Section 4: Types

Fortran provides an abstract means whereby data may be categorized without relying on a particular physical representation. This abstract means is the concept of type.

An intrinsic type is one that is defined by the language. The intrinsic types are integer, real, complex, character, and logical.

A derived type is one that is defined by a derived-type definition ((4.5.1)).

A derived type may be used only where its definition is accessible (4.5.1.1). An intrinsic type is always accessible.

A type is specified in several contexts by a **type specifier**.

R401	*type-spec*	**is**	*intrinsic-type-spec*
		or	*derived-type-spec*

C401 (R401) The *derived-type-spec* shall not specify an abstract type.

4.1 The concept of type

A type has a name, a set of valid values, a means to denote such values (constants), and a set of operations to manipulate the values.

NOTE 4.1

For example, the logical type has a set of two values, denoted by the lexical tokens .TRUE. and .FALSE., which are manipulated by logical operations.

An example of a less restricted type is the integer type. This type has a processor-dependent set of integer numeric values, each of which is denoted by an optional sign followed by a string of digits, and which may be manipulated by integer arithmetic operations and relational operations.

4.1.1 Set of values

For each type, there is a set of valid values. The set of valid values may be completely determined, as is the case for logical, or may be determined by a processor-dependent method, as is the case for integer, character, and real. For complex, the set of valid values consists of the set of all the combinations of the values of the individual components. For derived types, the set of valid values is as defined in 4.5.7.

4.1.2 Constants

The syntax for literal constants of each intrinsic type is specified in 4.4.

The syntax for denoting a value indicates the type, type parameters, and the particular value.

A constant value may be given a name (5.1.2.10, 5.2.9).

A structure constructor (4.5.9) may be used to construct a constant value of derived type from an appropriate sequence of initialization expressions (7.1.7). Such a constant value is considered to be a scalar even though the value may have components that are arrays.

4.1.3 Operations

For each of the intrinsic types, a set of operations and corresponding operators is defined intrinsically. These are described in Section 7. The intrinsic set may be augmented with operations and operators defined by functions with the OPERATOR interface (12.3.2.1). Operator definitions are described in Sections 7 and 12.

For derived types, there are no intrinsic operations. Operations on derived types may be defined by the program (4.5.10).

4.2 Type parameters

A type may be parameterized. In this case, the set of values, the syntax for denoting the values, and the set of operations on the values of the type depend on the values of the parameters.

The intrinsic types are all parameterized. Derived types may be defined to be parameterized.

A type parameter is either a kind type parameter or a length type parameter.

A kind type parameter may be used in initialization and specification expressions within the derived-type definition (4.5.1) for the type; it participates in generic resolution (16.2.3). Each of the intrinsic types has a kind type parameter named KIND, which is used to distinguish multiple representations of the intrinsic type.

NOTE 4.2

By design, the value of a kind type parameter is known at compile time. Some parameterizations that involve multiple representation forms need to be distinguished at compile time for practical implementation and performance. Examples include the multiple precisions of the intrinsic real type and the possible multiple character sets of the intrinsic character type.

A type parameter of a derived type may be specified to be a kind type parameter in order to allow generic resolution based on the parameter; that is to allow a single generic to include two specific procedures that have interfaces distinguished only by the value of a kind type parameter of a dummy argument. Generics are designed to be resolvable at compile time.

A length type parameter may be used in specification expressions within the derived-type definition for the type, but it may not be used in initialization expressions. The intrinsic character type has a length type parameter named LEN, which is the length of the string.

NOTE 4.3

The adjective "length" is used for type parameters other than kind type parameters because they often specify a length, as for intrinsic character type. However, they may be used for other purposes. The important difference from kind type parameters is that their values need not be known at compile time and might change during execution.

A type parameter value may be specified with a type specification (4.4, 4.5.8).

R402	*type-param-value*	**is**	*scalar-int-expr*
		or	*
		or	:

C402 (R402) The *type-param-value* for a kind type parameter shall be an initialization expression.

C403 (R402) A colon may be used as a *type-param-value* only in the declaration of an entity or component that has the POINTER or ALLOCATABLE attribute.

A **deferred type parameter** is a length type parameter whose value can change during execution of the program. A colon as a *type-param-value* specifies a deferred type parameter.

