536 An entity with the PROTECTED attribute shall be a procedure pointer or variable.

C537 An entity with the PROTECTED attribute shall not be in a common block.

C538 A nonpointer object that has the PROTECTED attribute and is accessed by use association shall not appear in a variable definition context (16.5.7) or as the *data-target* or *proc-target* in a *pointer-assignment-stmt*.

C539 A pointer that has the PROTECTED attribute and is accessed by use association shall not appear in a pointer association context (16.5.8).

Other than within the module in which an entity is given the PROTECTED attribute, or within any of its descendant submodules,

(1) if it is a nonpointer object, it is not definable, and

(2) if it is a pointer, its association status shall not be changed except that it may become undefined if its target is deallocated other than through the pointer (16.4.2.1.3) or if its target becomes undefined by execution of a RETURN or END statement.

If an object has the PROTECTED attribute, all of its subobjects have the PROTECTED attribute.

NOTE 5.15

An example of the PROTECTED attribute:

```
MODULE temperature
  REAL, PROTECTED :: temp_c, temp_f
CONTAINS
  SUBROUTINE set_temperature_c(c)
    REAL, INTENT(IN) :: c
    temp_c = c
    temp_f = temp_c*(9.0/5.0) + 32
  END SUBROUTINE
END MODULE
```

The PROTECTED attribute ensures that the variables `temp_c` and `temp_f` cannot be modified other than via the `set_temperature_c` procedure, thus keeping them consistent with each other.

5.3.14 SAVE attribute

The SAVE attribute specifies that a variable retains its association status, allocation status, definition status, and value after execution of a RETURN or END statement unless it is a pointer and its target becomes undefined (16.4.2.1.3(4)). If it is a local variable of a subprogram it is shared by all instances (12.5.2.3) of the subprogram.

Giving a common block the SAVE attribute confers the attribute on all variables in the common block.

C540 An entity with the SAVE attribute shall be a common block, variable, or procedure pointer.

C541 The SAVE attribute shall not be specified for a dummy argument, a function result, an automatic data object, or an object that is in a common block.

A **saved** entity is an entity that has the SAVE attribute. An **unsaved** entity is an entity that does not have the SAVE attribute.

The SAVE attribute has no effect on entities declared in a main program. If a common block has the SAVE attribute in any scoping unit that is not a main program, it shall have the SAVE attribute in every scoping unit that is not a main program.

5.3.15 TARGET attribute

The **TARGET attribute** specifies that a data object may have a pointer associated with it (7.4.2). An object without the TARGET attribute shall not have an accessible pointer associated with it.

C542 An entity with the TARGET attribute shall be a variable.

C543 An entity with the TARGET attribute shall not have the POINTER attribute.

NOTE 5.16

In addition to variables explicitly declared to have the TARGET attribute, the objects created by allocation of pointers (6.3.1.2) have the TARGET attribute.

If an object has the TARGET attribute, then all of its nonpointer subobjects also have the TARGET attribute.

NOTE 5.17

Examples of TARGET attribute specifications are:

```
TYPE (NODE), TARGET :: HEAD
REAL, DIMENSION (1000, 1000), TARGET :: A, B
```

For a more elaborate example see C.2.2.

NOTE 5.18

Every object designator that starts from a target object will have either the TARGET or POINTER attribute. If pointers are involved, the designator might not necessarily be a subobject of the original target object, but because pointers may point only to targets, there is no way to end up at a nonpointer that is not a target.

5.3.16 VALUE attribute

The **VALUE attribute** specifies a type of argument association (12.4.1.2) for a dummy argument.

C544 An entity with the VALUE attribute shall be a scalar dummy data object.

C545 An entity with the VALUE attribute shall not have the ALLOCATABLE, INTENT(INOUT), INTENT(OUT), POINTER, or VOLATILE attributes.

C546 If an entity has the VALUE attribute, any length type parameter value in its declaration shall be omitted or specified by an initialization expression.

5.3.17 VOLATILE attribute

The **VOLATILE attribute** specifies that an object may be referenced, defined, or become undefined, by means not specified by the program.

C547 An entity with the VOLATILE attribute shall be a variable that is not an INTENT(IN) dummy argument.

An object may have the VOLATILE attribute in a particular scoping unit without necessarily having it in other scoping units (11.2.1, 16.4.1.3). If an object has the VOLATILE attribute, then all of its subobjects also have the VOLATILE attribute.

NOTE 5.19

The Fortran processor should use the most recent definition of a volatile object when a value is required. Likewise, it should make the most recent Fortran definition available. It is the programmer's responsibility to manage any interaction with non-Fortran processes.

A pointer with the VOLATILE attribute may additionally have its association status, dynamic type and type parameters, and array bounds changed by means not specified by the program.

