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. A named data object shall not be explicitly specified to have a particular attribute more than once in a scoping unit.

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**	*type-spec* [[, *attr-spec*] ... ::] *entity-decl-list*
R502	*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 (*type-name*)
R503	*attr-spec*	**is**	PARAMETER
		or	*access-spec*
		or	ALLOCATABLE
		or	DIMENSION (*array-spec*)
		or	EXTERNAL
		or	INTENT (*intent-spec*)
		or	INTRINSIC
		or	OPTIONAL
		or	POINTER
		or	SAVE
		or	TARGET
R504	*entity-decl*	**is**	*object-name* [(*array-spec*)] [* *char-length*] [*initialization*]
		or	*function-name* [* *char-length*]
R505	*initialization*	**is**	= *initialization-expr*
		or	=> NULL ( )
R506	*kind-selector*	**is**	([KIND =] *scalar-int-initialization-expr*)

Constraint: The same *attr-spec* shall not appear more than once in a given *type-declaration-stmt*.

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

Constraint: The ALLOCATABLE attribute may be used only when declaring an array that is not a dummy argument or a function result.

Constraint: An array declared with a POINTER or an ALLOCATABLE attribute shall be specified with an *array-spec* that is a *deferred-shape-spec-list* (5.1.2.4.3).

Constraint: An *array-spec* for an *object-name* that is a function result that does not have the POINTER attribute shall be an *explicit-shape-spec-list*.

Constraint: If the POINTER attribute is specified, the TARGET, INTENT, EXTERNAL, or INTRINSIC attribute shall not be specified.

Constraint: If the TARGET attribute is specified, the POINTER, EXTERNAL, INTRINSIC, or PARAMETER attribute shall not be specified.

Constraint: The PARAMETER attribute shall not be specified for dummy arguments, pointers, allocatable arrays, functions, or objects in a common block.

Constraint: The INTENT and OPTIONAL attributes may be specified only for dummy arguments.

Constraint: An entity shall not have the PUBLIC attribute if its type has the PRIVATE attribute.

Constraint: 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.

Constraint: An entity shall not have the EXTERNAL attribute if it has the INTRINSIC attribute.

Constraint: An entity in an *entity-decl-list* shall not have the EXTERNAL or INTRINSIC attribute specified unless it is a function.

Constraint: An array shall not have both the ALLOCATABLE attribute and the POINTER attribute.

Constraint: The * *char-length* option is permitted only if the type specified is character.

Constraint: The *function-name* shall be the name of an external function, an intrinsic function, a function dummy procedure, or a statement function.

Constraint: The *initialization* shall appear if the statement contains a PARAMETER attribute (5.1.2.1).

Constraint: If *initialization* appears, a double colon separator shall appear before the *entity-decl-list*.

Constraint: *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 array, an external name, an intrinsic name, or an automatic object.

Constraint: If => appears in *initialization*, the object shall have the POINTER attribute. If = appears in *initialization*, the object shall not have the POINTER attribute.

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

NOTE 5.2

The double colon separator in a *type-declaration-stmt* is required only if an *attr-spec* or *initialization* is specified; otherwise, the separator is optional.

A name that identifies a specific intrinsic function in a scoping unit has a type as specified in 13.13. 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.

The *specification-expr* (7.1.6.2) of a *char-len-param-value* (5.1.1.5) or an *array-spec* (5.1.2.4) may be a nonconstant expression provided the specification expression is in an interface body (12.3.2.1), the specification part of a subprogram, or the *type-spec* of a FUNCTION statement (12.5.2.2). If a *specification-expr* involves a reference to a specification function (7.1.6.2), the expression is considered to be a nonconstant expression. If the data object being declared depends on the value of such a nonconstant expression and 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 a nonconstant 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* becomes defined with the value determined from *initialization-expr* in accordance with the rules of intrinsic assignment (7.5.1.4). 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, or by inclusion of the *object-name* in a SAVE statement (5.2.4).

