Section 4: Data 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 data 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 derived by composition of other types. Objects of derived type have **components**. Each component is of an intrinsic type or of a derived type. A type definition (4.5.1) is required to supply the name of the type and the names and types of its components.

NOTE 4.1

For example, if the complex type were not intrinsic but had to be derived, a type definition would be required to supply the name "complex" and declare two components, each of type real. In addition, arithmetic operators would have to be defined.

A derived type may be used only where its definition is accessible (4.5.1). An intrinsic type is always accessible.

4.1 The concept of data type

A data 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 4.2

For example, the logical data 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 data type is the integer data type. This data 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 data 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 and real. For complex or derived types, the set of valid values consists of the set of all the combinations of the values of the individual components.

4.1.2 Constants

For each of the intrinsic data types, the syntax for literal constants of that type is specified in this standard. These literal constants are described in 4.4 for each intrinsic type.

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

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

A structure constructor (4.5.8) 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 data types, a set of operations and corresponding operators are 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, the only intrinsic operation is assignment with agreement of type and type parameters. All other operations shall be defined by the program (4.5.9).

4.2 Type parameters

A data 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 data types are all parameterized. Derived types may be defined to be parameterized.

A type parameter is either a kind type parameter or a nonkind 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 (16.1.2.3). Each of the intrinsic types has a kind type parameter named KIND, which is used to distinguish multiple representations of the intrinsic type.

NOTE 4.3

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 nonkind 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 nonkind type parameter named LEN, which is the length of the string.

NOTE 4.4

A typical use of a nonkind type parameter is to specify a size. An example is the length of an entity of intrinsic character type.

A type parameter value may be specified with a type specification (5.1, 4.5.7).

R401	*type-param-value*	**is**	*scalar-int-expr*
		or	*
		or	:

C401 (R401) The *type-param-value* for a kind type parameter shall be an initialization expression.

C402 (R401) 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 nonkind type parameter whose value can change during execution of the program. A colon as a *type-param-value* specifies a deferred type parameter.

NOTE 4.5

Any entity with a deferred type parameter is required to have the ALLOCATABLE or POINTER attribute.

The values of the deferred type parameters of an object are determined by sucessful execution of an ALLOCATE statement (6.3.1), execution of a derived-type intrinsic assignment statement (7.5.1.2), execution of a pointer assignment statement (7.5.2), or by argument association (12.4.1.2).

NOTE 4.6

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 nonkind type parameter for a dummy argument that assumes the type parameter value from the corresponding actual argument. 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 data 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 scalar object of derived type is referred to as a structure.

Variables may be objects or subobjects. The data 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 when 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 (4.5.1.5, 12.3.2.1) whose generic specifier is ASSIGNMENT (=) define an assignment that is not defined intrinsically or redefine an intrinsic derived-type assignment (7.5.1.3).

NOTE 4.7

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

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

4.4 Intrinsic data types

The intrinsic data 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 (∗∗), negation (unary –), and identity (unary +), are defined intrinsically for this set of types.

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 (13.11.57). The decimal exponent range of a representation method is returned by the intrinsic function RANGE (13.11.92). The intrinsic function SELECTED_INT_KIND (13.11.101) 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 (R503).

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

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

R402	*signed-digit-string*	**is**	[*sign*] *digit-string*
R403	*digit-string*	**is**	*digit* [*digit*] ...
R404	*signed-int-literal-constant*	**is**	[*sign*] *int-literal-constant*
R405	*int-literal-constant*	**is**	*digit-string* [_ *kind-param*]
R406	*kind-param*	**is**	*digit-string*
		or	*scalar-int-constant-name*

C403 (R406) A *scalar-int-constant-name* shall be a named constant of type integer.

R407	*sign*	**is**	+
		or	–

C404 (R406) The value of *kind-param* shall be nonnegative.

C405 (R405) 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 4.8

Examples of signed integer literal constants are:

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

where SHORT is a scalar integer named constant.

R408	*boz-literal-constant*	**is**	*binary-constant*
		or	*octal-constant*
		or	*hex-constant*
R409	*binary-constant*	**is**	B ' *digit* [*digit*] ... '
		or	B " *digit* [*digit*] ... "

C406 (R409) *digit* shall have one of the values 0 or 1.

R410	*octal-constant*	**is**	O ' *digit* [*digit*] ... '
		or	O " *digit* [*digit*] ... "

C407 (R410) *digit* shall have one of the values 0 through 7.

R411	*hex-constant*	**is**	Z ' *hex-digit* [*hex-digit*] ... '
		or	Z " *hex-digit* [*hex-digit*] ... "
R412	*hex-digit*	**is**	*digit*
		or	A
		or	B
		or	C
		or	D
		or	E
		or	F

In these constants, the binary, octal, and hexadecimal digits are interpreted according to their respective number systems. The *hex-digit*s A through F may be represented by their lower-case equivalents.

A *boz-literal-constant* is treated as if the constant were an *int-literal-constant* with a *kind-param* that specifies the representation method with the largest decimal exponent range supported by the processor.

