Section 5: Data object declarations and specifications

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 (5.3). All of its attributes may be included in a type declaration statement or may be specified individually in separate specification statements.

NOTE 5.1

For example:

```
INTEGER :: INCOME, EXPENDITURE
```

declares the two data objects named INCOME and EXPENDITURE to have the type integer.

```
REAL, DIMENSION (-5:+5) :: X, Y, Z
```

declares three data objects with names X, Y, and Z. These all have default real type and are explicit-shape rank-one arrays with a lower bound of –5, an upper bound of +5, and therefore a size of 11.

5.1 Type declaration statements

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

R502 *declaration-type-spec*
is *type-spec*
or CLASS (*derived-type-spec*)
or CLASS (*)

C501 (R502) In a *declaration-type-spec*, every *type-param-value* that is not a colon or an asterisk shall be a *specification-expr*.

C502 (R502) In a *declaration-type-spec* that uses the CLASS keyword, *derived-type-spec* shall specify an extensible type.

NOTE 5.2

A *declaration-type-spec* is used in a nonexecutable statement; a *type-spec* is used in an array constructor or an ALLOCATE statement.

R503 *type-spec*
is INTEGER [*kind-selector*]
or REAL [*kind-selector*]
or DOUBLE PRECISION
or COMPLEX [*kind-selector*]
or CHARACTER [*char-selector*]
or LOGICAL [*kind-selector*]
or TYPE (*derived-type-spec*)
or TYPE (*type-alias-name*)

C503 (R503) A *type-alias-name* shall be the name of a type alias.

R504 *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 SAVE
or TARGET
or VALUE
or VOLATILE

R505 *entity-decl* **is** *object-name* [(*array-spec*)] [* *char-length*] [*initialization*]
or *function-name* [* *char-length*]

C504 (R505) If a *type-param-value* in an *entity-decl* is not a colon or an asterisk, it shall be a *specification-expr*.

R506 *object-name* **is** *name*

C505 (R506) The *object-name* shall be the name of a data object.

R507 *initialization* **is** = *initialization-expr*
or => NULL ( )

R508 *kind-selector* **is** ([KIND =] *scalar-int-initialization-expr*)

C506 (R501) The same *attr-spec* shall not appear more than once in a given *type-declaration-stmt*.

C507 An entity shall not be explicitly given any attribute more than once in a scoping unit.

C508 (R501) An entity declared with the CLASS keyword shall be a dummy argument or have the ALLOCATABLE or POINTER attribute.

C509 (R501) An array declared with a POINTER or an ALLOCATABLE attribute shall be specified with an *array-spec* that is a *deferred-shape-spec-list* (5.1.2.5.3).

C510 (R501) An *array-spec* for an *object-name* that is a function result that does not have the ALLOCATABLE or POINTER attribute shall be an *explicit-shape-spec-list*.

C511 (R501) If the POINTER attribute is specified, the ALLOCATABLE, TARGET, EXTERNAL, or INTRINSIC attribute shall not be specified.

C512 (R501) If the TARGET attribute is specified, the POINTER, EXTERNAL, INTRINSIC, or PARAMETER attribute shall not be specified.

C513 (R501) The PARAMETER attribute shall not be specified for a dummy argument, a pointer, an allocatable entity, a function, or an object in a common block.

C514 (R501) The INTENT, VALUE, and OPTIONAL attributes may be specified only for dummy arguments.

C515 (R501) The SAVE attribute shall not be specified for an object that is in a common block, a dummy argument, a procedure, a function result, an automatic data object, or an object with the PARAMETER attribute.

C516 An entity shall not have both the EXTERNAL attribute and the INTRINSIC attribute.

C517 (R501) An entity in an *entity-decl-list* shall not have the EXTERNAL or INTRINSIC attribute specified unless it is a function.

C518 (R505) The * *char-length* option is permitted only if the type specified is character.

C519 (R505) The *function-name* shall be the name of an external function, an intrinsic function, a function dummy procedure, or a statement function.

C520 (R501) The *initialization* shall appear if the statement contains a PARAMETER attribute (5.1.2.10).

C521 (R501) If *initialization* appears, a double-colon separator shall appear before the *entity-decl-list*.

C522 (R505) *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 name, an intrinsic name, or an automatic object.

C523 (R505) If => appears in *initialization*, the object shall have the POINTER attribute. If = appears in *initialization*, the object shall not have the POINTER attribute.

C524 (R503) The value of *scalar-int-initialization-expr* in *kind-selector* shall be nonnegative and shall specify a representation method that exists on the processor.

C525 (R501) If the VOLATILE attribute is specified, the PARAMETER, INTRINSIC, EXTERNAL, or INTENT(IN) attribute shall not be specified.

C526 (R501) If the VALUE attribute is specified, the PARAMETER, EXTERNAL, POINTER, ALLOCATABLE, DIMENSION, VOLATILE, INTENT(INOUT), or INTENT(OUT) attribute shall not be specified.

C527 (R501) If the VALUE attribute is specified for a dummy argument of type character, the length parameter shall be omitted or shall be specified by an initialization expression with the value one.

C528 (R501) The VALUE attribute is permitted only for a scalar dummy argument of a subprogram or interface body that has a *language-binding-spec*.

C529 (R501) The ALLOCATABLE, POINTER, and OPTIONAL attributes shall not be specified for a dummy argument of a subprogram or interface body that has a *language-binding-spec*.

C530 (R504) A *language-binding-spec* shall appear only in the specification part of a module.

