Date: 14 January 1998
To: J3
From: Van Snyder
Subject: Tutorial on Object Oriented Programming in Fortran
References: 97-182, 97-183, 97-194, 97-195, 97-196, 97-230, 97-261, 98-104, 98-105

1 Introduction

Object oriented programming is a method for abstract data type programming that rests on four software construction principles: Encapsulation, inheritance, polymorphism, and, sometimes, dynamic binding.

Encapsulation means that one exploits the properties of an abstract data type without depending on its implementation. An abstract data type consists of a definition of a structure to represent data (a Fortran type), together with a set of operations.

An **object** in the object oriented programming sense is an instance of the data structure – a Fortran object – together with the operations of the type of the object.

Operations of the type are implemented by procedures. A procedure that is unavoidably associated with a type, that is, one that is inevitably accessible if the type is accessible, is called a **type bound procedure**. In other literature, a type bound procedure is known as a **primitive operation of the type** or a **method**.

The mechanism of re-use in object oriented programming is to derive a new abstract data type, called an **extended type** or **child type** from an existing abstract data type, called a **parent type**. In the process, the extended type **inherits** the data structure and operations from the parent. The extended type may define additional components, or additional operations. New operations of the extended type that have the same name as operations inherited from the parent, and for which the only difference of characteristic is that (one or more) dummy arguments of the parent type are replaced by dummy arguments of the extended type, are said to **over-ride** the parent type's operations.

Inheritance provides a more structured and more incremental form of re-use than simple procedure re-use.

The terms *parent* and *child* are relative; a child type can be the parent of other child types.

A parent type, together with all descendant types, is called a **class of types**, or more simply a **class**. The parent type of all types within a class is called the **base type** of the class.

An important principle of object oriented programming is that data entities may be allowed to represent different types of objects from the same class at different moments. This is a form of **data polymorphism**. A data entity that is only permitted to contain objects of a single type is called a **monomorphic** entity.

Procedure polymorphism is used to allow a single name to refer to different procedures, depending on the type of the actual arguments. This facility is particularly powerful when used in conjunction with data polymorphism.

The process of deciding which of several procedures to invoke is called **dispatch** or **binding**. **Static dispatch** or **static binding** means it is possible to deduce the type of arguments, and therefore the particular procedure, from the text of the program. This is the variety of procedure polymorphism provided by generic interface blocks in Fortran. **Dynamic dispatch** or **dynamic binding** means that the decision which procedure to invoke is delayed until the moment the procedure invocation is executed. Dynamic dispatch is used when a procedure that has monomorphic dummy arguments

is invoked with polymorphic actual arguments. Since polymorphic data objects can contain objects having types within a class, dynamic dispatch decides which of several over-riding procedures to invoke.

2 Encapsulation

The unit of encapsulation in Fortran in the module. No changes to the syntax and semantics of modules are necessary in order to support object oriented programming, but one change may be useful and desirable. See paper 98-105, *Enhancing Modules II – Extensibility for object oriented programming.*

3 Inheritance

Inheritance consists of two parts, *viz.* data inheritance and procedure inheritance.

3.1 Data inheritance

The mechanism of data inheritance presently proposed to be implemented into Fortran is **type extension**: A new Fortran type is constructed from an existing one by adding zero or more components. The J3/data subgroup have agreed on specifications and syntax for data type extension, and J3 have accepted these.

An extensible type that is not the extension of another is declared by adjoining the `EXTENSIBLE` attribute to its declaration, e.g.

```
TYPE, EXTENSIBLE :: POINT
  PRIVATE; REAL :: X, Y
END TYPE POINT
```

declares an extensible type named `POINT` that has two private components.

A child type is declared by adjoining the `EXTENDS` attribute, with an "argument" consisting of the parent type, to the type declaration, e.g.

```
TYPE, EXTENDS(POINT) :: COLOR_POINT
  PRIVATE; INTEGER :: COLOR
END TYPE COLOR_POINT
```

declares an extended type `COLOR_POINT` that is a child of type `POINT`. A child type has exactly one parent type. This is called **single inheritance**.

