Fortran 2003:
the latest Fortran Standard

John Reid
JKR Associates
Convener, ISO Fortran Committee WG5
jkr@rl.ac.uk

Computer Languages for Scientific Computing
Institute of Physics, London
22 April 2005

ftp://ftp.numerical.rl.ac.uk/pub/jkr/f2003.pdf

The new Standard

The new Standard was published in November 2004.

Content was decided by WG5 (ISO) in 1997; considered all the requirements of users, expressed via national bodies.

Draft is available via the web as N1601 in

`ftp://ftp.nag.co.uk/sc22wg5/`

and I have written a summary as N1579.

Also, see:

Fortran 95/2003 explained,
Metcalf, Reid and Cohen, OUP, 2004.

Summary of Fortran 2003

Fortran 2003 is an upward-compatible extension of Fortran 95 with these new features:

- Exception handling (TR)
- Allocatable components and dummy arguments (TR)
- Interoperability with C
- Object-orientation: procedure pointers and structure components, structure finalization, type extension and inheritance, polymorphism
- Many minor enhancements

Important extension

- Enhanced module facilities (TR)

1 Exceptions (TR)

1.1 Requirements for handling exceptions

- Access IEEE conditions on IEEE hardware
- Support other aspects of IEEE
- Recognize partial support and provide enquiries
- Provide control on the degree of support
- Allow partial support on non-IEEE hardware

We found that it was impossible to do all this with a procedure library or a non-intrinsic module.

With an intrinsic module, we can make the USE statement control the compiler’s action.

1.2 Intrinsic modules

IEEE_EXCEPTIONS supports exceptions – at least overflow and divide-by-zero.

IEEE_ARITHMETIC supports other IEEE features. It behaves as if it has a USE statement for IEEE_EXCEPTIONS.

IEEE_FEATURES provides control over the features needed.

Example:

```
USE, INTRINSIC :: IEEE_EXCEPTIONS
USE, INTRINSIC :: IEEE_FEATURES, &
                  ONLY:IEEE_INVALID_FLAG
```

1.3 A few of the procedures

Inquiry:

- `IEEE_SUPPORT_INF([X])`
- `IEEE_SUPPORT_NAN([X])`

Elemental functions:

- `IEEE_IS_NAN(X)`

Elemental subroutines:

- `IEEE_GET_FLAG(FLAG,FLAG_VALUE)`
- `IEEE_SET_FLAG(FLAG,FLAG_VALUE)`

Non-elemental subroutines:

- `IEEE_GET_ROUNDING_MODE (ROUND_VALUE)`
- `IEEE_SET_ROUNDING_MODE (ROUND_VALUE)`

2. Allocatable array extensions (TR)

Just too late for inclusion in Fortran 95, we realized that allocatable arrays have significant advantages over pointer arrays:

- Efficiency: always contiguous in memory
- No memory leaks

Fortran 2003 allows

- Allocatable dummy arguments
- Allocatable function results
- Allocatable components of structures

Functionality provided in Fortran 90 by pointers.

3. Interoperating with C

Any entity involved in interoperating with C must be such that equivalent declarations of it may be made in the two languages.

Enforced within the Fortran program by requiring all such entities to be **interoperable.**

We will explain in turn what this requires for types, variables, and procedures.

They are all requirements on the syntax so that the compiler knows at compile time whether an entity is interoperable.

3.1 Interoperability of intrinsic types

Intrinsic module `ISO_C_BINDING` contains named constants holding kind type parameter values.

For example:

`C_INT`	`int`
`C_SHORT`	`short int`
`C_LONG`	`long int`
`C_FLOAT`	`float`
`C_DOUBLE`	`double`
`C_LONG_DOUBLE`	`long double`
`C_FLOAT_COMPLEX`	`float _Complex`
`C_BOOL`	`_Bool`
`C_CHAR`	`char`

Lack of support is indicated with a negative value.

