GG

EUROPEAN ORGANIZATION FOR NUCLEAR RESEARCH

CERN–CN /95/1

February 2, 1995

Abstract Data Types in Fortran 90

SW 3508

Michael Metcalf

Computing and Networks Division, CERN

Submitted to *Computer Standards & Interfaces*

Abstract Data Types in Fortran 90

Michael Metcalf
CERN, Geneva, Switzerland
(metcalf@cern.ch)

Abstract

Progress in programming languages has been marked by successive waves of new techniques, each accompanied by miraculous claims from its proponents. Only after a certain time lag do these techniques become standardized. The functionality required to support one of these techniques, abstract data types (ADTs), is now part of the latest Fortran standard, but the newest wave is object orientation (OO). This paper describes the implementation of ADTs in Fortran and speculates on whether the language will withstand the OO onslaught.

1. Fortran and abstraction

It is now forty years since work began on the first Fortran compiler [1]. Fortran was the first high–level programming language but since then tremendous developments in programming techniques in procedural languages have occurred. The most recent Fortran standard [2][3] consolidates one of these developments, structured programming, and incorporates two others, data structures and abstract data types. An early paper [4] referred to an abstract data type as "a class of objects defined by a representation–dependent specification", with attributes "that specify the names and define the abstract meanings of the operations associated with an object". This is just how they are implemented in Fortran 90, following initial suggestions made by Lawrie Schonfelder of Liverpool University to the ISO Fortran standardization committee, meeting in Vienna in 1982.

In this paper we shall see how a derived–data type may be defined, and how such types can be packaged into modules together with a set of defined operations on objects of the types. We shall see how these, combined with function overloading, provide all we need for abstract data types. We shall examine examples and include complex types such as linked lists that require other new language features: pointers, recursion and dynamic memory allocation.

We shall conclude by noting that this round of standardization did not (and could not) take the final hurdle and incorporate the messaging and other features (inheritance, polymorphism) that conventional wisdom deems necessary for the full implementation of object–oriented programming. But the Fortran standards committees are giving very serious consideration to adding these to the next major Fortran revision, due in the year 2000.

2. Features for ADTs

2.1 Derived–data types

In order to use an object of a derived–data type we must first define the form of the type. For a very simple string data type, defining the effective length of the string in one component and the string's contents in a second, we might write

```
TYPE string
   INTEGER       length
   CHARACTER(80) string_data
END TYPE string
```

Now we can create structures of that type:

```
TYPE(string) str1, str2
```

To select components of a derived type object, we use the % qualifier:

```
str1%length = 7
```

The form of a literal constant of a derived type is shown by

```
str2 = string(7, 'Fortran')
```

which is known as a structure constructor. Each component is assigned the value in the corresponding position in the constructor.

2.2 *Structure–valued functions*

For an operation between derived–data types, or between a derived type and an intrinsic type, we must define the meaning of the operator. Given

```
CHARACTER    char1
TYPE(string) str1,  str2,  str3
```

we might wish to write

```
str3  = str1//str2        ! must define operation
```

where we note the use of the intrinsic operator symbol // between two objects of derived type. This requires us to define the exact meaning of the // symbol in this context, and this we do using a function like

```
FUNCTION string_concat(s1, s2)
   TYPE (string), INTENT(IN) :: s1, s2
   TYPE (string)                string_concat
   string_concat%string_data = s1%string_data(1:s1%length) // &
      s2%string_data(1:s2%length)
   string_concat%length      = s1%length + s2%length
END FUNCTION string_concat
```

that takes its two INTENT(IN) arguments as operands and returns the result of the operation as the function result. The function is of the same string type as its arguments.

The association between an operator and its corresponding function is made via an interface block. An example of an interface for this string concatenation is

```
INTERFACE OPERATOR(//)
   MODULE PROCEDURE string_concat
END INTERFACE
```

We could also write the statement above as

```
str3  = str1.concat.str2 ! must define operation
```

where we note the use of a named operator, *.concat.* . A difference is that, for an intrinsic operator token like //, the usual precedence rules apply, whereas for named operators their precedence is the highest as a unary operator or the lowest as a binary one. Thus, assuming the obvious definitions, in

```
vector3 =  matrix    *    vector1  + vector2
vector3 = (matrix .times. vector1) + vector2
```

the two expressions are equivalent only if the appropriate parentheses are added as shown.