A child type has all of the components of the parent type, and additional ones may be declared. In this example, objects of type `COLOR_POINT` have `X` and `Y` components inherited from type `POINT`, and a `COLOR` component. Objects of child type are considered also to have a component having the same name as the parent type. See papers 97-183 and 97-196 for more details, including specifications, syntax and semantics of type constructors.

Extended types can be further extended.

In extensible types, as for Fortran-95 types, the default component visibility is `PUBLIC`.

3.2 Procedure inheritance

Procedure inheritance rests on type bound procedures. The term *type bound* implies at least that if the type is accessible, then the type bound procedures are accessible as well. It may eventually have

other implications that are still controversial within the J3/data subgroup.

Type bound procedures are declared within the body of a type declaration by using one of three statements. A `PROCEDURE` statement creates a generic name by which a collection of specific procedures is known, similar to the way that generic interface blocks function in Fortran-95. The differences will be explained later. In addition to generic procedure names, one can declare generic operators using an `OPERATOR(`*defined-operator*`)` statement, or define assignment using an `ASSIGNMENT(=)` statement. In each case, the set of specific procedures follows `=>`. An example of type bound procedure declarations is

```
TYPE T
CONTAINS
  PROCEDURE P => P1, P2, P3
  OPERATOR(-) => REAL_MINUS_T, T_MINUS_REAL, T_MINUS_T
  ASSIGNMENT(=) => T_EQUALS_T
END TYPE T
```

The specific procedure to use for each reference is determined using existing Fortran-95 rules for generic procedure dis-ambiguation.

When an extended type is declared it inherits all of the type bound procedures of its parent. Some or all of the specific procedures bound to the parent type can be over-ridden by providing specific procedures that have arguments of the extended type where an inherited procedure has arguments of the parent type. Additional procedures that do not over-ride procedures inherited from the parent type can be declared to be type bound to the extended type.

If a procedure from the parent type is over-ridden in an extended type, the over-riding procedure is used for that operation on the extended type.

3.2.1 Over-ride semantics

The precise meaning of over-riding is still a subject of controversy within the J3/data subgroup.

One point of view is that type bound procedures have a dummy argument that is distinguished as the *receiver of a message* sent to an object requesting that some action be taken. This is the point of view one adopts when programming in C++, Smalltalk, Objective-C, Oberon and numerous other object oriented programming languages.

In this case, when a child type inherits a type bound procedure from its parent, the type of the receiver is considered to be of the child type, for purposes of generic dis-ambiguation and type checking. If a procedure has additional arguments of the parent type, the essential feature is that they have that type. Therefore, when the procedure is inherited into a child type, the types of additional arguments of the parent type are not considered to become of the child type. An over-riding procedure in the child type would be required to have a receiver of the child type, and all other arguments of the same type. All arguments must have the same type parameters and rank as for corresponding arguments of the procedure to be over-ridden.

Another point of view is that the send-a-message-to-an-object paradigm is alien to Fortran; all dummy arguments of type bound procedures should be co-equal; there should be no distinguished receiver of a message; the Fortran-95 interpretation of procedure invocation should continue in the context of object oriented programming. This is the point of view one adopts when programming in Eiffel, CLOS, Haskell, Ada-95 and numerous other object oriented programming languages.

In this case, when a child type inherits a type bound procedure from its parent, the type of all arguments of the parent type is assumed to be of the child type, for purposes of generic dis-ambiguation and type

checking. If a procedure has several arguments of the parent type, the essential feature is that they have the same type. An over-riding procedure in the child type would be required to have arguments of the child type in all positions where the over-ridden procedure has arguments of the parent type, and in all other positions to have arguments of exactly the same type as the procedure to be over-ridden. All arguments must have the same type parameters and rank as for corresponding arguments of the procedure to be over-ridden.

