Date:	12 August 1998
To:	J3
From:	Van Snyder
Subject:	Specifications and syntax for enumeration types
References:	98-165r1, 98-171r1

1 Introduction

Enumeration types are necessary, or at least very useful, for C interoperability, because the C standard allows a processor to choose a different integer representation (a different kind parameter, from the Fortran perspective) for each enumeration. It is difficult to predict what representation will be chosen.

Enumerations have important uses beyond C interoperability. In particular, using them as array dimensions, do inductors, and subscripts allows compilers to provide subscript bounds checking at very low run-time cost – frequently zero cost.

2 Enumeration types

To declare enumeration types and their literals, the `TYPE` statement is extended:

type-definition-stmt	**is**	TYPE [,*enum-spec-list*] :: *enum-def-list*
enum-def	**is**	*type-name* => *literals*
enum-spec	**is**	*access-spec*
	or	BIND(C)
literals	**is**	ORDERED [(*kind-selector*)] ■
		■ (*ordered-enum-list*)
	or	UNORDERED [(*kind-selector*)] ■
		■ (*unordered-enum-list*)
ordered-enum	**is**	*named-constant* [(*explicit-shape-spec*)]
unordered-enum	**is**	*named-constant* ■
		■ [= *scalar-int-initialization-expr*]
	or	*named-constant* = *boz-literal-constant*

Values of enumeration types are represented by integers.

If BIND(C) is specified, C representational rules apply, and *kind-selector* is not allowed.

If *kind-selector* is not specified, the kind of integer used to represent enumerations is separately selected for each enumeration type by the processor.

Notice that "::" is not optional, just as it is not optional in the case of initializing a pointer object by using "=> `NULL()`" in a *type-declaration-stmt*.

Enumeration types cannot be parameterized. Literals of enumeration types can be renamed during USE association.

What is the effect of USE, ONLY on an enumeration type? Does it make just the type available, or the type and the literals? Either way is probably wrong for some circumstances. It would be useful to have two syntaxes, one to say "use only the type," say, USE, ONLY: T, and another to say "use only the type and its literals," say, USE, ONLY: T()." ???

Objects of enumeration types can be declared by using
`TYPE(`*`type-name`*`) :: `*`enumeration-variable`*.

BIND(C) objects of enumeration types cannot appear in COMMON, in EQUIVALENCE, or within SEQUENCE derived types. Maybe it's ok for non-BIND(C) objects.

The intrinsic function `INT` may be used to retrieve the numeric representation of an enumeration literal or object of enumeration type. In the case of ordered enumerations, or of unordered enumerations in which no explicit value is provided for the k'th literal, the first literal is represented by zero, and the k'th literal is represented by `SIZE`(*k-1*'th literal) + `INT`(*k-1*'th literal).

The size of unordered enumeration literals, or of ordered scalar enumeration literals, is one. The size of an array enumeration literal is the number of values. The `SIZE` intrinsic function may be used to retrieve the size of an enumeration literal.

If *explicit-shape-spec* is specified for an ordered enumeration, the size must be positive. If `E` is an enumeration literal with bounds e_1:e_2, `E`(e_1) denotes the first value, etc., `E` and `E`($k:l$) are sequences of values of the type of `E`, and `INT(E)` and `INT(E(`$k:l$`))` are sequences of integers. `LBOUND(E)` returns e_1 and `UBOUND(E)` returns e_2.

It is useful to allow ordered enumeration literals to have a size other than one so that one can declare a type with literals having representations, say, 0, 1 and 10, while still guaranteeing that there are no gaps or duplications in the set of values of the type. Another application is to define an enumeration type with one literal of a specified size, which is used as an array bound. If a variable of the type is then used as a subscript, array bounds checking has no cost (at least at the point of use as a subscript – but maybe it does where the variable gets a value). Here's an example:

```
TYPE :: E => ORDERED( EV(10) )
REAL :: X(EV)
TYPE(E) :: SUB
DO SUB = TINY(E), HUGE(E) ! No check needed for value of SUB here
  PRINT *, X(SUB)         ! Bounds checking for X is FREE!
END DO
```

You also can simulate unsigned integers – there's no arithmetic (not directly, anyway), but you have a better chance of getting the right representation than with `SELECTED_INT_KIND`. Here's an example: `TYPE :: B => BV(256)`.

It is possible for two literals of an unordered enumeration type to have the same representation.

The intrinsic function `KIND` may be applied to a value of enumeration type to determine the kind of integer used to represent values of the type. The kind of a value of a BIND(C) enumeration could be -1 if the C processor uses a representation for the type for which the Fortran processor has no kind.

The only intrinsic operations defined on values of unordered enumeration types are assignment (=), equality (`.EQ.` or ==), and inequality (`.NE.` or /=).

2.1 Additional features of ordered enumerations

- All numeric relational operators are defined on values of ordered enumeration types.
- Scalar values of ordered enumeration types may be used in `SELECT CASE` constructs and `DO` constructs. Array ones may be used in `CASE` statements.

- TINY and HUGE are defined for ordered enumeration types, and return the first and last literal of the type, respectively (not an integer). Thus if one has a variable E of an ordered enumeration type, it is permitted to write DO E = TINY(E), HUGE(E), to use TINY(E) and HUGE(E) for array dimensions, etc.

- Scalar values of ordered enumeration types may be used in array dimensions and scalar or array ones may be used in subscripts. If an array has a dimension bound given by a value of an ordered enumeration type, the other bound of that dimension shall be of the same type, or omitted (in which case it is taken to be TINY or HUGE, as appropriate), and a subscript for that dimension shall be of the same type as the bound. A subscript triplet must consist of scalar values of an enumeration type. An omitted lower or upper bound of a subscript triplet is taken to be TINY or HUGE, respectively. An increment of a subscript triplet is an integer. Should increments of subscript triplets of enumeration types be prohibited? *Straw vote*

- An elemental constructor having the same name as the type is defined. It takes a single integer argument and returns a value of the enumeration type. One can guard against an out-of-range argument by writing, e.g.

  ```
  IF ( I >= INT(TINY(E)) .AND. I <= INT(HUGE(E)) ) E = <type-of-E>(I)
  ```

  A constructor is not provided for unordered enumeration types because different values of the type may have the same representation, and there may be integers between the smallest one that represents a value of the type and the largest one that represents a value of the type that do not represent values of the type.

- Two elemental intrinsic functions are defined, say SUCC and PRED (spelling negotiable) that return the successor and predecessor of a value of an ordered enumeration type. The result is the same type as the argument, not an integer.

  Should SUCC and PRED be provided? *Straw Vote*

  Should SUCC(HUGE(E)) be an error, or TINY(E)? The obvious anti-symmetric question applies to PRED. Whatever choice is made for the behavior of SUCC and PRED, one can guard against the error, or detect wrap-around, similarly to guarding against the error in the constructor. *Straw Vote*