2.3 Assignment

For assignment between two objects of the same derived–data type, as shown so far, assignment applies on a component–by–component basis (but can be overridden). However, between an object of one derived–data type and an object of a different derived or intrinsic type, we must define the meaning of the assignment. For

```
str3  = char1            ! must define assignment
```

we have to provide a subroutine with two arguments, one corresponding to the left–hand side and the other to the right–hand side of the assignment. This might be

```
SUBROUTINE c_to_s_assign(s, c)
   TYPE (string), INTENT(OUT)   :: s
   CHARACTER(LEN=*), INTENT(IN) :: c
   s%string_data = c
   s%length      = LEN(c)
END SUBROUTINE c_to_s_assign
```

The association between an assignment and its corresponding subroutine is also made via an interface block:

```
INTERFACE assignment(=)
   MODULE PROCEDURE c_to_s_assign
END INTERFACE
```

2.4 Intrinsic function overloading

Just as we overload the intrinsic operator symbol // to indicate concatenation of two objects of our string type, so we might wish to use the name of the intrinsic function LEN to extract the length of a string, as in

```
length = LEN(str3)
```

Here also we are able to use overloading. By use of an interface block

```
INTERFACE LEN
   MODULE PROCEDURE string_len
END INTERFACE
```

we indicate to a compiler that a function *string_len* should be invoked wherever LEN has an argument of type string.

2.5 Information hiding

To make a full–blown abstract data type we have to place the procedures defining the operators, assignments and overloaded function definitions in a module along with the type definition and all the interface blocks.

An example of a module containing the facilities we have so far introduced, as well as the definition of string–to–character assignment, follows (with a : standing for code that has already appeared). As already stated, this is for a very simple string type, of fixed maximum length and with no error recovery, and serves only as an example of the basic principles.

```
MODULE string_type
   TYPE string
      INTEGER length
      CHARACTER(LEN=80)   :: string_data
   END TYPE string
   INTERFACE assignment(=)
      MODULE PROCEDURE c_to_s_assign, s_to_c_assign
   END INTERFACE
   INTERFACE LEN
      MODULE PROCEDURE string_len
   END INTERFACE
   INTERFACE OPERATOR(//)
      MODULE PROCEDURE string_concat
   END INTERFACE
CONTAINS
   SUBROUTINE c_to_s_assign(s, c)
      :
   END SUBROUTINE c_to_s_assign
   SUBROUTINE s_to_c_assign(c, s)
      TYPE (string), INTENT(IN)     :: s
      CHARACTER(LEN=*), INTENT(OUT) :: c
      c = s%string_data(1:s%length)
   END SUBROUTINE s_to_c_assign
   FUNCTION string_len(s)
      INTEGER         string_len
      TYPE(string) :: s
      string_len = s%length
   END FUNCTION string_len
   FUNCTION string_concat(s1, s2)
      :
   END FUNCTION string_concat
END MODULE string_type
```

The user of this module simply has to add the statement

```
USE string_type
```

to a procedure to gain access to its facilities. However, in general we would want to hide as much of the internal details of a module as possible, and we can do this by making parts of the module private, as in

```
PRIVATE string_len
```

or by making the whole contents of the module private by default, and making just those entities public that are to be exposed to users. This high degree of information hiding is essential to achieve good abstraction: the details of operations and even types can be changed without user code being

affected, and the internal state of the module is protected from deliberate or inadvertent outside interference during execution (remember those awful COMMON variables!).

2.6 Pointers and dynamic allocation

It is possible in Fortran 90 to give an object the POINTER attribute, and such a pointer can be a component of a derived type:

```
TYPE entry
   REAL value
   TYPE(entry), POINTER :: next
END TYPE entry
```

where the type definition defines a data component, *value*, and a pointer component, *next*, that can point to the next entry in a linked list. We can define the beginning of a linked list of such entries:

```
TYPE(entry), POINTER :: chain
```

After suitable allocations and definitions, the first two entries could be addressed as

```
chain%value          chain%next%value
chain%next           chain%next%next
```

but we would normally define additional pointers to point at, for instance, the first and current entries in the list.

A pointer has an association status that is one of:

- undefined (initial state);
- associated (after allocation or a pointer assignment):

```
ALLOCATE(chain)          ! space allocation to pointer
first => chain           ! pointer assignment
```

- disassociated:

```
DEALLOCATE (chain)       ! for returning storage
NULLIFY (first)          ! for setting to 'null'
```

Some care has to be taken not to leave a pointer 'dangling' by use of DEALLOCATE on a target without NULLIFYing a pointer referring to it.