3.2 Interoperability of derived types

For a derived type to be interoperable, it must be given the `BIND` attribute explicitly:

```
TYPE, BIND(C) :: MYTYPE
  :
END TYPE MYTYPE
```

Each component must have interoperable type and type parameters, must not be a pointer, and must not be allocatable. This allows Fortran and C types to correspond.

3.3 Interoperability of variables

A scalar Fortran variable is interoperable if it is of interoperable type and type parameters, and is neither a pointer nor allocatable.

An array Fortran variable is interoperable if it is of interoperable type and type parameters, and is of explicit shape or assumed size. It interoperates with a C array of the same type, type parameters and shape, but with reversal of subscripts.

For example, a Fortran array declared as

```
INTEGER :: A(18, 3:7, *)
```

is interoperable with a C array declared as

```
int b[][5][18]
```

3.4 Interoperability with C pointers

For interoperating with C pointers (addresses), the module contains a derived type `C_PTR` that is interoperable with any C pointer type and a named constant `C_NULL_PTR`.

The module also contains the procedures:

`C_LOC(X)` returns the C address of `X`.

`C_ASSOCIATED (C_PTR1[, C_PTR2])` is an inquiry function that is like `ASSOCIATED`.

`C_F_POINTER (CPTR, FPTR [, SHAPE]))` is a subroutine that constructs a Fortran pointer from a scalar of type `C_PTR`.

3.5 Interoperability of procedures

A new attribute, `VALUE`, has been introduced for scalar dummy arguments. Does copy-in without copy-out.

A Fortran procedure is interoperable if it has an explicit interface and is declared with the `BIND` attribute:

```
FUNCTION FUNC(I, J, K, L, M), BIND(C)
```

All the dummy arguments must be interoperable.

For a function, the result must be scalar and interoperable.

3.6 Binding labels

The Fortran procedure has a ‘binding label’, which has global scope and is the name by which it is known to the C processor.

By default, it is the lower-case version of the Fortran name.

An alternative binding label may be specified:

```
FUNCTION FUNC(I, J, K, L, M), &
         BIND(C, NAME='C_Func')
```

4 Object orientation

4.1 Procedure pointers

A pointer or pointer component may be a procedure:

```
PROCEDURE(proc),POINTER :: p => NULL()
            ! Has the interface of proc
PROCEDURE(),POINTER :: q
            ! Implicit interface
       :
 p => fun
```

Association with target is as for a dummy procedure.

4.2 Procedures bound by name to a type

Like a procedure component with a fixed target:

```
TYPE T
   : ! Component declarations
CONTAINS
   PROCEDURE :: proc => my_proc
   PROCEDURE :: proc2
END TYPE T
TYPE(T) :: A
     :
CALL a%proc(x,y)
```

4.3 Procedures bound to a type as operators

A procedure may be bound to a type as an operator or a defined assignment.

Accessible wherever an object of the type is accessible.

4.4 Type extension

A derived type may be extended:

```
TYPE :: matrix(kind,n)
   INTEGER, KIND :: kind
   INTEGER, NONKIND :: n
   REAL(kind) :: element(n,n)
END TYPE

TYPE,EXTENDS(matrix)::factored_matrix
 LOGICAL :: factored=.FALSE.
 REAL(matrix%kind) :: &
            factors(matrix%n,matrix%n)
END TYPE
```

All the type parameters, components, and bound procedures of the parent type are inherited by the extended type and they are known by the same names.

4.5 Polymorphic entities

A polymorphic entity

```
CLASS (matrix(kind(0.0),10)) :: f
```

has a varying **dynamic** type that is the declared type or an extension of it.

Allows code to be written for objects of a given type and used later for objects of any extension.

4.6 SELECT TYPE construct

Access to the extended parts is available thus:

```
SELECT TYPE (f)
   TYPE IS (matrix)
      : ! Block of statements
   CLASS IS (factored_matrix)
      : ! Block of statements
   CLASS DEFAULT
END SELECT
```

4.7 Unlimited polymorphic

An object may be declared as **unlimited polymorphic**

```
CLASS (*) :: upoly
```

so that any extensible type extends it.