Suppose one has bound a `DISTANCE` function to the type `POINT` declared above, using, e.g. `PROCEDURE DISTANCE => POINT_DIST`, and suppose `POINT_DIST` is declared

```
REAL FUNCTION POINT_DIST ( A, B ) ; TYPE(POINT), INTENT(IN) :: A, B
  POINT_DIST = SQRT( (B%X-A%X)**2 + (B%Y-A%Y)**2 )
  RETURN
END FUNCTION POINT_DIST
```

This controversy concerns whether it would be more generally useful if `POINT_DIST` is considered to have one argument of type `COLOR_POINT` and one of type `POINT` (the first point of view above) or two arguments of type `COLOR_POINT` (the second point of view above), when it is inherited into type `COLOR_POINT`.

This controversy is also discussed in paper 97-261.

3.2.2 Function result type

If a function is inherited from a parent type, and the function result is monomorphic and of the parent type, it cannot be considered to be monomorphic of the child type because the child type may have additional components that the function inherited from the parent doesn't compute.

It could be considered to be polymorphic in the class of the parent type (see section 4), or it could be considered to be monomorphic of the parent type. The latter may compromise the usefulness of polymorphic expressions.

4 Polymorphism

4.1 Data polymorphism

The J3/data subgroup have recommended and J3 have accepted specifications and syntax to declare polymorphic objects, that is, objects that may contain data of different types within the same class at different instants. An example declaration for an object that could contain data of either of the types `POINT` or `COLOR_POINT` declared above is

```
OBJECT(POINT), POINTER :: POINT_DATUM
```

Because data of different types may require different amounts of storage, polymorphic objects must be declared with the `POINTER` attribute if they are not dummy arguments.

A polymorphic object may be allocated with the declared base type, e.g. `ALLOCATE ( POINT_DATUM )` creates an object of type `POINT`. Pointer assignment may be used to assign a datum of any type in the class.

4.2 Procedure polymorphism

When a procedure is invoked with monomorphic actual arguments corresponding to monomorphic dummy arguments, the static (compile-time declared) types of the actual arguments must match the types of the dummy arguments, as is presently the case for Fortran-95. The procedure to be invoked can be selected by the compiler, and the correspondence of actual to dummy argument types can be verified by the compiler. The difference in the case of extensible types is that one interprets dummy argument types to account for inheritance, as described in section 3.2. This is one of the differences between type bound procedure declarations and generic interface blocks. Leaving aside other issues for now, one could consider the declaration `PROCEDURE P => P1, P2, P3` above to be equivalent to

```
INTERFACE P
  MODULE PROCEDURE P1, P2, P3
END INTERFACE P
```

When a child of type `T`, say type `T'`, is declared. the above interface is considered to be extended by

```
INTERFACE P
  MODULE PROCEDURE P1', P2', P3'
END INTERFACE P
```

in which any of `P1'` etc. are the procedures `P1` etc. inherited from type `T` with dummy argument(s) of type `T` considered to be changed to be of type `T'`, or procedures that over-ride `P1` etc.

When a procedure is invoked with monomorphic actual arguments corresponding to polymorphic dummy arguments, the static (compile-time declared) types of the actual arguments must match or be descendant from the declared class of the corresponding dummy arguments. As in the case of monomorphic dummy arguments, the procedure to be invoked can be selected by the compiler, and the correspondence of actual to dummy argument types can be verified by the compiler.

When a procedure is invoked with polymorphic actual arguments corresponding to polymorphic dummy arguments, the classes of the actual arguments must match or be descendant from the declared class of the corresponding dummy arguments. As in the above cases, the procedure to be invoked can be selected by the compiler, and the correspondence of actual to dummy argument classes can be verified by the compiler.

When a procedure is invoked with polymorphic actual arguments corresponding to monomorphic dummy arguments, the compiler cannot always determine the procedure to be invoked, nor can it verify completely the correspondence of actual argument types to dummy argument types – the best it can do is to verify that the base type of the actual argument class is the same as or descendant from the type of the dummy arguments. The compiler carries out all of the generic dis-ambiguation that is possible (for those parts of the characteristic that can be statically determined), and postpones the remainder until the program runs. This is *dynamic dispatch* or *dynamic binding*. Since the inheritance model for Fortran is specified to be single inheritance, the run-time selection is always within one class. No matter which model of procedure inheritance is used, the selection is determined using a single type. In the first model, it is the type of the receiver; in the second, it is the type of a set of arguments. This is called **single dispatch**.