C531 (R501) If a *language-binding-spec* is specified, the POINTER, PARAMETER, ALLOCATABLE, EXTERNAL, or INTRINSIC attribute shall not be specified. The entity declared shall be a variable.

C532 (R501) If a *language-binding-spec* with a *bind-spec-list* appears, the *entity-decl-list* shall consist of a single *entity-decl*.

A name that identifies a specific intrinsic function in a scoping unit has a type as specified in 13.10. 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.

A function result may be declared to have the POINTER attribute.

A *specification-expr* in an *array-spec* or a *type-param-value* in a *declaration-type-spec* corresponding to a nonkind type parameter shall be an initialization expression unless it is in an interface body (12.3.2.1), the specification part of a subprogram, or the *declaration-type-spec* of a FUNCTION statement (12.5.2.1). If the data object being declared depends on the value of a *specification-expr* that is not an initialization expression, and it is not a dummy argument, such an object is called an **automatic data object**.

NOTE 5.3

An automatic object shall neither appear in a SAVE or DATA statement nor be declared with a SAVE attribute nor be initially defined by an *initialization*.

If a *length-selector* (5.1.1.5) is an expression that is not an initialization expression, the length is declared at the entry of the procedure and is not affected by any redefinition or undefinition of the variables in the specification expression during execution of the procedure.

If an *entity-decl* contains *initialization* and the *object-name* does not have the PARAMETER attribute, 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 *object-name* is initially defined with the value specified by the *initialization-expr*; if necessary, the value is converted according to the rules of intrinsic assignment (7.5.1.4) to a value that agrees in type, type parameters, and shape with the *object-name*. 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 ( ), *object-name* shall be a pointer, and its initial association status is disassociated. Use of =>NULL ( ) in a scoping unit is a reference to the intrinsic function NULL.

The presence of *initialization* implies that *object-name* is saved, except for an *object-name* in a named common block or an *object-name* with the PARAMETER attribute. The implied SAVE attribute may be reaffirmed by explicit use of the SAVE attribute in the type declaration statement, by inclusion of the *object-name* in a SAVE statement (5.2.11), or by the appearance of a SAVE statement without a *saved-entity-list* in the same scoping unit.

NOTE 5.4

Examples of type declaration statements are:

```
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
TYPE (PERSON) :: CHAIRMAN
TYPE(NODE), POINTER :: HEAD => NULL ( )
TYPE (matrix (kind=8, dim=1000)) :: mat
```

5.1.1 Type specifiers

The **type specifier** in a type declaration statement specifies the type of the entities in the entity declaration list. This explicit type declaration may override or confirm the implicit type that could otherwise be indicated by the first letter of an entity name (5.3).

5.1.1.1 INTEGER

The INTEGER type specifier is used to declare entities of intrinsic type integer (4.4.1). The kind selector, if present, specifies the integer representation method. If the kind selector is absent, the kind type parameter is KIND (0) and the entities declared are of type default integer.

5.1.1.2 REAL

The REAL type specifier is used to declare entities of intrinsic type real (4.4.2). The kind selector, if present, specifies the real approximation method. If the kind selector is absent, the kind type parameter is KIND (0.0) and the entities declared are of type default real.

5.1.1.3 DOUBLE PRECISION

The DOUBLE PRECISION type specifier is used to declare entities of intrinsic type double precision real (4.4.2). The kind parameter value is KIND (0.0D0). An entity declared with a type specifier REAL (KIND (0.0D0)) is of the same kind as one declared with the type specifier DOUBLE PRECISION.

5.1.1.4 COMPLEX

The COMPLEX type specifier is used to declare entities of intrinsic type complex (4.4.3). The kind selector, if present, specifies the real approximation method of the two real values making up the real and imaginary parts of the complex value. If the kind selector is absent, the kind type parameter is KIND (0.0) and the entities declared are of type default complex.

5.1.1.5 CHARACTER

The CHARACTER type specifier is used to declare entities of intrinsic type character (4.4.4).

R509 *char-selector* **is** *length-selector*
or (LEN = *type-param-value* , ■
■ KIND = *scalar-int-initialization-expr*)
or (*type-param-value* , ■
■ [KIND =] *scalar-int-initialization-expr*)
or (KIND = *scalar-int-initialization-expr* ■
■ [, LEN = *type-param-value*])

R510 *length-selector* **is** ([LEN =] *type-param-value*)
or * *char-length* [,]

R511 *char-length* **is** (*type-param-value*)
or *scalar-int-literal-constant*

C533 (R509) The value of *scalar-int-initialization-expr* shall be nonnegative and shall specify a representation method that exists on the processor.

C534 (R511) The *scalar-int-literal-constant* shall not include a *kind-param*.

C535 (R511) A *char-length* type parameter value of * may be used only in the following ways:

J3 internal note

Unresolved issue 337

Because C535 explicitly mentions the bnf term char-length, it "clearly" applies only to that bnf term, which is used only in the obsolescent form of the character specifier. I seriously doubt that was the intent. This also makes me notice that the bnf rule for char-length should probably be in obsolescent font.

(1) to declare a dummy argument,

(2) to declare a named constant,

(3) in the *type-spec* of an ALLOCATE statement wherein each *allocate-object* is a dummy argument of type CHARACTER with an assumed character length, or

(4) in an external function, to declare the character length parameter of the function result.

C536 A function name shall not be declared with an asterisk *type-param-value* unless it is of type CHARACTER and is the name of the result of an external function or the name of a dummy function.

