Date: 21 December 1997
To: J3
From: Van Snyder
Subject: Enhancing Modules I – Separating interface from implementation
References: 97-114, 97-196, 97-230

1 Introduction

Six long-standing and well-known problems related to modules are solved by adding one keyword to the language, and using it in four ways related to modules, procedures and types.

The additions proposed here advance the goals of the new work item instituted to enhance modules to support construction of interval arithmetic and other data types.

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.

1.1 Terminology

The **module interface** comprises those parts of a module by which its users are expected to communicate with the facilities provided by the module. In Fortran 95, a module interface would comprise most entities with `PUBLIC` visibility. It is generally agreed that it is "good practice" to attempt to keep interfaces stable.

The **implementation of entities of a module** comprises those parts of a module that its author would like to be able to modify without affecting user's of the module. In Fortran 95, the implementation of a module would comprise all entities with `PRIVATE` visibility, procedure bodies, and, sometimes, the `PRIVATE` components of `PUBLIC` types. It is generally agreed that it is "good practice" that users do *not* depend on the implementations of entities of modules.

An **opaque type** is a type about which nothing other than its name is exposed to users of the type. Public types with private components are sometimes called opaque types, but several operations permitted for objects of these types require the compiler to know their sizes; changes to private components of public types can therefore cause users of a module to be re-compiled. A better description of public Fortran 95 types with private components is "translucent types."

Encapsulation is the practice of packaging entities in such a way that users of a behavior of an abstract data type cannot depend on its implementation. In Fortran, `PRIVATE` entities are used for encapsulation.

1.2 Separating interface and implementation

Separating interfaces of modules from implementations of entities of modules provides at least five benefits.

1. Bodies of procedures, and entities that are not intended to be known to users of modules, can be changed without requiring users of those modules to be compiled. It has been argued that this goal can be realized by sufficiently good "quality of implementation of compilers" (QOI). Vendors are only now, however, beginning to think about how to do so. Failure to separate interface from implementation, as a facility of the language, has apparently made achieving this goal unnecessarily difficult, and delayed its realization.

2. Suppose a module X uses another module Y which in turn uses a third module Z in its implementation, but not in its interface. It is not necessary to re-compile module X if module Z is changed.

3. Opaque types can be defined, without an additional new mechanism.

4. Providers of libraries of software could, in principle, remove their trade secrets from Fortran 95 modules, and publish only the remaining text, as authoritative documentation of the interface. Few or none take the trouble to do so. Separate interfaces and implementations of modules, as a facility of the language, allow exactly the same text to be used to declare the interface during development and for publication of a module, with no additional effort or cost.

5. In languages that allow separating interface from implementation, a tactic that is occasionally used to decrease the bulk of individual modules to more manageable size is to "cross use" interfaces. That is, the interface of module A is USE'ed into the implementation of module B, and the interface of module B is USE'ed into the implementation of module A. If interface and implementation are not separated, such "cross use" introduces a circularity of dependence.

1.3 Separating implementation into several parts

Modules sometimes become too large to manage conveniently, and sometimes they become too large to compile. Using Fortran 95 facilities, it is sometimes impossible to split modules into several parts, and still preserve encapsulation: It may be necessary to convert `PRIVATE` entities to `PUBLIC` entities in order to split a module into separate parts of manageable size.

2 Specifications for proposals

It is possible to provide for separate interface and implementation, and separate parts of implementation, with only minor additions to syntax and semantics of modules, types and procedures. The proposals are designed:

- To achieve the goals set forth in the introduction,
- To preserve compatibility to existing Fortran standards,
- To introduce the smallest possible number of additional concepts, but
- Not to introduce any requirement to declare the interface to a procedure more than once.

The mechanisms proposed here allow program authors to package modules as several separate but dependent parts. The **main part** corresponds to the present Fortran module, and can be used in exactly the same way, including the possibility to write entire procedures in the main part.

To separate interface from implementation, and separate implementation into several parts, it is possible to:

- Declare the interfaces of procedures in the main part, but defer defining the procedures to **separate parts** of the module, and
- Declare derived types in the main part, but defer defining some or all of their components to separate parts.

The main part of a module is accessible by use association; separate parts are not. All entities of separate parts are private entities of those separate parts – `PUBLIC` and `PRIVATE` are ignored in separate parts – and are only accessible therein, and in separate parts thereof. This has two useful effects:

- A separate part of one module can access the main part of another by use association, and vice-versa simultaneously, without causing circularity of dependence.

