Date: 8 December 1997
To: J3
From: Van Snyder
Subject: Enhancing Modules II – Extensibility for object-oriented programming
References: 97-182, 97-183, 97-196, 97-230, 98-104

1 Introduction

The benefit that most distinguishes object oriented programming from other methodologies is that one may modify, replace or extend the behavior of an object by creating a new one that inherits properties from the original one, and one may then extend, replace, or add new behaviors, without modifying the original object.

This permits one to retain confidence in the integrity of the implementation of the original object, whereas modifying an object in order to extend its behavior would compromise that confidence.

One of the principles of "good" software engineering is that users of a resource (procedure, type, or datum) should not depend on the implementation of that resource. The `PRIVATE` statement and attribute enforce this discipline in Fortran 95. When a type is extended, however, it is usually useful if procedures that over-ride a behavior of a parent type to provide different behavior for objects of the extension type, or that provide new behavior, have access to private resources of the parent type. Given the facilities of modules in Fortran 95, and presently proposed extensions to support object oriented programming, one could implement an extension type in one of two ways.

In a different module. Access to the module in which the parent type is implement is gained by use association; entities with private visibility are not accessible.

By modifying the module containing the parent type. Private resources are accessible; modifying the module compromises the confidence one places in the integrity of the implementation of the parent type, and causes otherwise unnecessary re-compilation and re-certification cascades.

Neither alternative is acceptable.

The second is not possible if one must extend a type for which source code is not available.

In C++ and Java, this unavoidable tension between users and extenders of a type is relieved by providing a third type of visibility, *protected* visibility, that is not presently provided by Fortran.

Visibility in C++ and Java is based on *classes*, which are roughly equivalent to derived types in Fortran, with the added facility to define procedures inside of them. Fortran has modules, which C++ and Java lack, and visibility control in Fortran is based on modules. Components of a base type having protected visibility in C++ or Java are accessible in extension types, but not to other users. The C++ equivalent of module variables is static class members. There is no proposal to implement anything equivalent to C++ static class members into Fortran, and this is not necessary. To provide the equivalent of protected visibility for extensible type components, and for module variables, while simultaneously preserving encapsulation, it is sufficient to:

- Allow extending a module without modifying it,
- Allow control of which modules are extensible,
- Allow extension modules access to private resources of the extended module,

- Prohibit users of extensible or extension modules to gain access to private entities of the modules.

A change to modules that is parallel in functionality, syntax and semantics to changes to types that are already under way to provide object oriented programming features is proposed. Without this change it will be difficult effectively to exploit object oriented programming features presently planned to be introduced into Fortran.

Rather than write throughout "If this proposal were implemented, one could ..." this proposal is written as though it were already implemented. Where practice is constrained by the current design of Fortran, "In Fortran 95..." is written.

2 Specifications for proposals

Module extensibility is based on the same principles as type extensibility. By analogy with extensible types, a module that is extended is called a **parent module** and its extension is called a **child module**.

A child module has access to all entities of its parent module, including entities with private visibility, by host association. If the facility advocated in 98-104 to divide modules into main and separate parts is implemented, a child module would have access only to the main part of its parent module. The inaccessibility of the separate parts of the parent module would provide visibility control similar to C++ private visibility.

Superficial examination of 98-104 may suggest that child modules are redundant to separate parts of modules.

A child module is different from a separate part of a module in the following ways:

- A child module can be accessed by use association; a separate part cannot.
- A main part can declare an interface for a procedure contained in a separate part. A parent module can not declare an interface for a procedure contained in a child module.

A child module is similar to a separate part in the following ways:

- All entities, including private entities, that are accessible in or part of a parent module are accessible in its child module, as though by host association. The relation between parent and child modules provides a facility very much like protected visibility in C++ or Java.
- Entities of a child module are not accessible in its parent module.

Neither a parent module nor a child module shall access the other by use association – this would cause a circularity of dependence.

If the proposal advocated in 98-104 to divide modules into main and separate parts is implemented, parent and child modules may independently be composed of main parts and separate parts.

3 Syntax for extensible and extension modules

Extensible and extension modules are analogous to extensible and extension types. As such, the same syntactic modifications are applied to `MODULE` statements as are applied to `TYPE` statements (see O-O proposals).

An extensible type is declared by adjoining an `EXTENSIBLE` attribute to a `TYPE` declaration, e.g. `TYPE, EXTENSIBLE :: POINT`. By analogy, an extensible module is declared by adjoining the same attribute keyword, and using the same punctuation, e.g. `MODULE, EXTENSIBLE :: POINTS`.

An extension type is declared by adjoining an `EXTENDS(`*parent-type*`)` keyword to a `TYPE` declaration, e.g. `TYPE, EXTENDS(POINT) :: COLOR_POINT`. By analogy, an extension module is declared by adjoining the same attribute keyword, and using the same punctuation, e.g. `MODULE, EXTENDS(POINTS) :: COLOR_POINTS`.

For consistency, modules that are neither extensible nor extensions can be declared using syntax analogous to syntax for type and object declarations, e.g. `MODULE :: M`.

4 An illustrative example

This example illustrates extensible modules and extension modules.

```
module, extensible :: points ! an extensible module
  type, extensible :: point  ! an extensible type, see 0-0 proposals 97-196, 97-230
    private; real :: x, y
  contains
    procedure dist => point_dist
    procedure draw => point_draw
  end type point
  private point_dist, point_draw  ! accessible only by using dist and draw
  real function point_dist ( a, b )
    type(point) :: a, b
    point_dist = sqrt( (b%x - a%x)**2 + (b%y - a%y)**2 )
  end function point_dist
  subroutine point_draw ( p )
    type(point) :: p
    ! Whatever is necessary to draw a monochrome point.
  end subroutine point_draw
end module points

module, extends(points) :: color_points ! an extension module
  type, extends(point) :: color_point   ! extension of type "point"
    private; integer :: color
  contains
    procedure draw => color_point_draw ! DRAW is visible to USE'ers
  end type
  private color_point_draw
  subroutine color_point_draw ( p )
    type(color_point), pointer :: p
    ! Whatever is necessary to draw a color point.
  end subroutine color_point_draw
end module color_points
```