Date: 10 July 1997
To: J3
From: Van Snyder
Subject: White Paper on Object Oriented Programming

Summary of recommended changes

In order to support object oriented programming, this report recommends the following changes. The intent of this report is to explain features necessary and useful for object oriented programming, enumerate all the desirable or necessary changes in a single document, and show their relation. Not all of them are immediately necessary. The design of those that are implemented immediately should be such that the remainder can be implemented later without undue difficulty.

Un-define intrinsic operations Ada uses **limited** to undefine assignment and intrinsic equality, and defines inequality to be (in Fortran notation) .NOT.equality. Either a single word (e.g. LIMITED) or an "instruction" with "parameters" (e.g. DELETE(=), DELETE(=,==) etc.) could be used as an attribute of a type in Fortran. See 97-145. .... 2.1

Defined pointer assignment Use INTERFACE ASSIGNMENT (=>). 2.2.1

Destructor procedures Numerous possibilities. 2.2.2

Type extension Use the mechanism described in 97-183 or 97-195. .... 3.1

Conversion within derivation class Use type name as a view converter and as an extended *structure-constructor*. .... 3.1.1.

Module extension Use A%B for a module name to indicate it is a "child unit" of module A. See 97-114. .... 3.2.1.

Separate specification and implementation of modules See 97-114. .... 3.2.2

Polymorphic data objects Allow non-pointer polymorphic data objects analogous to automatic arrays, by providing executable initialization expressions. See also 97-183 and 97-195. 4.1.1

Conversion to polymorphic type Define a "type" name for polymorphic object declarations; use it for view conversions. 4.1.3

Over-riding procedure definitions Mechanism of 97-182 is workable, but preferably without shifting actual-to-formal argument correspondence. Alternatives should be debated. See also 97-195. .... 4.2.2

Dispatching calls – dynamic binding Mechanism of 97-182 is workable, if polymorphic actual arguments that appear between (...), and that have the same run-time type as the "argument" before %, can correspond to monomorphic formal arguments. See also 97-195. .... 4.2.3

Abstract types and procedures Use an attribute of types and procedures, e.g. TYPE, ABSTRACT :: foo, ABSTRACT SUBROUTINE and ABSTRACT FUNCTION (I prefer SUBROUTINE, ABSTRACT :: and FUNCTION, ABSTRACT ::). See 97-182 and 97-195. ...5

1 Introduction

The most widely used definition of Object Oriented Programming revolves around three concepts – *data abstaction*, *inheritance* or *extension*, and *polymorphism*.

Data abstraction means that a data type consists of a set of values, together with the operations on those values. Both the sets of values, and the operations on them, can be defined by the program.

Inheritance means that a new data type can be derived from an existing data type by adding to the existing set of data values, adding to the operations, or changing some of the existing operations.

Polymorphism allows defining variables that have different types at different times, and operations that apply to more than one type.

Fortran 95 has rudimentary forms of data abstraction and procedural polymorphism, but no inheritance. Data abstraction and polymorphism must be extended, and inheritance must be introduced.

Other facilities that are necessary and useful for software engineering, but lacking from Fortran, influence the use of the three basic concepts of object oriented programming. Extension of Fortran to include support for object oriented programming and software engineering could be done piecemeal, but the greatest benefit, and the greatest difficulty, comes from integrating all of the necessary facilities.

This may be too difficult to do all at once. Nonetheless, all of the necessary facilities should be kept in mind during the design of those to be introduced into the present revision of Fortran, so as not to make later addition of the other facilities unnecessarily difficult.

Chapter 12 of **Ada As a Second Language** by Norman H. Cohen (ISBN 0-07-011607-5), entitled *Classwide Programming*, is an excellent discussion of the ideas outlined here.

2 Data abstraction

The principal principle supporting data abstraction is *information hiding*, that is, use of an object should depend neither on its representation, nor on the methods by which operations on the object are carried out. One habit of software engineering that helps to conceal details of representation is to hide all accesses and operations in procedures. An alternative is described in 97-114. To prevent dependence on methods of operation, data representing the state of objects or collections thereof are maintained as *private* resources of the abstraction.

2.1 Un-defining intrinsic assignment and intrinsic equality

To allow use of type-dependent procedures, while maintaining the value of the semantic suggestiveness of well-known operational symbols, Fortran allows a program to provide definitions for them. There are deficiencies, however, in the specification of defined assignment. One of those deficiencies is addressed by 97-145. Nonetheless, as remarked in 97-145, it is still possible to "subvert an abstraction" by neglecting to import defined assignment.

This could be prevented by allowing an attribute of a type to indicate that intrinsic assignment, and perhaps intrinsic equality, are not defined for the type. In Ada, this attribute is called *limited*.

