Revisiting OOP Syntax

by Craig T. Dedo
October 25, 1998

I believe that it would be helpful for J3 to revisit the issue of syntax for OOP in Fortran 2000, for the reasons that Dr. Werner W. Schulz explains in his e-mail, which is quoted in full below.

I believe that one of the strengths of the Fortran language is that it has a relatively easy and straightforward grammar and syntax, especially compared with most of the other languages whcih are popular right now. The current syntax that is proposed for OOP could be made much easier for application developers to use.

Please think over the issues in this message which Dr. Schulz sent to comp-fortran-90 mailing list and give them your careful and thoughtful consideration.

[Begin e-mail from Dr. Werner W. Schulz]
Date: Mon, 10 Aug 1998 17:09:57 +0100 (BST)
Message-ID: <Pine.SOL.3.96.980810160414.14987F-100000@taurus.cus.cam.ac.uk>
MIME-Version: 1.0
Content-Type: TEXT/PLAIN; charset=US-ASCII
Subject: OOP in Fortran 2000
From: "Dr W.W. Schulz" <wws20@cus.cam.ac.uk>
To: fortran90 mailing list <comp-fortran-90@mailbase.ac.uk>
X-List: comp-fortran-90@mailbase.ac.uk
X-Unsub: To leave, send text 'leave comp-fortran-90' to mailbase@mailbase.ac.uk
X-List-Unsubscribe: <mailto:mailbase@mailbase.ac.uk?body=leave%20comp-fortran-90>
Reply-To: "Dr W.W. Schulz" <wws20@cus.cam.ac.uk>
Sender: comp-fortran-90-request@mailbase.ac.uk
Errors-To: comp-fortran-90-request@mailbase.ac.uk
Precedence: list
X-UIDL: d3e7f741ac3f7b463eccb58127b2ce74
X-Mozilla-Status: 9003

There have been several comments in comp.lang.fortran (clf) about the lack of generic classes and functions in the F2000 proposal. That is a heavy setback.

But the OOP proposal itself contains several problematic or far from ideal constructs.

The various documents of J3 are available at ftp://ftp.ncsa.uiuc.edu/x3j3/doc/year/.
9x-000.txt is a list of all submitted papers, minutes, current draft, etc.
for a particular year. Some relevant papers for OOP are
year 98: 152, 140, 137, 136, 133, 108, 100
year 97: 230, 196, 195, 194, 183, 182 (and earlier)
Please read the full papers for more details.

Let me quote from 98-152r1.txt since it is the latest and should be close to the current discussion in J3. (Fortran Words are always in UPPER case, otherwise the same word may be used in a different sense.)

EXAMPLES from 98-152:

a) Classes are constrcuted from TYPE with the new attributes
EXTENSIBLE or EXTENDS(parent-TYPE):

```
    TYPE,EXTENSIBLE :: vector_2d
        REAL x,y
    CONTAINS
        PROCEDURE,PASS_OBJ :: length => length_2d
    END TYPE

    REAL FUNCTION length_2d(v)
        CLASS(vector_2d) v
        length_2d = SQRT(v%x**2+v%y**2)
    END FUNCTION


    TYPE,EXTENDS(vector_2d) :: vector_3d
        REAL z
    CONTAINS
        PROCEDURE,PASS_OBJ :: length => length_3d
    END TYPE

    REAL FUNCTION length_3d(self)
        CLASS(vector_3d) self
        length_3d = SQRT(self%x**2+self%y**2+self%z**2)
    END FUNCTION
```

Usually one would put these TYPEs and the corresponding type-bound procedures
into one or several modules, so a complete vector_3d would look like this:

```
    MODULE vector_3d_mod

      USE vector_2d_mod
      IMPLICIT NONE

      TYPE,EXTENDS(vector_2d) :: vector_3d
         REAL z
      CONTAINS
         PROCEDURE,PASS_OBJ :: length   => length_3d
         PROCEDURE,PASS_OBJ :: distance => distance_3d
      END TYPE

      PRIVATE :: length_3d, distance_3d

    CONTAINS

      REAL FUNCTION length_3d(self)
         CLASS(vector_3d), INTENT(IN) :: self
         length_3d = SQRT(self%x**2+self%y**2+self%z**2)
      END FUNCTION
```

```
      REAL FUNCTION distance_3d(self,p)
         CLASS(vector_3d), INTENT(IN) :: self
         TYPE(vector_3d),  INTENT(IN) :: p
         distance_3d = SQRT((self%x-p%x)**2+(self%y-p%x)**2+(self%z-p%x)**2)
      END FUNCTION

   END MODULE vector_3d_mod
```

b) Invocation of these constructs:

```
   TYPE(vector_2d) vec
   TYPE(vector_3d) x
   REAL size
   ...
   size = vec%length()    ! Invokes length_2d(vec).
   size = x%length()      ! Invokes length_3d(x).
```