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 ( )
```

NOTE 5.5

Type declaration statements in FORTRAN 77 required different attributes of an entity to be specified in different statements (INTEGER, SAVE, DATA, etc.). This standard allows the attributes of an entity to be specified in a single extended form of the type statement. For example,

```
INTEGER, DIMENSION (10, 10), SAVE :: A, B, C
REAL, PARAMETER :: PI = 3.14159265, E = 2.718281828
```

To retain compatibility and consistency with FORTRAN 77, most of the attributes that may be specified in the extended type statement may alternatively be specified in separate statements.

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 would 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 (4.3.1.1). 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 (4.3.1.2). 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 (4.3.1.2). 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 (4.3.1.3). 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 (4.3.2.1).

R507	*char-selector*	**is**	*length-selector*
		or	(LEN = *char-len-param-value* , ■
			■ KIND = *scalar-int-initialization-expr*)
		or	(*char-len-param-value* , ■
			■ [KIND =] *scalar-int-initialization-expr*)
		or	(KIND = *scalar-int-initialization-expr* ■
			■ [, LEN = *char-len-param-value*])
R508	*length-selector*	**is**	([LEN =] *char-len-param-value*)
		or	* *char-length* [,]
R509	*char-length*	**is**	(*char-len-param-value*)
		or	*scalar-int-literal-constant*
R510	*char-len-param-value*	**is**	*specification-expr*
		or	*

Constraint: The value of *scalar-int-initialization-expr* shall be nonnegative and shall specify a representation method that exists on the processor.

Constraint: The *scalar-int-literal-constant* shall not include a *kind-param*.

Constraint: A function name shall not be declared with an asterisk *char-len-param-value* unless it is the name of the result of an external function or the name of a dummy function.

Constraint: A function name declared with an asterisk *char-len-param-value* shall not be array-valued, pointer-valued, recursive, or pure.

Constraint: The optional comma in a *length-selector* is permitted only in a *type-spec* in a *type-declaration-stmt*.

Constraint: The optional comma in a *length-selector* is permitted only if no double colon separator appears in the *type-declaration-stmt*.

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 *char-len-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.

An **assumed character length parameter** is a type parameter of a dummy argument that is specified with an asterisk *char-len-param-value*.

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 * may be used only in the following ways:

(1) It may be used to declare a dummy argument of a procedure, in which case the dummy argument assumes the length of the associated actual argument when the procedure is invoked.

(2) It may be used to declare a named constant, in which case the length is that of the constant value.

(3) In an external function, the name of the function result may be specified with a character length parameter value of *; in this case, 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.

NOTE 5.6

An interface body may be specified for a dummy or external function whose result has a character length parameter of * only if the function is not invoked. This is because this characteristic has to be specified to be the same in the interface body as in the procedure definition, but in order to invoke such a procedure, the calling routine is required to specify a length other than *.

The length specified for a character-valued statement function or statement function dummy argument of type character shall be a constant specification expression.

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 5.7

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 (4.3.2.2).

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*. 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* (4.4.1). 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**. A scalar entity of numeric sequence type (4.4.1) is a **numeric sequence structure**. A scalar entity of character sequence type (4.4.1) is a **character sequence structure**.

A declaration for a nonsequence derived-type dummy argument shall specify a derived type that is accessed by use association or host association because the same definition shall be used to declare both the actual and dummy arguments to ensure that both are of the same derived type. This restriction does not apply to arguments of sequence type (4.4.2).

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 PARAMETER attribute

The **PARAMETER attribute** specifies entities that are named constants. The *object-name* becomes defined with the value determined from the *initialization-expr* that appears on the right of the equals, in accordance with the rules of intrinsic assignment (7.5.1.4). The appearance of a PARAMETER attribute in a specification requires that the = *initialization-expr* option appear for all objects in the *entity-decl-list*.

Any named constant that appears in the initialization expression shall have been defined previously in the same type declaration statement, defined in a prior PARAMETER statement or type declaration statement using the PARAMETER attribute, or made accessible by use association or host association. A named constant shall not be referenced in any other context unless it has been defined in a prior PARAMETER statement or type declaration statement using the PARAMETER attribute, or made accessible by use association or host association.

A named constant shall not appear within a format specification (10.1.1).

NOTE 5.8

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 ( ))
```

5.1.2.2 Accessibility attribute

The **accessibility attribute** specifies the accessibility of entities.

R511	*access-spec*	**is**	PUBLIC
		or	PRIVATE

Constraint: An *access-spec* shall appear only in the *specification-part* of a module.

Entities that are declared with a PRIVATE attribute are not accessible outside the module. Entities that are declared with a PUBLIC attribute may be made accessible in other program units by the

USE statement. Entities without an explicitly specified *access-spec* have default accessibility, which is PUBLIC unless the default has been changed by a PRIVATE statement (5.2.3).