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 (13.11.57). The decimal precision and decimal exponent range of an approximation method are returned by the intrinsic functions PRECISION (13.11.86) and RANGE (13.11.92). The intrinsic function SELECTED_REAL_KIND (13.11.102) returns a kind value based on specified precision and decimal range requirements.

NOTE 4.9

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 SIGN, and

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

NOTE 4.10

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

```
          ( X .GE. 0.0 )
```

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

```
          ( X .LT. 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 4.11

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 specific 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 and the type specifier for the double precision real type uses the keyword DOUBLE PRECISION (R503).

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

R413	*signed-real-literal-constant*	**is**	[*sign*] *real-literal-constant*
R414	*real-literal-constant*	**is**	*significand* [*exponent-letter exponent*] [_ *kind-param*]
		or	*digit-string exponent-letter exponent* [_ *kind-param*]
R415	*significand*	**is**	*digit-string* **.** [*digit-string*]
		or	**.** *digit-string*
R416	*exponent-letter*	**is**	E
		or	D
R417	*exponent*	**is**	*signed-digit-string*

C408 (R414) If both *kind-param* and *exponent-letter* are present, *exponent-letter* shall be E.

C409 (R414) 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 4.12

Examples of signed real literal constants are:

```
-12.78
+1.6E3
2.1
-16.E4_8
0.45E-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 (13.11.57).

The type specifier for the complex type uses the keyword COMPLEX (R503). 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 data entity is of type **default complex**.

R418	*complex-literal-constant*	**is**	(*real-part* , *imag-part*)
R419	*real-part*	**is**	*signed-int-literal-constant*
		or	*signed-real-literal-constant*
		or	*named-constant*
R420	*imag-part*	**is**	*signed-int-literal-constant*
		or	*signed-real-literal-constant*
		or	*named-constant*

C410 (R418) 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 4.13

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 (13.11.57). The intrinsic function SELECTED_CHAR_KIND (13.11.100) 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 data type**. The character set defined by ISO/IEC 10646-1:1993 UCS-4 is referred to as the **ISO 10646 character set** of the **ISO 10646 character data type**.

The type specifier for the character type uses the keyword CHARACTER (R503).

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

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

R421	*char-literal-constant*	**is**	[*kind-param* _] ' [*rep-char*] ... '
		or	[*kind-param* _] " [*rep-char*] ... "

C411 (R421) 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 one of the following:

(1) Any character in the processor-dependent character set in fixed source form. A processor may restrict the occurrence of some or all of the control characters.

(2) Any graphic character in the processor-dependent character set in free source form.

NOTE 4.14

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 4.15

Examples of character literal constants are:

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

both of which have the value DON'T and

```
''
```

which has the zero-length character string as its value.

NOTE 4.16

Examples of nondefault character literal constants, where the processor supports the corresponding character sets, are:

```
BOLD_FACE_'This is in bold face '
ITALICS_'This is in italics '
```

where BOLD_FACE and ITALICS are named constants whose values are the kind type parameters for bold face and italic characters, respectively.

4.4.4.1 Collating sequence

Each implementation defines a collating sequence for the character set of each kind of character. A **collating sequence** is a one-to-one mapping of the characters into the nonnegative integers such that each character corresponds to a different nonnegative integer. The intrinsic functions CHAR (13.11.19) and ICHAR (13.11.50) provide conversions between the characters and the integers according to this mapping.

NOTE 4.17

For example:

```
   ICHAR ( 'X' )
```

returns the integer value of the character 'X' according to the collating sequence of the processor.

For the default character type, the only constraints on the collating sequence are the following:

(1) ICHAR ('A') < ICHAR ('B') < ... < ICHAR ('Z') for the twenty-six letters.

(2) ICHAR ('0') < ICHAR ('1') < ... < ICHAR ('9') for the ten digits.

(3) ICHAR (' ') < ICHAR ('0') < ICHAR ('9') < ICHAR ('A') or
ICHAR (' ') < ICHAR ('A') < ICHAR ('Z') < ICHAR ('0').

(4) ICHAR ('a') < ICHAR ('b') < ... < ICHAR ('z').

(5) ICHAR (' ') < ICHAR ('0') < ICHAR ('9') < ICHAR ('a') or
ICHAR (' ') < ICHAR ('a') < ICHAR ('z') < ICHAR ('0').

Except for blank, there are no constraints on the location of the special characters and underscore in the collating sequence, nor is there any specified collating sequence relationship between the upper-case and lower-case letters.

The sequence of numerical codes defined by the ASCII standard is called the **ASCII collating sequence** in this standard.

NOTE 4.18

The intrinsic functions ACHAR (13.11.2) and IACHAR (13.11.45) provide conversions between these characters and the integers of the ASCII collating sequence.

The intrinsic functions LGT, LGE, LLE, and LLT (13.11.61-13.11.64) provide comparisons between strings based on the ASCII collating sequence. International portability is guaranteed if the set of characters used is limited to the letters, digits, underscore, and special characters.

4.4.5 Logical type

The **logical type** has two values which represent true and false.

A processor shall provide one or more **representation methods** for data of type logical. 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 (13.11.57).

The type specifier for the logical type uses the keyword LOGICAL (R503).

If the kind type parameter is not specified, the default kind value is KIND (.FALSE.) and the data entity is of type **default logical**.