The intrinsic function ASSOCIATED can test the association status of a defined pointer:

```
IF (ASSOCIATED(chain)) THEN
```

or whether association exists between a defined pointer and a defined target (which may, itself, be a pointer):

```
IF (ASSOCIATED(first, chain)) THEN
```

We have just met the statement to allocate space to a pointer. We can use this statement also to allocate space to arrays as long as they have the POINTER (or ALLOCATABLE) attribute, and such a pointer array can appear as a component of a derived–data type. We can thus modify the example above to make *value* an array which, for each entry in the list, has a different, dynamically–defined

length. The data type becomes

```
TYPE entry
   REAL, DIMENSION(:), POINTER :: value
   TYPE(entry), POINTER        :: next
END TYPE entry
```

and for a given entry named *chain* we can allocate space to *value* thus:

```
ALLOCATE(chain%value(n))        ! n is a variable
```

2.7 Recursion

In an example we shall meet in the next Section, the traversal of a data structure will require that a procedure be able to call itself recursively. This too is possible in Fortran 90, by adding the RECURSIVE keyword to the procedure header line:

```
RECURSIVE FUNCTION finish(tree)
```

We now have all the elements we require to manipulate quite general data structures such as lists and trees.

3. Examples

Given the original initiative of Schonfelder, it came as no surprise that one of the first examples of an ADT was that for a varying character–string type proposed by him to become a standard Fortran 90 module [5]. Indeed, this proposal has since been developed and formally published as a standard [6], and a model implementation makes use of all the features so far described. A paper published in 1991 discussed how a histogramming package might be implemented in Fortran 90, with a histogram as a derived–data type [7]. A completely implemented example concerned with the construction of sets of electron configurations and their angular momentum couplings has been given by Scott et al. [8].

3.1 Interval arithmetic

An example using an arithmetic type is given in outline in [3]. This is for a data type for interval arithmetic, in which the type contains the absolute bounds on any given calculation. The basic type is

```
TYPE interval
   REAL lower, upper
END TYPE interval
```

and we can define the interface for the addition operator as

```
INTERFACE OPERATOR(+)
   MODULE PROCEDURE add_intervals
END INTERFACE
```

giving access to the corresponding function in a module:

```
FUNCTION add_intervals(a,b)
   TYPE(interval)                 add_intervals
   TYPE(interval), INTENT(IN) :: a, b
   add_intervals%lower = a%lower + b%lower    ! Production code would
   add_intervals%upper = a%upper + b%upper    ! allow for roundoff.
END FUNCTION add_intervals
```

This shows addition for two scalars. In the general case, a function must be provided for each rank or pair of ranks for which it is needed. For example, the module below provides summation for scalars and rank–one arrays of intervals:

```
MODULE interval_addition
   TYPE interval
      REAL lower, upper
   END TYPE interval
   INTERFACE OPERATOR(+)
      MODULE PROCEDURE add00, add11
   END INTERFACE
CONTAINS
   FUNCTION add00 (a, b)
      TYPE (interval)                 add00
      TYPE (interval), intent(in) :: a, b
      add00%lower = a%lower + b%lower          ! Production code would
      add00%upper = a%upper + b%upper          ! allow for roundoff.
   END FUNCTION add00
   FUNCTION add11 (A, B)
      TYPE (interval), DIMENSION(:), INTENT(IN)       :: a
      TYPE (interval), DIMENSION(SIZE(a))             :: add11
      TYPE (interval), DIMENSION(SIZE(a)), INTENT(IN) :: b
      add11%lower = a%lower + b%lower          ! These are whole array
      add11%upper = a%upper + b%upper          ! assigmnents.
   END FUNCTION add11
END MODULE interval_addition
```

where, once again, the appropriate function is selected by the compiler depending on the rank of the operands. In a similar fashion, elemental versions of defined assignments must be provided explicitly. Such a module might be further extended to provide, for instance, overloaded mathematical functions like SIN and COS.

3.2 Tree structure

As a second example we take that of a tree structure where, as an experiment, a module providing many of the facilities required to support and manipulate tree structures has been written. It is called *eagle* and its use in a physics experiment has been reported in [9]. For the purposes of this paper, a tree data structure consists of a set of connected nodes, arranged in levels. The top node points to some nonzero number of nodes at the second level. All other nodes point at zero or more nodes at the next lower level, and point back to exactly one node at the next higher level. The set of nodes pointed at by a single higher node is referred to as a layer. This is thus a standard tree of mother nodes each connected to a set of daughter nodes.