C537 A function name declared with an asterisk *type-param-value* shall not be array-valued, pointer-valued, recursive, or pure.

C538 (R510) The optional comma in a *length-selector* is permitted only in a declaration-type-spec in a *type-declaration-stmt*.

C539 (R510) The optional comma in a *length-selector* is permitted only if no double-colon separator appears in the *type-declaration-stmt*.

C540 (R509) The length specified for a character-valued statement function or for a statement function dummy argument of type character shall be an initialization expression.

The *char-selector* in a CHARACTER *type-spec* and the * *char-length* in an *entity-decl* or in a *component-decl* of a type definition specify character length. The * *char-length* in an *entity-decl* or a *component-decl* specifies an individual length and overrides the length specified in the *char-selector*, if any. If a * *char-length* is not specified in an *entity-decl* or a *component-decl*, the *length-selector* or *type-param-value* specified in the *char-selector* is the character length. If the length is not specified in a *char-selector* or a * *char-length*, the length is 1.

If the character length parameter value evaluates to a negative value, the length of character entities declared is zero. A character length parameter value of : indicates a deferred type parameter (4.2). A *char-length* type parameter value of * has the following meaning:

(1) If used to declare a dummy argument of a procedure, the dummy argument assumes the length of the associated actual argument.

(2) If used to declare a named constant, the length is that of the constant value.

(3) If used in the *type-spec* of an ALLOCATE statement, each *allocate-object* assumes its length from the associated actual argument.

(4) If used to specify the character length parameter of a function result, any scoping unit invoking the function shall declare the function name with a character length parameter value other than * or access such a definition by host or use association. When the function is invoked, the length of the result variable in the function is assumed from the value of this type parameter.

The kind selector, if present, specifies the character representation method. If the kind selector is absent, the kind type parameter is KIND ('A') and the entities declared are of type default character.

NOTE 5.5

Examples of character type declaration statements are:

```
CHARACTER (LEN = 10, KIND = 2)  A
CHARACTER B, C *20
```

5.1.1.6 LOGICAL

The LOGICAL type specifier is used to declare entities of intrinsic type logical (4.4.5).

The kind selector, if present, specifies the representation method. If the kind selector is absent, the kind type parameter is KIND (.FALSE.) and the entities declared are of type default logical.

5.1.1.7 Derived type

A TYPE type specifier is used to declare entities of the derived type specified by the *type-name* of the *derived-type-spec*. The components of each such entity are declared to be of the types specified by the corresponding *component-def* statements of the *derived-type-def* (4.5.1). When a data entity is declared explicitly to be of a derived type, the derived type shall have been defined previously in the scoping unit or be accessible there by use or host association. If the data entity is a function result, the derived type may be specified in the FUNCTION statement provided the derived type is defined within the body of the function or is accessible there by use or host association.

A scalar entity of derived type is a **structure**. If a derived type has the SEQUENCE property, a scalar entity of the type is a **sequence structure**.

5.1.1.8 Polymorphic entities

A **polymorphic** entity is a data entity that is able to be of differing types during program execution. The type of a data entity at a particular point during execution of a program is its **dynamic type**. The **declared type** of a data entity is the data type that it is declared to have, either explicitly or implicitly.

A CLASS type specifier is used to declare polymorphic objects. The declared type of a polymorphic object is the specified type if the CLASS type specifier contains a type name, or no type if the CLASS type specifier contains an asterisk. An object declared with the CLASS(*) specifier is an **unlimited polymorphic** object.

An unlimited polymorphic object is **type-compatible** with all entities of extensible type; any other polymorphic entity is type-compatible with entities of the same type or of any of its extension types. A nonpolymorphic entity is type-compatible only with entities of the same type. An entity is said to be type-compatible with a type if it is type-compatible with entities of that type.

A polymorphic allocatable object may be allocated to be of any type with which it is type-compatible. A polymorphic pointer or dummy argument may, during program execution, be associated with objects with which it is type-compatible.

The dynamic type of an allocatable polymorphic object that is currently allocated is the type with which it was allocated. The dynamic type of a polymorphic pointer that is currently associated is the dynamic type of its target. The dynamic type of a nonallocatable nonpointer polymorphic dummy argument is the dynamic type of its associated actual argument. The dynamic type of an unallocated allocatable or a disassociated pointer is the same as its declared type. The dynamic type of an entity identified by an associate name (8.1.4) is the dynamic type of the selector with which it is associated. The dynamic type of an object that is not polymorphic is its declared type.

NOTE 5.6

Only components of the declared type of a polymorphic object may be designated by component selection (6.1.2).

5.1.2 Attributes

The additional attributes that may appear in the attribute specification of a type declaration statement further specify the nature of the entities being declared or specify restrictions on their use in the program.

5.1.2.1 Accessibility attribute

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

R512 *access-spec* **is** PUBLIC
or PRIVATE

C541 (R512) An *access-spec* shall appear only in the *specification-part* of a module.

Identifiers that are declared with a PRIVATE attribute are not accessible outside the module. Identifiers that are declared with a PUBLIC attribute may be made accessible in other program units by the USE statement. Identifiers without an explicitly specified *access-spec* have default accessibility. Default accessibility for a module is PUBLIC unless it has been changed by a PRIVATE statement (5.2.1).

