Section 4: Intrinsic and derived 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. Each data type has a name. The names of the intrinsic types are predefined by the language; the names of any derived types shall be defined in type definitions (4.4.1). A data type is characterized by a set of values, a means to denote the values, and a set of operations that can manipulate and interpret the values.

NOTE 4.1

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.

The means by which a value is denoted indicates both the type of the value and a particular member of the set of values characterizing that type. Intrinsic data types are parameterized. In this case, the set of values is constrained by the value of the parameter or parameters. For example, the character data type has a length parameter that constrains the set of character values to those whose length is equal to the value of the parameter.

An intrinsic type is one that is predefined by the language. The intrinsic types are integer, real, complex, character, and logical. The phrase "defined intrinsically" will be used later in this section to mean "predefined" in this sense.

In addition to the intrinsic types, application specific types may be derived. Objects of derived type have **components**. Each component is of an intrinsic type or of a derived type. A type definition (4.4.1) is required to supply the name of the type and the names and types of its components.

NOTE 4.2

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.

Means are provided to denote values of a derived type (4.4.4) and to define operations that can be used to manipulate objects of a derived type (4.4.5). A derived type shall be defined in the program, whereas an intrinsic type is predefined.

A derived type may be used only where its definition is accessible (4.4.1). 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.

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. For parameterized types, the set of valid values depends on the values of the parameters.

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 4.3 for each intrinsic type. Within a program, all literal constants that have the same form have the same value.

A constant value may be given a name (5.1.2.1, 5.2.9).

A constant value of derived type may be constructed (4.4.4) using a structure constructor from an appropriate sequence of constant expressions (7.1.6.1). 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. All other operations shall be defined by the program (4.4.5).

4.2 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. Once a derived type is defined, an object may be declared to be of that type. 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. The components of a structure are subobjects.

Variables may be objects or subobjects. The data type of a variable determines 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 (7.5.1.3) and an ASSIGNMENT interface block (12.3.2.1) define an assignment that is not defined intrinsically or redefine an intrinsic derived-type assignment.

NOTE 4.3

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.

4.3 Intrinsic data types

The intrinsic data types are:

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

NOTE 4.4

In addition to these intrinsic types, this standard provides derived types to allow the creation of new data types. See C.1.1 for an example.

4.3.1 Numeric types

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.

4.3.1.1 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 (13.14.52). The decimal exponent range of a representation method is returned by the intrinsic function RANGE (13.14.87). The intrinsic function SELECTED_INT_KIND (13.14.94) 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 is the keyword INTEGER (R502).

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*.

R401	*signed-digit-string*	**is**	[*sign*] *digit-string*
R402	*digit-string*	**is**	*digit* [*digit*] ...
R403	*signed-int-literal-constant*	**is**	[*sign*] *int-literal-constant*
R404	*int-literal-constant*	**is**	*digit-string* [_ *kind-param*]
R405	*kind-param*	**is**	*digit-string*
		or	*scalar-int-constant-name*
R406	*sign*	**is**	+
		or	–

Constraint: The value of *kind-param* shall be nonnegative.

Constraint: 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 4.5

Examples of signed integer literal constants are:

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

where SHORT is a scalar integer named constant.

In a DATA statement (5.2.10), an unsigned binary, octal, or hexadecimal literal constant shall correspond to an integer scalar variable.

R407 *boz-literal-constant* **is** *binary-constant*
or *octal-constant*
or *hex-constant*

Constraint: A *boz-literal-constant* may appear only in a DATA statement.

R408 *binary-constant* **is** B ' *digit* [*digit*] ... '
or B " *digit* [*digit*] ... "

Constraint: *digit* shall have one of the values 0 or 1.

R409 *octal-constant* **is** O ' *digit* [*digit*] ... '
or O " *digit* [*digit*] ... "

Constraint: *digit* shall have one of the values 0 through 7.

R410 *hex-constant* **is** Z ' *hex-digit* [*hex-digit*] ... '
or Z " *hex-digit* [*hex-digit*] ... "

R411 *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. If the processor supports lower-case letters in the source form, the *hex-digit*s A through F may be represented by their lower-case equivalents.