Note the correspondence between PASS_OBJ in the TYPE declaration and invoking object as in vec%length().
So far these have been monomorphic objects (vec,x).

c) Polymorphism is enabled by this construct:

```
   CLASS(vector_2d), POINTER :: y
   y => vec
   y => x
```

There is also a non-pointer CLASS, but that is allowed only as a scalar dummy argument in procedures. It is necessary in connection with PASS_OBJ. Note that CLASS is different from class in Java,C++, Eiffel, and similar OOP languages and OOP literature but rather more like Ada95's OOP version.

d) PROCEDUREs can 'point' to NULL(procname), i.e. they represent abstract or deferred procedures whose interface is defined by procname (see proposal for procedure pointers for details).

e) Visibility: PRIVATE can be added to PROCEDURE as an attribute.

f) Overriding of procedure characteristics:

"When overriding a type-bound procedure without the PASS_OBJ attribute, all characteristics of the overriding procedure shall be the same as that of the procedure being overridden."

"When overriding a type-bound procedure with the PASS_OBJ attribute, only the characteristics of the dummy argument used for passing the invoking object shall be different."

i.e. the dummy arguments have to have the same type declaration as in the original extensible TYPE.

(Question: Is there ever a reasonable type-bound procedure which does not require the invoking object to be passed?)

CRITICISM of the proposed Syntax:

a) Next to TYPE(type-name)

```
TYPE(type-name), POINTER
```

there are also:

```
TYPE(type-name), EXTENSIBLE/EXTENDS(parent-type)
TYPE(type-name), POINTER     ! same as before but type-name is extensible
```

and

```
CLASS(base-type)
CLASS(base-type), POINTER
```

Note that extensible TYPEs share the rules for pure TYPEs but are otherwise separate though this is not obvious in variable declarations.

The problem construct is the non-pointer CLASS construct since it is not safe from run-time errors (the same problem appears in Ada95).
Example:

```
CLASS(vector_2d) :: vc2      ! dummy argument in some procedure
TYPE(vector_2d)  :: t2       ! obvious versions from above
TYPE(vector_3d)  :: t3
TYPE(vector_4d)  :: t4
```

If the actual run-time of vc2 is actually a vector_3d then the following happens:

```
vc2 = t2    ! run-time error, not enough fields to assign
vc2 = t3    ! ok.
vc2 = t4    ! uses first three fields and skips fourth
```

The statements are legal but not run-time safe.

Currently the non-pointer CLASS construct is needed since J3 doesn't want to introduce a SELF construct which would be safe (see below).

I personally don't like the names either since they are in conflict with common usage in OOP literature notwithstanding the fact that Ada95, Modula-3 etc use TYPE.
(There is also the awkwardness that a sub-TYPE of a parent-TYPE is not necessarily a subtype of the parenttype. I can give references and examples for anyone interested in this subtlety. It is better to separate class and type instead of mixing TYPE and type as in Fortran. It wasn't so bad with F90's TYPEs but under OOP it does become an issue.)

b) extensible TYPEs with deferred procedures are not specially marked or limited. This can lead to run-time errors if, for example, vector_2d and vector_3d have a concrete LENGTH function while the programmer decided to defer (again) the LENGTH function on -say- vector_4d. If one now invoke the LENGTH function of a polymorphic object of base vector_1d which happens to a vector_4d, a run-time error occurs.

c) Asymmetry of procedure argument list. The PASS_OBJ attribute requires a dummy non-pointer CLASS argument which is not present in the invocation statement.

d) The PROCEDURE declaration is not transparent. Neither does it reveal whether a FUNCTION or a SUBROUTINE is meant nor is the interface (argument list) directly visible (except for NULL(procname)). Without these features the PROCEDURE declaration is largely useless and should be scrapped.
(Similar criticism applies to the procedure pointer construct.)