NOTE 5.20

If the target of a pointer is referenced, defined, or becomes undefined, by means not specified by the program, while the pointer is associated with the target, then the pointer shall have the VOLATILE attribute. Usually a pointer should have the VOLATILE attribute if its target has the VOLATILE attribute. Similarly, all members of an EQUIVALENCE group should have the

NOTE 5.20 (cont.)

> VOLATILE attribute if one member has the VOLATILE attribute.

An allocatable object with the VOLATILE attribute may additionally have its allocation status, dynamic type and type parameters, and array bounds changed by means not specified by the program.

5.4 Attribute specification statements

5.4.1 Accessibility statements

R517	*access-stmt*	**is**	*access-spec* [[::] *access-id-list*]
R518	*access-id*	**is**	*use-name*
		or	*generic-spec*

C548 (R517) An *access-stmt* shall appear only in the *specification-part* of a module. Only one accessibility statement with an omitted *access-id-list* is permitted in the *specification-part* of a module.

C549 (R518) Each *use-name* shall be the name of a named variable, procedure, derived type, named constant, or namelist group.

An *access-stmt* with an *access-id-list* specifies the accessibility attribute (5.3.2), PUBLIC or PRIVATE, of each *access-id* in the list. An *access-stmt* without an *access-id* list specifies the default accessibility that applies to all potentially accessible identifiers in the *specification-part* of the module. The statement

```
PUBLIC
```

specifies a default of public accessibility. The statement

```
PRIVATE
```

specifies a default of private accessibility. If no such statement appears in a module, the default is public accessibility.

NOTE 5.21

Examples of accessibility statements are:

```
MODULE EX
   PRIVATE
   PUBLIC :: A, B, C, ASSIGNMENT (=), OPERATOR (+)
```

5.4.2 ALLOCATABLE statement

R519	*allocatable-stmt*	**is**	ALLOCATABLE [::] ■
			■ *object-name* [(*array-spec*)] ■
			■ [, *object-name* [(*array-spec*)]] ...

This statement specifies the ALLOCATABLE attribute (5.3.3) for a list of objects.

NOTE 5.22

An example of an ALLOCATABLE statement is:

```
REAL A, B (:), SCALAR
```

NOTE 5.22 (cont.)

```
ALLOCATABLE :: A (:, :), B, SCALAR
```

5.4.3 ASYNCHRONOUS statement

R520	*asynchronous-stmt*	**is**	ASYNCHRONOUS [::] *object-name-list*

The ASYNCHRONOUS statement specifies the ASYNCHRONOUS attribute (5.3.4) for a list of objects.

5.4.4 BIND statement

R521	*bind-stmt*	**is**	*language-binding-spec* [::] *bind-entity-list*
R522	*bind-entity*	**is**	*entity-name*
		or	/ *common-block-name* /

C550 (R521) If the *language-binding-spec* has a NAME= specifier, the *bind-entity-list* shall consist of a single *bind-entity*.

The BIND statement specifies the BIND attribute (5.3.5) for a list of variables and common blocks.

5.4.5 DATA statement

R523	*data-stmt*	**is**	DATA *data-stmt-set* [[,] *data-stmt-set*] ...

This statement specifies explicit initialization (5.2.2).

A variable, or part of a variable, shall not be explicitly initialized more than once in a program. If a nonpointer object has been specified with default initialization in a type definition, it shall not appear in a *data-stmt-object-list*.

A variable that appears in a DATA statement and has not been typed previously may appear in a subsequent type declaration only if that declaration confirms the implicit typing. An array name, array section, or array element that appears in a DATA statement shall have had its array properties established by a previous specification statement.

Except for variables in named common blocks, a named variable has the SAVE attribute if any part of it is initialized in a DATA statement, and this may be confirmed by explicit specification.

R524	*data-stmt-set*	**is**	*data-stmt-object-list* / *data-stmt-value-list* /
R525	*data-stmt-object*	**is**	*variable*
		or	*data-implied-do*
R526	*data-implied-do*	**is**	(*data-i-do-object-list* , *data-i-do-variable* = ■
			■ *scalar-int-expr* , *scalar-int-expr* [, *scalar-int-expr*])
R527	*data-i-do-object*	**is**	*array-element*
		or	*scalar-structure-component*
		or	*data-implied-do*
R528	*data-i-do-variable*	**is**	*scalar-int-variable*

C551 (R525) In a *variable* that is a *data-stmt-object*, any subscript, section subscript, substring starting point, and substring ending point shall be an initialization expression.

C552 (R525) A variable whose designator appears as a *data-stmt-object* or a *data-i-do-object* shall not be a dummy argument, made accessible by use association or host association, in a named common block unless the DATA statement is in a block data program unit, in a blank common block, a function name, a function result name, an automatic object, or an allocatable variable.