- One would not need to depend on QOI to be assured that a "compilation cascade" would not result from changing the implementation of a public procedure, the definition of a type having private components, a private entity declaration, or a module used in the separate part.

It is important to be able to declare separate but accessible procedures, including their interfaces, in the main part, as opposed simply to declaring and defining a separate part of a module in which complete procedures (interface and implementation) are written. The former advances all the goals set forth in the introduction, while the latter advances only the goal set forth in section 1.3. Furthermore, if procedure interfaces appear only where the procedure bodies are defined, then in modules that are split into several parts it would be necessary to read all of the parts, not just the main part, to determine the set of visible entities of the module, and their interfaces.

A separate part of a module can in turn have separate parts. This is useful for partitioning private entities of large or complex modules into non-interacting subsets.

A part that has separate parts is called a **senior part**. A part that is a separate part of another part is called a **junior part**. These terms are relative to each other; a part could be both a senior and a junior part.

A junior part is logically a continuation of its senior part, with the additional properties that a procedure may be declared, or a type declared and partly defined, in the senior part, and the remainder of the definition provided in the junior part. This is prohibited in the case of a single part.

The only entities of junior parts that are accessible in their senior part are those declared in the senior part.

Junior parts can communicate by using private data and private declarations of separate procedures in the main part. It is better, however, to use an extra layer of separate parts, as this permits the interfaces to these private procedures to change without affecting users of the main part.

Some compilers presently materialize some module procedures in-line instead of calling them. Procedures wholly defined in the main part could continue to be processed in the same way. Superficially, it appears that putting procedure bodies into junior parts subverts this possibility. There are at least three methods, however, to supply this functionality:

- Provide a compiler option that requests "aggressively in-line procedures in junior parts." This means that USE'ers of modules depend on junior parts, and therefore changes to junior parts require re-compiling USE'ers. This subverts the first goal in section 1.2, but none of the other goals in the introduction. If it hurts, don't do it. At least you have a choice.
- It is unlikely that commercial software vendors will provide source text of the junior parts of their modules, so their procedures can't be in-lined in the above way. A semi-compiled form of a junior part that could not easily be changed or examined by an end-user (something like ANDF, for example) would allow in-lining procedures from junior parts, even if source text is not available.
- Foist the in-lining job (and attendant register allocation, etc., jobs) onto the linker.

3 Syntax to separate modules into parts

The facilities necessary and useful to separate modules into parts are

- Declaration that the body that corresponds to a procedure interface is defined in a specified junior part.
- Declaration that an opaque type definition is completed in a specified junior part.
- Declaration for a junior part.
- Declaration that a procedure body in a junior part completes a procedure declared in its senior part.

- Declaration that a type definition in a junior part completes an opaque type declaration, and perhaps partial definition, begun in the senior part.

3.1 Declaring that a junior part exists

The region of a module part that follows a `CONTAINS` statement is herein called a **contains division** of the module. A new statement, *viz.* `SEPARATE ::` *junior-part-name* introduces a **separate division** of a module part. A separate or contains division continues from the statement that introduces it until the next `SEPARATE`, `CONTAINS` or `END MODULE` statement. A module part may contain several separate divisions or several contains divisions, in any order. Different separate divisions of the same senior part may name the same junior part, in which case they are considered to be concatenated. A junior part name is not a global name. It is local to the junior part, and its senior part. It shall not be the same as its senior part's name. Junior parts of distinct senior parts are distinct, even if they have the same name.

3.1.1 Declaring that a procedure body is separate

A syntactic structure identical to an interface body in a separate division indicates that the body of the procedure is defined in the specified junior part. For example

```
SEPARATE :: POINTS_A
  REAL FUNCTION POINT_DIST ( A, B )
    TYPE(POINT) :: A, B
  END FUNCTION POINT_DIST
```

Although syntactically identical to an interface body, the procedure interface declaration must have a critical semantic difference in order to be useful: Definitions, especially derived type definitions, that appear outside of the interface declaration must be visible by host association. If the semantics of interface bodies are not changed so as to allow visibility of derived type definitions by host association, these syntactically identical structures must be called by a different name. In the remainder of this document the term **procedure interface declaration** is used.

All of the procedure characteristics shall be declared in the procedure interface declaration. Declarations that do not contribute to the characteristics may be placed within the procedure interface declaration, and are visible within the procedure when it is defined in the junior part.