The bodies of the PROCEDUREs are kept separately elsewhere, most commonly in the same module. I view this separation as awkward for two reasons: firstly, constructs that belong together should stay together in one linguistic unit, secondly, the separation requires (for all practical purposes) a module hierarchy that follows that of the extensible TYPE hierarchy, an unnecessary doubling of names, etc. is a consequence.

The PROCEDURE declaration can take attributes but FUNCTIONs and SUBROUTINEs cannot (why not?). (Adding attributes to FUNCTIONs and SUBROUTINEs would help to make Fortran syntax more regular in any case. Currently procedures must be declared PUBLIC/PRIVATE in separate attribute statements which cause a number of limitations in Fortran syntax.)

e) There seems to be a general tendency in J3 to miss more appropriate names:

EXTENSIBLE/EXTENDS is used when everyone in OOP talks of inheritance (so why not use INHERIT <class> similar to USE <module>?). Inheritance is not always extension, sometimes only a redefinition.
The attribute syntax is also not very suitable for multiple inheritance should that be added in the future. A separate statement INHERIT <class> is better suited.

NON_OVERRIDABLE (15 chars!) is an attribute to PROCEDURE to prevent redefinition of procedures later on. FINAL seems to be a good word, too, and it is ten characters shorter (FROZEN didn't get a majority vote!). The longest words in FORTRAN95 so far are ALLOCATABLE, EQUIVALENCE and UNFORMATTED with 11 chars each.

Arguments against TYPE (monomorphic use) and CLASS (polymorphic use) I have already noted.

What are the ALTERNATIVES?
Let me propose a different Fortranese:

a) Classes:

```
CLASS :: vector_2d       ! Attention: CLASS is different from above
  SELF :: me             ! here: me is of CLASS vector_2d
  REAL :: x, y
  !CONTAINS necessary?

  FUNCTION length()
    length = SQRT( x**2 +y**2 )
  END FUNCTION length

  FUNCTION distance( p )
```

```
      LIKE(me), INTENT(IN) :: p
      distance = SQRT(  (x-p%x)**2 +(y-p%y)**2 )
    END FUNCTION distance
  END CLASS vector_2d

  CLASS :: vector_3d
    INHERIT  :: vector_2d          ! me, x, y are taken over from vector_2d
    REDEFINE :: length, distance   ! but me now means a vector_3d class

    REAL :: z

    FUNCTION length()
      length = SQRT( x**2 +y**2 +z**2 )
    END FUNCTION length

    FUNCTION distance( p )
      LIKE(me), INTENT(IN) :: p    ! p is now vector_3d, not _2d
      distance = SQRT(  (x-p%x)**2 +(y-p%y)**2 +(z-p%z)**2 )
    END FUNCTION distance

  END CLASS vector_3d
```

The SELF construct allows a dynamic type change under inheritance.
LIKE(me) is also a dynamic type declaration and always changes in
line with the actual type of the current object.

Invocation is the same as the J3 proposal but note that the asymmetry in
the argument list is gone since 'SELF :: me' always stands in for the
invoking object:

```
    CLASS(vector_2d) :: vec
    size = vec%length()      ! one could even skip the parentheses
                             ! no one needs to know whether length is a
                             ! variable or a function
```

'me' inside the CLASS definition refers to the current object 'vec'.

Some restrictions are that class procedure names cannot be used as actual arguments
to dummy procedure arguments (obviously) and class procedures should not contain saved
local variables (ex- or implicitly).

b) Polymorphic objects:

```
    REF(vector_2d) :: poly_vec
```

REF always has the (implicit) POINTER attribute and only pointer assignment is
allowed (=>) but not assignment (=).
The variable poly_vec can point to any variable that inherits from
the ancestral class incl. this class itself (nothing new).

Class procedures with arguments declared with LIKE cannot be invoked
from polymorphic objects since this could result in run-time errors.

(This is an example of a sub-CLASS derived from a parent-CLASS not being a sub-type of the parent-type, see distance function in vector_2d and vector_3d.)