5 Abstract types

Most object oriented programming languages provide for **abstract types**. Abstract types are types for which the characteristics of necessary type bound procedures are declared, but for which no procedures are provided. In a graphical system, for example, the types `POINT` and `COLOR_POINT` used above might have an abstract `DRAW` procedure. Other types descendant from them, say `CIRCLE` and `COLOR_CIRCLE` would provide concrete `DRAW` procedures. Most languages prohibit creating objects of abstract type.

Some languages require descendants of abstract types to be concrete; others allow them to be abstract. Some languages prohibit concrete types to have abstract descendants; others allow it.

The mechanism under consideration to indicate an abstract type is to provide `NULL(`*abstract-interface-name*`)` after `=>`, rather than a concrete procedure name (*abstract-interface-name* is defined in paper 97-248r2 for use with procedure pointers).

Since intrinsic assignment is defined by the standard, providing a null procedure name for a defined assignment procedure, in which both arguments are of the type to which the assignment is bound, serves to declare that intrinsic assignment doesn't exist. It does not make the type abstract.

6 Syntax of type bound procedure reference

In most object oriented programming languages that consider invocation of a type bound procedure to be sending a message to an object, the syntax to reference a type bound procedure is the same as the syntax to reference a component of a derived type. If `U` and `V` are two objects of the type `POINT` used in an example above, the type bound `DISTANCE` function would be invoked by `U%DISTANCE(V)`. One is expected to think of this as "Send a message to `U` asking it to compute its `DISTANCE` from `V`."

The advantage of this syntax is that the possibility for conflict of names is reduced because a type bound procedure name is visible only when qualified by an object of the type to which it is bound.

The disadvantages of this syntax are

- The correspondence between actual and dummy arguments is not as transparent as in the case of the usual Fortran procedure reference syntax. If the first dummy argument of a specific procedure corresponding to a type bound generic name is designated to be the receiver of the message, then the first actual argument written within parentheses (`V` above) corresponds to the second dummy argument in the procedure declaration, etc.

- Designating the first dummy argument, or any other dummy argument in a fixed position, to be the receiver makes it difficult to write a full spectrum of type bound operators.

- Allowing a specification of which dummy argument is the receiver requires restrictions in order to avoid ambiguities. Consider the following example:

```
TYPE :: T
CONTAINS; PROCEDURE P => P1, P2
...
TYPE(T) :: U
...
SUBROUTINE P1 ( A, B )
  TYPE(T), RECEIVER :: A ! First dummy argument corresponds to
                         ! actual argument before the "%"
  REAL :: B
  ...
SUBROUTINE P2 ( A, B )
  REAL :: A
  TYPE(T), RECEIVER :: B ! Second dummy argument corresponds to
                         ! actual argument before the "%"
  ...
CALL U%P(5.0)            ! Does this invoke P1 or P2?
```

Whatever restrictions are put in place to avoid this ambiguity could not apply to type bound operators or type bound defined assignment without severly compromising their usefulness. Such an inconsistency makes compilers expensive and slow, and discourages learning and use of the language.

If the same syntax is used to reference type bound procedures and other procedures, the `DISTANCE` function bound to type `POINT` would be invoked `DISTANCE(U, V)`.

The advantages of this syntax are

- The positional correspondence between actual arguments and dummy arguments is not modified to account for a distinguished receiver of a message.
- The rules for reference to a procedure by way of a generic type bound name are the same as the rules for reference to a procedure by way of a type bound operator or type bound defined assignment, and the same as existing rules in Fortran-95 when inheritance is taken into account.
- Type bound procedures can be in the same generic set with procedures that are not type bound. For example:

```
TYPE :: T
CONTAINS; PROCEDURE P => P1, P2
...
INTERFACE P ! perhaps in a different scoping unit from the above
  MODULE PROCEDURE P3
END INTERFACE
```

  is reasonable and should be allowed. Unlike `P1` and `P2`, `P3` is not a type bound procedure, and therefore would not be inherited into types descendant from type `T`, and `P3` would not necessarily be accessible everywhere that `T` is accessible. See paper 98-104 for another proposal to achieve this effect.

