(.AND.), inclusive disjunction (.OR.), logical equivalence (.EQV.), and logical nonequivalence (.NEQV.) as described in 7.2.4. There is also a set of intrinsically defined relational operators that compare the values of data entities of other types and yield a value of type default logical. These operations are described in 7.2.3.

4.5 Derived types

Additional types may be derived from the intrinsic types and other derived types. A type definition is required to define the name of the type and the names and attributes of its **components**.

The type specifier for a derived type uses the keyword TYPE followed by the name of the type in parentheses (R503).

A derived type may be parameterized by multiple type parameters, each of which is defined to be either a kind or nonkind type parameter. There is no concept of a default value for a type parameter of a derived type; it is required to explicitly specify, assume, or defer the values of all type parameters of a derived-type entity.

The **ultimate components** of an object of derived type are the components that are of intrinsic type or have the POINTER or ALLOCATABLE attribute, plus the ultimate components of the components of the object that are of derived type and have neither the ALLOCATABLE nor POINTER attribute.

NOTE 4.17

The ultimate components of objects of the derived type `kids` defined below are `name`, `age`, and `other_kids`.

```
type :: person
  character(len=20) :: name
  integer :: age
end type person

type :: kids
  type(person) :: oldest_child
  type(person), allocatable, dimension(:) :: other_kids
end type kids
```

By default, no storage sequence is implied by the order of the component definitions. However, a storage order is implied for a sequence type (4.5.1.2). If the derived type has the BIND attribute, the storage sequence is that required by the companion processor (2.5.10, 15.2.3).

A derived type may have procedures bound to it. A type-bound procedure is accessed via an object of the type.

4.5.1 Derived-type definition

R423	*derived-type-def*	**is**	*derived-type-stmt*
			[*type-param-def-stmt*] ...
			[*private-or-sequence*] ...
			[*component-part*]
			[*type-bound-procedure-part*]
			end-type-stmt
R424	*derived-type-stmt*	**is**	TYPE [[, *type-attr-spec-list*] ::] *type-name* ■
			■ [(*type-param-name-list*)]
R425	*type-attr-spec*	**is**	*access-spec*

or EXTENSIBLE
or EXTENDS ([*access-spec* ::] *parent-type-name* ■
■ [= *initialization-expr*])
or BIND (C)

C414 (R424) A derived type *type-name* shall not be the same as the name of any intrinsic type defined in this standard.

C415 (R424) The same *type-attr-spec* shall not appear more than once in a given *derived-type-stmt*.

C416 (R424) EXTENSIBLE and EXTENDS shall not both appear.

C417 (R425) A *parent-type-name* shall be the name of an accessible extensible type (4.5.6).

C418 (R423) If EXTENDS or EXTENSIBLE appears, neither BIND(C) nor SEQUENCE shall appear.

R426 *private-or-sequence* **is** *private-components-stmt*
or *sequence-stmt*

C419 (R423) The same *private-or-sequence* shall not appear more than once in a given *derived-type-def*.

R427 *end-type-stmt* **is** END TYPE [*type-name*]

C420 (R427) If END TYPE is followed by a *type-name*, the *type-name* shall be the same as that in the corresponding *derived-type-stmt*.

Derived types with the BIND attribute are subject to additional constraints as specified in 15.2.3.

NOTE 4.18

An example of a derived-type definition is:

```
TYPE PERSON
   INTEGER AGE
   CHARACTER (LEN = 50) NAME
END TYPE PERSON
```

An example of declaring a variable CHAIRMAN of type PERSON is:

```
TYPE (PERSON) :: CHAIRMAN
```

4.5.1.1 Accessibility

Types that are defined in a module or accessibile in that module by use association have either the PUBLIC or PRIVATE attribute. Types for which an *access-spec* is not explicitly specified in that module have the default accessibility attribute for that module. The default accessibility attribute for a module is PUBLIC unless it has been changed by a PRIVATE statement (5.2.1). Only types that have the PUBLIC attribute in that module are available to be accessed from that module by USE association.

The accessibility of a type does not affect, and is not affected by, the accessibility of its components and bindings.

If a type definition is private, then the type name, and thus the structure constructor (4.5.9) for the type, are accessible only within the module containing the definition.

NOTE 4.19

An example of a type with a private name is:

```
TYPE, PRIVATE :: AUXILIARY
   LOGICAL :: DIAGNOSTIC
   CHARACTER (LEN = 20) :: MESSAGE
END TYPE AUXILIARY
```

Such a type would be accessible only within the module in which it is defined.

4.5.1.2 Sequence type

R428 *sequence-stmt* **is** SEQUENCE

C421 (R431) If SEQUENCE appears, all derived types specified in component definitions shall be sequence types.

C422 (R423) If SEQUENCE appears, a *type-bound-procedure-part* shall not appear.

If the **SEQUENCE statement** is present, the type is a **sequence type**. The order of the component definitions in a sequence type specifies a storage sequence for objects of that type. If there are no type parameters and all of the ultimate components of objects of the type are of type default integer, default real, double precision real, default complex, or default logical and are not pointers or allocatable, the type is a **numeric sequence type**. If there are no type parameters and all of the ultimate components of objects of the type are of type default character and are not pointers or allocatable, the type is a **character sequence type**.

