Effective Object-Orientation: Discussion Paper

Type Extension

- extend types by adding new components
- base type component layout unchanged in extended type
- extensible type definitions marked by new keyword CLASS

Example Syntax:

```
CLASS name
        ... 0 or more components ...
END CLASS

CLASS newname, EXTENDS name
        ... 0 or more additional components ...
END CLASS
```

- components in the base class visible in the extended class
- the type name of the base class visible in the extended class as the aggregate of its members
- variables of an extensible/extended class declared normally (with the TYPE keyword)

Example Syntax:

```
CLASS POINT
        REAL X,Y,Z
END CLASS
CLASS SPACETIME_POINT, EXTENDS POINT
        REAL TIME
END CLASS
...
TYPE(POINT) W
! W contains only POINT components (W%X, W%Y, W%Z)
TYPE(SPACETIME_POINT) V
! V%X is the X coordinate
! V%POINT is of TYPE(POINT), contains V%X, V%Y, V%Z
! V%TIME is the TIME coordinate
```

Note: W and V are just ordinary variables, there is no additional overhead - they are just like variables of an ordinary derived type.

Polymorphic Variables

- refer to an object of any type extended from a particular base type (or the type itself)
- direct access only to components of the base type

Example Syntax:

```
CLASS(POINT) A
TYPE(POINT) B
! B is only of TYPE(POINT)
! A can be of TYPE(POINT) or any type extended from POINT.

B = A
! If A happens to be of an extended type (e.g. SPACETIME_POINT) at runtime,
! this is still ok, with the POINT components being assigned normally.
```

- intrinsic function for runtime type enquiry: whether two objects are of exactly the same type (the signature to support this can be generated automatically from the CLASS definitions). Typically one of the objects will be of a specific TYPE().

Example Syntax:

```
CLASS(POINT) VAR
TYPE(SPACETIME_POINT) SPT
... MATCHING_TYPE(VAR,SPT) ... has value .TRUE. iff VAR is a
                                        SPACETIME_POINT.
```

- when passed to a TYPE(...) dummy, must either match the dummy's type or be of an extended type, in which case only the TYPE(...) part is effectively passed

Further Example:

```
CLASS(POINT) VAR
CLASS(SPACETIME_POINT) V2
INTERFACE
        SUBROUTINE SUB1(A)
                USE POINT_MODULE
                TYPE(POINT) A
        END SUBROUTINE
        SUBROUTINE SUB2(B)
                USE SPACETIME_POINT_MODULE
                TYPE(SPACETIME_POINT) B
        END SUBROUTINE
END INTERFACE
...
CALL SUB1(V2) ! SUB1 expects a TYPE(POINT), so V2%POINT is passed
CALL SUB2(VAR) ! SUB2 expects a TYPE(SPACETIME_POINT), so it is an
                ! error if VAR is not a SPACETIME_POINT (or of a type
                ! extended from SPACETIME_POINT)
```

Dummy Arguments

- assume the type from the actual argument
- accept the base type or any type extended from it

Example Syntax:

```
SUBROUTINE SUB(A)
        CLASS(POINT) A
        TYPE(SPACETIME_POINT) B
        INTRINSIC MATCHING_TYPE
        IF (MATCHING_TYPE(A,B)) THEN
                PRINT *,"A is a SPACETIME_POINT"
        END IF
END
```

Polymorphic Pointers

- can point to the base type or any type extended from it
- ALLOCATE only gives you an object of the base type
- however, DEALLOCATE operates on any acceptable object
- allows generic List/Graph/Tree/Stack/etc. code to be written

Example Syntax:

```
CLASS LIST
        CLASS(LIST),POINTER :: NEXT = NULL()
END CLASS
...
INTEGER FUNCTION CARDINAL(L)
        CLASS(LIST) L
        CLASS(LIST),POINTER :: P
        CARDINAL = 1
        P = L%NEXT
        DO WHILE (ASSOCIATED(P))
                P = P%NEXT
                CARDINAL = CARDINAL + 1
        END DO
END FUNCTION
...
CLASS INTLIST, EXTENDS LIST
        INTEGER VALUE
END CLASS
CLASS REALLIST, EXTENDS LIST
        REAL VALUE
END CLASS
```

Polymorphic Local Variables

These can be handled by pointers, but the necessary memory allocation and deallocation, although formulaic, is tedious, verbose and error-prone. Therefore allow it to be handled automatically with non-pointer locals. Note that the CLASS notation is effectively marking that these operations are more expensive.

- begin life undefined
- assume type on assignment
- can be passed to a dummy argument when undefined, in which case they become of the dummy's type (but still with undefined components).

Example Syntax:

```
SUBROUTINE SUB
        CLASS(POINT) TMP1,TMP2
        TMP1 = SPACETIME_POINT(1.0,1.0,1.0,0.0)
        !  TMP1 is created as a TYPE(SPACETIME_POINT)
        CALL INITIALISE_HYPERSPACE_POINT(TMP2)
        ! TMP2 is created with the type required by INITIALISE_...
END SUBROUTINE
```