NOTE 5.7

An example of an accessibility specification is:

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

5.1.2.2 ALLOCATABLE attribute

An object with the **ALLOCATABLE attribute** is one for which space is allocated by an ALLOCATE statement (6.3.1) or by a derived-type intrinsic assignment statement (7.5.1.5). If it is an array it shall be a deferred-shape array.

5.1.2.3 ASYNCHRONOUS attribute

The base object of 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)

The ASYNCHRONOUS attribute may also be conferred implicitly by the use of a variable in an asynchronous input/output statement (9.5.1.4).

NOTE 5.8

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

5.1.2.4 BIND attribute

The BIND attribute specifies that a variable is interoperable with a C variable with external linkage, as described in 15.2.7.

NOTE 5.9

The C 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 standard). The name of such a C identifier may 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 linked (12.5.3, 15.2.7.1) with a Fortran entity.

This standard does not require a processor to provide a means of linking Fortran entities with C entities whose names are specified using the universal character name facility.

R513 *language-binding-spec* **is** BIND (C [, *bind-spec-list*])

R514 *bind-spec* **is** NAME = *scalar-char-initialization-expr*
or BINDNAME = *scalar-char-initialization-expr*

C542 (R513) A *bind-spec-list* shall not have more than one NAME= specifier.

C543 (R513) A *bind-spec-list* that appears in a BIND statement or a type declaration statement shall not have more than one BINDNAME= specifier.

C544 (R514) The *scalar-char-initialization-expr* in a *bind-spec* shall be of default character kind.

The BIND attribute shall not be specified for a variable or common block with characteristics that prevent interoperation with a C global variable (15.2.7). The BIND attribute implies the SAVE attribute, which may be confirmed by explicit specification.

NOTE 5.10

Specifying the BIND attribute for an entity might have no discernable effect for a processor that is its own companion processor.

5.1.2.5 DIMENSION attribute

The **DIMENSION attribute** specifies entities that are arrays. The rank or shape is specified by the *array-spec*, if there is one, in the *entity-decl*, or by the *array-spec* in the DIMENSION *attr-spec* otherwise. An *array-spec* in an *entity-decl* specifies either the rank or the rank and shape for a single array and overrides the *array-spec* in the DIMENSION *attr-spec*. To declare an array in a type declaration statement, either the DIMENSION *attr-spec* shall appear, or an *array-spec* shall appear in the *entity-decl*. The appearance of an *array-spec* in an *entity-decl* specifies the DIMENSION attribute for the entity. The DIMENSION attribute alternatively may be specified in the specification statements DIMENSION, ALLOCATABLE, POINTER, TARGET, or COMMON.

R515 *array-spec* **is** *explicit-shape-spec-list*
or *assumed-shape-spec-list*
or *deferred-shape-spec-list*
or *assumed-size-spec*

C545 (R515)The maximum rank is seven.

NOTE 5.11

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

5.1.2.5.1 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.

R516 *explicit-shape-spec* **is** [*lower-bound* :] *upper-bound*

R517 *lower-bound* **is** *specification-expr*

R518 *upper-bound* **is** *specification-expr*

C546 (R516) An explicit-shape array whose bounds are not initialization expressions shall be a dummy argument, a function result, or an automatic array of a procedure.

An **automatic array** is an explicit-shape array that is declared in a subprogram, is not a dummy argument, and has bounds that are not initialization expressions.

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 any redefinition or undefinition of the specification expression variables 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 number of sets of bounds specified is the rank.

5.1.2.5.2 Assumed-shape array

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

R519 *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 $s + d - 1$. The lower bound is *lower-bound*, if present, and 1 otherwise.

5.1.2.5.3 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 with the ALLOCATABLE attribute shall be declared with a *deferred-shape-spec-list*. Nonkind type parameters may be deferred.

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. An array with the POINTER attribute shall be declared with a *deferred-shape-spec-list*. Nonkind type parameters may be deferred.

R520 *deferred-shape-spec* **is** :

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, or

(2) in a pointer assignment statement (7.5.2).

The bounds of the array target or allocatable array are unaffected by any subsequent redefinition or undefinition of variables involved in the bounds' specification expressions.

5.1.2.5.4 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.

R521 *assumed-size-spec* **is** [*explicit-shape-spec-list* ,] [*lower-bound* :] *

C547 The function name of an array-valued function shall not be declared as an assumed-size array.

C548 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 dumy 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 equals 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 or in a reference to the intrinsic function LBOUND.

The bounds of the first *n* - 1 dimensions are those specified by the *explicit-shape-spec-list*, if present, in the *assumed-size-spec*. The lower bound of the last dimension is lower-bound, if present, and 1 otherwise. 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 any redefinition or undefinition of the specification expression variables during execution of the procedure.

5.1.2.6 EXTERNAL attribute

The **EXTERNAL attribute** specifies that a name is an external procedure, a dummy procedure, or a procedure pointer. This attribute may be explicitly specified by a type declaration statement, an EXTERNAL statement (12.3.2.2), a procedure declaration statement (12.3.2.3), or by the appearance of the name as a specific procedure name in an interface body (12.3.2.1).

Any name that is used in a scoping unit as the *function-name* of a *function-reference* or as the *subroutine-name* of a *call-stmt* implicitly acquires the EXTERNAL attribute in that scoping unit if it is not the name of an accessible statement function or internal procedure, is not accessed by host or USE association, and is not explicitly given the EXTERNAL attribute.

If a name that has the EXTERNAL attribute also has an explicitly specified type or appears as a function name in a function reference (12.4) or interface body (12.3.2.1) then it is the name of a function.