R422	*logical-literal-constant*	**is**	.TRUE. [_ *kind-param*]
		or	.FALSE. [_ *kind-param*]

C412 (R422) The value of *kind-param* shall specify a representation method that exists on the processor.

The optional kind type parameter following the trailing delimiter specifies the kind type parameter of the logical constant; if it is not present, the constant is of type default logical.

The intrinsic operations defined for data entities of logical type are: negation (.NOT.), conjunction (.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 data types may be derived from the intrinsic data 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.

Ultimately, a derived type is resolved into **ultimate components** that are either of intrinsic type or are allocatable or pointers.

NOTE 4.19

See C.1.1 for an example

By default, no storage sequence is implied by the order of the component definitions. However, if the definition of a derived type contains a SEQUENCE statement, 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 the definition of a derived type specifies BIND(C), the storage sequence is that required by the companion processor (2.5.10) for an entity of a C struct type with which an entity of the derived type is interoperable (15.2.4).

4.5.1 Derived-type definition

R423 *derived-type-def* **is** *derived-type-stmt*
    [*type-param-def-stmt*] ...
    [*data-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)

C413 (R424) A derived type *type-name* shall not be the same as the name of any intrinsic type defined in this standard.

C414 (R424) The same *type-attr-spec* shall not appear more than once in a given *derived-type-stmt*.

C415 (R424) EXTENSIBLE and EXTENDS shall not both appear.

C416 (R425) A *parent-type-name* shall be the name of an accessible extensible type (4.5.3) or of an accessible type alias (4.6) for an extensible type.

C417 (R424) If EXTENDS or EXTENSIBLE appears, neither BIND(C) nor SEQUENCE shall appear.

C418 (R423) If BIND(C) appears, SEQUENCE, a *type-bound-procedure-part*, or a *type-param-name-list* shall not appear.

C419 (R423) If BIND(C) appears, any derived type specified in a component definition shall be defined with the BIND(C) *type-attr-spec*.

C420 (R423) If BIND(C) appears, neither POINTER nor ALLOCATABLE shall appear in any component definition in the type.

R426 *type-param-def-stmt* **is** INTEGER [*kind-selector*] [[, *type-param-attr-spec*] ::] ■
    ■ *type-param-name-list*

C421 (R426) A *type-param-name* in a *type-param-def-stmt* in a *derived-type-def* shall be one of the *type-param-name*s in the *derived-type-stmt* of that *derived-type-def*.

R427 *type-param-attr-spec* **is** KIND
    or NONKIND

R428 *data-component-part* **is** [*private-sequence-stmt*] ...
    [*component-def-stmt*] ...

R429 *private-sequence-stmt* **is** PRIVATE
    or SEQUENCE

C422 (R429) A PRIVATE statement is permitted only if the type definition is within the specification part of a module.

C423 (R428) The same *private-sequence-stmt* shall not appear more than once in a given *derived-type-def*.

C424 (R428) If SEQUENCE appears, all derived types specified in component definitions shall be sequence types.

C425 (R423) If SEQUENCE appears, a *type-bound-procedure-part* shall not appear.

R430 *component-def-stmt* **is** *data-component-def-stmt*
    or *proc-component-def-stmt*

R431 *data-component-def-stmt* **is** *declaration-type-spec* [[, *component-attr-spec-list*] ::] ■

■ *component-decl-list*

R432 *component-attr-spec* **is** POINTER
or DIMENSION (*component-array-spec*)
or ALLOCATABLE
or *access-spec*

R433 *component-decl* **is** *component-name* [(*component-array-spec*)] ■
■ [∗ *char-length*] [*component-initialization*]

R434 *component-array-spec* **is** *explicit-shape-spec-list*
or *deferred-shape-spec-list*

R435 *component-initialization* **is** = *initialization-expr*
or => NULL ( )

C426 (R431) No *component-attr-spec* shall appear more than once in a given *component-def-stmt*.

C427 (R431) A component declared with the CLASS keyword (5.1.1.8) shall have the ALLOCATABLE or POINTER attribute.

C428 (R431) 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.

C429 (R431) 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.

C430 (R431) If the POINTER or ALLOCATABLE attribute is specified, each *component-array-spec* shall be a *deferred-shape-spec-list*.

C431 (R431) If neither the POINTER attribute nor the ALLOCATABLE attribute is specified, each *component-array-spec* shall be an *explicit-shape-spec-list*.

C432 (R434) Each bound in the *explicit-shape-spec* shall not contain references to specification functions or any object designators other than named constants or subobjects thereof.

C433 (R431) A component shall not have both the ALLOCATABLE and the POINTER attribute.

C434 (R433) The ∗ *char-length* option is permitted only if the type specified is character.

C435 (R430) Each *type-param-value* within a *component-def-stmt* shall either be a colon or a specification expression that does not contain references to specification functions or any object designators other than named constants or subobjects thereof.

C436 (R431) If *component-initialization* appears, a double-colon separator shall appear before the *component-decl-list*.

C437 (R431) 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*.