2.2 Storage management

Storage management should be entirely the responsibility of a data type. Unfortunately, deficiencies in Fortran 95 require users of an abstraction sometimes to take on some of the responsibility for storage management, and, worse, in some situations, prevent both the abstraction and the user thereof to take on the responsibility.

2.2.1 Defined pointer assignment

One technique to prevent "memory leakage" is to maintain a "reference count" for each object that is dynamically allocated. The inability to define pointer assignment requires users of objects to undertake this responsibility. This, in turn, requires exposing some of the representation or mechanism of the managed objects.

2.2.2 Destructors

A program can also "lose track" of memory usage when objects contain pointers to dynamically allocated memory. This can be somewhat ameliorated by providing a *destroy the object* procedure, that users of the object explicitly invoke. Neglecting to do so causes a memory leak. No amount of diligence, however, can cause a procedure that must be invoked explicitly to be invoked for objects that have no names – temporary intermediate results created during evaluation of expressions involving user-defined types and operators, for example. A facility to define a procedure that is invoked automatically whenever an object ceases to exist could prevent this source of memory leakage. In C++ this is called a *destructor*.

Some possible mechanisms:

-
  ```
  INTERFACE DESTRUCTOR
      SUBROUTINE DSUB (D) ! D is to be destroyed
      TYPE(T), INTENT(INOUT) :: D
  END INTERFACE DESTRUCTOR
  ```
- DESTRUCTOR SUBROUTINE DSUB (D) ! Same interface for DSUB as above
- TYPE, DESTRUCTOR(DSUB) :: T ! Same interface for DSUB as above

Only one destructor subroutine may be defined for each type.

3 Inheritance

There is no mechanism for inheritance in Fortran 95. Many languages provide inheritance by *abstract data type extension*, which consists of *data representation extension* and *procedure extension*.

3.1 Data representation extension

A mechanism for data representation extension is proposed in 97-183: The data representation for a new type of object is created by adding zero or more components onto the representation of an existing derived type. Both the base type and the extended type are Fortran derived types. Henceforth, if a type T2 is created by extension, in zero or more steps, from a type T1, the type T2 will be said to be *descendant from* T1, and T1 will be said to be *ancestral to* T2.

3.1.1 Conversion

Part of the definition in 97-183 is that an extension type is implicitly considered to have a component having the same name and type as the ancestral type. The implied component is used to construct a value of an extension type by giving a value of the base type and values for components that extend the base type, and to access all of the components of the base type as a single component.

A simple extension of the *structure-constructor* could serve for the first use. In the case of constructing an object of an extension type from an object of the base type, plus additional fields, a *structure-constructor* for the extension type would necessarily have "arguments" for the additional fields. A slight additional generalization is useful. Suppose T1 is a base type, T2 is an extension of T1, T3 is an extension of T2, X1, X2 and X3 are objects of types T1, T2 and T3 respectively, and C2 and C3 are additional components necessary to extend X1 to X2 and X2 to X3, respectively. One should be able to construct X3 in the following ways:

- X3 = T3(X2, C3)
- X3 = T3(T2(X1, C2), C3)
- X3 = T3(X1, C2, C3)
- X3 = T3(*components-of-X1*, C2, C3)

The last two are progressively more general short-hand notations for the second method.

To extract the base-type-part from an extension type, simply considering the type name to be a function that performs a *view conversion* (see 4.1.2) provides the desired functionality.

3.2 Abstract data type extension

The method proposed in 97-183 provides to extend data representations by adding components, and to extend operations by *over-riding* existing operations (see 4.2.2). These two capabilities, however, leave two shortcomings.

The mechanism in Fortran 95 to construct abstract data types by packaging data representations together with operations on them is the module. Providing additional features of modules would address these shortcomings.

Other considerations concerning abstract data type extension are intimately connected to polymorphism (see 4).

3.2.1 Module extensions

An *extension module* is a natural container for extension types, related new types, over-riding operations, and new operations.

Suppose a module A is to be extended. Its extensions, and their extensions, might be named A%B, A%B%C, etc. Extensions in A%B of types defined in A should have visibility of resources of their base types defined in A. USE association does not provide this visibility. Thus, for purposes of visibility, A%B should be considered to be incorporated into A, and A%B%C incorporated into A%B (hence the notational suggestion). Therefore A%B would have access to private entities defined in A, and A%B%C would have access to private entities defined in A%B and (by induction) in A.

3.2.2 Separating module specification and implementation

It is sometimes desirable for some of the resources of a base type to be completely private resources of the module that packages the type. In C++ the distinction between resources that can and cannot be accessed by a type derived from another by inheritance is specified by different accessibility levels, *viz. private* and *protected*. In Ada 95, the distinction is provided by separating the package into a specification part and an implementation part. The specification part of a base module is visible in an extension module; the implementation part is not.