4.3.1.2 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 (13.14.52). The decimal precision and decimal exponent range of an approximation method are returned by the intrinsic functions PRECISION (13.14.81) and RANGE (13.14.87). The intrinsic function SELECTED_REAL_KIND (13.14.95) returns a kind value based on specified precision and decimal range requirements.

NOTE 4.6

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 4.7

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 4.8

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. ANSI/IEEE 754-1985, IEEE standard for binary floating point arithmetic, introduced a negative zero with specific properties. For example when the exact result of an operation is negative but rounding produces a zero, the IEEE 754 specified value is -0.0. This standard includes adjustments intended to permit IEEE 754 compliant processors to behave in accordance with that standard without violating this standard.

The type specifier for the real type is the keyword REAL and the type specifier for the double precision real type is the keyword DOUBLE PRECISION (R502).

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.

R412 *signed-real-literal-constant* **is** [*sign*] *real-literal-constant*

R413 *real-literal-constant* **is** *significand* [*exponent-letter exponent*] [_ *kind-param*]
or *digit-string exponent-letter exponent* [_ *kind-param*]

R414 *significand* **is** *digit-string* **.** [*digit-string*]
or **.** *digit-string*

R415 *exponent-letter* **is** E
or D

R416 *exponent* **is** *signed-digit-string*

Constraint: If both *kind-param* and *exponent-letter* are present, *exponent-letter* shall be E.

Constraint: 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 4.9

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.

4.3.1.3 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 (13.14.52).

The type specifier for the complex type is the keyword COMPLEX (R502). 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**.

R417	*complex-literal-constant*	**is**	(*real-part* , *imag-part*)
R418	*real-part*	**is**	*signed-int-literal-constant*
		or	*signed-real-literal-constant*
R419	*imag-part*	**is**	*signed-int-literal-constant*
		or	*signed-real-literal-constant*

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 signed integer literal constants, they are converted to the default real approximation method and the constant is of type default complex. If only one of the parts is a signed integer literal constant, the signed integer literal constant is converted to the approximation method selected for the signed real literal constant and the kind type parameter value of the complex literal constant is that of the signed real literal constant.

NOTE 4.10

Examples of complex literal constants are:

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

4.3.2 Nonnumeric types

The **nonnumeric types** are provided for nonnumeric processing. The intrinsic operations defined for each of these types are given below.

4.3.2.1 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 (13.14.52). Any character of a particular representation method representable in the processor may occur in a character string of that representation method.

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

The type specifier for the character type is the keyword CHARACTER (R502).

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

R420 *char-literal-constant* **is** [*kind-param* _] ' [*rep-char*] ... '
or [*kind-param* _] " [*rep-char*] ... "

Constraint: 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 4.11

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 4.12

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 4.13

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.

4.3.2.1.1 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 (13.14.19) and ICHAR (13.14.45) provide conversions between the characters and the integers according to this mapping.

NOTE 4.14

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'), if the processor supports lower-case letters.

(5) ICHAR (' ') < ICHAR ('0') < ICHAR ('9') < ICHAR ('a') or
ICHAR (' ') < ICHAR ('a') < ICHAR ('z') < ICHAR ('0'), if the processor supports lower-case letters.

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.

ISO/IEC 646:1991 (International Reference Version) assigns numerical codes to a set of characters that includes the letters, digits, underscore, and special characters; the sequence of such codes is called in this standard the **ASCII collating sequence**.

NOTE 4.15

> ISO/IEC 646:1991 is the international equivalent of ANSI X3.4-1986, commonly known as ASCII. The intrinsic functions ACHAR (13.14.2) and IACHAR (13.14.40) provide conversions between these characters and the integers of the ASCII collating sequence.

The intrinsic functions LGT, LGE, LLE, and LLT (13.14.56-13.14.59) 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.

4.3.2.2 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 (13.14.52).

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

R421	*logical-literal-constant*	**is**	.TRUE. [_ *kind-param*]
		or	.FALSE. [_ *kind-param*]

Constraint: 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.

The type specifier for the logical type is the keyword LOGICAL (R502).

4.4 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 types of its components. The **direct components** of a derived type are

(1) The components of that type and

(2) For any nonpointer component that is of derived type, the direct components of that derived type.