C553 (R525) A *data-i-do-object* or a *variable* that appears as a *data-stmt-object* shall not be an object designator in which a pointer appears other than as the entire rightmost *part-ref*.

C554 (R528) The *data-i-do-variable* shall be a named variable.

C555 (R526) A *scalar-int-expr* of a *data-implied-do* shall involve as primaries only constants, subobjects of constants, or DO variables of the containing *data-implied-do*s, and each operation shall be intrinsic.

C556 (R527) The *array-element* shall be a variable.

C557 (R527) The *scalar-structure-component* shall be a variable.

C558 (R527) The *scalar-structure-component* shall contain at least one *part-ref* that contains a *subscript-list*.

C559 (R527) In an *array-element* or a *scalar-structure-component* that is a *data-i-do-object*, any subscript shall be an expression whose primaries are either constants, subobjects of constants, or DO variables of this *data-implied-do* or the containing *data-implied-do*s, and each operation shall be intrinsic.

R529	*data-stmt-value*	**is**	[*data-stmt-repeat* *] *data-stmt-constant*
R530	*data-stmt-repeat*	**is**	*scalar-int-constant*
		or	*scalar-int-constant-subobject*

C560 (R530) The *data-stmt-repeat* shall be positive or zero. If the *data-stmt-repeat* is a named constant, it shall have been declared previously in the scoping unit or made accessible by use association or host association.

R531	*data-stmt-constant*	**is**	*scalar-constant*
		or	*scalar-constant-subobject*
		or	*signed-int-literal-constant*
		or	*signed-real-literal-constant*
		or	*null-init*
		or	*structure-constructor*

C561 (R531) If a DATA statement constant value is a named constant or a structure constructor, the named constant or derived type shall have been declared previously in the scoping unit or made accessible by use or host association.

C562 (R531) If a *data-stmt-constant* is a *structure-constructor*, it shall be an initialization expression.

R532	*int-constant-subobject*	**is**	*constant-subobject*

C563 (R532) *int-constant-subobject* shall be of type integer.

R533	*constant-subobject*	**is**	*designator*

C564 (R533) *constant-subobject* shall be a subobject of a constant.

C565 (R533) Any subscript, substring starting point, or substring ending point shall be an initialization expression.

The *data-stmt-object-list* is expanded to form a sequence of pointers and scalar variables, referred to as "sequence of variables" in subsequent text. A nonpointer array whose unqualified name appears as a *data-stmt-object* or *data-i-do-object* is equivalent to a complete sequence of its array elements in array element order (6.2.2.2). An array section is equivalent to the sequence of its array elements in array element order. A *data-implied-do* is expanded to form a sequence of array elements and structure components, under the control of the *data-i-do-variable*, as in the DO construct (8.1.6.4).

The *data-stmt-value-list* is expanded to form a sequence of *data-stmt-constant*s. A *data-stmt-repeat* indicates the number of times the following *data-stmt-constant* is to be included in the sequence; omission of a *data-stmt-repeat* has the effect of a repeat factor of 1.

A zero-sized array or a *data-implied-do* with an iteration count of zero contributes no variables to the expanded sequence of variables, but a zero-length scalar character variable does contribute a variable to the expanded sequence. A *data-stmt-constant* with a repeat factor of zero contributes no *data-stmt-constant*s to the expanded sequence of scalar *data-stmt-constant*s.

The expanded sequences of variables and *data-stmt-constant*s are in one-to-one correspondence. Each *data-stmt-constant* specifies the initial value or *null-init* for the corresponding variable. The lengths of the two expanded sequences shall be the same.

A *data-stmt-constant* shall be *null-init* if and only if the corresponding *data-stmt-object* has the POINTER attribute. The initial association status of a pointer *data-stmt-object* is disassociated.

A *data-stmt-constant* other than *null-init* shall be compatible with its corresponding variable according to the rules of intrinsic assignment (7.4.1.2). The variable is initially defined with the value specified by the *data-stmt-constant*; if necessary, the value is converted according to the rules of intrinsic assignment (7.4.1.3) to a value that agrees in type, type parameters, and shape with the variable.

If a *data-stmt-constant* is a *boz-literal-constant*, the corresponding variable shall be of type integer. The *boz-literal-constant* is treated as if it were an *int-literal-constant* with a *kind-param* that specifies the representation method with the largest decimal exponent range supported by the processor.