If a name that is not the name of a block data program unit has the EXTERNAL attribute then it may be used as an actual argument, as a procedure name in a procedure reference (12.4), or as the target of a procedure pointer assignment (7.5.2).

A dummy argument that has the EXTERNAL attribute is a dummy procedure or a dummy procedure pointer. A name that has the EXTERNAL attribute and is not a dummy argument is the name of an external procedure, a procedure pointer, or a block data program unit.

NOTE 5.12

The EXTERNAL attribute for a block data program unit can only be specified by an EXTERNAL statement (12.3.2.2); it is not possible to do so in a type declaration statement.

5.1.2.7 INTENT attribute

The **INTENT attribute** specifies the intended use of a dummy argument.

R522 *intent-spec* **is** IN
or OUT
or INOUT

C549 The INTENT attribute shall not be specified for a dummy argument that is a dummy procedure.

NOTE 5.13

A dummy procedure pointer is not a dummy procedure. Therefore, INTENT may be specified for a dummy procedure pointer.

C550 A nonpointer object with the INTENT (IN) attribute shall not appear in a variable definition context (16.8.7).

C551 A pointer object with the INTENT (IN) attribute shall not appear as

(1) A *pointer-object* in a *pointer-assignment-stmt* or *nullify-stmt*,

(2) An *allocate-object* in an *allocate-stmt* or *deallocate-stmt*, or

(3) An actual argument in a reference to a procedure if the associated dummy argument is a pointer with the INTENT (OUT) or INTENT (INOUT) attribute.

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 (16.7.2.1.3).

The INTENT (OUT) attribute for a nonpointer dummy argument specifies that it shall be defined before a reference to the dummy argument is made within the procedure and any actual argument that becomes associated with such a dummy argument shall be definable. On invocation of the procedure, such a dummy argument becomes undefined except for components of an object of derived type for which default initialization has been specified. The INTENT (OUT) attribute for a pointer dummy argument specifies that on invocation of the procedure the dummy argument becomes disassociated. Any actual argument associated with such a pointer dummy shall be a pointer variable.

The INTENT (INOUT) attribute for a nonpointer dummy argument specifies that it is intended for use both to receive data from and to return data to the invoking scoping unit. Such a dummy argument may be referenced or defined. Any actual argument that becomes associated with such a dummy argument shall be definable. The INTENT (INOUT) attribute for a pointer dummy argument specifies that it is intended for use both to receive a pointer association from and to return a pointer association to the invoking scoping unit. Any actual argument associated with such a pointer dummy shall be a pointer variable.

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 5.14

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 5.15

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 prohibitted, 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 5.16

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 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 argument corresponding to an INTENT (INOUT) dummy argument always shall 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.

5.1.2.8 INTRINSIC attribute

The **INTRINSIC attribute** confirms that a name is the specific name (13.10) or generic name (13.8, 13.9) of an intrinsic procedure. The INTRINSIC attribute allows the specific name of an intrinsic procedure that is listed in 13.10 and not marked with a bullet (•) to be used as an actual argument (12.4).

Declaring explicitly that a generic intrinsic procedure name has the INTRINSIC attribute does not cause that name to lose its generic property.

If the specific name of an intrinsic procedure (13.10) is used as an actual argument, the name shall be explicitly specified to have the INTRINSIC attribute.

The following constraint applies to syntax rules R504, R1201, and R1201:

C552 If the name of a generic intrinsic procedure is explicitly declared to have the INTRINSIC attribute, and it is also the generic name in 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 16.1.2.3.

5.1.2.9 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 may be used to determine whether an actual argument has been associated with a dummy argument having the OPTIONAL attribute.

5.1.2.10 PARAMETER attribute

The **PARAMETER attribute** specifies entities that are named constants. The *object-name* has the value specified by the *initialization-expr* that appears on the right of the equals; if necessary, the value is converted according to the rules of intrinsic assignment (7.5.1.4) to a value that agrees in type, type parameters, and shape with the *object-name*.

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 5.17

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

5.1.2.11 POINTER attribute

An object with the **POINTER attribute** shall neither be referenced nor defined unless it is pointer associated with a target object that may be referenced or defined. If the pointer is an array, it shall be declared with a *deferred-shape-spec-list*.

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 5.18

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.

5.1.2.12 SAVE attribute

An object with the **SAVE attribute**, in the scoping unit of a subprogram, retains its association status, allocation status, definition status, and value after execution of a RETURN or END statement unless the object is a pointer and its target becomes undefined (16.7.2.1.3(3)). The object is shared by all instances (12.5.2.3) of the subprogram. Such an object is called a **saved object**. An object that does not have the SAVE attribute is called an **unsaved** object.

An object with the SAVE attribute, declared in the scoping unit of a module, retains its association status, allocation status, definition status, and value after a RETURN or END statement is executed in a procedure that accesses the module unless the object is a pointer and its target becomes undefined.

A procedure pointer with the SAVE attribute retains its association status when execution of an instance of any subprogram completes.

The SAVE attribute may appear in declarations in a main program and has no effect.

5.1.2.13 TARGET attribute

An object with the **TARGET attribute** may have a pointer associated with it (7.5.2). An object without the TARGET attribute shall not have an accessible pointer associated with it.

NOTE 5.19

In addition to variables explicitly declared to have the TARGET attribute, the objects created by allocation of pointers (6.3.1.3) have the TARGET attribute.

