Date: 4 February 1997
To: X3J3
From: Van Snyder
Subject: Comments on SC22/WG5/N1188: Class inheritance and dynamic binding polymorphism in Fortran 2000

I am pleased that the basic ideas for class inheritance and dynamic binding are based on Ada-95 rather than C++. The complication of "friends" and the upside-down nature of "virtual" are distasteful as compared with the simplicity of the relation between types and modules, and the type-safety of class-wide pointers.

A Quibble: I'd rather see

```
TYPE, INHERIT :: HUMAN
  CHARACTER(LEN=4) :: FIRST_NAME
END TYPE HUMAN
TYPE, INHERIT(HUMAN) :: MAN
  LOGICAL :: BEARDED = .FALSE.
END TYPE MAN
TYPE, INHERIT(HUMAN) :: WOMAN
END TYPE WOMAN

CLASS(HUMAN) :: OBJ ! or TYPE(HUMAN), CLASS :: OBJ
  ! in either case, OBJ is a "class-wide" object
TYPE(HUMAN)  :: OBJ1
TYPE(MAN)    :: OBJ2
TYPE(WOMAN)  :: OBJ3
```

instead of having an INHERIT statement in the body of the type. The reason is that INHERIT applies to the TYPE, not its components.

It seems that HUMAN(OBJ2) is simply a TRANSFER function, whereas MAN(OBJ1,.false.) is a derived type constructor. In either case, the result shouldn't have the POINTER attribute, except that if OBJ2 has the POINTER or TARGET attribute, and OBJ1 has the POINTER attribute, then OBJ1 => HUMAN(OBJ2) should be allowed, but OBJ2 =>MAN (OBJ1, .false.) should be prohibited, even if OBJ1 has the TARGET or POINTER attribute and OBJ2 has the pointer attribute – we don't allow I => J+3, even if J has the target or pointer attribute. If OBJ has the POINTER attribute, and OBJ2 has the TARGET or POINTER attribute

then OBJ =>OBJ2 is allowed. MAN(OBJ1) shouldn't be allowed (also see remarks below about auto-specialization).

It seems that a class-wide variable, especially an all-classes-wide variable, without the pointer attribute, might be a difficulty for the compiler. The compiler might not be able to know how much storage to allow for

```
CLASS() :: T3
```

without reading every module in sight, whereas it would know how much to allow for

```
CLASS(), POINTER :: T3
```

Even

```
MODULE ONE
  CLASS(HUMAN) :: OBJ
  ...
MODULE TWO
  USE ONE
  TYPE, INHERIT(HUMAN) :: CHILD
  ...
```

is a problem if some modules USE ONE but not TWO, and others use both. So class-wide objects probably *must* have the POINTER attribute.

CLASS() is a problem for dynamic dispatching. If we have compiled modules A and B separately (or purchased compiled versions thereof, but received no source text), and then we have

```
MODULE C
USE A, ONLY: TA, PA
USE B, ONLY: TB, PB
CLASS() :: Z
INTERFACE PC
  SUBROUTINE PA(X) ! X is of TYPE(TA)
  SUBROUTINE PB(Y) ! Y is of TYPE(TB)
END INTERFACE
...
CALL PC(Z)
```

how does one make the run-time decision whether to call PA or PB if TA, TB, and an ancestor type of the value currently in Z all have the same CLASS_KIND? I don't think CLASS() can be make to work. I think all we

can say about CLASS_KIND is that every CLASS_KIND in an explicitly rooted class hierarchy is unique. I don't think we can guarantee uniqueness in class hierarchies rooted at () without a world-wide database, or too much complication.

I'm worried about auto-specialization. If OBJ1 is passed to a procedure that ordinarily takes an argument of type MAN, what value is given to the BEARDED component? There's the same problem for assignment. I'd prefer MAN(OBJ1,.false.) and WOMAN(OBJ1) to auto-specialization.

Suppose there is

```
INTERFACE FOO
  SUBROUTINE BAR (X)
    TYPE(MAN) :: X
  END SUBROUTINE BAR
  SUBROUTINE BAZ (Y)
    TYPE(WOMAN) :: Y
  END SUBROUTINE BAZ
END INTERFACE
 ...
CALL FOO (OBJ1)
```

Does BAR or BAZ get invoked?

Auto-generalization, on the other hand, is simple (because of single inheritance): If objects don't have the SEQUENCE attribute, and the super-class doesn't have the same storage layout as the collection of variables in the sub-class that the sub-class inherited from the super-class, make a copy of the type of the super-class. Otherwise, just pass the sub-class object. One might suggest (require?) in the standard that the fields inherited from the super-class always come first in sub-classes, and always have the same layout as they would have in the super-class, even without the SEQUENCE attribute. Then one would never need to make a copy to auto-generalize.

If we have a class-wide pointer to the class HUMAN, and it happens to point to a MAN, then it should be allowed to invoke a procedure P that takes a MAN. If its value happens to be of type WOMAN, and there's no procedure P (or in a generic named P) that takes WOMAN or HUMAN, there would be a run-time complaint; if its value happens to be of type HUMAN, and there's no generic for HUMAN, there would also be a run-time complaint. One shouldn't try to create a more specialized object – just call the most specialized procedure corresponding to the actual type of object held by a class-wide variable, if any, else announce an error. Maybe that's the intent of auto-specialization, but I didn't get that from N1188.

There's also an important issue that N1188 doesn't address. C++ has a visibility attribute intermediate between PUBLIC and PRIVATE, namely PROTECTED. Procedures of a derived class have access to PROTECTED members of a base class, but other procedures do not. That's not a problem in the proposed scheme for F2000 if the base and derived types are declared in the same module, but what if we have

```
MODULE TWO
  USE ONE, ONLY: HUMAN
  TYPE, INHERIT(HUMAN) :: CHILD
  ...
```

Does a procedure declared in module TWO have access to the private fields of HUMAN? "Yes" and "No" are both unacceptable answers. Ada-95 solves this problem by way of "child" units. If we have the following,

```
MODULE ONE%TWO
  TYPE, INHERIT(HUMAN) :: CHILD
  ...
```

the answer in TWO should be "No" but the answer in ONE%TWO should be "Yes". The effect is as though the body of ONE were incorporated into ONE%TWO by INCLUDE instead of USE. This view, however, doesn't allow private TYPEs in ONE to be inaccessible in ONE%TWO. Should they be? I think so, but I could be convinced otherwise.

A better solution would be to separate module interfaces (specification parts) and implementations into separate program units. Then, USE refers to an interface module (or an undistinguished module), but not to an implementation module, and child units implicitly incorporate the text of their parent unit's interface module, but not their parent unit's implementation module. So things declared in the implementation module are truly PRIVATE, even to the extent of being invisible to child units. (I think I've worked out all the details of this – see X3J3/97-114 or
`http://gyre.jpl.nasa.gov/~vsnyder/fortran/modules.html`.)

Implementing "child" units into Fortran could be postponed, but one must give the notion some thought, so as not to "paint Fortran into a corner."