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

> **NOTE 4.16**
> See C.1.1 for an example

By default, derived types defined in the specification part of a module are accessible (5.1.2.2, 5.2.3) in any scoping unit that accesses the module. This default may be changed to restrict the accessibility of such types to the host module itself. A particular type definition may be declared to be public or private regardless of the default accessibility declared for the module. In addition, a type may be accessible while its components are private.

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.

Default initialization is specified for a component of an object of derived type when initialization appears in the component declaration. The object will be initialized as specified in the derived-type definition (14.7.3, 14.7.5) even if the definition is private or inaccessible. Default initialization applies to dummy arguments with INTENT (OUT). Unlike explicit initialization, default initialization (4.4.1) does not imply that the object has the SAVE attribute. If a component has default initialization, it is not required that default initialization be specified for other components of the derived type.

The type specifier for derived types is the keyword TYPE followed by the name of the type in parentheses (R502).

4.4.1 Derived-type definition

R422 *derived-type-def* **is** *derived-type-stmt*
  [*private-sequence-stmt*] ...
  component-def-stmt
  [*component-def-stmt*] ...
  end-type-stmt

R423 *derived-type-stmt* **is** TYPE [[, *access-spec*] ::] *type-name*

R424 *private-sequence-stmt* **is** PRIVATE
  or SEQUENCE

Constraint: An *access-spec* (5.1.2.2) or a PRIVATE statement within the definition is permitted only if the type definition is within the specification part of a module.

Constraint: A derived type *type-name* shall not be the same as the name of any intrinsic type defined in this standard nor the same as any other accessible derived type *type-name*.

Constraint: The same *private-sequence-stmt* shall not appear more than once in a given *derived-type-def*.

Constraint: If SEQUENCE is present, all derived types specified in component definitions shall be sequence types.

R425 *component-def-stmt* **is** *type-spec* [[, *component-attr-spec-list*] ::] *component-decl-list*

R426 *component-attr-spec* **is** POINTER
  or DIMENSION (*component-array-spec*)

R427 *component-array-spec* **is** *explicit-shape-spec-list*
  or *deferred-shape-spec-list*

Constraint: If a component of a derived type is of a type declared to be private, either the derived-type definition shall contain the PRIVATE statement or the derived type shall be private.

Constraint: No *component-attr-spec* shall appear more than once in a given *component-def-stmt*.

Constraint: If the POINTER attribute is not specified for a component, a *type-spec* in the *component-def-stmt* shall specify an intrinsic type or a previously defined derived type.

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

Constraint: If the POINTER attribute is specified, each *component-array-spec* shall be a *deferred-shape-spec-list*.

Constraint: If the POINTER attribute is not specified, each *component-array-spec* shall be an *explicit-shape-spec-list*.

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

R428 *component-decl* **is** *component-name* [(*component-array-spec*)] ■
■ [∗ *char-length*] [*component-initialization*]

R429 *component-initialization* **is** = *initialization-expr*
or => NULL ( )

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

Constraint: The character length specified by the *char-length* in a *component-decl* or the *char-selector* in a *type-spec* (5.1, 5.1.1.5) shall be a constant specification expression (7.1.6.2).

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

Constraint: If => appears in *component-initialization*, the POINTER attribute shall appear in the *component-attr-spec-list*. If = appears in *component-initialization*, the POINTER attribute shall not appear in the *component-attr-spec-list*.

R430 *end-type-stmt* **is** END TYPE [*type-name*]

Constraint: 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 4.17

The double colon separator in a *component-def-stmt* is required only if a *component-attr-spec* or *component-initialization* is specified; otherwise, it is optional.

If the **SEQUENCE statement** is present, the type is a sequence type. If all of the ultimate components are of type default integer, default real, double precision real, default complex, or default logical and are not pointers, the type is a **numeric sequence type**. If all of the ultimate components are of type default character and are not pointers, the type is a **character sequence type**.

NOTE 4.18

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 non-SEQUENCE structure in memory as best suited to the particular architecture.

If *initialization-expr* appears for a nonpointer component, that component in any object of the type is initially defined or becomes defined as specified in (14.7.5) with the value determined from *initialization-expr*. The *initialization-expr* is evaluated in the scoping unit of the type definition. The evaluation rules are those that would be in effect for intrinsic assignment (7.5.1.4) if *component-name* were a variable assigned the value of *initialization-expr*. If *component-name* 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 4.28). Explicit initialization in a type declaration statement (5.1) overrides default initialization (see Note 4.27).