NOTE 4.20

An example of a numeric sequence type is:

```
TYPE NUMERIC_SEQ
   SEQUENCE
   INTEGER :: INT_VAL
   REAL    :: REAL_VAL
   LOGICAL :: LOG_VAL
END TYPE NUMERIC_SEQ
```

NOTE 4.21

A structure resolves into a sequence of components. Unless the structure includes a SEQUENCE statement, the use of this terminology in no way implies that these components are stored in this, or any other, order. Nor is there any requirement that contiguous storage be used. The sequence merely refers to the fact that in writing the definitions there will necessarily be an order in which the components appear, and this will define a sequence of components. This order is of limited significance since a component of an object of derived type will always be accessed by a component name except in the following contexts: the sequence of expressions in a derived-type value constructor, intrinsic assignment, the data values in namelist input data, and the inclusion of the structure in an input/output list of a formatted data transfer, where it is expanded to this sequence of components. Provided the processor adheres to the defined order in these cases, it is otherwise free to organize the storage of the components for any nonsequence structure in memory as best suited to the particular architecture.

4.5.1.3 Determination of derived types

Derived-type definitions with the same type name may appear in different scoping units, in which case they may be independent and describe different derived types or they may describe the same type.

Two data entities have the same type if they are declared with reference to the same derived-type definition. The definition may be accessed from a module or from a host scoping unit. Data entities in different scoping units also have the same type if they are declared with reference to different derived-type definitions that specify the same type name, all have the SEQUENCE property or all have the BIND attribute, have no components with PRIVATE accessibility, and have type parameters and components that agree in order, name, and attributes. Otherwise, they are of different derived types. A data entity declared using a type with the SEQUENCE property or with the BIND attribute is not of the same type as an entity of a type declared to be PRIVATE or that has any components that are PRIVATE.

NOTE 4.22

An example of declaring two entities with reference to the same derived-type definition is:

```
TYPE POINT
   REAL  X,  Y
END TYPE POINT
TYPE (POINT) :: X1
CALL SUB (X1)
   ...
CONTAINS
   SUBROUTINE SUB (A)
      TYPE (POINT) :: A
         ...
   END SUBROUTINE SUB
```

The definition of derived type POINT is known in subroutine SUB by host association. Because the declarations of X1 and A both reference the same derived-type definition, X1 and A have the same type. X1 and A also would have the same type if the derived-type definition were in a module and both SUB and its containing program unit accessed the module.

NOTE 4.23

An example of data entities in different scoping units having the same type is:

```
PROGRAM PGM
   TYPE EMPLOYEE
      SEQUENCE
      INTEGER        ID_NUMBER
      CHARACTER (50) NAME
   END TYPE EMPLOYEE
   TYPE (EMPLOYEE) PROGRAMMER
   CALL SUB (PROGRAMMER)
      ...
END PROGRAM PGM
SUBROUTINE SUB (POSITION)
   TYPE EMPLOYEE
      SEQUENCE
      INTEGER        ID_NUMBER
      CHARACTER (50) NAME
```

NOTE 4.23 (cont.)

```
   END TYPE EMPLOYEE
   TYPE (EMPLOYEE) POSITION
   ...
END SUBROUTINE SUB
```

The actual argument PROGRAMMER and the dummy argument POSITION have the same type because they are declared with reference to a derived-type definition with the same name, the SEQUENCE property, and components that agree in order, name, and attributes.

Suppose the component name ID_NUMBER was ID_NUM in the subroutine. Because all the component names are not identical to the component names in derived type EMPLOYEE in the main program, the actual argument PROGRAMMER would not be of the same type as the dummy argument POSITION. Thus, the program would not be standard conforming.

NOTE 4.24

The requirement that the two types have the same name applies to the *type-name*s of the respective *derived-type-stmt*s, not to *type-alias* names or to local names introduced via renaming in USE statements.

4.5.2 Derived-type parameters

R429	*type-param-def-stmt*	**is**	INTEGER [*kind-selector*] , *type-param-attr-spec* :: ■ ■ *type-param-name-list*

C423 (R429) A *type-param-name* in a *type-param-def-stmt* in a *derived-type-def* shall be one of the *type-param-names* in the *derived-type-stmt* of that *derived-type-def*.

C424 (R429) Each *type-param-name* in the *derived-type-stmt* in a *derived-type-def* shall appear as a *type-param-name* in a *type-param-def-stmt* in that *derived-type-def*.

R430	*type-param-attr-spec*	**is**	KIND
		or	NONKIND

The derived type is parameterized if the *derived-type-stmt* has any *type-param-names*.

Each type parameter is itself of type integer.

A type parameter is either a kind type parameter or a nonkind type parameter (4.2). If it is a kind parameter it is said to have the KIND attribute. Its *type-param-attr-spec* explicitly specifies whether a type parameter is kind or nonkind.

A type parameter may be used as a primary in a specification expression (7.1.6) in the *derived-type-def*. A kind type parameter may also be used as a primary in an initialization expression (7.1.7) in the *derived-type-def*.

NOTE 4.25

The following example uses derived-type parameters.

```
     TYPE humongous_matrix(k, d)
       INTEGER, KIND :: k
       INTEGER(selected_int_kind(12)), NONKIND :: d
         !-- Specify a nondefault kind for d.
```

NOTE 4.25 (cont.)

```
      REAL(k) :: element(d,d)
    END TYPE
```

In the following example, `dim` is declared to be a kind parameter, allowing generic overloading of procedures distinguished only by `dim`.

```
    TYPE general_point(dim)
      INTEGER, KIND :: dim
      REAL :: coordinates(dim)
    END TYPE
```

4.5.2.1 Type parameter order

Type parameter order is an ordering of the type parameters of a derived type; it is used for derived-type specifiers.

The type parameter order of a nonextended type is the order of the type parameter list in the derived-type definition. The type parameter order of an extended type consists of the type parameter order of its parent type followed by any additional type parameters in the order of the type parameter list in the derived-type definition.