R436 *proc-component-def-stmt* **is** PROCEDURE ([*proc-interface*]) , ■
■ *proc-component-attr-spec-list* :: *proc-decl-list*

NOTE 4.20
See 12.3.2.3 for definitions of *proc-interface* and *proc-decl*.

R437 *proc-component-attr-spec* **is** POINTER
or PASS_OBJ
or *access-spec*

C438 (R436) The same *proc-component-attr-spec* shall not appear more than once in a given *proc-component-def-stmt*.

C439 (R436) POINTER shall appear in each *proc-component-attr-spec-list*.

C440 (R436) If PASS_OBJ appears, the procedure component shall have an explicit interface that has a scalar, nonpointer, nonallocatable dummy variable of type *type-name*. The first such dummy argument shall be polymorphic if and only if *type-name* is extensible.

C441 (R436) All of the nonkind type parameters of a passed-object dummy argument shall be assumed.

R438 *type-bound-procedure-part* **is** *contains-stmt*
  [*binding-private-stmt*]
  proc-binding-stmt
  [*proc-binding-stmt*] ...

R439 *binding-private-stmt* **is** PRIVATE

C442 (R438) A *binding-private-stmt* is permitted only if the type definition is within the specification part of a module.

R440 *proc-binding-stmt* **is** *specific-binding*
  or *generic-binding*
  or *final-binding*

C443 (R440) No *proc-binding-stmt* shall specify a binding that overrides (4.5.3.2) one that is inherited (4.5.3.1) from the parent type and has the NON_OVERRIDABLE binding attribute.

R441 *specific-binding* **is** PROCEDURE [(*abstract-interface-name*)] ■
  ■ [[, *binding-attr-list*] ::] *binding-name* [=> *binding*]

C444 (R441) If => *binding* appears, the double-colon separator shall appear.

C445 (R441) The *abstract-interface-name* shall appear if and only if the binding is to NULL() and is not overriding an inherited binding.

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

R442 *generic-binding* **is** GENERIC [(*abstract-interface-name*)] ■
  ■ [, *binding-attr-list*] :: *generic-spec* => *binding-list*

C446 (R442) The *abstract-interface-name* shall appear if and only if *binding-list* is a single binding that is NULL().

C447 (R442) If *generic-spec* is *generic-name*, *generic-name* shall not be the name of a specific binding of the type.

C448 (R442) If *generic-spec* is OPERATOR (*defined-operator*), the interface of each binding shall be as specified in 12.3.2.1.1.

C449 (R442) If *generic-spec* is ASSIGNMENT (=), the interface of each binding shall be as specified in 12.3.2.1.2.

C450 (R442) If *generic-spec* is *dtio-generic-spec*, the interface of each binding shall be as specified in 9.5.4.4.3. The type of the **dtv** argument shall be *type-name*.

R443 *final-binding* **is** FINAL [::] *final-subroutine-name-list*

C451 (R443) 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).

C452 (R443) A *final-subroutine-name* shall not be one previously specified as a final subroutine for that type.

C453 (R443) 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.

R444 *binding-attr* **is** PASS_OBJ

		or	NON_OVERRIDABLE
		or	*access-spec*

C454 (R444) The same *binding-attr* shall not appear more than once in a given *binding-attr-list*.

C455 (R440) If PASS_OBJ appears, the interface specified by *abstract-interface-name* or the procedure specified by *binding* shall have a scalar, nonpointer, nonallocatable dummy argument of type *type-name*. The first such dummy variable shall be polymorphic if and only if *type-name* is extensible.

C456 (R442) PASS_OBJ shall not appear in a *generic-binding* that has a *dtio-generic-spec*.

C457 (R442) PASS_OBJ shall appear in a *generic-binding* that has OPERATOR (*defined-operator*) or ASSIGNMENT (=).

C458 (R442) The PASS_OBJ attribute shall be specified for an overriding binding if and only if it is specified for all generic bindings, both inherited and declared within the type definition, with the same *generic-spec*.

C459 (R442) 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 *type-definition* that has the same *generic-spec*.

R445	*binding*	**is**	*procedure-name*
		or	NULL()

C460 (R445) The *procedure-name* shall be the name of an accessible module procedure or external procedure that has an explicit interface. If PASS_OBJ appears, the procedure shall have a scalar, nonpointer, nonallocatable dummy argument of type *type-name*. The first such dummy argument shall be polymorphic if and only if *type-name* is extensible.

C461 (R445) All of the nonkind type parameters of a passed-object dummy argument shall be assumed.

R446	*end-type-stmt*	**is**	END TYPE [*type-name*]

C462 (R446) If END TYPE is followed by a *type-name*, the *type-name* shall be the same as that in the corresponding *derived-type-stmt*.

NOTE 4.21

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 Derived-type parameters

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

Each type parameter is itself of type integer. This may be confirmed by a *type-param-def-stmt*. A type parameter is default integer unless its kind is explicitly specified in a *type-param-def-stmt*.

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. A *type-param-attr-spec* explicitly specifies whether a type parameter is kind or nonkind. The KIND attribute may also be implicitly conferred as described below. If a type parameter is not given the KIND attribute, either explicitly or implicitly, it is a nonkind type parameter. It is not required to explicitly specify a type parameter to be a nonkind parameter, but such specification is allowed.