NOTE 5.23

Examples of DATA statements are:

```
CHARACTER (LEN = 10)  NAME
INTEGER, DIMENSION (0:9) :: MILES
REAL, DIMENSION (100, 100) :: SKEW
TYPE (NODE), POINTER :: HEAD_OF_LIST
TYPE (PERSON) MYNAME, YOURNAME
DATA NAME / 'JOHN DOE' /, MILES / 10 * 0 /
DATA ((SKEW (K, J), J = 1, K), K = 1, 100) / 5050 * 0.0 /
DATA ((SKEW (K, J), J = K + 1, 100), K = 1, 99) / 4950 * 1.0 /
DATA HEAD_OF_LIST / NULL() /
DATA MYNAME / PERSON (21, 'JOHN SMITH') /
DATA YOURNAME % AGE, YOURNAME % NAME / 35, 'FRED BROWN' /
```

The character variable NAME is initialized with the value JOHN DOE with padding on the right because the length of the constant is less than the length of the variable. All ten elements of the integer array MILES are initialized to zero. The two-dimensional array SKEW is initialized so that the lower triangle of SKEW is zero and the strict upper triangle is one. The structures MYNAME and YOURNAME are declared using the derived type PERSON from Note 4.18. The pointer HEAD_OF_LIST is declared using the derived type NODE from Note 4.37; it is initially disassociated. MYNAME is initialized by a structure constructor. YOURNAME is initialized by supplying a separate value for each component.

5.4.6 DIMENSION statement

R534 *dimension-stmt* **is** DIMENSION [::] *array-name* (*array-spec*) ■
■ [, *array-name* (*array-spec*)] ...

This statement specifies the DIMENSION attribute (5.3.6) for a list of objects.

NOTE 5.24

An example of a DIMENSION statement is:

```
DIMENSION A (10), B (10, 70), C (:)
```

5.4.7 INTENT statement

R535 *intent-stmt* **is** INTENT (*intent-spec*) [::] *dummy-arg-name-list*

This statement specifies the INTENT attribute (5.3.8) for the dummy arguments in the list.

NOTE 5.25

An example of an INTENT statement is:

```
SUBROUTINE EX (A, B)
   INTENT (INOUT) :: A, B
```

5.4.8 OPTIONAL statement

R536 *optional-stmt* **is** OPTIONAL [::] *dummy-arg-name-list*

This statement specifies the OPTIONAL attribute (5.3.10) for the dummy arguments in the list.

NOTE 5.26

An example of an OPTIONAL statement is:

```
SUBROUTINE EX (A, B)
   OPTIONAL :: B
```

5.4.9 PARAMETER statement

The **PARAMETER statement** specifies the PARAMETER attribute (5.3.11) and the values for the named constants in the list.

R537 *parameter-stmt* **is** PARAMETER (*named-constant-def-list*)
R538 *named-constant-def* **is** *named-constant* = *initialization-expr*

If a named constant is defined by a PARAMETER statement, it shall not be subsequently declared to have a type or type parameter value that differs from the type and type parameters it would have if declared implicitly (5.5). A named array constant defined by a PARAMETER statement shall have its shape specified in a prior specification statement.

The value of each named constant is that specified by the corresponding initialization expression; if necessary, the value is converted according to the rules of intrinsic assignment (7.4.1.3) to a value that agrees in type, type parameters, and shape with the named constant.

NOTE 5.27

An example of a PARAMETER statement is:

```
PARAMETER (MODULUS = MOD (28, 3), NUMBER_OF_SENATORS = 100)
```

5.4.10 POINTER statement

R539 *pointer-stmt* **is** POINTER [::] *pointer-decl-list*
R540 *pointer-decl* **is** *object-name* [(*deferred-shape-spec-list*)]
or *proc-entity-name*

This statement specifies the POINTER attribute (5.3.12) for a list of entities.

NOTE 5.28

An example of a POINTER statement is:

```
TYPE (NODE) :: CURRENT
POINTER :: CURRENT, A (:, :)
```

5.4.11 PROTECTED statement

R541 *protected-stmt* **is** PROTECTED [::] *entity-name-list*

The **PROTECTED statement** specifies the PROTECTED attribute (5.3.13) for a list of entities.

5.4.12 SAVE statement

R542 *save-stmt* **is** SAVE [[::] *saved-entity-list*]
R543 *saved-entity* **is** *object-name*
or *proc-pointer-name*
or / *common-block-name* /
R544 *proc-pointer-name* **is** *name*

C566 (R542) If a SAVE statement with an omitted saved entity list appears in a scoping unit, no other appearance of the SAVE *attr-spec* or SAVE statement is permitted in that scoping unit.

A SAVE statement with a saved entity list specifies the SAVE attribute (5.3.14) for a list of entities. A SAVE statement without a saved entity list is treated as though it contained the names of all allowed items in the same scoping unit.

NOTE 5.29

An example of a SAVE statement is:

```
SAVE A, B, C, / BLOCKA /, D
```

5.4.13 TARGET statement

R545 *target-stmt* **is** TARGET [::] *object-name* [(*array-spec*)] ■
■ [, *object-name* [(*array-spec*)]] ...

This statement specifies the TARGET attribute (5.3.15) for a list of objects.