4.5.3 Components

R431	*component-part*	**is**	[*component-def-stmt*] ...
R432	*component-def-stmt*	**is**	*data-component-def-stmt*
		or	*proc-component-def-stmt*
R433	*data-component-def-stmt*	**is**	*declaration-type-spec* [[, *component-attr-spec-list*] ::] ■
			■ *component-decl-list*
R434	*component-attr-spec*	**is**	POINTER
		or	DIMENSION (*component-array-spec*)
		or	ALLOCATABLE
		or	*access-spec*
R435	*component-decl*	**is**	*component-name* [(*component-array-spec*)] ■
			■ [* *char-length*] [*component-initialization*]
R436	*component-array-spec*	**is**	*explicit-shape-spec-list*
		or	*deferred-shape-spec-list*
R437	*component-initialization*	**is**	= *initialization-expr*
		or	=> *null-init*

C425 (R433) No *component-attr-spec* shall appear more than once in a given *component-def-stmt*.

C426 (R433) A component declared with the CLASS keyword (5.1.1.8) shall have the ALLOCATABLE or POINTER attribute.

C427 (R433) If the POINTER attribute is not specified for a component, the *declaration-type-spec* in the *component-def-stmt* shall specify an intrinsic type or a previously defined derived type.

C428 (R433) If the POINTER attribute is specified for a component, the *declaration-type-spec* in the *component-def-stmt* shall specify an intrinsic type or any accessible derived type including the type being defined.

C429 (R433) If the POINTER or ALLOCATABLE attribute is specified, each *component-array-spec* shall be a *deferred-shape-spec-list*.

C430 (R433) If neither the POINTER attribute nor the ALLOCATABLE attribute is specified, each *component-array-spec* shall be an *explicit-shape-spec-list*.

C431 (R436) Each bound in the *explicit-shape-spec* shall either be an initialization expression or be a specification expression that does not contain references to specification functions or any object designators other than named constants or subobjects thereof.

C432 (R433) A component shall not have both the ALLOCATABLE and the POINTER attribute.

C433 (R435) The * *char-length* option is permitted only if the type specified is character.

C434 (R432) Each *type-param-value* within a *component-def-stmt* shall either be a colon, be an initialization expression, or be a specification expression that contains neither references to specification functions nor any object designators other than named constants or subobjects thereof.

NOTE 4.26

Since a type parameter is not an object, a bound for an *explicit-shape-spec* or a *type-param-value* may contain a *type-param-name*.

C435 (R433) If *component-initialization* appears, a double-colon separator shall appear before the *component-decl-list*.

C436 (R433) If => appears in *component-initialization*, POINTER shall appear in the *component-attr-spec-list*. If = appears in *component-initialization*, POINTER or ALLOCATABLE shall not appear in the *component-attr-spec-list*.

R438 *proc-component-def-stmt* **is** PROCEDURE ([*proc-interface*]) , ■
■ *proc-component-attr-spec-list* :: *proc-decl-list*

NOTE 4.27

See 12.3.2.3 for definitions of *proc-interface* and *proc-decl*.

R439 *proc-component-attr-spec* **is** POINTER
or PASS [(*arg-name*)]
or NOPASS
or *access-spec*

C437 (R438) The same *proc-component-attr-spec* shall not appear more than once in a given *proc-component-def-stmt*.

C438 (R438) POINTER shall appear in each *proc-component-attr-spec-list*.

C439 (R438) If the procedure pointer component has an implicit interface or has no arguments, NOPASS shall be specified.

C440 (R438) If PASS (*arg-name*) appears, the interface shall have a dummy argument named *arg-name*.

C441 (R438) PASS and NOPASS shall not both appear in the same *proc-component-attr-spec-list*.

4.5.3.1 Array components

A data component is an array if its *component-decl* contains a *component-array-spec* or its *data-component-def-stmt* contains the DIMENSION attribute. If the *component-decl* contains a *component-array-spec*, it specifies the array rank, and if the array is explicit shape (5.1.2.5.1), the array bounds; otherwise, the *component-array-spec* in the DIMENSION attribute specifies the array rank, and if the array is explicit shape, the array bounds.

NOTE 4.28

A type definition may have a component that is an array. For example:

```
TYPE LINE
   REAL, DIMENSION (2, 2) :: COORD    !
                                      ! COORD(:,1) has the value of (/X1, Y1/)
                                      ! COORD(:,2) has the value of (/X2, Y2/)
   REAL                   :: WIDTH    ! Line width in centimeters
   INTEGER                :: PATTERN  ! 1 for solid, 2 for dash, 3 for dot
END TYPE LINE
```

An example of declaring a variable LINE_SEGMENT to be of the type LINE is:

```
TYPE (LINE)          :: LINE_SEGMENT
```

The scalar variable LINE_SEGMENT has a component that is an array. In this case, the array is a subobject of a scalar. The double colon in the definition for COORD is required; the double colon in the definition for WIDTH and PATTERN is optional.

NOTE 4.29

A derived type may have a component that is allocatable. For example:

```
TYPE STACK
   INTEGER              :: INDEX
   INTEGER, ALLOCATABLE :: CONTENTS (:)
END TYPE STACK
```

For each scalar variable of type STACK, the shape of the component CONTENTS is determined by execution of an ALLOCATE statement or assignment statement, or by argument association.

NOTE 4.30

Default initialization of an explicit-shape array component may be specified by an initialization expression consisting of an array constructor (4.8), or of a single scalar that becomes the value of each array element.

4.5.3.2 Pointer components

A component is a pointer if its *component-attr-spec-list* contains the POINTER attribute. Pointers have an association status of associated, disassociated, or undefined. If no default initialization is specified, the initial association status is undefined. To specify that the default initial status of a pointer component is to be disassociated, the pointer assignment symbol (=>) shall be followed by a reference to the intrinsic function NULL ( ) with no argument. No mechanism is provided to specify a default initial status of associated.