The values of the deferred type parameters of an object are determined by successful execution of an ALLOCATE statement (6.3.1), execution of an intrinsic assignment statement (7.4.1.3), execution of a pointer assignment statement (7.4.2), or by argument association (12.4.1.2).

NOTE 4.4

Deferred type parameters of functions, including function procedure pointers, have no values. Instead, they indicate that those type parameters of the function result will be determined by execution of the function, if it returns an allocated allocatable result or an associated pointer result.

An **assumed type parameter** is a length type parameter for a dummy argument that assumes the type parameter value from the corresponding actual argument; it is also used for an associate name in a SELECT TYPE construct that assumes the type parameter value from the corresponding selector. An asterisk as a *type-param-value* specifies an assumed type parameter.

4.3 Relationship of types and values to objects

The name of a type serves as a type specifier and may be used to declare objects of that type. A declaration specifies the type of a named object. A data object may be declared explicitly or implicitly. Data objects may have attributes in addition to their types. Section 5 describes the way in which a data object is declared and how its type and other attributes are specified.

Scalar data of any intrinsic or derived type may be shaped in a rectangular pattern to compose an array of the same type and type parameters. An array object has a type and type parameters just as a scalar object does.

A variable is a data object. The type and type parameters of a variable determine which values that variable may take. Assignment provides one means of defining or redefining the value of a variable of any type. Assignment is defined intrinsically for all types where the type, type parameters, and shape of both the variable and the value to be assigned to it are identical. Assignment between objects of certain differing intrinsic types, type parameters, and shapes is described in Section 7. A subroutine and a generic interface (4.5.1, 12.3.2.1) whose generic specifier is ASSIGNMENT (=) define an assignment that is not defined intrinsically or redefine an intrinsic derived-type assignment (7.4.1.4).

NOTE 4.5

For example, assignment of a real value to an integer variable is defined intrinsically.

The type of a variable determines the operations that may be used to manipulate the variable.

4.4 Intrinsic types

The intrinsic types are:

numeric types:	integer, real, and complex
nonnumeric types:	character and logical

The **numeric types** are provided for numerical computation. The normal operations of arithmetic, addition (+), subtraction (–), multiplication (*), division (/), exponentiation (**), identity (unary +), and negation (unary –), are defined intrinsically for the numeric types.

R403	*intrinsic-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*]

R404	*kind-selector*	**is**	([KIND =] *scalar-int-initialization-expr*)

C404 (R404) The value of *scalar-int-initialization-expr* shall be nonnegative and shall specify a representation method that exists on the processor.

4.4.1 Integer type

The set of values for the **integer type** is a subset of the mathematical integers. A processor shall provide one or more **representation methods** that define sets of values for data of type integer. Each such method is characterized by a value for a type parameter called the **kind** type parameter. The kind type parameter of a representation method is returned by the intrinsic inquiry function KIND (13.7.59). The decimal exponent range of a representation method is returned by the intrinsic function RANGE (13.7.96). The intrinsic function SELECTED_INT_KIND (13.7.105) returns a kind value based on a specified decimal range requirement. The integer type includes a zero value, which is considered neither negative nor positive. The value of a signed integer zero is the same as the value of an unsigned integer zero.

The type specifier for the integer type uses the keyword INTEGER.

If the kind type parameter is not specified, the default kind value is KIND (0) and the type specified is **default integer**.

Any integer value may be represented as a *signed-int-literal-constant*.

R405	*signed-int-literal-constant*	**is**	[*sign*] *int-literal-constant*
R406	*int-literal-constant*	**is**	*digit-string* [_ *kind-param*]
R407	*kind-param*	**is**	*digit-string*
		or	*scalar-int-constant-name*
R408	*signed-digit-string*	**is**	[*sign*] *digit-string*
R409	*digit-string*	**is**	*digit* [*digit*] ...
R410	*sign*	**is**	+
		or	–

C405 (R407) A *scalar-int-constant-name* shall be a named constant of type integer.

C406 (R407) The value of *kind-param* shall be nonnegative.

C407 (R406) The value of *kind-param* shall specify a representation method that exists on the processor.

The optional kind type parameter following *digit-string* specifies the kind type parameter of the integer constant; if it is not present, the constant is of type default integer.

An integer constant is interpreted as a decimal value.