Internally, the module allocates a node, and allocates data at that node if the corresponding argument is present in the call to *new_node*, based on the basic node definition. At each node are stored, as far as the user is concerned:

- an index number supplied by the user,

- a pointer to its mother node,
- an optional fixed character component,
- an optional integer array,
- an optional real array,
- and the optional pointers to nodes with specified index numbers (the daughter nodes).

The node contains also various internal variables, such a running index number maintained by the module. The corresponding type definitions are firstly for a pointer data type for use to create a pointer array:

```
TYPE ptr
   TYPE(data), pointer :: pp
END TYPE ptr
```

and secondly for the basic node data type:

```
TYPE, PUBLIC :: data
PRIVATE
   INTEGER                 index, amount(3), running_index
   CHARACTER(max_char)     header
   INTEGER, POINTER     :: j(:), link(:)
   REAL, POINTER        :: y(:)
   TYPE(ptr), POINTER   :: p(:)
   TYPE(data), POINTER  :: back
   TYPE(state), POINTER:: own_state
END TYPE data
```

where the pointer array *p* holds the dynamically defined pointers actually required at each node. Note that the internal structure of the type is inaccessible outside the module.

The module supports an arbitrary number of independent trees; they may be manipulated simultaneously.

A single reference pointer to another node (even in a different tree) may also be stored at each node. A reference pointer is one between any two nodes and is not part of the tree structure as such.

The user interfaces are:

start must be called to initialise a tree immediately before the first call to *new_node* for that tree.

new_node stores the data provided at the node whose index number is supplied as a second argument, and sets up pointers to all the specified daughter nodes that will be stored in subsequent calls.

retrieve retrieves a specified node; the data arrays (if present) are accessed via pointers, as are the pointers to any daughter nodes.

next like *retrieve*, for the node with the next following running index number.

next_in_layer like *next*, for the next node in the current layer.

previous retrieves the data in the mother node of the most recent node accessed.

dump_tree writes a complete tree to a specified unit.

restore_tree reads a complete tree from a specified unit.

set_reference establishes a reference pointer between a node of one tree and a node of the same or another tree.

get_reference like *retrieve*, but for the data at the node that is the target of the reference pointer at the specified node.

finish deallocates all the storage occupied by a complete tree.

These interfaces are simple. To add a new node to a tree, with a name, some integer data, and pointers to three daughter nodes, we write, for example,

```
CALL new_node(tree_name, index, node_name,                          &
     integer_data = (/ (i, i = 1, 10) /), links = (/ 2, 3, 5/))
```

and to retrieve data we write

```
CALL retrieve(tree_name, index, back, name, j, y, link)
```

where *back* is the index of the mother of the node index, and *j*, *y*, and *link* are pointers to any optional integer data, real data or pointers stored at that node. These names are chosen by the user, and retrieved data can be referred to directly, say as *j*(16).

The code consists of less than 800 lines of Fortran 90. This demonstrates one of the remarkable features of the language – the succinct way in which it can be used to express algorithms. As an example, the complete code to traverse a tree deallocating all its associated storage is simply:

```
   RECURSIVE SUBROUTINE finish (tree)
!
! Traverse a complete tree or subtree, deallocating all storage
      TYPE(data), POINTER :: tree
      INTEGER loop
!
      DO loop = 1, size(tree%p)          ! loop over children
         CALL finish (tree%p(loop)%pp)   ! delete all their subtrees
      END DO
      DEALLOCATE(tree%j, tree%y, tree%p, tree%link)
      DEALLOCATE(tree)
   END SUBROUTINE finish
```

Future plans might be to add the following features: replace a node; add a node; extend a component; allow an error exit if insufficient storage is available for allocation for dynamic memory; and further validation and navigation facilities. The existing code is available on request to the author.

4. Fortran and the future

Fortran has always had a slightly (sometimes even decidedly) old–fashioned image. In the 1960s, the block–structured language ALGOL was regarded as superior to Fortran. In the 1970s the more powerful PL/1 was expected to replace Fortran (and COBOL). ALGOL's successors Pascal (with its emphasis on safety) and ADA (with its ADTs) caused Fortran proponents some concern in the 1980s. Meanwhile, it ploughed on as the workhorse of scientific computing. However, in the late 1980s, two developments did begin seriously to affect Fortran's predominance in this field – UNIX and object orientation – and this trend was certainly accentuated by the uncertainty that surrounded the new standard at the end of that decade.