NOTE 4.31

A derived type may have a component that is a pointer. For example:

```
TYPE REFERENCE
```

NOTE 4.31 (cont.)

```
   INTEGER                          :: VOLUME, YEAR, PAGE
   CHARACTER (LEN = 50)             :: TITLE
   CHARACTER, DIMENSION (:), POINTER :: ABSTRACT => NULL()
END TYPE REFERENCE
```

Any object of type REFERENCE will have the four nonpointer components VOLUME, YEAR, PAGE, and TITLE, plus a pointer to an array of characters holding ABSTRACT. The size of this target array will be determined by the length of the abstract. The space for the target may be allocated (6.3.1) or the pointer component may be associated with a target by a pointer assignment statement (7.4.2).

NOTE 4.32

A pointer component of a derived type may have as its target an object of that derived type. The type definition may specify that in objects declared to be of this type, such a pointer is default initialized to disassociated. For example:

```
TYPE NODE
   INTEGER              :: VALUE = 0
   TYPE (NODE), POINTER :: NEXT_NODE => NULL ( )
END TYPE
```

A type such as this may be used to construct linked lists of objects of type NODE. See C.1.4 for an example.

4.5.3.3 The passed-object dummy argument

A **passed-object dummy argument** is a distinguished dummy argument of a procedure pointer component or type-bound procedure. It affects procedure overriding (4.5.6.2) and argument association (12.4.1.1).

If NOPASS is specified, the procedure pointer component or type-bound procedure has no passed-object dummy argument.

If neither PASS nor NOPASS is specified or PASS is specified without *arg-name*, the first dummy argument of a procedure pointer component or type-bound procedure is its passed-object dummy argument.

If PASS (*arg-name*) is specified, the dummy argument named *arg-name* is the passed-object dummy argument of the procedure pointer component or named type-bound procedure.

C442 The passed-object dummy argument shall be a scalar, nonpointer, nonallocatable dummy data object with the same declared type as the type being defined; all of its nonkind type parameters shall be assumed; it shall be polymorphic if and only if the type being defined is extensible.

NOTE 4.33

If a procedure is bound to several types as a type-bound procedure, different dummy arguments might be the passed-object dummy argument in different contexts.

4.5.3.4 Default initialization for components

Default initialization provides a means of automatically initializing pointer components to be disassociated (4.5.3.2), and nonpointer nonallocatable components to have a particular value. Allocatable components are always initialized to not allocated.

If *initialization-expr* appears for a nonpointer component, that component in any object of the type is initially defined (16.5.3) or becomes defined as specified in 16.5.5 with the value determined from *initialization-expr*. An *initialization-expr* in the EXTENDS *type-attr-spec* is for the parent component (4.5.6.1). If necessary, the value is converted according to the rules of intrinsic assignment (7.4.1.3) to a value that agrees in type, type parameters, and shape with the component. If the component is of a type for which default initialization is specified for a component, the default initialization specified by *initialization-expr* overrides the default initialization specified for that component. When one initialization **overrides** another it is as if only the overriding initialization were specified (see Note 4.35). Explicit initialization in a type declaration statement (5.1) overrides default initialization (see Note 4.34). Unlike explicit initialization, default initialization does not imply that the object has the SAVE attribute.

A subcomponent (6.1.2) is **default-initialized** if the type of the object of which it is a component specifies default initialization for that component, and the subcomponent is not a subobject of an object that is default-initialized or explicitly initialized.

NOTE 4.34

It is not required that initialization be specified for each component of a derived type. For example:

```
TYPE DATE
    INTEGER DAY
    CHARACTER (LEN = 5) MONTH
    INTEGER :: YEAR = 1994       ! Partial default initialization
END TYPE DATE
```

In the following example, the default initial value for the YEAR component of TODAY is overridden by explicit initialization in the type declaration statement:

```
TYPE (DATE), PARAMETER :: TODAY = DATE (21, "Feb.", 1995)
```

NOTE 4.35

The default initial value of a component of derived type may be overridden by default initialization specified in the definition of the type. Continuing the example of Note 4.34:

```
TYPE SINGLE_SCORE
    TYPE(DATE) :: PLAY_DAY = TODAY
    INTEGER SCORE
    TYPE(SINGLE_SCORE), POINTER :: NEXT => NULL ( )
END TYPE SINGLE_SCORE
TYPE(SINGLE_SCORE) SETUP
```

The PLAY_DAY component of SETUP receives its initial value from TODAY, overriding the initialization for the YEAR component.

NOTE 4.36

Arrays of structures may be declared with elements that are partially or totally initialized by default. Continuing the example of Note 4.35 :

```
TYPE MEMBER (NAME_LEN)
    INTEGER, NONKIND :: NAME_LEN
    CHARACTER (LEN = NAME_LEN) NAME = ''
```

NOTE 4.36 (cont.)

```
    INTEGER :: TEAM_NO, HANDICAP = 0
    TYPE (SINGLE_SCORE), POINTER :: HISTORY => NULL ( )
END TYPE MEMBER
TYPE (MEMBER) LEAGUE (36)        ! Array of partially initialized elements
TYPE (MEMBER) :: ORGANIZER = MEMBER ("I. Manage",1,5,NULL ( ))
```

ORGANIZER is explicitly initialized, overriding the default initialization for an object of type MEMBER.

Allocated objects may also be initialized partially or totally. For example:

```
ALLOCATE (ORGANIZER % HISTORY)   ! A partially initialized object of type
                                 ! SINGLE_SCORE is created.
```

4.5.3.5 Component order

Component order is an ordering of the nonparent components of a derived type; it is used for intrinsic formatted input/output and structure constructors (where component keywords are not used). Parent components are excluded from the component order of an extensible type.