A separate procedure is a module procedure.

3.1.2 Declaring that a type definition is separate

A type declaration that appears in a separate division declares an opaque type. Zero or more components can be declared. For example

```
SEPARATE :: S
   TYPE :: OPAQUE_TYPE_NAME
   ! zero or more component declarations
   END TYPE OPAQUE_TYPE_NAME
```

indicates that the remainder of the definition of type `OPAQUE_TYPE_NAME` is deferred to a junior part named `S`.

As is the case for non-opaque types, private components are not accessible by use association, but are visible in the senior part in which the type declaration appears and all junior parts thereof.

The complete definition of an opaque type is visible only in the junior part in which the opaque type definition is completed, and junior parts thereof. Elsewhere, because their sizes are invisible:

- Objects of opaque type shall be defined with the `POINTER` attribute.
- Allocation, deallocation, and intrinsic assignment are not defined.
- An object of opaque type shall neither appear as an *input-item* in a `READ` statement, nor be accessed during namelist editing, nor shall an *output-item* in a `WRITE, PRINT` or `INQUIRE` statement have an opaque type, unless an accesible interface specifies derived type editing for the type.

Opaque types are not extensible.

The senior part that contains a separate procedure or type declaration is not required to be the main part of a module.

3.2 Declaring a junior part

A junior part is introduced by a `SEPARATE` statement that gives the names of the senior and junior parts. It is ended by an `END SEPARATE` statement E.g.

```
SEPARATE(POINTS) :: POINTS_A
...
END SEPARATE POINTS_A
```

declares that `POINTS_A` is a junior part of `POINTS`.

3.3 Defining a procedure declared in a senior part

The body for a procedure declared in a senior part and defined in a junior part is placed in a "contains division" and introduced by an abbreviated procedure header that indicates it is a separate procedure, the category of procedure, and its name. If it is a function, its result name may be declared in the senior part or the junior part, but not both. This is logically a continuation of the interface in the senior part, not a host association to it. Therefore, declarations in the procedure interface declaration in the senior part are accessible here, and shall not be duplicated here. E.g.

```
SEPARATE(POINTS) :: POINTS_A
CONTAINS
  SEPARATE FUNCTION POINT_DIST RESULT(HOW_FAR)
  ! don't re-declare dummy arguments, or result type
  ...
  END FUNCTION POINT_DIST
```

In a separate division of a junior part, it is possible to defer defining a procedure declared in a senior part to an even more junior part. This is useful, for example, if one wants to use an intermediate junior part only for the purpose of encapsulating private data and procedures shared by some subset of the procedures of the module, but wants to put the procedures in separate junior parts. In this case, one uses only the procedure header, without a body or `END` statement, e.g.

```
SEPARATE(POINTS) :: POINTS_A
SEPARATE :: EVEN_MORE_JUNIOR_PART
  SEPARATE FUNCTION POINT_DIST ! No body, because it's in a "separate division"
...
SEPARATE(POINTS_A) :: EVEN_MORE_JUNIOR_PART
```

```
CONTAINS
  SEPARATE FUNCTION POINT_DIST RESULT(HOW_FAR)
  ... ! Body required, because it's in a "contains division"
  END FUNCTION POINT_DIST
```

3.4 Completing the definition of an opaque type

The completion of a definition of an opaque type is indicated by adjoining a `SEPARATE` annotation onto a declaration of the same type, before any separate or contains divisions. E.g.

```
TYPE, SEPARATE :: OPAQUE_TYPE_NAME
! invisible components, and invisible type bound procedure declarations
END TYPE OPAQUE_TYPE_NAME
```

indicates that invisible components of the type `OPAQUE_TYPE_NAME` are defined here. Since these components are invisible to USE'ers or more senior parts, their declaration can't affect the sizes of objects. There is no point to declaring them to be `PRIVATE`, because only junior parts have access to the complete type definition, and those junior parts also have access to private components.

A definition of an opaque type in a junior part is logically a continuation of the declaration in the main part; no component names may be re-declared.

It is possible in a separate division of a junior part to define some of an opaque type, and then defer completion or further continuation to an even more junior part. E.g.

```
SEPARATE :: MORE_JUNIOR_THAN_S
  TYPE, SEPARATE :: OPAQUE_TYPE_NAME
  ! Zero or more additional components.
  END TYPE OPAQUE_TYPE_NAME
```

Additional components are visible at the point of continuation and in junior parts, but not in the senior part. The size is still not known, so the restrictions for opaque types apply.