NOTE 4.6

Examples of signed integer literal constants are:

```
473
+56
-101
21_2
21_SHORT
1976354279568241_8
```

where SHORT is a scalar integer named constant.

R411 *boz-literal-constant* **is** *binary-constant*
or *octal-constant*
or *hex-constant*
R412 *binary-constant* **is** B ' *digit* [*digit*] ... '
or B " *digit* [*digit*] ... "

C408 (R412) *digit* shall have one of the values 0 or 1.

R413 *octal-constant* **is** O ' *digit* [*digit*] ... '
or O " *digit* [*digit*] ... "

C409 (R413) *digit* shall have one of the values 0 through 7.

R414 *hex-constant* **is** Z ' *hex-digit* [*hex-digit*] ... '
or Z " *hex-digit* [*hex-digit*] ... "
R415 *hex-digit* **is** *digit*
or A
or B
or C
or D
or E
or F

Binary, octal and hexadecimal constants are interpreted according to their respective number systems. The *hex-digit*s A through F represent the numbers ten through fifteen, respectively; they may be represented by their lower-case equivalents.

C410 (R411) A *boz-literal-constant* shall appear only as a *data-stmt-constant* in a DATA statement, as the actual argument associated with the dummy argument A of the numeric intrinsic functions DBLE, REAL or INT, or as the actual argument associated with the X or Y dummy argument of the intrinsic function CMPLX.

4.4.2 Real type

The **real type** has values that approximate the mathematical real numbers. A processor shall provide two or more **approximation methods** that define sets of values for data of type real. Each such method has a **representation method** and is characterized by a value for a type parameter called the **kind** type parameter. The kind type parameter of an approximation method is returned by the intrinsic inquiry function KIND (13.7.59). The decimal precision and decimal exponent range of an approximation method are returned by the intrinsic functions PRECISION (13.7.90) and RANGE (13.7.96). The intrinsic function SELECTED_REAL_KIND (13.7.106) returns a kind value based on specified precision and decimal range requirements.

NOTE 4.7

See C.1.2 for remarks concerning selection of approximation methods.

The real type includes a zero value. Processors that distinguish between positive and negative zeros shall treat them as equivalent

(1) in all relational operations,

(2) as actual arguments to intrinsic procedures other than those for which it is explicitly specified that negative zero is distinguished, and

(3) as the *scalar-numeric-expr* in an arithmetic IF.

NOTE 4.8

On a processor that can distinguish between 0.0 and −0.0,

```
( X >= 0.0 )
```

evaluates to true if X = 0.0 or if X = −0.0,

```
( X < 0.0 )
```

evaluates to false for X = −0.0, and

```
IF (X) 1,2,3
```

causes a transfer of control to the branch target statement with the statement label "2" for both X = 0.0 and X = −0.0.

In order to distinguish between 0.0 and −0.0, a program should use the SIGN function. SIGN(1.0,X) will return −1.0 if X < 0.0 or if the processor distinguishes between 0.0 and −0.0 and X has the value −0.0.

NOTE 4.9

Historically some systems had a distinct negative zero value that presented some difficulties. Fortran standards were specified such that these difficulties had to be handled by the processor and not the user. The IEEE standard introduced a negative zero with particular properties. For example, when the exact result of an operation is negative but rounding produces a zero, the value specified by the IEEE standard is −0.0. This standard includes adjustments intended to permit IEEE-compliant processors to behave in accordance with that standard without violating this standard.

The type specifier for the real type uses the keyword REAL. The keyword DOUBLE PRECISION is an alternate specifier for one kind of real type.

If the type keyword REAL is specified and the kind type parameter is not specified, the default kind value is KIND (0.0) and the type specified is **default real**. If the type keyword DOUBLE PRECISION is specified, the kind value is KIND (0.0D0) and the type specified is type **double precision real**. The decimal precision of the double precision real approximation method shall be greater than that of the default real method.

R416	*signed-real-literal-constant*	**is**	[*sign*] *real-literal-constant*
R417	*real-literal-constant*	**is**	*significand* [*exponent-letter exponent*] [_ *kind-param*]
		or	*digit-string exponent-letter exponent* [_ *kind-param*]
R418	*significand*	**is**	*digit-string* . [*digit-string*]
		or	. *digit-string*

R419	*exponent-letter*	**is**	E
		or	D
R420	*exponent*	**is**	*signed-digit-string*

C411 (R417) If both *kind-param* and *exponent-letter* are present, *exponent-letter* shall be E.