The component order of a nonextended type is the order of the declarations of the components in the derived-type definition. The component order of an extended type consists of the component order of its parent type followed by any additional components in the order of their declarations in the extended derived-type definition.

4.5.3.6 Component accessibility

R440 *private-components-stmt* **is** PRIVATE

C443 (R440) A *private-components-stmt* is permitted only if the type definition is within the specification part of a module.

The default accessibility for the components of a type is private if the type definition contains a *private-components-stmt*, and public otherwise. The accessibility of a component may be explicitly declared by an *access-spec*; otherwise its accessibility is the default for the type definition in which it is declared.

If a component is private, that component name is accessible only within the module containing the definition.

NOTE 4.37

Type parameters are not components. They are effectively always public.

NOTE 4.38

The accessibility of the components of a type is independent of the accessibility of the type name. It is possible to have all four combinations: a public type name with a public component, a private type name with a private component, a public type name with a private component, and a private type name with a public component.

NOTE 4.39

An example of a type with private components is:

NOTE 4.39 (cont.)

```
MODULE DEFINITIONS
   TYPE POINT
      PRIVATE
      REAL :: X, Y
   END TYPE POINT
END MODULE DEFINITIONS
```

Such a type definition is accessible in any scoping unit accessing the module via a USE statement; however, the components X and Y are accessible only within the module.

NOTE 4.40

The following example illustrates the use of an individual component *access-spec* to override the default accessibility:

```
     TYPE MIXED
       PRIVATE
       INTEGER :: I
       INTEGER, PUBLIC :: J
     END TYPE MIXED

     TYPE (MIXED) :: M
```

The component M%J is accessible in any scoping unit where M is accessible; M%I is accessible only within the module containing the TYPE MIXED definition.

4.5.4 Type-bound procedures

R441 *type-bound-procedure-part* **is** *contains-stmt*
[*binding-private-stmt*]
proc-binding-stmt
[*proc-binding-stmt*] ...

R442 *binding-private-stmt* **is** PRIVATE

C444 (R441) A *binding-private-stmt* is permitted only if the type definition is within the specification part of a module.

R443 *proc-binding-stmt* **is** *specific-binding*
or *generic-binding*
or *final-binding*

C445 (R443) No *proc-binding-stmt* shall specify a binding that overrides (4.5.6.2) one that is inherited (4.5.6.1) from the parent type and has the NON_OVERRIDABLE binding attribute.

R444 *specific-binding* **is** PROCEDURE ■
■ [[, *binding-attr-list*] ::] *binding-name* [=> *binding*]

C446 (R444) If => *binding* appears, the double-colon separator shall appear.

If => *binding* does not appear, it is as though it had appeared with a procedure name the same as the binding name.

R445 *generic-binding* **is** GENERIC ■
■ [, *binding-attr-list*] :: *generic-spec* => *binding-list*

C447 (R445) If *generic-spec* is *generic-name*, *generic-name* shall not be the name of a nongeneric binding of the type.

C448 (R445) If *generic-spec* is OPERATOR (*defined-operator*), the interface of each binding shall be as specified in 12.3.2.1.1.

C449 (R445) If *generic-spec* is ASSIGNMENT (=), the interface of each binding shall be as specified in 12.3.2.1.2.

C450 (R445) If *generic-spec* is *dtio-generic-spec*, the interface of each binding shall be as specified in 9.5.3.7. The type of the `dtv` argument shall be *type-name*.

R446	*binding-attr*	**is**	PASS [(*arg-name*)]
		or	NOPASS
		or	NON_OVERRIDABLE
		or	*access-spec*

C451 (R446) The same *binding-attr* shall not appear more than once in a given *binding-attr-list*.

C452 (R444, R445) If the interface of the binding has no dummy argument of the type being defined, NOPASS shall appear.

C453 (R444, R445) If PASS (*arg-name*) appears, the interface of the binding shall have a dummy argument named *arg-name*.

C454 (R443) PASS and NOPASS shall not both appear in the same *binding-attr-list*.

C455 (R445) A *generic-binding* for which *generic-spec* is not *generic-name* shall have a passed-object dummy argument (4.5.3.3).

C456 (R445) An overriding binding shall have a passed-object dummy argument if and only if the binding that it overrides has a passed-object dummy argument.

C457 (R445) Within the *specification-part* of a module, each *generic-binding* shall specify, either implicitly or explicitly, the same accessibility as every other *generic-binding* in the same *derived-type-def* that has the same *generic-spec*.

R447	*binding*	**is**	*procedure-name*

C458 (R447) The *procedure-name* shall be the name of an accessible module procedure or an external procedure that has an explicit interface.

Each binding in a *proc-binding-stmt* specifies a **type-bound procedure**. A type-bound procedure may have a passed-object dummy argument **??**. A *generic-binding* specifies a type-bound generic interface.

The interface of a binding is that of the procedure specified by *procedure-name*.

NOTE 4.41

An example of a type and a type-bound procedure is:

```
TYPE, EXTENSIBLE :: POINT
  REAL :: X, Y
CONTAINS
  PROCEDURE, PASS :: LENGTH => POINT_LENGTH
END TYPE POINT
...
```

NOTE 4.41 (cont.)

and in the *module-subprogram-part* of the same module:

```
REAL FUNCTION POINT_LENGTH (A, B)
  CLASS (POINT), INTENT (IN) :: A, B
  POINT_LENGTH = SQRT ( (A%X - B%X)**2 + (A%Y - B%Y)**2 )
END FUNCTION POINT_LENGTH
```

The same *generic-spec* may be used in several *generic-binding*s within a single derived-type definition.

The default accessibility for the procedure bindings of a type is private if the type definition contains a *binding-private-stmt*, and public otherwise. The accessibility of a procedure binding may be explicitly declared by an *access-spec*; otherwise its accessibility is the default for the type definition in which it is declared.

