Subject: Enumerations that are new types, and their enumerators
From: Van Snyder
Reference: 98-194r1

1 Number

TBD

2 Title

Enumerations that are new types, and their enumerators

3 Submitted By

J3

4 Status

For consideration.

5 Basic Functionality

Provide enumerations that are new types, not a shorthand for named constants of integer type.

6 Rationale

Enumerations that are aliases for integer kinds provide none of the benefits of strong typing. Enumerations have far broader uses than C interoperoperability. For example, using them as array dimensions, do inductors and subscripts allows compilers to provide subscript checking at low run-time cost — frequently zero cost.

7 Estimated Impact

This is a moderate project, requiring changes (but not extensive ones) in Sections 4, 5, 6, 9, 10, 12, 13 and maybe 15.

8 Detailed Specification

Provide for ordered and unordered enumerations. For ordered enumerations, an explicit value cannot be specified for any enumerator other than the first one of the type, and all relational operations are defined between enumerators of the type. For unordered enumerations, an explicit value can be specified for any enumerator of the type, and the only relational operations defined are equality and inequality.

The following is based on 98-194r1, which proposed also to provide C interoperable enumerations. C interoperability is not an essential part of this proposal; it could be removed without significant compromise to the facility.

One way to declare enumerations and their enumerators is to extend the TYPE statement. This is used as a vehicle to illustrate some of the advocated features.

R1	*type-definition-stmt*	**is**	TYPE [,*enum-spec-list*] :: *enum-definition-list*
R2	*enum-definition*	**is**	*type-name* => *enumerators*
R3	*enum-spec*	**is**	*access-spec*
		or	BIND(C)
R4	*enumerators*	**is**	ORDERED [(*kind-selector*)] (*first-ordered-enum* ■
			■ [, *ordered-enum-list*])
		or	UNORDERED [(*kind-selector*)] (*unordered-enum-list*)

R5 *first-ordered-enum* **is** *named-constant* [(*explicit-shape-spec*)] ■
■ [= *enum-initializer*]

R6 *ordered-enum* **is** *named-constant* [(*explicit-shape-spec*)]

R7 *unordered-enum* **is** *named-constant* [= *enum-initializer*]

R8 *enum-initializer* **is** *scalar-int-initialization-expr*
or *boz-literal-constant*

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 the enumeration is selected by the processor. The processor is not required to select the same kind for different enumerations. If *kind-selector* is specified it shall be a valid integer kind type parameter. The only reason to allow to specify a *kind-selector* is to allow storage association with objects of integer type. If this is not desired, the ability to specify a *kind-selector* should not be included.

The "::" is not optional, because of the presence of the => symbol, 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. Enumerators of enumeration types can be renamed during USE association.

What is the effect of USE, ONLY on an enumeration? Does it make just the type available, or the type and the enumerators? 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 enumerators," say, USE, ONLY: T()." ???

Objects of enumeration types can be declared by using
TYPE(*type-name*) :: *enumeration-variable* or
TYPE(*type-name*), PARAMETER :: *enumeration-constant-name* = *initialization-expr* or
TYPE(*type-name*) ... FUNCTION ....

BIND(C) objects of enumeration types cannot appear in COMMON, in EQUIVALENCE, or as components of 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 enumerator or object of enumeration type. If no *enum-initializer* is specified for the first enumerator, it is represented by zero. If it is an array enumerator its first value is represented by zero. If no *enum-initializer* is specified for the k'th enumerator, it is represented by SIZE($k-1$'th enumerator) + INT($k-1$'th enumerator). If an *enum-initializer* is specified the enumerator is represented by the value of the *enum-initializer*. If the enumerator is an array enumerator, its first value is *enum-initializer*.

The size of scalar enumerators is one. The size of an array enumerator is the number of values. The SIZE intrinsic function may be used to retrieve the size of an enumerator.

If *explicit-shape-spec* is specified for an enumerator, the size shall be positive. If E is an enumerator 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. INT(E(k)) = INT(E(e_1)) + $k - e_1 + 1$. LBOUND(E) returns e_1 and UBOUND(E) returns e_2.

It is useful to allow enumerators of ordered enumerations to have a size other than one so that one can declare a type with enumerators 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 with one enumerator 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(E)
TYPE(E) :: V
DO V = TINY(V), HUGE(V) ! No check needed for value of V here
  PRINT *, X(V)         ! 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 enumerators of an unordered enumeration 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 value of a BIND(C) enumerator could be −1 if the companion processor uses a representation for its 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 /=).

If the proposal to add function result type, kind and rank to the criteria for generic resolution is adopted, enumerators should be considered to be generic functions with no arguments and scalar result — even though they are referenced without an empty argument list. This would allow enumerators in different enumerations to have the same name.

8.1 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 objects of ordered enumeration types, and return the first and last enumerator of the type, respectively (not an integer). Thus if one has a variable V of an ordered enumeration type, it is permitted to write DO V = TINY(V), HUGE(V), to use TINY(V) and HUGE(V) for array dimensions, etc.
- The name of an ordered enumeration type may be used as a specification for a dimension of an explicit-shape array. The bounds in that case are TINY and HUGE for the type. Scalar values of ordered enumeration types may be used as array bounds. If an array has a dimension bound given by a value of an object 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 for the type, as appropriate). If the bound for a dimension is specified by the name or a value of an enumeration type, a subscript for that dimension shall be of the same type as the bound. A subscript range shall consist of scalar values of an enumeration type. An omitted lower or upper bound of a subscript range is taken to be TINY or HUGE for the type, respectively. An increment of a subscript triplet is an integer. This also applies to declarations of and references to array enumerators.

  Should increments for subscript ranges of enumeration types be prohibited? *Straw vote*
- For each ordered enumeration, 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(V)) .AND. I <= INT(HUGE(V)) ) V = <type-of-V>(I)
  ```

  A constructor is not provided for unordered enumerations because different enumerators 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(V)) be an error, or TINY(V)? 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*

8.2 Formatted input/output

Formatted input and output of values of enumeration types uses the text of the enumerator, without regard to case. For objects of unordered enumeration types other than enumerators, several enumerators may have the same representation. In this case, the output is processor dependent. For elements of array enumerators or objects having values that correspond to elements of array enumerators, the subscript shall be included. This could cause an arbitrarily large amount of output, or require an arbitrarily large amount of input, in the case that the bounds of an array enumerator are given by another enumerator, etc.

8.3 Alternatives for descoping of ambition

The facility could be simplified by removing array enumerators, and prohibiting enumerators of an unordered enumeration from having the same representation. Descoping the proposal in either or both of these ways should not result in abandoning it altogether.

9 History