C412 (R417) The value of *kind-param* shall specify an approximation method that exists on the processor.

A real literal constant without a kind type parameter is a default real constant if it is without an exponent part or has exponent letter E, and is a double precision real constant if it has exponent letter D. A real literal constant written with a kind type parameter is a real constant with the specified kind type parameter.

The exponent represents the power of ten scaling to be applied to the significand or digit string. The meaning of these constants is as in decimal scientific notation.

The significand may be written with more digits than a processor will use to approximate the value of the constant.

NOTE 4.10

Examples of signed real literal constants are:

```
-12.78
+1.6E3
2.1
-16.E4_8
0.45D-4
10.93E7_QUAD
.123
3E4
```

where QUAD is a scalar integer named constant.

4.4.3 Complex type

The **complex type** has values that approximate the mathematical complex numbers. The values of a complex type are ordered pairs of real values. The first real value is called the **real part**, and the second real value is called the **imaginary part**.

Each approximation method used to represent data entities of type real shall be available for both the real and imaginary parts of a data entity of type complex. A **kind** type parameter may be specified for a complex entity and selects for both parts the real approximation method characterized by this kind type parameter value. The kind type parameter of an approximation method is returned by the intrinsic inquiry function KIND (13.7.59).

The type specifier for the complex type uses the keyword COMPLEX. There is no keyword for double precision complex. If the type keyword COMPLEX is specified and the kind type parameter is not specified, the default kind value is the same as that for default real, the type of both parts is default real, and the type specified is **default complex**.

R421	*complex-literal-constant*	**is**	(*real-part* , *imag-part*)
R422	*real-part*	**is**	*signed-int-literal-constant*
		or	*signed-real-literal-constant*
		or	*named-constant*

R423 *imag-part* **is** *signed-int-literal-constant*
or *signed-real-literal-constant*
or *named-constant*

C413 (R421) Each named constant in a complex literal constant shall be of type integer or real.

If the real part and the imaginary part of a complex literal constant are both real, the kind type parameter value of the complex literal constant is the kind type parameter value of the part with the greater decimal precision; if the precisions are the same, it is the kind type parameter value of one of the parts as determined by the processor. If a part has a kind type parameter value different from that of the complex literal constant, the part is converted to the approximation method of the complex literal constant.

If both the real and imaginary parts are integer, they are converted to the default real approximation method and the constant is of type default complex. If only one of the parts is an integer, it is converted to the approximation method selected for the part that is real and the kind type parameter value of the complex literal constant is that of the part that is real.

NOTE 4.11

Examples of complex literal constants are:

```
(1.0, -1.0)
(3, 3.1E6)
(4.0_4, 3.6E7_8)
( 0., PI)
```

where PI is a previously declared named real constant.

4.4.4 Character type

The **character type** has a set of values composed of character strings. A **character string** is a sequence of characters, numbered from left to right 1, 2, 3, ... up to the number of characters in the string. The number of characters in the string is called the **length** of the string. The length is a type parameter; its value is greater than or equal to zero. Strings of different lengths are all of type character.

A processor shall provide one or more **representation methods** that define sets of values for data of type character. Each such method is characterized by a value for a type parameter called the **kind** type parameter. The kind type parameter of a representation method is returned by the intrinsic inquiry function KIND (13.7.59). The intrinsic function SELECTED_CHAR_KIND (13.7.104) returns a kind value based on the name of a character type. Any character of a particular representation method representable in the processor may occur in a character string of that representation method.

The character set defined by ISO/IEC 646:1991 is referred to as the **ASCII character set** or the **ASCII character type**. The character set defined by ISO/IEC 10646-1:2000 UCS-4 is referred to as the **ISO 10646 character set** or the **ISO 10646 character type**.

4.4.4.1 Character type specifier

The type specifier for the character type uses the keyword CHARACTER.

If the kind type parameter is not specified, the default kind value is KIND ('A') and the type specified is **default character**.