A public type-bound procedure is accessible via any accessible object of the type. A private type-bound procedure is accessible only within the module containing the type definition.

NOTE 4.42

The accessibility of a type-bound procedure is not affected by a PRIVATE statement in the *component-part*; the accessibility of a data component is not affected by a PRIVATE statement in the *type-bound-procedure-part*.

4.5.5 Final subroutines

R448 *final-binding* **is** FINAL [::] *final-subroutine-name-list*

C459 (R448) A *final-subroutine-name* shall be the name of a module procedure with exactly one dummy argument. That argument shall be nonoptional and shall be a nonpointer, nonallocatable, nonpolymorphic variable of the derived type being defined. All nonkind type parameters of the dummy argument shall be assumed. The dummy argument shall not be INTENT(OUT).

C460 (R448) A *final-subroutine-name* shall not be one previously specified as a final subroutine for that type.

C461 (R448) A final subroutine shall not have a dummy argument with the same kind type parameters and rank as the dummy argument of another final subroutine of that type.

The FINAL keyword specifies a list of **final subroutines**. A final subroutine might be executed when a data entity of that type is finalized (4.5.5.1).

A derived type is **finalizable** if it has any final subroutines or if it has any nonpointer, nonallocatable component whose type is finalizable. A nonpointer data entity is finalizable if its type is finalizable.

NOTE 4.43

Final subroutines are effectively always "accessible". They are called for entity finalization regardless of the accessibility of the type, its other type-bound procedure bindings, or the subroutine name itself.

NOTE 4.44

Final subroutines are not inherited through type extension and cannot be overridden. The final subroutines of the parent type are called after calling any additional final subroutines of an extended type.

4.5.5.1 The finalization process

Only finalizable entities are **finalized**. When an entity is finalized, the following steps are carried out in sequence:

(1) If the dynamic type of the entity has a final subroutine whose dummy argument has the same kind type parameters and rank as the entity being finalized, it is called with the entity as an actual argument. Otherwise, if there is an elemental final subroutine whose dummy argument has the same kind type parameters as the entity being finalized, it is called with the entity as an actual argument. Otherwise, no subroutine is called at this point.

(2) Each finalizable component that appears in the type definition is finalized. If the entity being finalized is an array, each finalizable component of each element of that entity is finalized separately.

(3) If the entity is of extended type and the parent type is finalizable, the parent component is finalized.

If several entities are to be finalized as a consequence of an event specified in 4.5.5.2, the order in which they are finalized is processor-dependent. A final subroutine shall not reference or define an object that has already been finalized.

4.5.5.2 When finalization occurs

The target of a pointer is finalized when the pointer is deallocated. An allocatable entity is finalized when it is deallocated.

A nonpointer, nonallocatable object that is not a dummy argument or function result is finalized immediately before it would become undefined due to execution of a RETURN or END statement (16.5.6, item (3)). If the object is defined in a module and there are no longer any active procedures referencing the module, it is processor-dependent whether it is finalized. If the object is not finalized, it retains its definition status and does not become undefined.

If an executable construct references a function, the result is finalized after execution of the innermost executable construct containing the reference.

If an executable construct references a structure constructor, the entity created by the structure constructor is finalized after execution of the innermost executable construct containing the reference.

If a specification expression in a scoping unit references a function, the result is finalized before execution of the first executable statement in the scoping unit.

When a procedure is invoked, a nonpointer, nonallocatable object that is an actual argument associated with an INTENT(OUT) dummy argument is finalized.

When an intrinsic assignment statement is executed, *variable* is finalized after evaluation of *expr* and before the definition of *variable*.

NOTE 4.45

If finalization is used for storage management, it often needs to be combined with defined assignment.

If an object is allocated via pointer allocation and later becomes unreachable due to all pointers to that object having their pointer association status changed, it is processor dependent whether it is finalized. If it is finalized, it is processor dependent as to when the final subroutines are called.

4.5.5.3 Entities that are not finalized

If program execution is terminated, either by an error (e.g. an allocation failure) or by execution of a STOP or END PROGRAM statement, entities existing immediately prior to termination are not finalized.

NOTE 4.46

A nonpointer, nonallocatable object that has the SAVE attribute or which occurs in the main program is never finalized as a direct consequence of the execution of a RETURN or END statement.

A variable in a module is not finalized if it retains its definition status and value, even when there is no active procedure referencing the module.

4.5.6 Extensible types

A derived type that has the EXTENSIBLE or EXTENDS attribute is an **extensible type**.

A type that has the EXTENSIBLE attribute is a **base type**. A type that has the EXTENDS attribute is an **extended type**. The **parent type** of an extended type is the type named in the EXTENDS attribute specification.

NOTE 4.47

The name of the parent type might be a *type-alias* name or a local name introduced via renaming in a USE statement.

A base type is an **extension type** of itself only. An extended type is an extension of itself and of all types for which its parent type is an extension.

4.5.6.1 Inheritance

An extended type includes all of the type parameters, components, and nonfinal procedure bindings of its parent type. These are said to be **inherited** by the extended type from the parent type. They retain all of the attributes that they had in the parent type. Additional type parameters, components, and procedure bindings may be declared in the derived-type definition of the extended type.

NOTE 4.48

Inaccessible components and bindings of the parent type are also inherited, but they remain inaccessible in the extended type. Inaccessible entities occur if the type being extended is accessed via use association and has a private entity.

NOTE 4.49

A base type is not required to have any components, bindings, or parameters; an extended type is not required to have more components, bindings, or parameters than its parent type.

An object of extended type has a scalar, nonpointer, nonallocatable, **parent component** with the type and type parameters of the parent type. The name of this component is the parent type name. Components of the parent component are **inheritance associated** (16.4.4) with the corresponding components inherited from the parent type.