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*.

With the exception stated below, a type parameter is implicitly given the KIND attribute if it appears in the *derived-type-def* as a primary in an expression that is required to be an initialization expression.

NOTE 4.22

In most cases, it is not necessary to explicitly declare anything about a type parameter; it is implicitly of type default integer and will implicitly get the KIND attribute if needed. For example, consider

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

Both k and d are default integer. The parameter k implicitly has the KIND attribute because it is used in a context that requires it to.

The following example uses explicit type parameter declarations.

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

In the following example, dim is explicitly declared to be a kind parameter, even though it is not required by anything shown here. This would allow generic overloading of procedures distinguished only by dim.

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

If a derived type has a component that is a pointer of a (possibly different) derived type, the appearance of a type parameter of the containing type in an expression for a kind type parameter of the component implicitly declares it to be a kind type parameter of the containing type only if the type definition for the component preceded that of the containing type.

NOTE 4.23

This rule is to avoid indeterminacy caused by mutually recursive derived type definitions. For example

```
TYPE type_1(a)
  INTEGER, KIND :: a  !-- required because we don't yet know whether
                      !-- type_2 has a kind type parameter.
  TYPE(type_2(a)), POINTER :: comp
END TYPE

TYPE type_2(b)
  !-- No explicit declaration of b needed here.
  TYPE(type_1(b)), POINTER :: comp
END TYPE
```

This is not at issue except with pointer components because a nonpointer component is required to be of a previously defined type.

4.5.1.2 Default initialization for components

If *initialization-expr* appears for a nonpointer component, that component in any object of the type is initially defined (16.8.3) or becomes defined as specified in 16.8.5 with the value determined from *initialization-expr*. An *initialization-expr* in the EXTENDS *type-attr-spec* is for the parent component. If necessary, the value is converted according to the rules of intrinsic assignment (7.5.1.5) 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 4.25). Explicit initialization in a type declaration statement (5.1) overrides default initialization (see Note 4.24). 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. If a subcomponent of an object is default-initialized, no subcomponent of that component is default-initialized (any default initialization of such subcomponents has been overridden by the default initialization of the higher subcomponent).

NOTE 4.24

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 4.25

The default initial value of a component of derived type may be overridden by default initialization specified in the definition of the type.

```
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 4.26

Arrays of structures may be declared with elements that are partially or totally initialized by default. For example:

```
TYPE MEMBER (NAME_LEN)
    CHARACTER (LEN = NAME_LEN) NAME = ''
    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.
```

4.5.1.3 Array components

A component is an array if its *component-decl* contains a *component-array-spec* or its *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, 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 4.27

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 4.28

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 4.29

Default initialization of an 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.

4.5.1.4 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 4.30

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

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

Any object of type REFERENCE will have the four 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 (7.5.2).

NOTE 4.31

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
   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.

4.5.1.5 Type-bound procedures

Each binding in a *proc-binding-stmt* specifies a **type-bound procedure**. A *generic-binding* specifies a type-bound generic interface.

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

NOTE 4.32

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

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

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-bindings* within a single derived-type definition.

A binding that specifies the NULL intrinsic instead of a procedure name is said to be deferred.

If a type has a deferred procedure binding then any extension of the type shall specify a procedure binding for each inherited deferred binding. This new binding may supply a specific procedure or may confirm that the binding is still deferred.

It is permissible to override (4.5.3.2) an inherited binding with a deferred binding.

NOTE 4.33

A binding may be to a procedure or may be deferred. A deferred binding that is not a *dtio-generic-spec* shall not be referenced. If a deferred binding is selected for use during data transfer (16.1.2.4.4), an error condition occurs.

4.5.1.6 The passed-object dummy argument

If PASS_OBJ is specified for a procedure pointer component or a type-bound procedure, the first dummy argument that has the same declared type as the derived type being defined is called the **passed-object dummy argument.** It affects type-bound procedure overriding (4.5.3.2) and argument association (12.4.1.1).

NOTE 4.34

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

4.5.1.7 Accessibility

By default, the names of derived types defined in the specification part of a module are accessible (5.1.2.1, 5.2.1) in any scoping unit that accesses the module. This default may be changed to restrict the accessibility of such type names to the host module itself. The name of a particular derived type may be declared to be public or private regardless of the default accessibility declared for the module. In addition, a type name may be accessible while some or all of its components are private.

The accessibility of a derived type name may be declared explicitly by an *access-spec* in its *derived-type-stmt* or in an *access-stmt* (5.2.1). The accessibility is the default if it is not declared explicitly. If a type definition is private, then the type name, and thus the structure constructor (4.5.8) for the type, is accessible only within the module containing the definition.

The default accessibility for the components of a type is private if the *data-component-part* contains a PRIVATE statement, and public otherwise. The default acessibility for the procedure bindings of a type is private if the *type-bound-procedure-part* contains a PRIVATE statement, and public otherwise. The accessibility of a component or 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.

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

NOTE 4.35

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

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 4.36

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

NOTE 4.37

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 4.38

An example of a type with private components is:

```
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.