R424 *char-selector* **is** *length-selector*
or (LEN = *type-param-value* , ■

■ KIND = *scalar-int-initialization-expr*)
or (*type-param-value* , ■
■ [KIND =] *scalar-int-initialization-expr*)
or (KIND = *scalar-int-initialization-expr* ■
■ [, LEN =*type-param-value*])

R425 *length-selector* **is** ([LEN =] *type-param-value*)
or * *char-length* [,]

R426 *char-length* **is** (*type-param-value*)
or *scalar-int-literal-constant*

C414 (R424) The value of *scalar-int-initialization-expr* shall be nonnegative and shall specify a representation method that exists on the processor.

C415 (R426) The *scalar-int-literal-constant* shall not include a *kind-param*.

C416 (R424 R425 R426) A *type-param-value* of * may be used only in the following ways:

(1) to declare a dummy argument,

(2) to declare a named constant,

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

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

C417 A function name shall not be declared with an asterisk *type-param-value* unless it is of type CHARACTER and is the name of the result of an external function or the name of a dummy function.

C418 A function name declared with an asterisk *type-param-value* shall not be an array, a pointer, recursive, or pure.

C419 (R425) The optional comma in a *length-selector* is permitted only in a declaration-type-spec in a *type-declaration-stmt*.

C420 (R425) The optional comma in a *length-selector* is permitted only if no double-colon separator appears in the *type-declaration-stmt*.

C421 (R424) The length specified for a character statement function or for a statement function dummy argument of type character shall be an initialization expression.

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

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

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

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

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

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

4.4.4.2 Character literal constant

A **character literal constant** is written as a sequence of characters, delimited by either apostrophes or quotation marks.

R427 *char-literal-constant* **is** [*kind-param* _] ' [*rep-char*] ... '
or [*kind-param* _] " [*rep-char*] ... "

C422 (R427) The value of *kind-param* shall specify a representation method that exists on the processor.

The optional kind type parameter preceding the leading delimiter specifies the kind type parameter of the character constant; if it is not present, the constant is of type default character.

For the type character with kind *kind-param*, if present, and for type default character otherwise, a **representable character**, *rep-char*, is defined as follows:

(1) In free source form, it is any graphic character in the processor-dependent character set.

(2) In fixed source form, it is any character in the processor-dependent character set. A processor may restrict the occurrence of some or all of the control characters.

NOTE 4.12

FORTRAN 77 allowed any character to occur in a character context. This standard allows a source program to contain characters of more than one kind. Some processors may identify characters of nondefault kinds by control characters (called "escape" or "shift" characters). It is difficult, if not impossible, to process, edit, and print files where some instances of control characters have their intended meaning and some instances may not. Almost all control characters have uses or effects that effectively preclude their use in character contexts and this is why free source form allows only graphic characters as representable characters. Nevertheless, for compatibility with FORTRAN 77, control characters remain permitted in principle in fixed source form.

The delimiting apostrophes or quotation marks are not part of the value of the character literal constant.

An apostrophe character within a character constant delimited by apostrophes is represented by two consecutive apostrophes (without intervening blanks); in this case, the two apostrophes are counted as one character. Similarly, a quotation mark character within a character constant delimited by quotation marks is represented by two consecutive quotation marks (without intervening blanks) and the two quotation marks are counted as one character.

A zero-length character literal constant is represented by two consecutive apostrophes (without intervening blanks) or two consecutive quotation marks (without intervening blanks) outside of a character context.

The intrinsic operation **concatenation** (//) is defined between two data entities of type character (7.2.2) with the same kind type parameter.

NOTE 4.13

Examples of character literal constants are:

```
"DON'T"
'DON''T'
```

both of which have the value DON'T and

Section 5: Data object declarations and specifications

Every data object has a type and rank and may have type parameters and other attributes that determine the uses of the object. Collectively, these properties are the **attributes** of the object. The type of a named data object is either specified explicitly in a type declaration statement or determined implicitly by the first letter of its name (5.3). All of its attributes may be included in a type declaration statement or may be specified individually in separate specification statements.

NOTE 5.1

For example:

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

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

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

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

5.1 Type declaration statements

R501 *type-declaration-stmt* **is** *declaration-type-spec* [[, *attr-spec*] ... ::] *entity-decl-list*
R502 *declaration-type-spec* **is** *intrinsic-type-spec*
or TYPE (*derived-type-spec*)
or CLASS (*derived-type-spec*)
or CLASS (*)

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

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

NOTE 5.2

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

C503 (R502) The TYPE(*derived-type-spec*) shall not specify an abstract type (4.5.3).