NOTE 5.30

An example of a TARGET statement is:

```
TARGET :: A (1000, 1000), B
```

5.4.14 VALUE statement

R546	*value-stmt*	**is**	VALUE [::] *dummy-arg-name-list*

The VALUE statement specifies the VALUE attribute (5.3.16) for a list of dummy arguments.

5.4.15 VOLATILE statement

R547	*volatile-stmt*	**is**	VOLATILE [::] *object-name-list*

The VOLATILE statement specifies the VOLATILE attribute (5.3.17) for a list of objects.

5.5 IMPLICIT statement

In a scoping unit, an **IMPLICIT statement** specifies a type, and possibly type parameters, for all implicitly typed data entities whose names begin with one of the letters specified in the statement. Alternatively, it may indicate that no implicit typing rules are to apply in a particular scoping unit.

R548	*implicit-stmt*	**is**	IMPLICIT *implicit-spec-list*
		or	IMPLICIT NONE
R549	*implicit-spec*	**is**	*declaration-type-spec* (*letter-spec-list*)
R550	*letter-spec*	**is**	*letter* [– *letter*]

C567 (R548) If IMPLICIT NONE is specified in a scoping unit, it shall precede any PARAMETER statements that appear in the scoping unit and there shall be no other IMPLICIT statements in the scoping unit.

C568 (R550) If the minus and second *letter* appear, the second letter shall follow the first letter alphabetically.

A *letter-spec* consisting of two *letter*s separated by a minus is equivalent to writing a list containing all of the letters in alphabetical order in the alphabetic sequence from the first letter through the second letter. For example, A–C is equivalent to A, B, C. The same letter shall not appear as a single letter, or be included in a range of letters, more than once in all of the IMPLICIT statements in a scoping unit.

In each scoping unit, there is a mapping, which may be null, between each of the letters A, B, ..., Z and a type (and type parameters). An IMPLICIT statement specifies the mapping for the letters in its *letter-spec-list*. IMPLICIT NONE specifies the null mapping for all the letters. If a mapping is not specified for a letter, the default for a program unit or an interface body is default integer if the letter is I, J, ..., or N and default real otherwise, and the default for an internal or module procedure is the mapping in the host scoping unit.

Any data entity that is not explicitly declared by a type declaration statement, is not an intrinsic function, and is not made accessible by use association or host association is declared implicitly to be of the type (and type parameters) mapped from the first letter of its name, provided the mapping is not null. The mapping for the first letter of the data entity shall either have been established by a prior IMPLICIT statement or be the default mapping for the letter. The mapping may be to a derived type that is inaccessible in the local scope if the derived type is accessible in the host scope. The data entity is treated as if it were declared in an explicit type declaration in the outermost scoping unit in which it appears. An explicit type specification in a FUNCTION statement overrides an IMPLICIT statement for the name of the result variable of that function subprogram.

NOTE 5.31

The following are examples of the use of IMPLICIT statements:

```
MODULE EXAMPLE_MODULE
   IMPLICIT NONE
   ...
   INTERFACE
      FUNCTION FUN (I)    ! Not all data entities need to
         INTEGER FUN      ! be declared explicitly
      END FUNCTION FUN
   END INTERFACE
CONTAINS
   FUNCTION JFUN (J)      ! All data entities need to
      INTEGER JFUN, J     ! be declared explicitly.
      ...
   END FUNCTION JFUN
END MODULE EXAMPLE_MODULE
SUBROUTINE SUB
   IMPLICIT COMPLEX (C)
   C = (3.0, 2.0)      ! C is implicitly declared COMPLEX
   ...
CONTAINS
   SUBROUTINE SUB1
      IMPLICIT INTEGER (A, C)
      C = (0.0, 0.0)   ! C is host associated and of
                       ! type complex
      Z = 1.0          ! Z is implicitly declared REAL
      A = 2            ! A is implicitly declared INTEGER
      CC = 1           ! CC is implicitly declared INTEGER
      ...
   END SUBROUTINE SUB1
   SUBROUTINE SUB2
      Z = 2.0          ! Z is implicitly declared REAL and
                       ! is different from the variable of
                       ! the same name in SUB1
      ...
   END SUBROUTINE SUB2
   SUBROUTINE SUB3
      USE EXAMPLE_MODULE  ! Accesses integer function FUN
                          ! by use association
      Q = FUN (K)         ! Q is implicitly declared REAL and
      ...                 ! K is implicitly declared INTEGER
   END SUBROUTINE SUB3
END SUBROUTINE SUB
```

NOTE 5.32

The following is an example of a mapping to a derived type that is inaccessible in the local scope:

```
     PROGRAM MAIN
       IMPLICIT TYPE(BLOB) (A)
```

NOTE 5.32 (cont.)

```
    TYPE BLOB
      INTEGER :: I
    END TYPE BLOB
    TYPE(BLOB) :: B
    CALL STEVE
  CONTAINS
    SUBROUTINE STEVE
      INTEGER :: BLOB
      ..
      AA = B
      ..
    END SUBROUTINE STEVE
  END PROGRAM MAIN
```

In the subroutine STEVE, it is not possible to explicitly declare a variable to be of type BLOB because BLOB has been given a different meaning, but implicit mapping for the letter A still maps to type BLOB, so AA is of type BLOB.