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 4.19

Default initialization of an array component may be specified by a constant expression consisting of an array constructor (4.5), or of a single scalar that becomes the value of each array element.

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 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.

The accessibility of a derived type may be declared explicitly by an *access-spec* in its *derived-type-stmt* or in an *access-stmt* (5.2.3). The accessibility is the default if it is not declared explicitly. If a type definition is private, then the type name, the structure constructor (4.4.4) for the type, any entity that is of the type, and any procedure that has a dummy argument or function result that is of the type are accessible only within the module containing the definition.

If a type definition contains a PRIVATE statement, the component names for the type are accessible only within the module containing the definition, even if the type itself is public (5.1.2.2). The component names and hence the internal structure of the type are inaccessible in any scoping unit accessing the module via a USE statement. Similarly, the structure constructor for such a type shall be employed only within the defining module.

NOTE 4.20

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
```

NOTE 4.21

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 4.22

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 4.23

An example of a private type 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 4.24

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 4.25

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 in a pointer assignment statement (7.5.2).

NOTE 4.26

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.1.3 for an example.

NOTE 4.27

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 4.28

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
```

NOTE 4.28 *(Continued)*

```
TYPE(SINGLE_SCORE) SETUP
```

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

NOTE 4.29

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

For example:

```
TYPE MEMBER
    CHARACTER (LEN = 20) 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.
```

4.4.2 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 have the same name, have the SEQUENCE property, and have components that do not have PRIVATE accessibility and agree in order, name, and attributes. Otherwise, they are of different derived types. A data entity declared using a type with the SEQUENCE property is not of the same type as an entity of a type declared to be PRIVATE or which has components that are PRIVATE.

NOTE 4.30

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 4.31

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 4.32

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 local names introduced via renaming in USE statements.

4.4.3 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.

4.4.4 Construction of derived-type values

A derived-type definition implicitly defines a corresponding **structure constructor** that allows a scalar value of derived type to be constructed from a sequence of values, one value for each component of the derived type.

R431 *structure-constructor* **is** *type-name* (*expr-list*)

The sequence of expressions in a structure constructor specifies component values that shall agree in number and order with the components of the derived type. 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. A structure constructor whose component values are all constant expressions is a derived-type constant expression. A structure constructor shall not appear before the referenced type is defined.

NOTE 4.33

This example illustrates a derived-type constant expression using a derived type defined in Note 4.20:

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

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 (4.5).

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 4.34

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 4.25

```
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.

4.4.5 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 procedure interface block (12.3.2.1). Arguments and function values may be of any derived or intrinsic type.

4.5 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.

R432 *array-constructor* **is** (/ *ac-value-list* /)

R433 *ac-value* **is** *expr*
or *ac-implied-do*

R434 *ac-implied-do* **is** (*ac-value-list* , *ac-implied-do-control*)

R435 *ac-implied-do-control* **is** *ac-do-variable* = *scalar-int-expr* , *scalar-int-expr* ■
■ [, *scalar-int-expr*]

R436 *ac-do-variable* **is** *scalar-int-variable*

Constraint: *ac-do-variable* shall be a named variable.

Constraint: Each *ac-value* expression in the *array-constructor* shall have the same type and kind type parameter.

If the *ac-value* expressions are of type character, each *ac-value* expression in the *array-constructor* shall have the same character length parameter.

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 (8.1.4.4).

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.

The type and type parameters of an array constructor are those of the *ac-value* expressions.

If every expression in an array constructor is a constant expression, the array constructor is a constant expression.

NOTE 4.35

An example is:

```
REAL :: X (3)
X = (/ 3.2, 4.01, 6.5 /)
```

NOTE 4.36

A one-dimensional array may be reshaped into any allowable array shape using the RESHAPE intrinsic function (13.14.90). An example is:

```
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 4.37

Examples of array constructors containing an implied-DO are:

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

and

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

NOTE 4.38

Using the type definition for PERSON in Note 4.20, an example of the construction of a derived-type array value is:

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

NOTE 4.39

Using the type definition for LINE in Note 4.21, 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.