NOTE 5.9

An example of an accessibility specification is:

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

5.1.2.3 INTENT attribute

An **INTENT attribute** specifies the intended use of the dummy argument.

R512	*intent-spec*	**is**	IN
		or	OUT
		or	INOUT

Constraint: The INTENT attribute shall not be specified for a dummy argument that is a dummy procedure or a dummy pointer.

Constraint: A dummy argument with the INTENT(IN) attribute, or a subobject of such a dummy argument, shall not appear as

(1) The *variable* of an *assignment-stmt*,

(2) The *pointer-object* of a *pointer-assignment-stmt*,

(3) A DO variable or implied-DO variable,

(4) An *input-item* in a *read-stmt*,

(5) A *variable-name* in a *namelist-stmt* if the *name-list-group-name* appears in a NML= specifier in a *read-stmt*,

(6) An *internal-file-unit* in a *write-stmt*,

(7) An IOSTAT= or SIZE= specifier in an input/output statement,

(8) A definable variable in an INQUIRE statement,

(9) A *stat-variable* or *allocate-object* in an *allocate-stmt* or a *deallocate-stmt*, or

(10) An actual argument in a reference to a procedure with an explicit interface when the associated dummy argument has the INTENT(OUT) or INTENT(INOUT) attribute.

The INTENT (IN) attribute specifies that the dummy argument shall neither be defined nor become undefined during the execution of the procedure.

The INTENT (OUT) attribute specifies that the dummy argument 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 (INOUT) attribute specifies that the dummy argument is intended for use both to receive data from and to return data to the invoking scoping unit. Any actual argument that becomes associated with such a dummy argument shall be definable.

If no INTENT attribute is specified for a dummy argument, its use is subject to the limitations of the associated actual argument (12.4.1.1, 12.4.1.2, 12.4.1.3).

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
```

5.1.2.4 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.

R513	*array-spec*	**is**	*explicit-shape-spec-list*
		or	*assumed-shape-spec-list*
		or	*deferred-shape-spec-list*
		or	*assumed-size-spec*

Constraint: 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
```

5.1.2.4.1 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.

R514	*explicit-shape-spec*	**is**	[*lower-bound* :] *upper-bound*
R515	*lower-bound*	**is**	*specification-expr*
R516	*upper-bound*	**is**	*specification-expr*

Constraint: An explicit-shape array whose bounds depend on the values of nonconstant 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 nonconstant specification expressions.

If an explicit-shape array has bounds that are nonconstant specification 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.

5.1.2.4.2 Assumed-shape array

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

R517 *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 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.

5.1.2.4.3 Deferred-shape array

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

An **allocatable array** is a named array that has the ALLOCATABLE attribute and a specified rank, but its bounds, and hence shape, are determined when space is allocated for the array by execution of an ALLOCATE statement (6.3.1).

The ALLOCATABLE attribute may be specified for an array in a type declaration statement or in an ALLOCATABLE statement (5.2.6). An array with the ALLOCATABLE attribute shall be declared with a *deferred-shape-spec-list* in a type declaration statement, an ALLOCATABLE statement, a DIMENSION statement (5.2.5), or a TARGET statement (5.2.8). The type and type parameters may be specified in a type declaration statement.

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 by pointer assignment (7.5.2) or by execution of an ALLOCATE statement (6.3.1). The POINTER attribute may be specified for an array in a type declaration statement, a component definition statement, or a POINTER statement (5.2.7). An array with the POINTER attribute shall be declared with a *deferred-shape-spec-list* in a type declaration statement, a POINTER statement, or a DIMENSION statement (5.2.5). The type and type parameters may be specified in a type declaration statement or a component definition statement.

R518 *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 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 that is inquiring about the allocation status or a property of the type or type parameters. The lower and upper bounds of each dimension are those specified in the ALLOCATE statement when the array is allocated.

The size, bounds, and shape of the target of 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 that is inquiring about argument presence, a property of the type or type parameters, or association status. The bounds of each dimension of an array pointer may be specified in two ways:

(1) They are specified in an ALLOCATE statement (6.3.1) when the target is allocated, or

(2) They are specified in a pointer assignment statement. The lower bound of each dimension is the result of the intrinsic function LBOUND (13.14.53) applied to the corresponding dimension of the target. The upper bound of each dimension is the result of the intrinsic function UBOUND (13.14.113) applied to the corresponding dimension of the target.

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

5.1.2.4.4 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.