5.6 NAMELIST statement

A **NAMELIST statement** specifies a group of named data objects, which may be referred to by a single name for the purpose of data transfer (9.5, 10.10).

R551 *namelist-stmt* **is** NAMELIST ■
■ / *namelist-group-name* / *namelist-group-object-list* ■
■ [[,] / *namelist-group-name* / ■
■ *namelist-group-object-list*] ...

C569 (R551) The *namelist-group-name* shall not be a name accessed by use association.

R552 *namelist-group-object* **is** *variable-name*

C570 (R552) A *namelist-group-object* shall not be an assumed-size array.

C571 (R551) A *namelist-group-object* shall not have the PRIVATE attribute if the *namelist-group-name* has the PUBLIC attribute.

The order in which the variables are specified in the NAMELIST statement determines the order in which the values appear on output.

Any *namelist-group-name* may occur more than once in the NAMELIST statements in a scoping unit. The *namelist-group-object-list* following each successive appearance of the same *namelist-group-name* in a scoping unit is treated as a continuation of the list for that *namelist-group-name*.

A namelist group object may be a member of more than one namelist group.

A namelist group object shall either be accessed by use or host association or shall have its type, type parameters, and shape specified by previous specification statements or the procedure heading in the same scoping unit or by the implicit typing rules in effect for the scoping unit. If a namelist group object is typed by the implicit typing rules, its appearance in any subsequent type declaration statement shall confirm the implied type and type parameters.

NOTE 5.33

An example of a NAMELIST statement is:

```
NAMELIST /NLIST/ A, B, C
```

5.7 Storage association of data objects

5.7.1 EQUIVALENCE statement

5.7.1.1 General

An **EQUIVALENCE statement** is used to specify the sharing of storage units by two or more objects in a scoping unit. This causes storage association (16.4.3) of the objects that share the storage units.

If the equivalenced objects have differing type or type parameters, the EQUIVALENCE statement does not cause type conversion or imply mathematical equivalence. If a scalar and an array are equivalenced, the scalar does not have array properties and the array does not have the properties of a scalar.

R553	*equivalence-stmt*	**is**	EQUIVALENCE *equivalence-set-list*
R554	*equivalence-set*	**is**	(*equivalence-object* , *equivalence-object-list*)
R555	*equivalence-object*	**is**	*variable-name*
		or	*array-element*
		or	*substring*

C572 (R555) An *equivalence-object* shall not be a designator with a base object that is a dummy argument, a pointer, an allocatable variable, a derived-type object that has an allocatable ultimate component, an object of a nonsequence derived type, an object of a derived type that has a pointer at any level of component selection, an automatic object, a function name, an entry name, a result name, a variable with the BIND attribute, a variable in a common block that has the BIND attribute, or a named constant.

C573 (R555) An *equivalence-object* shall not be a designator that has more than one *part-ref*.

C574 (R555) An *equivalence-object* shall not have the TARGET attribute.

C575 (R555) Each subscript or substring range expression in an *equivalence-object* shall be an integer initialization expression (7.1.7).

C576 (R554) If an *equivalence-object* is of type default integer, default real, double precision real, default complex, default logical, or numeric sequence type, all of the objects in the equivalence set shall be of these types.

C577 (R554) If an *equivalence-object* is of type default character or character sequence type, all of the objects in the equivalence set shall be of these types.

C578 (R554) If an *equivalence-object* is of a sequence derived type that is not a numeric sequence or character sequence type, all of the objects in the equivalence set shall be of the same type with the same type parameter values.

C579 (R554) If an *equivalence-object* is of an intrinsic type other than default integer, default real, double precision real, default complex, default logical, or default character, all of the objects in the equivalence set shall be of the same type with the same kind type parameter value.

C580 (R555) If an *equivalence-object* has the PROTECTED attribute, all of the objects in the equivalence set shall have the PROTECTED attribute.

C581 (R555) The name of an *equivalence-object* shall not be a name made accessible by use association.

C582 (R555) A *substring* shall not have length zero.

NOTE 5.34

The EQUIVALENCE statement allows the equivalencing of sequence structures and the equivalencing of objects of intrinsic type with nondefault type parameters, but there are strict rules regarding the appearance of these objects in an EQUIVALENCE statement.