A derived-type definition may have a component that is of a derived type. For example:

```
TYPE TRIANGLE
   TYPE (POINT) :: A, B, C
END TYPE TRIANGLE
```

An example of declaring a variable T to be of type TRIANGLE is:

```
TYPE (TRIANGLE) :: T
```

NOTE 4.39

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.

NOTE 4.40

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

```
          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 only accessible within the module containing the TYPE MIXED definition.

4.5.1.8 Sequence type

If the **SEQUENCE statement** is present, the type is a sequence type. If there are no type parameters and all of the ultimate components 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 are of type default character and are not pointers or allocatable, the type is a **character sequence type**.

NOTE 4.41

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 4.42

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, 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.

4.5.1.9 Final subroutines

The FINAL keyword specifies a list of **final subroutines**. A final subroutine might be executed when a data entity of that type is finalized (4.5.10).

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

4.5.2 Determination of derived types

A particular type name shall be defined at most once in a scoping unit. 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(C) 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(C) attribute is not of the same type as an entity of a type declared to be PRIVATE or which has any components that are PRIVATE.

NOTE 4.45

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 4.46

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
   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 4.47

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.

4.5.3 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 4.48

The name of the parent type is the name that appears in the EXTENDS attribute specification. This 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.

4.5.3.1 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 4.49

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 4.50

A base type is not required to have any components, and an extended type is not required to have more components 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** (16.7.4) with the corresponding components inherited from the parent type.

NOTE 4.51

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 4.52

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
```

4.5.3.2 Type-bound procedure overriding

If a nongeneric 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 specify PASS_OBJ or neither shall.

(2) If the procedure of the inherited binding is pure then that of 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) The corresponding dummy arguments shall have the same names and characteristics, except for the type of the passed-object dummy arguments.

(6) Either both shall be subroutines or both shall be functions having the same result characteristics (12.2.2).

(7) If the inherited binding is PUBLIC then the overriding binding shall not be PRIVATE.

If a binding is deferred and does not specify an abstract interface then it inherits the interface from the binding in the parent type.

NOTE 4.53

The following is an example of procedure overriding expanding on the example in Note 4.32.

```
TYPE, EXTENDS (POINT) :: POINT_3D
  REAL :: Z
CONTAINS
  PROCEDURE, PASS_OBJ :: 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.'
  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, and shall satisfy the requirements specified in 16.1.2.3.

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 or abstract interface 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 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.

4.5.4 Component order

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

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.

4.5.5 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.

4.5.6 Derived-type values

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

4.5.7 Derived-type specifier

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

R447	*derived-type-spec*	**is**	*type-name* [(*type-param-spec-list*)]
		or	*type-alias-name*
R448	*type-param-spec*	**is**	[*keyword* =] *type-param-value*

C463 (R447) *type-name* shall be the name of an accessible derived type.

C464 (R447) *type-alias-name* shall be the name of an accessible type alias that is an alias for a derived type.

C465 (R447) *type-param-spec-list* shall appear if and only if the type is parameterized.

C466 (R447) There shall be exactly one *type-param-spec* corresponding to each parameter of the type.

C467 (R448) 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*.

C468 (R448) Each *keyword* shall be the name of a parameter of the type.

C469 (R448) 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 (4.5.5). 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 acording to the rules of intrinsic assignment (7.5.1.4) to a value that agrees with the type parameters of the type parameter.

4.5.8 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.

R449	*structure-constructor*	**is**	*derived-type-spec* ([*component-spec-list*])
R450	*component-spec*	**is**	[*keyword* =] *expr*

C470 (R449) There shall be at most one *component-spec* corresponding to each component of the type.

C471 (R449) There shall be exactly one *component-spec* corresponding to each component that does not have a default initialization.

C472 (R450) 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 (R450) Each *keyword* shall be the name of a component of the type.

C474 (R449) 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 (R449) 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 (16.1.2.4.1).

NOTE 4.54

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 *expr* is assigned to the corresponding component in component order (4.5.4). 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 (7.5.1.4) to a value that agrees in type and type parameters with the corresponding component of the derived type. For nonpointer components, the shape of the expression shall conform with the shape of the component.

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

If a structure constructor for an extended type specifies a value for a parent component, it shall not specify a value for any component that is associated with the parent component (4.5.3.1).

NOTE 4.55

Because no parent components appear in the defined component ordering, a value for a parent component may only be specified with a component keyword. Examples of equivalent values using types defined in Note 4.52:

```
! 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 4.56

This example illustrates a derived-type constant expression using a derived type defined in Note 4.21:

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

This could also be written as

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

NOTE 4.57

An example constructor using the derived type GENERAL_POINT defined in Note 4.22 is

```
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 constructor expression shall evaluate to an object that would be an allowable target for such a pointer in a pointer assignment statement (7.5.2).

NOTE 4.58

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 4.30

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

the statement

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

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 not currently allocated. 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 currently allocated and has the same bounds (if any) and value as the expression.

NOTE 4.59

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

4.5.9 Derived-type operations and assignment

Any operation on derived-type entities or nonintrinsic assignment for derived-type entities shall be defined explicitly by a function or a subroutine and a generic interface (4.5.1.5, 12.3.2.1).

4.5.10 The finalization process

When a finalizable 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. If there is no such subroutine, but 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. If no final subroutine fulfills these requirements, no subroutine is called at this point.

(2) Each finalizable component that appears in the type definition is finalized. If the entity is an array, the components of each element are 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 at the same time, 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.

4.5.11 When finalization occurs

A pointer or 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 (16.8.6, 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 whose result is finalizable, the result is finalized after execution of the innermost executable construct containing the reference.

If an executable construct references a structure constructor for a finalizable type, 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 whose result is finalizable, 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 4.60

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.

4.5.12 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 4.61

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.

4.6 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 data type; it is not a new data type. The name of a type alias for a derived type may also be used in the *derived-type-spec* of a *structure-constructor.*

R451	*type-alias-stmt*	**is**	TYPEALIAS :: *type-alias-list*
R452	*type-alias*	**is**	*type-alias-name* => *declaration-type-spec*

C476 (R452) A *type-alias-name* shall not be the same as the name of any intrinsic type defined in this standard.

C477 (R452) A *declaration-type-spec* in a *type-alias* shall not use the CLASS keyword.

C478 (R452) A *declaration-type-spec* shall specify an intrinsic type or a previously defined derivd type.

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 4.62

The declarations for X, Y, and S

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

are equivalent to the declarations

```
COMPLEX(KIND(1.0D0)) :: X, Y
TYPE(DERIVED) :: S, T
```

4.7 Enumerations and enumerators

An enumeration is a type alias for an integer type. An enumerator is a named integer constant. An enumeration definition specifies the enumeration and a set of enumerators of the corresponding integer kind.

R453	*enum-alias-def*	**is**	*enum-def-stmt* *enumerator-def-stmt* [*enumerator-def-stmt*] ... *end-enum-stmt*
R454	*enum-def-stmt*	**is**	ENUM, BIND(C) :: *type-alias-name*
		or	ENUM [*kind-selector*] [::] *type-alias-name*
R455	*enumerator-def-stmt*	**is**	ENUMERATOR [::] *enumerator-list*
R456	*enumerator*	**is**	*named-constant* [= *scalar-int-initialization-expr*]
R457	*end-enum-stmt*	**is**	END ENUM [*type-alias-name*]

C479 (R455) If = appears for an enumerator in an ENUMERATOR statement, a double-colon separator shall appear before the *enumerator-list*.

C480 (R457) If END ENUM is followed by a *type-alias-name*, the *type-alias-name* shall be the same as that in the corresponding *enum-def-stmt*.

The *type-alias-name* of an enumeration is treated as if it were explicitly declared in a type alias statement as a type alias for an integer whose kind parameter is determined as follows:

(1) If BIND(C) is specified, the kind is selected such that an entity of type integer with that kind is interoperable (15.2) with an entity of the corresponding C enumeration type. The corresponding C enumeration type is the type that would be declared by a C enumeration specifier (6.7.2.2 of the C standard) that specified C enumeration constants with the same values as those specified by the *enum-alias-def*, in the same order as specified by the *enum-alias-def*.

The companion processor (2.5.10) shall be one that uses the same representation for the types declared by all C enumeration specifiers that specify the same values in the same order.

NOTE 4.63

If a companion processor uses an unsigned type to represent a given enumeration type, the Fortran processor will use the signed integer type of the same width for the enumeration, even though some of the values of the enumerators cannot be represented in this signed integer type. The values of any such enumerators will be interoperable with the values declared in the C enumeration.

NOTE 4.64

C guarantees the enumeration constants fit in a C int (6.7.2.2 of the C standard). Therefore, the Fortran processor can evaluate all enumerator values using the integer type with kind parameter C_INT, and then determine the kind parameter of the integer type that is interoperable with the corresponding C enumerated type.

(2) If *kind-selector* is specified, the kind is that specified by the *kind-selector*.

(3) If neither BIND(C) nor *kind-selector* is specified, the kind is that of default integer.

NOTE 4.65

The C standard specifies that two enumeration types are compatible only if they specify enumeration constants with the same names and same values in the same order. This standard further requires that a C processor that is to be a companion processor of a Fortran processor use the same representation for two enumeration types if they both specify enumeration constants with the same values in the same order, even if names are different.

An enumerator is treated as if it were explicitly declared with type *type-alias-name* and with the PARAMETER attribute. The enumerator is defined in accordance with the rules of intrinsic assignment (7.5) with the value determined as follows:

(1) If *scalar-int-initialization-expr* is specified, the value of the enumerator is the result of *scalar-int-initialization-expr*.

(2) If *scalar-int-initialization-expr* is not specified and the enumerator is the first enumerator in *enum-alias-def*, the enumerator has the value 0.

(3) If *scalar-int-initialization-expr* is not present and the enumerator is not the first enumerator in *enum-alias-def*, its value is the result of adding 1 to the value of the enumerator that immediately precedes it in the *enum-alias-def*.

NOTE 4.66

The declarations