Separating specification and implementation has benefits in addition to controlling visibility (see 97-114); additional visibility levels do not.

4 Polymorphism

There are two interconnected varieties of polymorphism: procedure polymorphism, and data polymorphism.

Fortran 95 already supports one form of procedure polymorphism by way of generic interfaces.

Data polymorphism can be described in terms of a *derivation class*. A derivation class T consists of an extensible type T, and all types derived from T by extension.

It is useful to be able to construct *polymorphic objects* of class T, that is, objects whose values may be of any type in the derivation class T. Coupling polymorphic objects with the polymorphic operations defined over the derivation class yields a particularly powerful combination.

4.1 Data polymorphism

A polymorphic variable is one that might have values of different types, from the derivation class declared for the variable, at different times.

When the base type of a derivation class is defined, one does not know all the extensions that might eventually occur. Thus, it is impossible statically to declare a polymorphic data object – how can one know how much storage to allocate? This does not affect polymorphic objects that have the pointer attribute.

4.1.1 Non-pointer polymorphic data objects

Automatic arrays are created when the scope in which they are declared comes into existence, and destroyed when its existence ceases. Their sizes can be different from one existence to another, but throughout a single existence, their sizes are fixed.

An "Automatic" non-pointer polymorphic data object could come into existence when the scope in which it is declared comes into existence. It might contain values of different types from one existence to another, but throughout a single existence, the type of value it contains is constrained, either to be fixed throughout the existence of the object, or, at the expense of run-time bookkeeping, it could contain a value of any type in the derivation class between the base of the class, and the type of the object at the instant it came into existence.

One way to create a non-pointer polymorphic object with a specific type when it comes into existence is to provide a declaration that takes a parameter – from a (function of a) formal argument, or a global variable, for example – that is examined when it comes into existence.

Another is to require that a non-pointer polymorphic object must have an initial value, from which it takes the (most extended) type it is allowed to have during its existence. Therefore initial value expressions must be allowed to be executable; this has other uses (on which others have already commented).

4.1.2 View conversion

Conversion of a poly- or monomorphic object to an ancestor of its actual type is a *view conversion*, not a value conversion: zero or more fields are ignored. An anonymous copy of the object is not created, and the value of the object remains unchanged. It is always possible to view-convert or copy a monomorphic object to an ancestral type, or a polymorphic object to the monomorphic base type of its derivation class; otherwise, if a run-time check verifies the converted type of a polymorphic object is ancestral to the actual type, conversion is allowed.

4.1.3 Conversion to polymorphism

It is sometimes necessary to convert a value of mono- or polymorphic type to an ancestral polymorphic type. This usually occurs as a result of a *dispatching call* (see 4.2.3) to a polymorphic function that returns a monomorphic result. Neither the mechanism described in 3.1.1, nor the "implied component" mechanism described in 97-183, addresses this need. Ada 95 uses the notation <type>'class(value). Doing so in Fortran would introduce an entirely new syntactic device. An alternative would be desirable. One is to define a name for a derivation class, e.g.

```
  CLASS(vector_2d) :: vectors
```

Then use `TYPE(vectors)` to declare polymorphic objects of the derivation class rooted at `vector_2d`, and use `vectors(...)` as a view converter to polymorphic type.

4.2 Procedure polymorphism

4.2.1 Primitive operation of a tagged type

In Ada 95, a *primitive operation of a tagged* (extensible) *type* is a procedure that is defined in the same specification part of a package (module) as the type, and that takes at least one argument of the type. It is not possible for a procedure to be a primitive operation of several extensible types – all formal arguments of monomorphic extensible type must be of the same type.

In C++, Modula-3 and the mechanism proposed in 97-182, primitive operations are declared within the definition of the type.

4.2.2 Overloading and overriding procedure definitions

A set of procedures that have the same name, but different *signatures* – for example, the sequences of types of arguments – are collectively called a *polymorphic procedure*. The procedures in the set are said to *overload* one another. When a polymorphic procedure reference appears, the signature of the reference determines the procedure that is invoked. In Ada 83 and Fortran 95, the signature of a reference must precisely match the signature of exactly one procedure.

If a primitive operation of a base type exists, and a primitive operation of the same name, for an extension of the base type, is defined, and the only differences in signature are that formal arguments of the base type are replaced by formal arguments of the extension type, the latter *over-rides* the former. Absent over-riding definitions, primitive operations of the base type are also primitive operations of the extension type, that is, they are *inherited* by the extension type.

Actual arguments to a primitive operation that correspond to formal arguments of monomorphic tagged type must be of the same type. If necessary, they are view-converted to the type of the primitive operation. Other actual arguments may be of any type consistent with their corresponding formal arguments.