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

NOTE 5.20

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 5.21

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.

5.1.2.14 VALUE attribute

The VALUE attribute specifies a type of argument association (12.4.1) for a dummy argument.

NOTE 5.22

The name of the VALUE attribute is intended to be suggestive. Although a processor is not required to use pass-by-value for an argument with the VALUE attribute, that might be a possible implementation. In particular, if the VALUE attribute is specified for a dummy argument, the processor shall use the same argument passing convention as the companion processor, which is often pass-by-value (15.2.6).

5.1.2.15 VOLATILE attribute

An object shall have the VOLATILE attribute if there is a reference to or definition of the object, or the object becomes undefined, by means not specified in this standard.

An object may have the VOLATILE attribute in a specific scoping unit without necessarily having it in other scoping units. If an object has the VOLATILE attribute then all of its subobjects also have the VOLATILE attribute.

NOTE 5.23

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 the interactions with the non-Fortran processes.

If the POINTER and VOLATILE attributes are both specified, then the volatility shall apply to the target of the pointer and to the pointer association.

NOTE 5.24

If the value of the target of a pointer can change by means outside of Fortran, while a pointer is associated with a 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 VOLATILE attribute if one member has the VOLATILE attribute.

The interpretation of a program containing objects with the volatile attribute is processor dependent.

5.2 Attribute specification statements

All attributes (other than type) may be specified for entities, independently of type, by separate attribute specification statements. The combination of attributes that may be specified for a particular entity is subject to the same restrictions as for type declaration statements regardless of

the method of specification. This also applies to PROCEDURE, EXTERNAL and INTRINSIC statements.

5.2.1 Accessibility statements

R523 *access-stmt* **is** *access-spec* [[::] *access-id-list*]

R524 *access-id* **is** *use-name*
or *generic-spec*

C553 (R523) 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.

C554 (R524) 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 (5.1.2.1), 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 5.25

Examples of accessibility statements are:

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

5.2.2 ALLOCATABLE statement

R525 *allocatable-stmt* **is** ALLOCATABLE [::] ■
■ *object-name* [(*deferred-shape-spec-list*)] ■
■ [, *object-name* [(*deferred-shape-spec-list*)]] ...

This statement specifies the ALLOCATABLE attribute (5.1.2.2) for a list of objects.

NOTE 5.26

An example of an ALLOCATABLE statement is:

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

5.2.3 ASYNCHRONOUS statement

R526 *asynchronous-stmt* **is** ASYNCHRONOUS [::] *object-name-list*

The ASYNCHRONOUS statement specifies the ASYNCHRONOUS attribute (5.1.2.3) for a list of objects.

5.2.4 BIND statement

R527 *bind-stmt* **is** *language-binding-spec* [::] *bind-entity-list*

R528 *bind-entity* **is** *entity-name*
or / *common-block-name* /

C555 (R527) If any *bind-entity* in a *bind-stmt* is an *entity-name*, the *bind-stmt* shall appear in the specification part of a module.

C556 (R527) If the *language-binding-spec* has a *bind-spec-list*, the *bind-entity-list* shall consist of a single *bind-entity*.

The BIND statement specifies the BIND attribute (5.1.2.4) for a list of entities.

5.2.5 DATA statement

R529 *data-stmt* **is** DATA *data-stmt-set* [[,] *data-stmt-set*] ...

This statement is used to specify explicit initialization (5.1).

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 a SAVE statement or a type declaration statement containing the SAVE attribute.

R530 *data-stmt-set* **is** *data-stmt-object-list* / *data-stmt-value-list* /

R531 *data-stmt-object* **is** *variable*
or *data-implied-do*

R532 *data-implied-do* **is** (*data-i-do-object-list* , *data-i-do-variable* = ■
■ *scalar-int-expr* , *scalar-int-expr* [, *scalar-int-expr*])

R533 *data-i-do-object* **is** *array-element*
or *scalar-structure-component*
or *data-implied-do*

R534 *data-i-do-variable* **is** *scalar-int-variable*

C557 (R531) 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.

C558 (R531) A variable whose designator is included in a *data-stmt-object-list* or a *data-i-do-object-list* 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.

C559 (R531) 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*.

C560 (R534) *data-i-do-variable* shall be a named variable.

C561 (R532) 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.

C562 (R533) The *array-element* shall be a variable.

C563 (R533) The *scalar-structure-component* shall be a variable.

C564 (R533) The *scalar-structure-component* shall contain at least one *part-ref* that contains a *subscript-list*.

C565 (R533) 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 the containing *data-implied-do*s, and each operation shall be intrinsic.

R535 *data-stmt-value* **is** [*data-stmt-repeat* *] *data-stmt-constant*

R536 *data-stmt-repeat* **is** *scalar-int-constant*
or *scalar-int-constant-subobject*

C566 (R536) 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.

R537 *data-stmt-constant* **is** *scalar-constant*
or *scalar-constant-subobject*
or *signed-int-literal-constant*
or *signed-real-literal-constant*
or NULL ( )
or *structure-constructor*

C567 (R537) 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.

C568 (R537) If a *data-stmt-constant* is a *structure-constructor*, it shall be an initialization expression.

R538 *int-constant-subobject* **is** *constant-subobject*

C569 (R538) *int-constant-subobject* shall be of type integer.

R539 *constant-subobject* **is** *designator*

C570 (R539) *constant-subobject* shall be a subobject of a constant.

C571 (R535) 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 in a *data-stmt-object-list* 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 implied-DO variable, as in the DO construct (8.1.5.4).