R503 *attr-spec* **is** *access-spec*
or ALLOCATABLE
or ASYNCHRONOUS
or DIMENSION (*array-spec*)
or EXTERNAL
or INTENT (*intent-spec*)
or INTRINSIC
or *language-binding-spec*

or OPTIONAL
or PARAMETER
or POINTER
or PROTECTED
or SAVE
or TARGET
or VALUE
or VOLATILE

R504 *entity-decl* **is** *object-name* [(*array-spec*)] [* *char-length*] [*initialization*]
or *function-name* [* *char-length*]

C504 (R504) If a *type-param-value* in a *char-length* in an *entity-decl* is not a colon or an asterisk, it shall be a *specification-expr*.

R505 *object-name* **is** *name*

C505 (R505) The *object-name* shall be the name of a data object.

R506 *initialization* **is** = *initialization-expr*
or => *null-init*

R507 *null-init* **is** *function-reference*

C506 (R507) The *function-reference* shall be a reference to the NULL intrinsic function with no arguments.

C507 (R501) The same *attr-spec* shall not appear more than once in a given *type-declaration-stmt*.

C508 An entity shall not be explicitly given any attribute more than once in a scoping unit.

C509 (R501) An entity declared with the CLASS keyword shall be a dummy argument or have the ALLOCATABLE or POINTER attribute.

C510 (R501) An array that has the POINTER or ALLOCATABLE attribute shall be specified with an *array-spec* that is a *deferred-shape-spec-list* (5.1.2.5.3).

C511 (R501) An *array-spec* for an *object-name* that is a function result that does not have the ALLOCATABLE or POINTER attribute shall be an *explicit-shape-spec-list*.

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

C513 (R501) If the TARGET attribute is specified, the POINTER, EXTERNAL, INTRINSIC, or PARAMETER attribute shall not be specified.

C514 (R501) The PARAMETER attribute shall not be specified for a dummy argument, a pointer, an allocatable entity, a function, or an object in a common block.

C515 (R501) The INTENT, VALUE, and OPTIONAL attributes may be specified only for dummy arguments.

C516 (R501) The INTENT attribute shall not be specified for a dummy procedure without the POINTER attribute.

C517 (R501) The SAVE attribute shall not be specified for an object that is in a common block, a dummy argument, a procedure, a function result, an automatic data object, or an object with the PARAMETER attribute.

C518 An entity shall not have both the EXTERNAL attribute and the INTRINSIC attribute.

C519 (R501) An entity in an *entity-decl-list* shall not have the EXTERNAL or INTRINSIC attribute specified unless it is a function.

C520 (R504) The * *char-length* option is permitted only if the type specified is character.

C521 (R504) The *function-name* shall be the name of an external function, an intrinsic function, a function dummy procedure, or a statement function.

C522 (R501) The *initialization* shall appear if the statement contains a PARAMETER attribute (5.1.2.10).

C523 (R501) If *initialization* appears, a double-colon separator shall appear before the *entity-decl-list*.

C524 (R504)*initialization* shall not appear if *object-name* is a dummy argument, a function result, an object in a named common block unless the type declaration is in a block data program unit, an object in blank common, an allocatable variable, an external name, an intrinsic name, or an automatic object.

C525 (R504) If => appears in *initialization*, the object shall have the POINTER attribute. If = appears in *initialization*, the object shall not have the POINTER attribute.

C526 (R501) If the VOLATILE attribute is specified, the PARAMETER, INTRINSIC, EXTERNAL, or INTENT(IN) attribute shall not be specified.

C527 (R501) If the VALUE attribute is specified, the PARAMETER, EXTERNAL, POINTER, ALLOCATABLE, DIMENSION, VOLATILE, INTENT(INOUT), or INTENT(OUT) attribute shall not be specified.

C528 (R501) If the VALUE attribute is specified for a dummy argument of type character, the length parameter shall be omitted or shall be specified by an initialization expression with the value one.

C529 (R501) The VALUE attribute shall not be specified for a dummy procedure.

C530 (R501) The ALLOCATABLE, POINTER, or OPTIONAL attribute shall not be specified for a dummy argument of a procedure that has a *proc-language-binding-spec*.

C531 (R503) A *language-binding-spec* shall appear only in the specification part of a module.