The disadvantages of this syntax are

- There is more possibility for conflict of names than in the previous case. This is a very small effect, however, because type bound procedure names are generic; identical generic names are not a problem, so long as characteristics are different – the generic set is simply expanded. It is reasonable to constrain a type bound procedure to have at least one dummy argument of the type to which it is bound. Thus, conflicts of both name and characteristic are unlikely.
- Restrictions on argument types may be necessary or desirable to avoid ambiguities. Consider

```
TYPE, EXTENSIBLE :: T1
CONTAINS; PROCEDURE P => P1
...
SUBROUTINE P1 ( A, B ); TYPE(T1) :: A, B
...
TYPE, EXTENDS(T1) :: T2
CONTAINS; PROCEDURE P => P2, P3
...
SUBROUTINE P2 ( A, B ); TYPE(T1) :: A; TYPE(T2) :: B
...
```

```
SUBROUTINE P3 ( A, B ); TYPE(T2) :: A; TYPE(T1) :: B
...
TYPE(T2) :: U, V
...
CALL P ( U, V )
```

Does `CALL P` invoke `P1`, `P2` or `P3`, or is it prohibited? It is difficult to argue that `P2` or `P3` is the correct choice. One might argue by analogy with generic interfaces that include elemental procedures that `P1` is the correct choice. In the case when the actual arguments are not in the same class, an analogous choice would not be so obvious.

The ambiguity can be avoided by prohibiting it: Monomorphic actual and dummy arguments of extensible type must all be the same type. This is not a severe restriction on functionality because a similar restriction when dummy arguments are polymorphic is not necessary. The ultimate effect is that one sometimes (but not always) loses control over whether dynamic dispatch occurs.

7 Why not use generic interface blocks to declare type bound procedures?

Essentially everything that is accomplished by declaring type bound procedures inside of type declarations could be accomplished by using generic interface blocks. The advantages of the former are

- Admirable terseness.
- An obvious interpretation that the type bound procedures are inheritable, which would otherwise require special rules – e.g. a procedure that appears in a generic interface block in the same scoping unit as type `T`, and has monomorphic argument(s) of type `T` is a type bound procedure of type `T`.
- An obvious interpretation that the type bound procedures are accessible everywhere the type is (unless they're private), which would otherwise require special annotations or special rules – e.g. type bound procedures can't be separated from their type by an `ONLY` clause of a `USE` statement.

8 Questions for J3

J3/data are unable to resolve two questions, concerning which guidance from the full J3 committee would be helpful. The consequences of the choices are discussed in sections 3.2.1 and 6, respectively.

1. Should a type bound procedure have a distinguished *receiver* argument of the type to which it is bound, which is the only argument that is considered to change to the child type when the procedure is inherited?

   Or

   Should all monomorphic dummy arguments of the type to which a procedure is bound be considered to be changed to the child type when the type bound procedure is inherited?

2. Should type bound procedures be invoked using the same syntax as for invoking a procedure using a procedure pointer that is a component of the type?

   Or

   Should type bound procedures be invoked using the same syntax as is used for "ordinary procedures" in Fortran-95?

8.1 The author's opinions

The author advocates the second choice for both questions above. Concerning the first question, the author believes that it is more likely to be useful that several arguments of the type to which a procedure is bound are the same type, and remain the same type as each other when the procedure is inherited.

Concerning the second question, the author believes nothing significant is gained by developing numerous new mechanisms so that invoking procedures using component reference syntax is useful, or, conversely, nothing significant is foregone by using existing mechanisms. Absent a compelling technical reason to choose one over the other, simplicity argues for the second. One of the advantages of Fortran as compared to other widely used languages is its relative simplicity, and relative ease to learn. These virtues should be preserved.