The new features of Fortran 2023

John Reid, JKR Associates and
Rutherford Appleton Laboratory

BCS Fortran Specialist Group
London, 29 September 2022

Abstract

The Fortran 2023 standard is near finalization.

It is subject to a DIS (Draft International Standard) vote that is active now. There is no sign of controversy. Final publication is expected in October 2023.

The changes were all minor. I will describe the most significant ones.

I have provided a summary of all the new features as N2194 on the WG5 site,

`https://wg5-fortran.org/`

Length limits

In free source form:

- Line-length limit raised to 10,000 characters.
- Limit of 255 continuation lines removed.
- Statement-length limit is raised to a million characters.

These relaxations are designed to support programs that are constructed mechanically.

These are hard limits. Processors are required to issue warnings if they are breached.

Length of character variables

If a deferred-length allocatable variable is defined by intrinsic assignment:

```
character(:), allocatable :: quotation
:
quotation = “How now, brown cow?”
```

it is allocated by the processor to the correct length.

Also messages returned through `iomsg` and `errmsg` specifiers, writing to a scalar character variable as an internal file, and intent `out` and `inout` character arguments of intrinsic procedures, such as in

```
call get command(command)
```

Conditional expressions

Conditional expressions added.

A simple example is

```
value = ( a>0.0 ? a : 0.0)
```

which is a short way of writing

```
if (a>0.0) then
  value = a
else
  value = 0.0
end if
```

The general form is

(*cond* ? *expr* [*: cond* ? *expr*]... : *expr*)

Conditional arguments

Conditional arguments added.

```
call sub( (x>0? x : y>0? y : z), 0 )
```

General form of conditional argument is

(*cond* ? *arg* [: *cond* ? *arg*] . . . : *arg*)

where each *arg* is an expression, a variable, or `.nil.` to specify absence.

The *args* other than `.nil.` must have the same rank, type, and kind. If one is allocatable or a pointer, they must all be. This ensures that generic resolution is at compile time.

Arrays with coarray components

An object of a type that has a coarray component is allowed to be an array or allocatable, but not a coarray (which would be confusing).

```
type mine
   a[*]
end type
type (mine), allocatable :: x(:), y
allocate (y) ! y%a unallocated
```

`allocate` is not an image control statement but `deallocate` needs to be.

Put with notify

Popular in the SHMEM community. Example:

```
use iso_fortran_env
type(notify_type) nx[*] ! Has count
                        that is initially 0
:
if(this_image()/=10) then
  x[10, notify=nx] = y
else if(this_image()==10) then
  notify wait (nx, until_count=1)
  z = x
end if
```

Notify variable can be changed only this way and each update is atomic.

Reduction specifier for do concurrent

```
do concurrent(i = 1, n)reduce(+:a) &
                       reduce(max:b)
   a = a + x(i)**2
   b = max(b,x(i))
end do
```

A reduction variable must appear as

var = var op expr or *var = expr op var* where *op* is

`+, *, .and., .or., .eqv.,` or `.neqv.`

or as

var = fun(var, expr) or *var = fun (expr, var)* where *fun is*

`max, min, iand, ieor,` or `ior`

All occurences in the construct must have the same form.

Simple procedures

A **pure** procedure changes variables only through its arguments (no side effects).

A **simple** procedure is a pure procedure that references variables only through its arguments , e.g. `sin(x)`.

It is an entirely local calculation that may be executed by a thread accessing only the arguments.

All the functions that are intrinsic or defined in intrinsic modules are simple. So are many intrinsic subroutines.

A user-written procedure declared simple in its prefix:

```
real simple elemental function convert(a)
```

is required to satisfy checkable restrictions.

Enumeration types

```
enumeration type :: colour
  enumerator :: red, orange, green
end type
type(colour) light, dark
:
if (light==red) dark = colour(3)
```

Indexing and comparison (<, <=, >, >=, ==,\=) are by position in the declaration.

The only values are the enumerators of the type.

An enumeration type is not a derived type but behaves like a derived type with no components.

Enumeration types (cont)

Can be used in `select case` construct:

```
select case (light)
  case (red)
  :
  case (orange:green)
  :
end select
```

Intrinsic `int` returns position in declaration and intrinsics `next` and `previous` added.

In formatted i/o, treated as an integer. Not allowed in list-directed and namelist i/o.