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 an implied-DO list 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 list. 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 ( ) for the corresponding variable. The lengths of the two expanded sequences shall be the same.

If a *data-stmt-constant* shall be NULL ( ) 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 ( ) shall be compatible with its corresponding variable according to the rules of intrinsic assignment (7.5.1.4). 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 (7.5.1.4) to a value that agrees in type, type parameters, and shape with the variable.

NOTE 5.27

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 4.21. The pointer HEAD_OF_LIST is declared using the derived type NODE from Note 4.31; it is initially disassociated. MYNAME is initialized by a structure constructor. YOURNAME is initialized by supplying a separate value for each component.

5.2.6 DIMENSION statement

R540 *dimension-stmt* **is** DIMENSION [::] *array-name* (*array-spec*) ■
■ [, *array-name* (*array-spec*)] ...

This statement specifies the DIMENSION attribute (5.1.2.5) and the array properties for each object named.

NOTE 5.28

An example of a DIMENSION statement is:

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

5.2.7 INTENT statement

R541 *intent-stmt* **is** INTENT (*intent-spec*) [::] *dummy-arg-name-list*

This statement specifies the INTENT attribute (5.1.2.7) for the dummy arguments in the list.

NOTE 5.29

An example of an INTENT statement is:

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

5.2.8 OPTIONAL statement

R542 *optional-stmt* **is** OPTIONAL [::] *dummy-arg-name-list*

This statement specifies the OPTIONAL attribute (5.1.2.9) for the dummy arguments in the list.

NOTE 5.30

An example of an OPTIONAL statement is:

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

5.2.9 PARAMETER statement

The **PARAMETER statement** specifies the PARAMETER attribute (5.1.2.10) and the values for the named constants in the list.

R543 *parameter-stmt* **is** PARAMETER (*named-constant-def-list*)

R544 *named-constant-def* **is** *named-constant = initialization-expr*

The named constant shall have its type, type parameters, and shape specified in a prior specification of the *specification-part* or declared implicitly (5.3). If the named constant is typed by the implicit typing rules, its appearance in any subsequent specification of the *specification-part* shall confirm this implied type and the values of any implied type parameters.

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 (7.5.1.4) to a value that agrees in type, type parameters, and shape with the named constant.

NOTE 5.31

An example of a PARAMETER statement is:

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

5.2.10 POINTER statement

R545 *pointer-stmt* **is** POINTER [::] *pointer-decl-list*

R546 *pointer-decl* **is** *object-name* [(*deferred-shape-spec-list*)]
or *proc-entity-name*

C572 (R546) A *proc-entity-name* shall also be declared in a *procedure-declaration-stmt*.

This statement specifies the POINTER attribute (5.1.2.11) for a list of objects and procedure entities.

NOTE 5.32

An example of a POINTER statement is:

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

5.2.11 SAVE statement

R547 *save-stmt* **is** SAVE [[::] *saved-entity-list*]

R548 *saved-entity* **is** *object-name*
or *proc-pointer-name*
or / *common-block-name* /

R549 *proc-pointer-name* **is** *name*

C573 (R549) A *proc-pointer-name* shall be the name of a procedure pointer.

C574 (R547) If a SAVE statement with an omitted saved entity list occurs in a scoping unit, no other explicit occurrence of the SAVE attribute or SAVE statement is permitted in the same scoping unit.

A SAVE statement with a saved entity list specifies the SAVE attribute (5.1.2.12) for all entities named in the list or included within a common block named in the list. 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.

If a particular common block name is specified in a SAVE statement in any scoping unit of a program other than the main program, it shall be specified in a SAVE statement in every scoping unit in which that common block appears except in the scoping unit of the main program. For a common block declared in a SAVE statement, the values in the common block storage sequence (5.5.2.1) at the time a RETURN or END statement is executed are made available to the next scoping unit in the execution sequence of the program that specifies the common block name or accesses the common block. If a named common block is specified in the scoping unit of the main program, the current values of the common block storage sequence are made available to each scoping unit that specifies the named common block. The definition status of each object in the named common block storage sequence depends on the association that has been established for the common block storage sequence.

A SAVE statement may appear in the specification part of a main program and has no effect.

NOTE 5.33

An example of a SAVE statement is:

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

5.2.12 TARGET statement

R550 *target-stmt* **is** TARGET [::] *object-name* [(*array-spec*)] ■
■ [, *object-name* [(*array-spec*)]] ...

This statement specifies the TARGET attribute (5.1.2.13) for a list of objects.

NOTE 5.34

An example of a TARGET statement is:

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

5.2.13 VALUE statement

R551 *value-stmt* **is** VALUE [::] *dummy-arg-name-list*

The VALUE statement specifies the VALUE attribute (5.1.2.14) for a list of dummy arguments.

5.2.14 VOLATILE statement

R552 *volatile-stmt* **is** VOLATILE [::] *object-name-list*

The VOLATILE statement specifies the VOLATILE attribute (5.1.2.15) for a list of objects.

5.3 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.

R553 *implicit-stmt* **is** IMPLICIT *implicit-spec-list*
or IMPLICIT NONE

R554 *implicit-spec* **is** *declaration-type-spec* (*letter-spec-list*)

R555 *letter-spec* **is** *letter* [– *letter*]

C575 (R553)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.

C576 (R555) If the minus and second letter appear, the second letter shall follow the first letter alphabetically.