A structure that appears in an EQUIVALENCE statement shall be a sequence structure. If a sequence structure is not of numeric sequence type or of character sequence type, it shall be equivalenced only to objects of the same type with the same type parameter values.

A structure of a numeric sequence type shall be equivalenced only to another structure of a numeric sequence type, an object of default integer type, default real type, double precision real type, default complex type, or default logical type such that components of the structure ultimately become associated only with objects of these types.

A structure of a character sequence type shall be equivalenced only to an object of default character type or another structure of a character sequence type.

An object of intrinsic type with nondefault kind type parameters shall not be equivalenced to objects of different type or kind type parameters.

Further rules on the interaction of EQUIVALENCE statements and default initialization are given in 16.4.3.3.

5.7.1.2 Equivalence association

An EQUIVALENCE statement specifies that the storage sequences (16.4.3.1) of the data objects specified in an *equivalence-set* are storage associated. All of the nonzero-sized sequences in the *equivalence-set*, if any, have the same first storage unit, and all of the zero-sized sequences in the *equivalence-set*, if any, are storage associated with one another and with the first storage unit of any nonzero-sized sequences. This causes the storage association of the data objects in the *equivalence-set* and may cause storage association of other data objects.

5.7.1.3 Equivalence of default character objects

A data object of type default character shall not be equivalenced to an object that is not of type default character and not of a character sequence type. The lengths of the equivalenced character objects need not be the same.

An EQUIVALENCE statement specifies that the storage sequences of all the default character data objects specified in an *equivalence-set* are storage associated. All of the nonzero-sized sequences in the *equivalence-set*, if any, have the same first character storage unit, and all of the zero-sized sequences in the *equivalence-set*, if any, are storage associated with one another and with the first character storage unit of any nonzero-sized sequences. This causes the storage association of the data objects in the *equivalence-set* and may cause storage association of other data objects.

NOTE 5.35

For example, using the declarations:

```
CHARACTER (LEN = 4) :: A, B
CHARACTER (LEN = 3) :: C (2)
```

NOTE 5.35 (cont.)

```
EQUIVALENCE (A, C (1)), (B, C (2))
```

the association of A, B, and C can be illustrated graphically as:

```
  1      2      3      4      5      6      7
 |---    --- A  ---    ---|
                      |---    --- B  ---    ---|
 |---    C(1)   ---|  |---    C(2)   ---|
```

5.7.1.4 Array names and array element designators

For a nonzero-sized array, the use of the array name unqualified by a subscript list as an *equivalence-object* has the same effect as using an array element designator that identifies the first element of the array.

5.7.1.5 Restrictions on EQUIVALENCE statements

An EQUIVALENCE statement shall not specify that the same storage unit is to occur more than once in a storage sequence.

NOTE 5.36

For example:

```
REAL, DIMENSION (2) :: A
REAL :: B
EQUIVALENCE (A (1), B), (A (2), B) ! Not standard conforming
```

is prohibited, because it would specify the same storage unit for A (1) and A (2).

An EQUIVALENCE statement shall not specify that consecutive storage units are to be nonconsecutive.

NOTE 5.37

For example, the following is prohibited:

```
REAL A (2)
DOUBLE PRECISION D (2)
EQUIVALENCE (A (1), D (1)), (A (2), D (2)) ! Not standard conforming
```

5.7.2 COMMON statement

5.7.2.1 General

The **COMMON statement** specifies blocks of physical storage, called **common blocks**, that can be accessed by any of the scoping units in a program. Thus, the COMMON statement provides a global data facility based on storage association (16.4.3).

The common blocks specified by the COMMON statement may be named and are called **named common blocks**, or may be unnamed and are called **blank common**.

R556	*common-stmt*	**is**	COMMON ■
			■ [/ [*common-block-name*] /] *common-block-object-list* ■
			■ [[,] / [*common-block-name*] / ■

■ *common-block-object-list*] ...

R557 *common-block-object* **is** *variable-name* [(*array-spec*)]
or *proc-pointer-name*

C583 (R557) An *array-spec* in a *common-block-object* shall be an *explicit-shape-spec-list*.

C584 (R557) Only one appearance of a given *variable-name* or *proc-pointer-name* is permitted in all *common-block-object-lists* within a scoping unit.

C585 (R557) A *common-block-object* shall not be a dummy argument, an allocatable variable, a derived-type object with an ultimate component that is allocatable, an automatic object, a function name, an entry name, a variable with the BIND attribute, or a result name.

C586 (R557) If a *common-block-object* is of a derived type, it shall be a sequence type (4.5.2) or a type with the BIND attribute and it shall have no default initialization.

C587 (R557) A *variable-name* or *proc-pointer-name* shall not be a name made accessible by use association.

In each COMMON statement, the data objects whose names appear in a common block object list following a common block name are declared to be in that common block. If the first common block name is omitted, all data objects whose names appear in the first common block object list are specified to be in blank common. Alternatively, the appearance of two slashes with no common block name between them declares the data objects whose names appear in the common block object list that follows to be in blank common.