C532 (R501) If a *language-binding-spec* is specified, the entity declared shall be an interoperable variable (15.2).

C533 (R501) If a *language-binding-spec* with a NAME= specifier appears, the *entity-decl-list* shall consist of a single *entity-decl*.

C534 (R503) The PROTECTED attribute is permitted only in the specification part of a module.

C535 (R501) The PROTECTED attribute is permitted only for a procedure pointer or named variable that is not in a common block.

C536 (R501) If the PROTECTED attribute is specified, the EXTERNAL, INTRINSIC, or PARAMETER attribute shall not be specified.

C537 A nonpointer object that has the PROTECTED attribute and is accessed by use association shall not appear in a variable definition context (16.5.7) or as the *data-target* or *proc-target* in a *pointer-assignment-stmt*.

C538 A pointer object that has the PROTECTED attribute and is accessed by use association shall not appear as

(1) A *pointer-object* in a *pointer-assignment-stmt* or *nullify-stmt*,
(2) An *allocate-object* in an *allocate-stmt* or *deallocate-stmt*, or
(3) An actual argument in a reference to a procedure if the associated dummy argument is a pointer with the INTENT(OUT) or INTENT(INOUT) attribute.

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

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

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

NOTE 5.3

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

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

If an *entity-decl* contains *initialization* and the *object-name* does not have the PARAMETER attribute, the entity is a variable with **explicit initialization**. Explicit initialization alternatively may be specified in a DATA statement unless the variable is of a derived type for which default initialization is specified. If *initialization* is =*initialization-expr*, the *object-name* is initially defined with the value specified by the *initialization-expr*; if necessary, the value is converted according to the rules of intrinsic assignment (7.4.1.3) to a value that agrees in type, type parameters, and shape with the *object-name*. A variable, or part of a variable, shall not be explicitly initialized more than once in a program. If the variable is an array, it shall have its shape specified in either the type declaration statement or a previous attribute specification statement in the same scoping unit.

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

The presence of *initialization* implies that *object-name* is saved, except for an *object-name* in a named common block or an *object-name* with the PARAMETER attribute. The implied SAVE attribute may be reaffirmed by explicit use of the SAVE attribute in the type declaration statement, by inclusion of the *object-name* in a SAVE statement (5.2.12), or by the appearance of a SAVE statement without a *saved-entity-list* in the same scoping unit.

NOTE 5.4

Examples of type declaration statements are:

NOTE 5.4 (cont.)

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

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

5.1.1 Declaration type specifiers

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

An *intrinsic-type-spec* in a type declaration statement is used to declare entities of intrinsic type.

5.1.1.1 TYPE

A TYPE type specifier is used to declare entities of a derived type.

Where a data entity is declared explicitly using the TYPE type specifier, the specified 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. If the derived type is specified in the FUNCTION statement and is defined within the body of the function, it is as if the function result variable was declared with that derived type immediately following the *derived-type-def* of the specified derived type.

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

5.1.1.2 CLASS

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

A CLASS type specifier is used to declare polymorphic objects. The declared type of a polymorphic object is the specified type if the CLASS type specifier contains a type name.

An object declared with the CLASS(*) specifier is an **unlimited polymorphic** object. An unlimited polymorphic entity is not declared to have a type. It is not considered to have the same declared type as any other entity, including another unlimited polymorphic entity.

A nonpolymorphic entity is **type compatible** only with entities of the same type. For a polymorphic entity, type compatibility is based on its declared type. A polymorphic entity that is not an unlimited polymorphic entity is type compatible with entities of the same type or any of its extensions. Even though an unlimited polymorphic entity is not considered to have a declared type, it is type compatible with all entities. An entity is said to be type compatible with a type if it is type compatible with entities of that type.

Two entities are **type incompatible** if neither is type compatible with the other.

An entity is type, kind, and rank compatible, or **TKR compatible**, with another entity if the first entity is type compatible with the second, the kind type parameters of the first entity have the same values as corresponding kind type parameters of the second, and both entities have the same rank.

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

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

NOTE 5.5

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

5.1.2 Attributes

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

5.1.2.1 Accessibility attribute

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

R508	*access-spec*	**is**	PUBLIC
		or	PRIVATE

C539 (R508) An *access-spec* shall appear only in the *specification-part* of a module.

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

NOTE 5.6

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