R519 *assumed-size-spec* **is** [*explicit-shape-spec-list* ,] [*lower-bound* :] *

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

Constraint: 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.

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 nonconstant specification expressions, the bounds are declared 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.

5.1.2.5 SAVE attribute

An object with the **SAVE attribute**, declared 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 (14.6.2.1.3(3)). The object is shared by all instances (12.5.2.4) of the subprogram. Such an object is called a **saved 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.

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

5.1.2.6 OPTIONAL attribute

The **OPTIONAL attribute** shall be specified only in the scoping unit of a subprogram or an interface block, and shall be specified only for dummy arguments. The OPTIONAL attribute specifies that the dummy argument need not be associated with an actual argument in a reference to the procedure (12.4.1.5). The PRESENT intrinsic function may be used to determine whether an actual argument has been associated with a dummy argument having the OPTIONAL attribute.

5.1.2.7 POINTER attribute

An object with the **POINTER attribute** shall neither be referenced nor defined unless, as a result of executing a pointer assignment (7.5.2) or an ALLOCATE statement (6.3.1), it becomes 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*.

NOTE 5.12

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.

5.1.2.8 TARGET attribute

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

NOTE 5.13

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.

5.1.2.9 ALLOCATABLE attribute

The **ALLOCATABLE attribute** may be specified for an array. Such an array shall be a deferred-shape array; the shape is determined when space is allocated for the array by execution of an ALLOCATE statement (6.3.1).

5.1.2.10 EXTERNAL attribute

The **EXTERNAL attribute** specifies an external function or a dummy function and permits the function name to be used as an actual argument. This attribute may also be declared via the EXTERNAL statement (12.3.2.2).

5.1.2.11 INTRINSIC attribute

The **INTRINSIC attribute** specifies the specific or generic name of an intrinsic function and permits the name to be used as an actual argument if it is a specific name of an intrinsic function (13.13). This attribute may also be declared via the INTRINSIC statement (12.3.2.3).

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 EXTERNAL and INTRINSIC statements.

5.2.1 INTENT statement

R520 *intent-stmt* **is** INTENT (*intent-spec*) [::] *dummy-arg-name-list*

Constraint: An *intent-stmt* shall appear only in the *specification-part* of a subprogram or an interface body (12.3.2.1).

Constraint: *dummy-arg-name* shall not be the name of a dummy procedure or a dummy pointer.

This statement specifies the INTENT attribute (5.1.2.3) for the dummy arguments in the list.

NOTE 5.14

An example of an INTENT statement is:

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

5.2.2 OPTIONAL statement

R521 *optional-stmt* **is** OPTIONAL [::] *dummy-arg-name-list*

Constraint: An *optional-stmt* shall occur only in the *specification-part* of a subprogram or an interface body (12.3.2.1).

This statement specifies the OPTIONAL attribute (5.1.2.6) for the dummy arguments in the list.

NOTE 5.15

An example of an OPTIONAL statement is:

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

5.2.3 Accessibility statements

R522 *access-stmt* **is** *access-spec* [[::] *access-id-list*]

R523 *access-id* **is** *use-name*
or *generic-spec*

Constraint: 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.

Constraint: Each *use-name* shall be the name of a named variable, procedure, derived type, named constant, or namelist group.

Constraint: A module procedure that has a dummy argument or function result of a type that has PRIVATE accessibility shall have PRIVATE accessibility and shall not have a generic identifier that has PUBLIC accessibility.

An *access-stmt* with an *access-id-list* specifies the accessibility attribute (5.1.2.2), PUBLIC or PRIVATE, of the entities in the list. An *access-stmt* without an *access-id* list specifies the default accessibility that applies to all potentially accessible entities 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 5.16

Examples of accessibility statements are:

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

5.2.4 SAVE statement

R524	*save-stmt*	**is**	SAVE [[::] *saved-entity-list*]
R525	*saved-entity*	**is**	*object-name*
		or	/ *common-block-name* /

Constraint: An *object-name* shall not be the name of an object in a common block, a dummy argument name, a procedure name, a function result name, an automatic data object name, or the name of an object with the PARAMETER attribute.

Constraint: 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 (5.1.2.5) for all objects 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 5.17

An example of a SAVE statement is:

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

5.2.5 DIMENSION statement

R526	*dimension-stmt*	**is**	DIMENSION [::] *array-name* (*array-spec*) ■
			■ [, *array-name* (*array-spec*)] ...