```
ENUM (SELECTED_INT_KIND (1)) :: DIGITS
  ENUMERATOR :: ZERO, ONE, TWO
END ENUM DIGITS

ENUM :: PRIMARY_COLORS
  ENUMERATOR :: RED = 4, BLUE = 9
  ENUMERATOR YELLOW
END ENUM

TYPE (DIGITS) :: X
```

are equivalent to the declarations

```
TYPEALIAS :: DIGITS => INTEGER (SELECTED_INT_KIND(1))
TYPE (DIGITS), PARAMETER :: ZERO = 0, ONE = 1, TWO = 2
TYPE (DIGITS) :: X

! The kind type parameter for PRIMARY_COLORS is processor dependent, but the
! processor is required to select a kind sufficient to represent the values
! 4, 9, and 10, which are the values of its enumerators.
! The following declaration is one possibility for PRIMARY_COLORS.
TYPEALIAS :: PRIMARY_COLORS => INTEGER (SELECTED_INT_KIND (2))
TYPE (PRIMARY_COLORS), PARAMETER :: RED = 4, BLUE = 9, YELLOW = 10
```

NOTE 4.67

There is no difference in the effect of declaring the enumerators in multiple ENUMERATOR statements or in a single ENUMERATOR statement. The order in which the enumerators in an enumeration definition are declared is significant, but the number of ENUMERATOR statements is not.

4.8 Construction of array values

An **array constructor** is defined as a sequence of scalar values and is interpreted as a rank-one array where the element values are those specified in the sequence.

R458	*array-constructor*	**is**	(/ *ac-spec* /)
		or	*left-square-bracket ac-spec right-square-bracket*
R459	*ac-spec*	**is**	*type-spec* ::
		or	[*type-spec* ::] *ac-value-list*
R460	*left-square-bracket*	**is**	[
R461	*right-square-bracket*	**is**	]
R462	*ac-value*	**is**	*expr*
		or	*ac-implied-do*
R463	*ac-implied-do*	**is**	(*ac-value-list* , *ac-implied-do-control*)
R464	*ac-implied-do-control*	**is**	*ac-do-variable* = *scalar-int-expr* , *scalar-int-expr* ■ ■ [, *scalar-int-expr*]
R465	*ac-do-variable*	**is**	*scalar-int-variable*

C481 (R465) *ac-do-variable* shall be a named variable.

C482 (R459) If *type-spec* is omitted, each *ac-value* expression in the *array-constructor* shall have the same type and kind type parameters.

C483 (R459) If *type-spec* specifies an intrinsic type, each *ac-value* expression in the *array-constructor* shall be of an intrinsic type that is compatible with intrinsic assignment to a variable of type *type-spec* as specified in Table 7.8.

C484 (R459) If *type-spec* specifies a derived type, all *ac-value* expressions in the *array-constructor* shall be of that derived type and shall have the same kind type parameter values as specified by *type-spec*.

If *type-spec* is omitted, each *ac-value* expression in the *array-constructor* shall have the same type and type parameters. The type and type parameters of the array constructor are those of the *ac-value* expressions.

If *type-spec* appears, it specifies the type and type parameters of the array constructor. Each *ac-value* expression in the *array-constructor* shall be compatible with intrinsic assignment to a variable of this type and type parameters. Each value is converted to the type parameters of the *array-constructor* in accordance with the rules of intrinsic assignment (7.5.1.5).

If an *ac-value* is a scalar expression, its value specifies an element of the array constructor. If an *ac-value* is an array expression, the values of the elements of the expression, in array element order (6.2.2.2), specify the corresponding sequence of elements of the array constructor. If an *ac-value* is an *ac-implied-do*, it is expanded to form an *ac-value* sequence under the control of the *ac-do-variable*, as in the DO construct (8.1.5.4).

For an *ac-implied-do*, the loop initialization and execution is the same as for a DO construct. The *ac-do-variable* of an *ac-implied-do* that is in another *ac-implied-do* shall not appear as the *ac-do-variable* of the containing *ac-implied-do*.

An empty sequence forms a zero-sized rank-one array.

NOTE 4.68

A one-dimensional array may be reshaped into any allowable array shape using the RESHAPE intrinsic function (13.11.95). An example is:

```
X = (/ 3.2, 4.01, 6.5 /)
Y = RESHAPE (SOURCE = [ 2.0, [ 4.5, 4.5 ], X ], SHAPE = [ 3, 2 ])
```

This results in Y having the 3×2 array of values:

```
2.0   3.2
4.5   4.01
4.5   6.5
```

NOTE 4.69

Examples of array constructors containing an implied-DO are:

```
(/ (I, I = 1, 1075) /)
```

and

```
[ 3.6, (3.6 / I, I = 1, N) ]
```

NOTE 4.70

Using the type definition for PERSON in Note 4.21, an example of the construction of a derived-type array value is:

```
(/ PERSON (40, 'SMITH'), PERSON (20, 'JONES') /)
```

NOTE 4.71

Using the type definition for LINE in Note 4.27, an example of the construction of a derived-type scalar value with a rank-2 array component is:

```
LINE (RESHAPE ( (/ 0.0, 0.0, 1.0, 2.0 /), (/ 2, 2 /) ), 0.1, 1)
```

The RESHAPE intrinsic function is used to construct a value that represents a solid line from (0, 0) to (1, 2) of width 0.1 centimeters.