Its declared type is regarded as different from that of any other entity.

5 Some minor enhancements

5.1 Parameterized derived types

Allowed any number of ‘kind’ and ‘length’ parameters. E. g.

```
TYPE matrix(kind,m,n)
   INTEGER, KIND :: kind
   INTEGER, LEN :: m,n
   REAL(kind) :: element(m,n)
END TYPE
     :
TYPE(matrix(KIND(0.0D0),10,20)) :: a
WRITE(*,*) a%kind, a%m, a%n
```

5.2 Access to the computing environment

New intrinsic procedures:

`COMMAND_ARGUMENT_COUNT ( )` is a function that returns the number of command arguments.

`GET_COMMAND` is a subroutine that returns the entire command.

`GET_COMMAND_ARGUMENT` is a subroutine that returns a command argument.

`GET_ENVIRONMENT_VARIABLE` is a subroutine that returns an environment variable.

5.3 Support for international character sets

Fortran 90/95 allows multi-byte character sets. A new intrinsic function has been introduced:

`SELECTED_CHAR_KIND(NAME)` returns a kind value when `NAME` has one of the values `DEFAULT`, `ASCII`, and `ISO_10646`.

Default or ASCII character data may be assigned to ISO 10646 character variables.

There is a standardized method (UTF-8) of representing 4-byte characters as strings of 1-byte characters in a file. Supported by `ENCODING='UTF-8'` on the `OPEN` statement.

Lengths of names and statements

Names of length up to 63 characters and statements of up to 256 lines are allowed.

Binary, octal and hex constants

Permitted as a principal argument in a call of the intrinsic function `INT`, `REAL`, `CMPLX`, or `DBLE`:

```
INT(O'345'),  REAL(Z'1234ABCD')
```

For `INT`, the ‘boz’ constant is treated as if it were an integer constant.

For the others, treated as having the value that a variable of the type and kind would have if its value was the bit pattern specified.

5.4 Derived type input/output

It may be arranged that when a derived-type object is encountered in an input/output list, a Fortran subroutine of the form

```
SUBROUTINE formatted_io  (dtv,unit,&
         iotype,v_list,iostat,iomsg)
SUBROUTINE unformatted_io(dtv,unit,&
                        iostat,iomsg)
```

is called.

For formatted input/output, the `DT` edit descriptor passes a character string and an integer array to control the action. An example is

```
DT 'linked-list' (10, -4, 2)
```

5.5 Asynchronous input/output

Input/output may be asynchronous. Requires `ASYNCHRONOUS='YES'` in the `OPEN` statement for the file and in the `READ` or `WRITE` statement.

Initiates a ‘pending’ input/output operation, terminated by a wait operation for the file:

```
WAIT(10)
```

or implicitly by an `INQUIRE`, a `CLOSE`, or a file positioning statement for the file.

5.6 Stream access input/output

`ACCESS='STREAM'` on the `OPEN` statement.

May be formatted or unformatted.

The file is positioned by ‘file storage units’, normally bytes.

The current position may be determined from a `POS=` specifier of an `INQUIRE` statement for the unit.

A required position may be indicated in a `READ` or `WRITE` statement by a `POS=` specifier.

6 Enhanced module facilities (TR)

If a huge module is split into several modules:

- Internal parts exposed
- Any change leads to compilation cascade

Solution:

- Submodules containing definitions of procedures whose interfaces are in the module itself
- Users continue to access the public parts of the module
- Submodules have full access by host association

7 Conclusions

Have attempted to give you a overview of a major revision.

Designed to preserve

- huge investment in existing codes
- relative ease for writing codes that run fast
- strength for processing of arrays.

Therefore conservative approach for object orientation and interfacing with C.

Which of the new features excites you most is a personal matter. Certainly, there is something for you – Fortran will be a more powerful language.