This statement specifies the DIMENSION attribute (5.1.2.4) and the array properties for each object named.

NOTE 5.18

An example of a DIMENSION statement is:

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

5.2.6 ALLOCATABLE statement

R527 *allocatable-stmt* **is** ALLOCATABLE [::] ■
■ *array-name* [(*deferred-shape-spec-list*)] ■
■ [, *array-name* [(*deferred-shape-spec-list*)]] ...

Constraint: The *array-name* shall not be a dummy argument or function result.

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

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

NOTE 5.19

An example of an ALLOCATABLE statement is:

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

5.2.7 POINTER statement

R528 *pointer-stmt* **is** POINTER [::] *object-name* [(*deferred-shape-spec-list*)] ■
■ [, *object-name* [(*deferred-shape-spec-list*)]] ...

Constraint: The INTENT attribute shall not be specified for an *object-name*.

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

Constraint: The PARAMETER attribute shall not be specified for an *object-name*.

This statement specifies the POINTER attribute (5.1.2.7) for a list of objects.

NOTE 5.20

An example of a POINTER statement is:

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

5.2.8 TARGET statement

R529 *target-stmt* **is** TARGET [::] *object-name* [(*array-spec*)] ■
■ [, *object-name* [(*array-spec*)]] ...

Constraint: The PARAMETER attribute shall not be specified for an *object-name*.

This statement specifies the TARGET attribute (5.1.2.8) for a list of objects.

NOTE 5.21

An example of a TARGET statement is:

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

5.2.9 PARAMETER statement

The **PARAMETER statement** specifies the PARAMETER attribute (5.1.2.1) and the values for the named constants in the list.

R530 *parameter-stmt* **is** PARAMETER (*named-constant-def-list*)

R531 *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.

Each named constant becomes defined with the value determined from the initialization expression that appears on the right of the equals, in accordance with the rules of intrinsic assignment (7.5.1.4).

A named constant that appears in the initialization expression shall have been defined previously in the same PARAMETER statement, defined in a prior PARAMETER statement or type declaration statement using the PARAMETER attribute, or made accessible by use association or host association.

A named constant shall not appear as part of a format specification (10.1.1).

NOTE 5.22

An example of a PARAMETER statement is:

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

5.2.10 DATA statement

R532 *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 or subobject 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.

R533 *data-stmt-set* **is** *data-stmt-object-list* / *data-stmt-value-list* /

R534 *data-stmt-object* **is** *variable*
or *data-implied-do*

R535 *data-implied-do* **is** (*data-i-do-object-list* , *data-i-do-variable* = ■
■ *scalar-int-expr* , *scalar-int-expr* [, *scalar-int-expr*])

R536 *data-i-do-object* **is** *array-element*
or *scalar-structure-component*
or *data-implied-do*

R537 *data-i-do-variable* **is** *scalar-int-variable*

Constraint: 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.

Constraint: A variable whose name or 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 array.

Constraint: A *data-i-do-object* or a *variable* that appears as a *data-stmt-object* shall not be a subobject of a pointer.

Constraint: *data-i-do-variable* shall be a named variable.

Constraint: 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.

Constraint: The *array-element* shall not have a constant parent.

Constraint: The *scalar-structure-component* shall not have a constant parent.

Constraint: The *scalar-structure-component* shall contain at least one *part-ref* that contains a *subscript-list*.

Constraint: 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.

R538 *data-stmt-value* **is** [*data-stmt-repeat* ∗] *data-stmt-constant*

R539 *data-stmt-repeat* **is** *scalar-int-constant*
or *scalar-int-constant-subobject*

R540 *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*

Constraint: 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.

Constraint: In a *scalar-int-constant-subobject* that is a *data-stmt-repeat* any subscript shall be an initialization expression.

Constraint: In a *scalar-constant-subobject* that is a *data-stmt-constant* any subscript, substring starting point, or substring ending point shall be an initialization expression.

Constraint: 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.

Constraint: If a *data-stmt-constant* is a *structure-constructor*, each component 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 (8.1.4.4).

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* is NULL ( ), the corresponding *data-stmt-object* shall have the POINTER attribute.

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

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), and the variable becomes initially defined with the *data-stmt-constant* in accordance with the rules of intrinsic assignment.

If a variable has the POINTER attribute, the corresponding *data-stmt-constant* shall be NULL ( ), and the pointer has an initial association status of disassociated.