Enum types

In Fortran 2018, an enumeration is an ordered collection of integer constants of a kind that interoperates with C. To get some safety, enum type has been added:

```
enum, bind(c) :: season
  enumerator :: spring=5, summer=7, &
                autumn, winter
end type
type(season) my_season, your_season
```

Values other than the enumerators can occur.

Intrinsic `int` returns the value.

Indexing and comparison (`<`, `<=`, `>`, `>=`, `==`, `\=`) are by value in the declaration.

Enum types (cont)

Numeric operations between enums and integers not permitted but comparisons are.

```
if(my_season==7)your_season=season(9)
```

An integer expression is said to `type-conform' with an enum type if it contains a primary that is an enumerator of the type.

In an assignment statement with a left-hand side of enum type, the right-hand side must conform with it.

Can be used in `select case` construct.

Using integer arrays for subscripts

A multiple subscript specifies a sequence of subscripts by *@ int-expr* where *int-exp is* a rank-one integer expression, e.g.

```
a(@[3,5])       ! a(3,5)
a(6,@[3,5],1)   ! a(6,3,5,1)
```

A multiple subscript triplet specifies a sequence of section subscripts by *@ [int-expr] : [int-expr] [: int-expr]* where each *int-expr* is a rank-one array or scalar, e.g.

```
a(@[3,5]:[9,10]:[2,3]) ! a(3:9:2, 5:10:3)
a(@[3,5]:[9,10]:2)     ! a(3:9:2, 5:10:2)
a(@[3,5]:[9,10])       ! a(3:9, 5:10)
```

Using integer arrays for ranks and bounds

Rank and lower bounds of an assumed-shape array:

```
real :: zz(lb_array+2:)
```

Rank and bounds of an explicit-shape array:

```
real :: zz(lb_array+2:n), x(ub_array)
real :: y(0:ub_array)
```

Bounds of an allocatable array:

```
allocate(x(:upper), y(lower:upper), &
         z(0:upper))
```

Bounds in a pointer remapping

```
y(lower:)  => x
z(0:upper) => x
```

`system_clock([count, & count_rate,count_max])`

In Fortran 2018, arguments can be integers of any kind (to allow long integers). Vendors inconsistent for non-long and differing kinds of arguments .

All integer arguments in a single call will have the same kind and range at least default integer.

Support of long integers recommended.

There may be any number of clocks, including zero. Which is referenced depends on the kind.

Whether an image has no clock, has one or more clocks of its own, or shares a clock with another image, is processor dependent.

More values of intrinsic kinds

Additional named constants in the module `iso_fortran_env` for kinds of intrinsic type:

`logical8` 8-bit logical

`logical16` 16-bit logical

`logical32` 32-bit logical

`logical64` 64-bit logical

`real16` 16-bit real

-2 does not support this but supports a larger size

-1 does not support this or a larger size.

Intrinsic procedure `c_f_pointer`

To bring `c_f_pointer` into line with pointer assignment, an extra optional argument `lower` has been added to specify the lower bounds of the pointer result.

More use of boz constants

Binary, octal, and hexadecimal (boz) constants will be allowed in

- an initialization of a named object of type integer or real,
- as the right-hand side of an intrinsic assignment to a variable is of type integer or real,
- as a value in an integer or real array constructor, or
- as an integer value in an `enum` constructor.

Example:

```
I = z'a51f'
```

Trig functions that work in degrees

`sind(x)` returns the sine function for real values of `x` in degrees.

Similarly for `cosd(x)` and `tand(x)`

`asind(x)` returns inverse sine function in degrees.

Similarly for `acosd(x)`, `atand(x)`, `atand(y,x)`, and `atan2d(y,x)`.

Trig functions that work in half revs

`sindpi(x)` returns the sine function for real values of `x` in half revolutions (180 degrees).

Similarly for `cospi(x)` and `tanpi(x)`

`asinpi(x)` returns inverse sine function in half revolutions .

Similarly for `acospi(x)`, `atanpi(x)`, `atanpi(y,x)`, and `atan2pi(y,x)`.

Select kind for logicals

`selected_logical_kind(bits)`

returns the kind value for a logical type whose storage size in bits is at least `bits`, or `-1` if no such type is available.

Obsolete and deleted features

No more features have been added to the lists of obsolete and deleted features.

Summary

We have discussed

- Lengths of lines and character variables
- Conditional expressions and arguments
- Arrays with coarray components
- Put with notify
- Reduction in `do concurrent`
- Simple procedures
- `enumeration` and `enum` types
- Using arrays for subscripts and bounds
- System clock clarifications
- Some smaller changes