3.4.1 Another use for the syntax to complete opaque types

Type-bound procedure declarations (see object-oriented programming proposals) can be added to a visible type (opaque or not, completed or not), by adjoining a `SEPARATE` annotation. This is an augmentation, equivalent to extending generic interfaces, not an extension or replacement. The specific names and characteristics of additional type-bound procedures shall not conflict with other visible type-bound procedures.

Type-bound procedures added in this way are not inherited into extension types, or accessible by run-time dispatch. It is not necessary to indicate, at the point of declaration and definition of the type, that there will be additional type-bound procedures associated to the type.

This does not make the type opaque.

4 An illustrative example

This example illustrates modules with main and separate parts, separate parts that have separate parts, and opaque types.

```
module color_points
separate :: color_points_a
  type :: color_point ! opaque type completed in color_points_a
                      ! No visible components
  end type color_point
  interface del;  module procedure color_point_del;  end interface
  interface dist; module procedure color_point_dist; end interface
  interface draw; module procedure color_point_draw; end interface
  interface new;  module procedure color_point_new;  end interface
  private color_point_del, color_point_dist, color_point_draw, color_point_new
  subroutine color_point_del ( p )
    type(color_point), pointer :: p
  end subroutine color_point_del
  real function color_point_dist ( a, b )
    type(color_point), pointer :: a, b
  end function color_point_dist
  subroutine color_point_draw ( p )
    type(color_point), pointer :: p
  end subroutine color_point_draw
  subroutine color_point_new ( p )
    type(color_point), pointer :: p
  end subroutine color_point_new
end module color_points

separate(color_points) :: color_points_a ! Junior part
  integer, save :: instance_count = 0
  type, separate :: color_point ! completion of type declared in color_points part
    real :: x, y
    integer :: color
  end type color_point
separate :: color_points_b
  subroutine inquire_palette (p)
    use palette_stuff
    type(palette) :: p
  end subroutine inquire_palette
contains
  separate subroutine color_point_del ! ( p )
    instance_count = instance_count - 1
    deallocate ( p )
  end subroutine color_point_del
  separate function color_point_dist result(dist) ! ( a, b )
    dist = sqrt( (b%x - a%x)**2 + (b%y - a%y)**2 )
  end function color_point_dist
  separate subroutine color_point_draw ! ( p )
  ...; call inquire_palette ( p ); ...
  end subroutine color_point_draw
  subroutine color_point_new ! ( p )
    instance_count = instance_count + 1
    allocate(p)
```

```
    end subroutine color_point_new
  end separate color_points_a

  separate(color_points_a) :: color_points_b ! Junior**2 part
    separate subroutine inquire_palette
    ! "use palette_stuff" not needed because it's in the senior part
      ... implementation of inquire_palette
    end subroutine inquire_palette
  end separate color_points_b

  program main
    use color_points
    ! components of "color_point" are not accessible here because "color_point"
    ! is opaque. "instance_count" and "inquire_palette" are not accessible here
    ! because they are not declared in the main part of "color_points"
    type(color_point), pointer :: C_1, C_2 ! POINTER because it's opaque
    real :: RC
    ...
    call new(c_1)       ! color_point_new
    ...
    call draw (c_1)     ! color_point_draw
    ...
    rc = dist(c_1, c_2) ! color_point_dist
    ...
    call del(c_1)       ! color_point_del
    ...
    end program main
```

5 Rationales

5.1 Specifying in which junior part an entity is completed

There are two reasons to specify the junior part in which a procedure or type is completed:

1. It prevents providing more than one procedure body for a given declaration, or completing a type definition more than once.

2. It helps a human find the body of a procedure or completion of a type. It is not unusual that 75% of the lifetime cost of a program is incurred during maintenance. Anything in the language design that helps the human during this phase reduces lifetime cost.

The attraction of *not* specifying the junior part in which an entity is completed is that it allows more freedom to re-arrange the assignment of entities to junior parts without (potentially) requiring re-compilation of USE'ers of the module.

5.2 Using `SEPARATE` instead of `MODULE` for junior part header

`SEPARATE` is used for a junior part header to emphasize that a junior part is not accessible by use association.