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 4.20. The pointer HEAD_OF_LIST is declared using the derived type NODE from Note 4.26; it is initially disassociated. MYNAME is initialized by a structure constructor. YOURNAME is initialized by supplying a separate value for each component.

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.

R541 *implicit-stmt* **is** IMPLICIT *implicit-spec-list*
or IMPLICIT NONE

R542 *implicit-spec* **is** *type-spec* (*letter-spec-list*)

R543 *letter-spec* **is** *letter* [– *letter*]

Constraint: 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.

Constraint: 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 5.24

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
```

NOTE 5.24 *(Continued)*

```
   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 5.25

An IMPLICIT statement may specify a *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 5.26

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 (9.4, 10.9).

R544	*namelist-stmt*	**is**	NAMELIST ■
			■ / *namelist-group-name* / *namelist-group-object-list* ■
			■ [[,] / *namelist-group-name* / *namelist-group-object-list*] ...
R545	*namelist-group-object*	**is**	*variable-name*

Constraint: A *namelist-group-object* shall not be an array dummy argument with a nonconstant bound, a variable with nonconstant character length, an automatic object, a pointer, a variable of a type that has an ultimate component that is a pointer, or an allocatable array.

Constraint: If a *namelist-group-name* has the PUBLIC attribute, no item in the *namelist-group-object-list* shall have the PRIVATE attribute or have private components.

Constraint: The *namelist-group-name* shall not be a name made accessible by use association.

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 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 5.27

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 statement, the COMMON statement, and the SEQUENCE statement provide for control of the order and layout of storage units. The general mechanism of storage association is described in 14.6.3.

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.

R546 *equivalence-stmt* **is** EQUIVALENCE *equivalence-set-list*

R547 *equivalence-set* **is** (*equivalence-object* , *equivalence-object-list*)

R548 *equivalence-object* **is** *variable-name*
or *array-element*
or *substring*

Constraint: An *equivalence-object* shall not be a dummy argument, a pointer, an allocatable array, an object of a nonsequence derived type, an object of a sequence derived type containing a pointer at any level of component selection, an automatic object, a function name, an entry name, a result name, a named constant, a structure component, or a subobject of any of the preceding objects.

Constraint: An *equivalence-object* shall not have the TARGET attribute.

Constraint: Each subscript or substring range expression in an *equivalence-object* shall be an integer initialization expression (7.1.6.1).

Constraint: 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.

Constraint: 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.

Constraint: 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.

Constraint: 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.

Constraint: The name of an *equivalence-object* shall not be a name made accessible by use association.

Constraint: A *substring* shall not have length zero.

NOTE 5.28

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.

Structures that appear in EQUIVALENCE statements shall be sequence structures. 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.

A numeric sequence structure may be equivalenced to another numeric sequence structure, 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 character sequence structure may be equivalenced to an object of default character type or another character sequence structure.

Other objects 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 14.6.3.3.

5.5.1.1 Equivalence association

An EQUIVALENCE statement specifies that the storage sequences (14.6.3.1) 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 5.29

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 5.30

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 5.31

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 (14.6.3).

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**.

R549 *common-stmt* **is** COMMON ■
■ [/ [*common-block-name*] /]*common-block-object-list* ■
■ [[,] / [*common-block-name*] /*common-block-object-list*] ...

R550 *common-block-object* **is** *variable-name* [(*explicit-shape-spec-list*)]

Constraint: Only one appearance of a given *variable-name* is permitted in all *common-block-object-list*s within a scoping unit.

Constraint: A *common-block-object* shall not be a dummy argument, an allocatable array, an automatic object, a function name, an entry name, or a result name.

Constraint: Each bound in the *explicit-shape-spec* shall be a constant specification expression (7.1.6.2).

Constraint: If a *common-block-object* is of a derived type, it shall be a sequence type (4.4.1) with no default initialization.

Constraint: If a *variable-name* appears with an *explicit-shape-spec-list*, it shall not have the POINTER attribute.

Constraint: A *variable-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 ob ject 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 (4.4.1) or character sequence type (4.4.1) appear in common, it is as if the individual components were enumerated directly in the common list.

NOTE 5.32

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 (14.6.3.1) 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. Variables made accessible by use association or host association do not contribute.

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.

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 pointer shall become storage associated only with pointers of the same type, type parameters, and rank.

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 5.33

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 (14.7.6).

(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 5.34

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.