NOTE 4.50

A component or type parameter declared in an extended type shall not have the same name as any accessible component or type parameter of its parent type.

NOTE 4.51

Examples:

```
TYPE, EXTENSIBLE :: POINT            ! A base type
  REAL :: X, Y
END TYPE POINT

TYPE, EXTENDS(POINT) :: COLOR_POINT   ! An extension of TYPE(POINT)
  ! Components X and Y, and component name POINT, inherited from parent
  INTEGER :: COLOR
END TYPE COLOR_POINT
```

4.5.6.2 Type-bound procedure overriding

If a specific binding specified in a type definition has the same binding name as a binding inherited from the parent type then the binding specified in the type definition **overrides** the one inherited from the parent type.

The overriding binding and the inherited binding shall satisfy the following conditions:

(1) Either both shall have a passed-object dummy argument or neither shall.
(2) If the inherited binding is pure then the overriding binding shall also be pure.
(3) Either both shall be elemental or neither shall.
(4) They shall have the same number of dummy arguments.
(5) Passed-object dummy arguments, if any, shall correspond by name and position.
(6) Dummy arguments that correspond by position shall have the same names and characteristics, except for the type of the passed-object dummy arguments.
(7) Either both shall be subroutines or both shall be functions having the same result characteristics (12.2.2).
(8) If the inherited binding is PUBLIC then the overriding binding shall not be PRIVATE.

NOTE 4.52

The following is an example of procedure overriding, expanding on the example in Note 4.41.

```
TYPE, EXTENDS (POINT) :: POINT_3D
  REAL :: Z
CONTAINS
  PROCEDURE, PASS :: LENGTH => POINT_3D_LENGTH
END TYPE POINT_3D
...
```

and in the *module-subprogram-part* of the same module:

```
REAL FUNCTION POINT_3D_LENGTH ( A, B )
  CLASS (POINT_3D), INTENT (IN) :: A
  CLASS (POINT), INTENT (IN) :: B
  IF ( EXTENDS_TYPE_OF(B, A) ) THEN
    POINT_3D_LENGTH = SQRT( (A%X-B%X)**2 + (A%Y-B%Y)**2 + (A%Z-B%Z)**2 )
    RETURN
  END IF
  PRINT *, 'In POINT_3D_LENGTH, dynamic type of argument is incorrect.'
```

NOTE 4.52 (cont.)

```
  STOP
END FUNCTION POINT_3D
```

A generic binding overrides an inherited binding if they both have the same *generic-spec* and satisfy the above conditions for overriding. A generic binding with the same *generic-spec* that does not satisfy the conditions extends the generic interface; it shall satisfy the requirements specified in 16.2.3.

If a generic binding in a type definition has the same *dtio-generic-spec* as one inherited from the parent, and the `dtv` argument of the procedure it specifies has the same kind type parameters as the `dtv` argument of one inherited from the parent type, then the binding specified in the type overrides the one inherited from the parent type. Otherwise, it extends the type-bound generic interface for the *dtio-generic-spec*.

A binding of a type and a binding of an extension of that type are said to correspond if the latter binding is the same binding as the former, overrides a corresponding binding, or is an inherited corresponding binding.

A binding that has the NON_OVERRIDABLE attribute in the parent type shall not be overridden.

4.5.7 Derived-type values

The set of values of a particular derived type consists of all possible sequences of component values consistent with the definition of that derived type.

4.5.8 Derived-type specifier

A derived-type specifier is used in several contexts to specify a particular derived type and type parameters.

R449	*derived-type-spec*	**is**	*type-name* [(*type-param-spec-list*)]
		or	*type-alias-name*
R450	*type-param-spec*	**is**	[*keyword* =] *type-param-value*

C462 (R449) *type-name* shall be the name of an accessible derived type.

C463 (R449) *type-alias-name* shall be the name of an accessible type alias that is an alias for a derived type.

C464 (R449) *type-param-spec-list* shall appear if and only if the type is parameterized.

C465 (R449) There shall be exactly one *type-param-spec* corresponding to each parameter of the type.

C466 (R450) The *keyword*= may be omitted from a *type-param-spec* only if the *keyword*= has been omitted from each preceding *type-param-spec* in the *type-param-spec-list*.

C467 (R450) Each *keyword* shall be the name of a parameter of the type.

C468 (R450) An asterisk may be used as a *type-param-value* in a *type-param-spec* only in the declaration or allocation of a dummy argument.

Type parameter values that do not have type parameter keywords specified correspond to type parameters in type parameter order (4.5.2.1). If a type parameter keyword is present, the value is assigned to the type parameter named by the keyword. If necessary, the value is converted according to the rules of intrinsic assignment (7.4.1.3) to a value of the same kind as the type parameter.

4.5.9 Construction of derived-type values

A derived-type definition implicitly defines a corresponding **structure constructor** that allows construction of values of that derived type. The type and type parameters of a constructed value are specified by a derived type specifier.

R451	*structure-constructor*	**is**	*derived-type-spec* ([*component-spec-list*])
R452	*component-spec*	**is**	[*keyword* =] *component-data-source*
R453	*component-data-source*	**is**	*expr*
		or	*data-target*
		or	*proc-target*

C469 (R451) At most one *component-spec* shall be provided for a component.

C470 (R451) If a *component-spec* is be provided for a component, no *component-spec* shall be provided for any component with which it is inheritance associated.

C471 (R451) A *component-spec* shall be provided for a component unless it has default initialization or is inheritance associated with another component for which a *component-spec* is provided or that has default initialization.

C472 (R452) The *keyword*= may be omitted from a *component-spec* only if the *keyword*= has been omitted from each preceding *component-spec* in the constructor.