5.3 Procedure interfaces are not repeated in the junior part

Procedure interfaces are not repeated in the junior part because some people commented that the proposal would not be acceptable if it required repeating procedure interface declarations. It also would require an exception to the rule that no attribute can explicitly be declared more than once.

Others have remarked

- It is desirable to have interface declarations "nearby" during development.
- Copying interface declarations from the senior part to the junior part would not be an onerous burden if one uses modern "cut and paste" editors.
- Copying interface declarations from the senior part to the junior part, and then making them into comments, is not nearly so useful as having the compiler check them.
- Very few users of other languages that require separating interface from implementation (e.g. Modula and Ada) complain about the burden to reproduce procedure interface declarations.

It is presently prohibited to repeat an interface declaration for a module procedure, for which consistency can easily be checked, but allowed for an external procedure, for which consistency cannot easily be checked.

5.4 Separating interface from implementation instead of separating modules into pieces

- Separating interface from implementation solves all of the problems enumerated in the introduction. Simply allowing one to separate modules into pieces addresses only one of the problems – facilitating development and maintenance of large modules.
- If the several pieces of a module are to be allowed access to private entities of each other, it would be necessary either that private entities be included in whatever data base the compiler produces as a result of compiling a module (typically a `.mod` file), or separate data bases would be necessary to record the public information and the private information about modules.
- If all of the pieces all symmetrically to have access to all of the others, it is not clear how to compile any of them until all of the others have been compiled.

5.5 Why are junior parts allowed to have junior parts?

Junior parts are allwed to have junior parts for two reasons:

1. Except for the "header" and `END` statement. the syntactic structure of a junior part is identical to the structure of a module. This makes it easier to explain and remember.
2. It provides a simple mechanism to partition and structure private parts of a module. One correspondent explained the following situation:

   A module that encapsulates a simulation of a physical process contains one interface to a procedure that selects one of six models, depending on arguments. Each model is implemented by a distinct private procedure, typically exceeding 2000 lines in length. Each model procedure was developed and is maintained by a distinct team.

   If junior parts can have junior parts, interfaces to the model procedures, and any data shared between the model procedures and the "steering" procedure can be placed in the same junior part as the "steering" procedure. Otherwise, the interfaces to the model procedures must be placed into the main part (and

probably declared to be private). If model procedures communicate with the "steering" procedure by way of module variables, those, too, would need to be placed in the main part. Changes to the interfaces or shared data could cause re-compilation and re-certification of USE'ers of the modules.

5.6 Why are junior parts continuations, not new scoping units?

If junior parts were new scoping units that accessed their senior parts by host association, instead of being continuations, it would allow declaration of new entities using the same names as entities in the senior part. This was rejected because such redeclarations are almost certainly errors. It also requires a difficult explanation for why the completion of a procedure or type begun in the senior part is *not* a new entity.

6 Questions for J3

1. Should modules be changed in this general way?
2. Should modules be changed in this general way *now* – can this change reasonably be considered to be within the scope of the charge to "enhance the language to make it possible for users to implement abstract data types such as interval arithmetic using modules?"
3. Should the junior part in which a procedure body or type completion appears be announced in the senior part? That is, should a separate division be introduced by `SEPARATE ::` *junior-part-name*, or should a senior part have only one separate division introduced by `SEPARATE ::` *junior-part-name-list*?
4. Should the junior parts of a senior part be announced at all? That is, should a senior part have a single separate division introduced by `SEPARATE`?
5. Should procedure interface declarations be repeated in the junior part? The author's preferences are, in order of most-preferred-first:
   (a) Procedure interface declarations shall be repeated in the junior part, and shall specify the same characteristics as (better: be identical to) corresponding interface declarations in the senior part.
   (b) A procedure interface declaration may optionally be repeated in the junior part; if so it shall specify the same characteristics as (better: be identical to) the interface declaration in the senior part.
   (c) Procedure interface declarations shall not be repeated in the junior part.
6. Should junior parts be continuations of senior parts, or new scoping units that access the senior part by host association?
7. Should the `SEPARATE` attribute be specified on procedure and type definitions in the junior part that complete one declared in the senior part? So long as a junior part is a continuation, not a new scoping unit, this isn't strictly necessary. A compiler (and human) could look in the senior part for an interface or type declaration of the same name in a separate division that names the junior part.
8. Should completely opaque types be included in the present change?
9. Should section 3.4.1 be removed from this proposal?