To	: X3J3
From	: John Cuthbertson
Subject	: Minimal OO Features in Fortran
References	: 92-183, 96-114, 96-142, 96-149, and 96-172

Introduction

Fortran 90/95 already possesses many of the features that are commonly that facilitate Object Oriented Programming (that is allowing the realisation of an object model into actual program code). This paper aims to summarise those features and describe the features that Fortran lacks.

Specification of an Object-types and Objects

There seems to be a consensus within the Fortran community that if Fortran is to be made an object oriented language it should be a "class-based language." In OO terms, a class is a structural entity used as a template for defining objects that have the same structural components as the class. This is exactly what the existing Fortran 90 derived-type already provides. In fact extending the derived-type mechanism is seen by some as the most natural way of introducing object functionality in Fortran; others agree with this in principle, but would prefer alternative syntax. Example syntax alternatives:

```
TYPE name                                   CLASS name
  0 or more component specs                   0 or more component specs
END TYPE                                    END CLASS
```

The same controversy exists over the syntax of defining an actual object.

At least two pieces of extra semantics, associated with objects but not structures, have currently been identified and so a different syntax defining both objects and classes may be desirable, although for the purposes of this document the TYPE keyword will be used to define classes and objects.

Inheritance

One of the features of OOP where Fortran provides no language support is inheritance, where a new class is generated by extending an existing class. This is sometimes termed type extension or Example:

```
TYPE name
  0 or more component specifications
END TYPE

TYPE  newname, EXTENDS name
  0 or more additional component specifications
END TYPE
```

or:

```
TYPE  newname
  EXTENDS [::] name
  0 or more additional component specifications
END TYPE
```

The components that have been specified in *name* are implicitly specified in *newname* (that is they are inherited by *newname*). There is the obvious extension of this syntax to allow selective inheritance, where the programmer can select which components from *name* that *newname* will contain.

Note: this model describes "single-inheritance" (a class extends only one class, but can be extended by any number of classes), but can easily be amended to cope with "multiple-inheritance".

In an extended class (*newname*), the layout of the components of the base class (*name*) is unchanged; the name of the base class is also visible as a component of the extended class, where it represents a sub object of the base class. The following example from 96/149 should serve as an illustration:

```
TYPE POINT
  REAL::X,Y,Z
END TYPE

TYPE SPACETIME_POINT
  EXTENDS::POINT
  REAL::TIME
END TYPE

TYPE(POINT)::W
! W contains only the POINT components which are accessed using
! W%X, W%Y, W%Z

TYPE(SPACETIME_POINT)::V
! V contains the components: V%X, V%Y, V%Z, and V%TIME
! V also contains V%POINT
```

In my opinion, and I'm sure that there are people who agree with me, enhancing Fortran with some inheritance mechanism should be a primary goal of X3J3. Inheritance is one of the most widely used Object Oriented Programming features, and is the most difficult to simulate in Fortran. A good measure of OOP support would be provided if inheritance alone was added to Fortran.

Methods

Another item of semantics that is applicable to objects, but not structures, are methods: that is the association of procedures with specific types or classes. This functionality (albeit in a more flexible form) is offered by "procedure pointer components" that is part of the Procedure Pointers requirement that has already been approved for inclusion into Fortran 2000.

For a particular class, a method can be thought of as a component that is procedure pointer, but cannot be modified (overridden) except in a class that extends the current one. Example:

```
INTERFACE
  SUBROUTINE SUB(a,b)
    REAL A,B
  END SUBROUTINE
END INTERFACE

TYPE POINT
  REAL::X,Y,Z
  PROCEDURE,POINTER,PARAMETER::FRED => SUB
END TYPE
```

An alternative syntax (as specified in 92-183) could be:

```
 TYPE POINT
     REAL::X,Y,Z
 CONTAINS
   INTERFACE FRED
      SUBROUTINE SUB(a,b)
       ...
      END SUBROUTINE
    END INTERFACE
END TYPE
```

The above syntax is purely for illustration purposes only. Both examples mean the same thing and would probably be implemented in the same way.

Such constant component procedure pointers could be initialised using the NULL() intrinsic. In this case the method is said to be deferred and should be initialised with a proper procedure in an extended class.

A reference to a constant component procedure pointer is given as:

```
CALL X%FRED(<actual-argument-list>)
```

Or for a function reference:

```
 Z = Y%SUE(<actual-argument-list>)
```

Methods and Type Extension

If the "constant procedure pointers" model of methods is adopted, then a method should only be overridden (that is the procedure pointer component is initialised with the name of another procedure) when a class is extended to produce another class. When a method is being overridden, the component name and attributes in both the base type and the extended type must match; otherwise a new component is being defined.