Any common block name or an omitted common block name for blank common may occur more than once in one or more COMMON statements in a scoping unit. The common block list following each successive appearance of the same common block name in a scoping unit is treated as a continuation of the list for that common block name. Similarly, each blank common block object list in a scoping unit is treated as a continuation of blank common.

The form *variable-name* (*array-spec*) specifies the DIMENSION attribute for that variable.

If derived-type objects of numeric sequence type (4.5.2) or character sequence type (4.5.2) appear in common, it is as if the individual components were enumerated directly in the common list.

NOTE 5.38

Examples of COMMON statements are:

```
COMMON /BLOCKA/ A, B, D (10, 30)
COMMON I, J, K
```

5.7.2.2 Common block storage sequence

For each common block in a scoping unit, a **common block storage sequence** is formed as follows:

(1) A storage sequence is formed consisting of the sequence of storage units in the storage sequences (16.4.3.1) of all data objects in the common block object lists for the common block. The order of the storage sequences is the same as the order of the appearance of the common block object lists in the scoping unit.

(2) The storage sequence formed in (1) is extended to include all storage units of any storage sequence associated with it by equivalence association. The sequence shall be extended only by adding storage units beyond the last storage unit. Data objects associated with an entity in a common block are considered to be in that common block.

Only COMMON statements and EQUIVALENCE statements appearing in the scoping unit contribute to common block storage sequences formed in that scoping unit.

5.7.2.3 Size of a common block

The **size of a common block** is the size of its common block storage sequence, including any extensions of the sequence resulting from equivalence association.

5.7.2.4 Common association

Within a program, the common block storage sequences of all nonzero-sized common blocks with the same name have the same first storage unit, and the common block storage sequences of all zero-sized common blocks with the same name are storage associated with one another. Within a program, the common block storage sequences of all nonzero-sized blank common blocks have the same first storage unit and the storage sequences of all zero-sized blank common blocks are associated with one another and with the first storage unit of any nonzero-sized blank common blocks. This results in the association of objects in different scoping units. Use association or host association may cause these associated objects to be accessible in the same scoping unit.

A nonpointer object of default integer type, default real type, double precision real type, default complex type, default logical type, or numeric sequence type shall be associated only with nonpointer objects of these types.

A nonpointer object of type default character or character sequence type shall be associated only with nonpointer objects of these types.

A nonpointer object of a derived type that is not a numeric sequence or character sequence type shall be associated only with nonpointer objects of the same type with the same type parameter values.

A nonpointer object of intrinsic type other than default integer, default real, double precision real, default complex, default logical, or default character shall be associated only with nonpointer objects of the same type and type parameters.

A data pointer shall be storage associated only with data pointers of the same type and rank. Data pointers that are storage associated shall have deferred the same type parameters; corresponding non-deferred type parameters shall have the same value. A procedure pointer shall be storage associated only with another procedure pointer; either both interfaces shall be explicit or both interfaces shall be implicit. If the interfaces are explicit, the characteristics shall be the same. If the interfaces are implicit, either both shall be subroutines or both shall be functions with the same type and type parameters.

An object with the TARGET attribute shall be storage associated only with another object that has the TARGET attribute and the same type and type parameters.

NOTE 5.39

A common block is permitted to contain sequences of different storage units, provided each scoping unit that accesses the common block specifies an identical sequence of storage units for the common block. For example, this allows a single common block to contain both numeric and character storage units.

Association in different scoping units between objects of default type, objects of double precision real type, and sequence structures is permitted according to the rules for equivalence objects (5.7.1).

5.7.2.5 Differences between named common and blank common

A blank common block has the same properties as a named common block, except for the following.

(1) Execution of a RETURN or END statement may cause data objects in a named common block to become undefined unless the common block has the SAVE attribute, but never causes data objects in blank common to become undefined (16.5.6).

(2) Named common blocks of the same name shall be of the same size in all scoping units of a program in which they appear, but blank common blocks may be of different sizes.

(3) A data object in a named common block may be initially defined by means of a DATA statement or type declaration statement in a block data program unit (11.3), but objects in blank common shall not be initially defined.

5.7.3 Restrictions on common and equivalence

An EQUIVALENCE statement shall not cause the storage sequences of two different common blocks to be associated.

Equivalence association shall not cause a derived-type object with default initialization to be associated with an object in a common block.

Equivalence association shall not cause a common block storage sequence to be extended by adding storage units preceding the first storage unit of the first object specified in a COMMON statement for the common block.

NOTE 5.40

For example, the following is not permitted:

```
COMMON /X/ A
REAL B (2)
EQUIVALENCE (A, B (2))    ! Not standard conforming
```