UNIX brought with it the now highly–successful general–purpose language C. This language has been further developed by Bjarne Stroustrup into C++, an object–oriented language influenced by Smalltalk. C came as a part of most UNIX systems and is widely used for all levels of systems programming. Its simplicity has meant also that it has made inroads into Fortran's traditional numerical computing community, although attempts to incorporate meaningful numerical features into the language are apparently making little headway (Jones [10]). Its standard lays less stress on portability than does Fortran's.

C++, with its fully fledged OO features is widely viewed as a proper superset of C, but in fact this is not the case, and Jones points out that potential problems of a lack of compatibility between the existing (and future) C and C++ standards are beginning to emerge. He quotes also an opinion that C++ is getting too complicated and will fall apart under its own weight. As a language, C++ is especially strong on those features required to handle effectively graphical interfaces, where objects really come into their own. But as a numerical language, Stroustrup himself, quoted by Woodyard and Mills [11], writes *"In some areas, such as interactive graphics, there is clearly enormous scope for object–oriented programming. For other areas, such as classical arithmetic types and computations based upon them, there appears to be hardly any scope for more than data abstraction and the facilities needed for the support of object–oriented programming seem unnecessary"*. This might be even more true given the existence of the array as a first–class object in Fortran 90, although with respect to abstraction (as opposed to numerical features), Woodyard and Mills regret the absence of a method for specifying the precedence of defined operators in both languages (but note the module as an advantage for Fortran 90).

C++, available widely also on PCs, has begun to dominate many programming applications, especially those based on windowing, but still does not benefit from having a recognised standard, something that writers of non–trivial applications are beginning to find a disadvantage. De Morgan [12] reports on the difficulties that the C++ standards committees, X3J16 and WG21, face in trying to produce an agreed document (as does Jones too). However, despite these formal obstacles, many scientists, whose work requires not only numerical calculations but also suitable interfaces to their programs, are investigating the use of C++ for all their work. (A survey I conducted on the Internet news group comp.lang.c++ in 1994 failed, however, to turn up many finished products.) On the other hand, large–scale computations still seem to be performed in Fortran, and here a supplementary standard, High–Performance Fortran, has been established by a consortium of academia and industry [13][14]. The Fortran committees, X3J3 and WG5, have set themselves the task not only to add similar powerful features to the next major language version, Fortran 2000, but also to add the OO features deemed vital to compete with OO languages, and especially C++, on their home ground. Whether this plan will succeed only time can tell, but the final irony may be that Fortran 90, once criticized for its size, will resist inroads from C++ because that language is seen to be larger still. In the meantime, what would be most helpful to users would be a standardized method of communication between languages, for both procedures and I/O, enabling numerical code in Fortran to be used via interfaces in C++. Could this be one of the standards community's most important tasks for the next decade?

Bibliography

1. M. Metcalf, *FORTRAN programming language,* in Encyclopedia of Physical Science and Technology, Academic Press, Orlando, 1986, 1992.

2. ISO, *ISO/IEC 1539 : 1991,* ISO, Geneva, Switzerland.

3. M. Metcalf and J. Reid, *Fortran 90 Explained,* Oxford University Press, Oxford and New York, 1990.

4. J. Guttag, *Abstract data types and the development of data structures,* Comm. ACM, 20, 6, 1977, 396–404.

5. J.L. Schonfelder and J.S. Morgan, *Dynamic strings in Fortran 90,* Software – Practice and Experience, 20, 12, 1990, 1259–1272.

6. ISO, *ISO/IEC 1539–2 : 1994,* ISO, Geneva, Switzerland.

7. M. Metcalf, *A derived–data type for data analysis,* Computers in Physics, Nov. 1991, 599–604.

8. N.S. Scott et al. *The formal specification of abstract data types and their implementation in Fortran 90,* Computer Physics Communications, 84, 1–3, 1994, 201–225.

9. M. Metcalf and R. Windmolders, *A novel use of Fortran,* CN/94–6, CERN, Geneva, Switzerland.

10. D. Jones, *The programming language standards scene: C,* Computer Standards & Interfaces, 16, 5&6, 1994, 495–503.

11. M. Woodyard and A. Mills, *A contrast of Fortran 90 and C++ abstract data types for scientific computing,* Fortran Journal, 6, 5, 1994, 7–10.

12. R. M. De Morgan, *The programming language standards scene: C++,* Computer Standards & Interfaces, 16, 5&6, 1994, 531–535.

13. HPFF, *High–Performance Fortran Language Specification,* Rice University, Houston, Texas, 1993, (see also Scientific Programming, Vol. 2, 1993).

14. C. Koebel et al., *High–Performance Fortran Handbook,* MIT Press, Cambridge, 1993.