Any, all, or none of the methods can be overridden when extending a type; additional methods can also be included in the extended type. In object oriented programming it is very common to extend a class just to override the methods. Example:

```
TYPE POINT
   REAL X, Y, Z
   POINTER, PARAMETER::METHOD1=>FRED
END TYPE

TYPE SPACETIME_POINT
   EXTENDS::POINT
   REAL TIME
   POINTER, PARAMETER::METHOD1=>SUE   ! Method 1 overridden
   POINTER, PARAMETER::METHOD2=>JACK  ! New method
END TYPE
```

A method need not be overridden, in which case the extended type has exactly the same methods and behaviour as the base type. If the second implementation strategy is adopted then intrinsic procedures such as the MATCH function, as described in 96-142, can easily be implemented.

<u>Deferred Methods</u>
A deferred method is a nullified procedure pointer component. Example:

```
TYPE POINT
  REAL::X,Y,Z
  POINTER,PARAMETER::METHOD1=>NULL()
END TYPE
```

The nullified procedure pointer component, METHOD1, of POINT should be initialised (overridden) in a class that extends POINT. Example:

```
TYPE SPACETIME_POINT
  EXTENDS::POINT
  REAL::TIME
  POINTER,PARAMTER::METHOD1=>FRED
END TYPE
```

As with a normal Fortran 90 disassociated pointer, a deferred method should not be referenced. Referencing a deferred method will probably result in a run-time error and usually indicates an error in the programmer's model. To offset this, the programmer could easily perform an ASSOCIATED test before the actual reference:

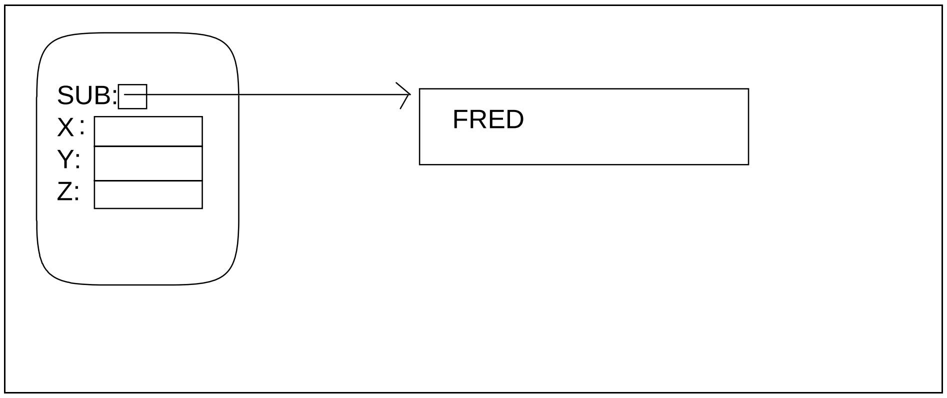
```
SUBROUTINE SUB(A)
  TYPE(POINT) :: A

  IF (ASSOCAIATED(A%METHOD1)) THEN
     CALL A%METHOD1()
  ENDIF
END SUBROUTINE
```

Alternatively the deferred method could be initialised with some programmer supplied default (that if called prints a message and halts) which in turn can be overridden.

Suggested Implementation Strategies for Methods

One of the major obstacles in getting approval for methods, and allowing them to be overridden in extended classes, is the execution overhead that would be involved. One way of implementing methods would be to actually store the value of the procedure pointer in the object.



Thus any reference of a method would at most involve a dereference. Because the methods are constants a processor, if it can resolve the reference at compile-time, would be free to substitute the component with the name of the actual procedure. Also the reference could be substituted for inline code. The disadvantage of this implementation strategy is that objects with a small number of data components, but a large number of associated methods, would be artificially large.

Another strategy would be to store a reference to a procedure table (some sort of structure that contains pointers to procedures) in the actual object. Objects of the same class/type would "point" to the same procedure table. Procedure references would involve two pointer dereferences, but determining whether two objects were of the same type would be trivial.

Conclusion

Fortran contains many of the features that make a programming language "object orientated", but it has no support for the functionality of inheritance. While it is possible to "simulate" inheritance with good modular design and code duplication, the result is often a piece of software that is larger than it really should be; it may also be the case that such software is harder to maintain that if it was developed using traditional software engineering techniques as change is more widespread.

Methods *can* be simulated in Fortran just now with good disciplined programming, the semantics of methods will also be superseded by "Procedure Pointers", and so they are of a lesser importance than inheritance.

Of all the so-called OO features, inheritance alone will provide a good measure of the support for OO programming that people want. Inheritance is the one key feature of object oriented programming that Fortran 2000 should provide.