C473 (R452) Each *keyword* shall be the name of a component of the type.

C474 (R451) The type name and all components of the type for which a *component-spec* appears shall be accessible in the scoping unit containing the structure constructor.

C475 (R451) If *derived-type-spec* is a type name that is the same as a generic name, the *component-spec-list* shall not be a valid *actual-arg-spec-list* for a function reference that is resolvable as a generic reference (12.4.4.1).

C476 (R453) A *data-target* shall correspond to a nonprocedure pointer component; a *proc-target* shall correspond to a procedure pointer component.

C477 (R453) A *data-target* shall have the same rank as its corresponding component.

NOTE 4.53

The form 'name(...)' is interpreted as a generic *function-reference* if possible; it is interpreted as a *structure-constructor* only if it cannot be interpreted as a generic *function-reference*.

In the absence of a component keyword, each *component-data-source* is assigned to the corresponding component in component order (4.5.3.5). If a component keyword is present, the *expr* is assigned to the component named by the keyword. If necessary, each value is converted according to the rules of intrinsic assignment (7.4.1.3) to a value that agrees in type and type parameters with the corresponding component of the derived type. For nonpointer nonallocatable components, the shape of the expression shall conform with the shape of the component.

If a component with default initialization has no corresponding *component-data-source*, then the default initialization is applied to that component.

NOTE 4.54

Because no parent components appear in the defined component ordering, a value for a parent

NOTE 4.54 (cont.)

component may be specified only with a component keyword. Examples of equivalent values using types defined in Note 4.51:

```
! Create values with components x = 1.0, y = 2.0, color = 3.
TYPE(POINT) :: PV = POINT(1.0, 2.0)        ! Assume components of TYPE(POINT)
                                           ! are accessible here.
...
COLOR_POINT( point=point(1,2), color=3)    ! Value for parent component
COLOR_POINT( point=PV, color=3)            ! Available even if TYPE(point)
                                           ! has private components
COLOR_POINT( 1, 2, 3)                      ! All components of TYPE(point)
                                           ! need to be accessible.
```

A structure constructor shall not appear before the referenced type is defined.

NOTE 4.55

This example illustrates a derived-type constant expression using a derived type defined in Note 4.18:

```
PERSON (21, 'JOHN SMITH')
```

This could also be written as

```
PERSON (NAME = 'JOHN SMITH', AGE = 21)
```

NOTE 4.56

An example constructor using the derived type GENERAL_POINT defined in Note 4.25 is

```
general_point(dim=3) ( (/ 1., 2., 3. /) )
```

A derived-type definition may have a component that is an array. Also, an object may be an array of derived type. Such arrays may be constructed using an array constructor (4.8).

Where a component in the derived type is a pointer, the corresponding *component-data-source* shall be an allowable *data-target* or *proc-target* for such a pointer in a pointer assignment statement (7.4.2).

NOTE 4.57

For example, if the variable TEXT were declared (5.1) to be

```
CHARACTER, DIMENSION (1:400), TARGET :: TEXT
```

and BIBLIO were declared using the derived-type definition REFERENCE in Note 4.31

```
TYPE (REFERENCE) :: BIBLIO
```

the statement

```
BIBLIO = REFERENCE (1, 1987, 1, "This is the title of the referenced &
                                &paper", TEXT)
```

NOTE 4.57 (cont.)

is valid and associates the pointer component ABSTRACT of the object BIBLIO with the target object TEXT.

If a component of a derived type is allocatable, the corresponding constructor expression shall either be a reference to the intrinsic function NULL with no arguments, an allocatable entity, or shall evaluate to an entity of the same rank. If the expression is a reference to the intrinsic function NULL, the corresponding component of the constructor has a status of unallocated. If the expression is an allocatable entity, the corresponding component of the constructor has the same allocation status as that allocatable entity and, if it is allocated, the same bounds (if any) and value. Otherwise the corresponding component of the constructor has an allocation status of allocated and has the same bounds (if any) and value as the expression.

NOTE 4.58

When the constructor is an actual argument, the allocation status of the allocatable component is available through the associated dummy argument.

4.5.10 Derived-type operations and assignment

Intrinsic assignment of derived-type entities is described in 7.4.1. This standard does not specify any intrinsic operations on derived-type entities. Any operation on derived-type entities or defined assignment (7.4.1.4) for derived-type entities shall be defined explicitly by a function or a subroutine, and a generic interface (4.5.1, 12.3.2.1).

4.6 Type aliases

Type aliasing provides a method of data abstraction. A **type alias** is an entity that may be used to declare entities of an existing type; it is not a new type. The name of a type alias for a derived type may also be used in the *derived-type-spec* of a *structure-constructor*.

R454	*type-alias-stmt*	**is**	TYPEALIAS :: *type-alias-list*
R455	*type-alias*	**is**	*type-alias-name* => *declaration-type-spec*

C478 (R455) A *type-alias-name* shall not be the same as the name of any intrinsic type defined in this standard.

C479 (R455) A *declaration-type-spec* in a *type-alias* shall not use the CLASS keyword.

C480 (R455) A *declaration-type-spec* shall specify an intrinsic type or a previously defined derived type. Each *type-param-value* shall be an initialization expression.

Explicit or implicit declaration of an entity or component using a type alias name has the same effect as using the *declaration-type-spec* for which it is an alias.

NOTE 4.59

The declarations for X, Y, and S

```
TYPEALIAS :: DOUBLECOMPLEX => COMPLEX(KIND(1.0D0)), &
             NEWTYPE => TYPE(DERIVED), &
             ANOTHERTYPE => TYPE(NEWTYPE)
TYPE(DOUBLECOMPLEX) :: X, Y
```