4.2.3 Dispatching calls

A useful consequence of the interaction between data polymorphism and procedure polymorphism is that the precise procedure used need not be known until the instant it is invoked. This capability is called *dynamic binding, delayed binding* or *late binding.* When the procedure to be invoked is selected at the instant of invocation, it is called a *dispatching call.*

In C++, all pointers to objects are polymorphic, but only *virtual functions* are dispatched. This design causes significant "fragility" – a common error is to neglect to declare a function to be virtual. The primitive operation for the base type is invoked, even when the actual type of the object is an extension for which an over-riding definition of the primitive operation exists.

Ada 95 and the methods proposed in 97-182 and 97-183 distinguish between monomorphic objects (of specific types), and polymorphic ("classwide") objects. If one uses polymorphic actual arguments when invoking a polymorphic procedure that takes monomorphic arguments, the precise type of the object is examined to determine the procedure to invoke. Several polymorphic arguments may appear, but their actual types must be identical when the procedure is invoked.

Procedures may have polymorphic formal arguments. Monomorphic or polymorphic actual arguments may be associated to polymorphic formal arguments so long as only a view conversion is required (see 4.1.2). A run-time check is required when it cannot be statically determined that the formal argument type is an ancestor of the actual argument type.

4.2.4 Comparison of linguistic devices

In C++, Smalltalk, Objective C, Modula-3, and the method proposed in 97-182, there is exactly one argument that determines the type of which a procedure is a primitive operation, and hence determines dispatching if it is polymorphic. In invocations, it appears in a special position, using a syntax that suggests the procedure is a component of the derived type; *variable%procedure* is proposed in 97-182. In C++, Smalltalk and Objective C it is accessed within the procedure by a pointer with a reserved name. In 97-182, it is proposed that it would be associated to the first argument. Languages that use one argument in a syntactically distinguished position frequently explicitly describe invocation of primitive operations of a type as *sending a message to an object.*

Ada 95 allows a primitive operation of a type to have several arguments of the type, with the restriction that all polymorphic actual arguments that correspond to monomorphic formal arguments must have the same type at the instant of invocation.

Languages that use the "send a message" method usually restrict the polymorphic object whose type determines the primitive operation to be a named object, or a subobject of one. The method of Ada 95 allows the object(s) that determine dispatching to be arbitrary expressions. To allow this capability using the "send a message" method in Fortran, it would be necessary to generalize the conditions under which the component selection operator could be used.

The syntactic device tentatively proposed in 97-182, *viz.*

```
type, extends(base) :: extension
contains
  procedure foo => bar
```

indicates three things:

1. `foo` must be invoked as `E%foo(...)`, where `E` is of `type(extension)`. `E` is an actual argument that is associated with the first dummy argument, the first actual argument within() is associated with the second dummy argument, etc. This introduces two irregularities: only one "actual argument" can be bound by %, and actual-to-formal argument correspondence is different from other procedures. The first could be removed by allowing % to be a general right-to-left argument-binding operator, e.g. `a%b%c` invokes a procedure `c` with arguments `b` and `a`, *viz.* `c(b,a)`. This would be difficult and time-consuming due to potential ambiguities related to derived type component references.

The second could be removed by using the type name, in the procedure, to denote the "formal argument" associated to the "object in whose context" the procedure is invoked, or one could use, e.g. SUBROUTINE OBJ%B(...), to associate A to OBJ in CALL A%B(...), and to associate actual arguments between (...) in the call to formal arguments between (...) in the declaration in the same way as for non-type-bound procedures.

2. `bar` is the specific name for the over-riding or over-rideable procedure `foo`. If the Ada 95 method to determine primitive operations of a type were used, the Fortran generic interface block mechanism could be used, allowing overloading `foo` as well over-riding; the `foo =>` `bar` mechanism is, however, admirably terse.

3. `foo => bar` is a primitive operation of `type(extension)`. This could be indicated by interpretations of the juxtaposition of type and procedure declaration in the same (specification) module, and dummy argument types, as in Ada 95. If the `foo => bar` mechanism is retained to serve the second purpose, using it for the third is natural.

5 Abstract types and procedures

When constructing a base class from which one expects others to derive extension classes, for which one wishes to require certain operations, but those operations make no sense for the base class alone, it is useful to be able to declare that these operations do not exist, and must be implemented in an extension.

Similarly, it is sometimes useful to declare the data representation for a base class, but to declare that no objects of that class may exist.

In C++ and Ada, such operations and data types are called *abstract* (as distinguished from *concrete*).

Abstract procedures have no body – only an interface – and cannot be invoked.

Abstract data types may or may not have components, but no objects of the type may be created.

If a data type is abstract, all procedures that take it as an argument must also be abstract.

If an operation of a type is abstract, then either the operation must be defined when the type is extended, or the extension type must be abstract as well.

An extension data type, and any of its operations, may be abstract even if its base type is not.