By the way, there are now three different versions possible for the distance function of the vector_nd classes. Here are some examples:

- LIKE(me): the most obvious choice since usually I want to compare two vectors of the same type
- REF(vector_1d): can only compute the distance of projection on x-axis of the invoking vector and any other one or more dimensional vector (maybe one should call this x_distance)
- CLASS(vector_2d): requires exactly a two-dimensional vector. (doesn't look very useful in this context, but maybe elsewhere)

c) Abstract classes:

```
CLASS, ABSTRACT :: abstract_vector
  SELF :: me

  FUNCTION, ABSTRACT :: length()
  END FUNCTION length

  FUNCTION, ABSTRACT :: distance(p)
    LIKE(me), INTENT(in) :: p
  END FUNCTION length

END CLASS abstract_vector

REF(abstract_vector)   :: av    ! is legal
CLASS(abstract_vector) :: bv    ! is illegal
```

Any class with at least one abstract procedure (or inherited) must be declared abstract as well. Only polymorphic objects can be declared with an abstract base class, but not monomorphic classes. Since polymorphic objects eventually must refer to a monomorphic object this presents no problem.
Once a prcoedure is made concrete (by redefining it upon inheritance) it cannot be redefined to ABSTRACT since this would lead to run-time errors under polymorphism.

d) CLASSes should be compilation units like MODULE, FUNCTION, etc. It is not necessary to encapsulate them inside MODULEs but it is allowed.
The class procedures interfaces are known and must be checked at compile time (like module procedures; the CLASS/REF declaration acts in a similar way to the USE module declaration).

e) Extra features:

-READONLY:
I would like to see that the class variables (x,y in vector_2d) are by default declared READONLY (as in Eiffel). They can also be PRIVATE but never PUBLIC.

Reason: Objects have a state (the variables) and behaviour(procedures). The language should ensure that objects are always in a consistent state. This is not possible with PUBLIC variables, esp. in large programming projects. Example: On top of the vector_2d version a unit_vector_2d CLASS is added by inheritance (i.e. x**2 +y**2 = 1.0 at all times is required). This is impossible to maintain if the x and y variables are accessible directly (polymorphism is the culprit).

The lack of READONLY currently requires to make all variables PRIVATE if one wants to impose some protection of objects with the added work of writing the trivial set_x, set_y, etc subroutines. READONLY would be the equivalent to INTENT(IN) in procedures and be at least as useful.

Polymorphism requires that there is only a one-way direction of redefining attributes: from PRIVATE to READONLY but not vice versa (to PUBLIC for procedures). (One can also allow FUNCTION to variable redefinition if one can drop the parentheses for argumentless class functions.)

-ALLOCATE:
I would like to see an enhanced ALLOCATE version so that one can point polymorphic objects to unnamed monomorphic objects at run-time, similar to the new construct in other languages.

-GENERIC CLASSES:
A possible syntax could be:

```
CLASS, GENERIC :: array(T)
  T, dimension(:), allocatable :: A
  ! plus many procedures
  SUBROUTINE set( i, value )
    INTEGER, INTENT(IN) :: I
    T, INTENT(IN)      :: value
    A(i) = value
  END SUBROUTINE set
END CLASS array

CLASS( array(REAL) ) :: x
```

I think it is very lamentable that the Fortran committees could not put this into the F2000 plan. We will have to wait until 2008 (TEN YEARS!) to get anything like it. This is unacceptable; the competition is not sleeping but far ahead already.

-PROCEDURE INTERFACE CHANGES:
Upon inheritance arguments can be changed in a covariant fashion. This is often required in real applications. Inheritance usually means specialisation and this in turn requires procedures with more specialized arguments. However, covariance is at odds with polymorphism which would require to exclude such procedures from use by polymorphic objects to avoid run-time errors.

-The F90 TYPE construct:
The F90 TYPE construct should be left to dissipate slowly since it is not really needed. Only CLASS and CLASS, POINTER and REF should be kept.

The proposed alternative syntax and semantics avoid run-time errors and emphasize type-safety, efficiency and clarity.
It is a little more restrictive than other OOP languages but not by much, while safer than C++, for example, though not yet as powerful. Added power will come from generic classes and procedures which should be included asap.

This bare bone version of OOP in Fortran is a more consistent and -in my view- more viable and elegant version since it embodies more of the underlying ideas of OOP and not just the techniques.

Like to hear your comments.

Cheers,
WWS

| Werner W Schulz |
| Dept of Chemistry email: wws20@cam.ac.uk |
| University of Cambridge Phone: (+44) (0)1223 336 502 |
| Lensfield Road Secretary: 1223 336 338 |
| Cambridge CB2 1EW Fax: 1223 336 536 |
| United Kingdom WWW: |

[End of e-mail from Dr. Werner W. Schulz]

[End of J3 / 98-216]