A *letter-spec* consisting of two letters 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 to 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 5.35

The following are examples of the use of IMPLICIT statements:

```
MODULE EXAMPLE_MODULE
   IMPLICIT NONE
   ...
INTERFACE
      FUNCTION FUN (I)   ! Not all data entities need
         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 5.36

An IMPLICIT statement may specify a *declaration-type-spec* of derived type.

For example, given an IMPLICIT statement and a type defined as follows:

```
IMPLICIT TYPE (POSN) (A-B, W-Z), INTEGER (C-V)
TYPE POSN
   REAL X, Y
   INTEGER Z
END TYPE POSN
```

variables beginning with the letters A, B, W, X, Y, and Z are implicitly typed with the type POSN and the remaining variables are implicitly typed with type INTEGER.

NOTE 5.37

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)
  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.

5.4 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).

R556 *namelist-stmt* **is** NAMELIST ■
■ / *namelist-group-name* / *namelist-group-object-list* ■
■ [[,] / *namelist-group-name* / *namelist-group-object-list*] ...

C577 (R556) The *namelist-group-name* shall not be a name made accessible by use association.

R557 *namelist-group-object* **is** *variable-name*

C578 (R557) A *namelist-group-object* shall not be an assumed-size array.

C579 (R556) A *namelist-group-object* shall not have the PRIVATE attribute if the *namelist-group-name* has the PUBLIC attribute.

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

Any *namelist-group-name* may occur in more than one NAMELIST statement 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 5.38

An example of a NAMELIST statement is:

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

5.5 Storage association of data objects

In general, the physical storage units or storage order for data objects is not specifiable. However, the EQUIVALENCE, COMMON, and SEQUENCE statements and the BIND(C) *type-attr-spec* provide for control of the order and layout of storage units. The general mechanism of storage association is described in 16.7.3.

5.5.1 EQUIVALENCE statement

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 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.

R558 *equivalence-stmt* **is** EQUIVALENCE *equivalence-set-list*

R559 *equivalence-set* **is** (*equivalence-object* , *equivalence-object-list*)

R560 *equivalence-object* **is** *variable-name*
or *array-element*
or *substring*

C580 (R560) An *equivalence-object* shall not be a designator with a base object that is a dummy argument, a pointer, an allocatable variable, an object of a derived type 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.

C581 (R560) An *equivalence-object* shall not be a designator that has more than one *part-ref*.

C582 (R560) An *equivalence-object* shall not have the TARGET attribute.

C583 (R560) Each subscript or substring range expression in an *equivalence-object* shall be an integer initialization expression (7.1.7).

C584 (R559) 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.

C585 (R559) 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.

C586 (R559) If an *equivalence-object* is of a 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.

C587 (R559) 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.

C588 (R560) The name of an *equivalence-object* shall not be a name made accessible by use association.

C589 (R560) A *substring* shall not have length zero.

NOTE 5.39

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 may be equivalenced 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 may be equivalenced 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 may be equivalenced only to objects of the same type and kind type parameters.

Further rules on the interaction of EQUIVALENCE statements and default initialization are given in 16.7.3.3.

5.5.1.1 Equivalence association

An EQUIVALENCE statement specifies that the storage sequences (16.7.3.1) 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.

5.5.1.2 Equivalence of default character objects

A data object of type default character may be equivalenced only with other objects of type default character. The lengths of the equivalenced 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 5.40

For example, using the declarations:

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

5.5.1.3 Array names and array element designators

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

5.5.1.4 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 5.41

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 5.42

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
```

5.5.2 COMMON statement

The **COMMON statement** specifies blocks of physical storage, called **common blocks**, that may be accessed by any of the scoping units in a program. Thus, the COMMON statement provides a global data facility based on storage association (16.7.3).

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**.

R561 *common-stmt* **is** COMMON ■
■ [/ [*common-block-name*] /] *common-block-object-list* ■
■ [[,] / [*common-block-name*] / *common-block-object-list*] ...

R562 *common-block-object* **is** *variable-name* [(*explicit-shape-spec-list*)]
or *proc-pointer-name*

C590 (R562) Only one appearance of a given *variable-name* is permitted in all *common-block-object-list*s within a scoping unit.

C591 (R562) 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.

C592 (R562) If a *common-block-object* is of a derived type, it shall be a sequence type (4.5.1) with no default initialization.

C593 (R562) 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* (*explicit-shape-spec-list*) declares *variable-name* to have the DIMENSION attribute and specifies the array properties that apply. If derived-type objects of numeric sequence type (4.5.1) or character sequence type (4.5.1) appear in common, it is as if the individual components were enumerated directly in the common list.

NOTE 5.43

Examples of COMMON statements are:

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

5.5.2.1 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 (16.7.3.1) 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 may 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 unit.

5.5.2.2 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.

5.5.2.3 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 become associated only with nonpointer objects of these types.

A nonpointer object of type default character or character sequence type shall become 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 become 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 become associated only with nonpointer objects of the same type and type parameters.

A data pointer shall become 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 nondeferred type parameters shall have the same value. A procedure pointer shall become 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 may become storage associated only with another object that has the TARGET attribute and the same type and type parameters.

NOTE 5.44

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 (5.5.1).

5.5.2.4 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 name has been declared in a SAVE statement, but never causes data objects in blank common to become undefined (16.8.6).

(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 (11.4), but objects in blank common shall not be initially defined.

5.5.2.5 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 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 5.45

For example, the following is not permitted:

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

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