EUROPEAN ORGANIZATION FOR NUCLEAR RESEARCH

CERN – CN /90/24

October 23, 1990

A derived-data type for data analysis

[redacted]hael Metcalf

Computing and Networks Division, CERN

To be submitted to *Computers in Physics*

A DERIVED-DATA TYPE FOR DATA ANALYSIS – A CASE STUDY

Michael Metcalf

CERN, Geneva, Switzerland

Abstract

A frequent requirement in physics data analysis is the ability to form and manipulate histograms; this is done currently using calls to the HBOOK subroutine package. Could its interfaces be improved by instead defining a histogram to be a Fortran data type?

1. Histogramming in FORTRAN 77

One of the most commonly occurring operations in high-energy physics data analysis is the formation, manipulation, representation, and storage and retrieval of histograms. For some time now, a single package of subroutines, HBOOK[1], has successfully dominated this application domain. It is, of course, a product of its times, and relies on many FORTRAN 77 features which find diminishing acceptance in computer science circles. Not the least of these is storage association. To use HBOOK we start by defining a working space in a COMMON block:

```
COMMON/PAWC/HMEMOR(20000)
```

and the whole of the underlying data structure is open to deliberate or inadvertent read and write access.

Next, it makes use of static storage. We have to tell it how much space we have defined; (in order not to get 20000 wrong, let's use a symbolic constant):

```
CALL HLIMIT(LENGTH)
```

Some non-standard systems may let us extend this area later, but that is not generally possible.

Now we tell HBOOK we want to build a histogram with certain characteristics (rank, title, number of channels, channel edges), and that appears thus:

```
CALL HBOOK1(ID, CHTITL, NX, XMI, XMA, VMX)
```

and here we must make sure all the arguments are given, and in the right order. During the execution of our program, we fill the histogram using a succession of calls:

```
CALL HFILL(ID, X, Y, WEIGHT)
```

and possibly write or read histograms to or from an external file. Finally, we print the result:

```
CALL HPRINT(ID)
```

In a more complicated application, we might have wanted to form, say, the sum of two histograms, and once again a subroutine call is our interface:

```
CALL HOPERA(ID1, CHOPER, ID2, ID3, C1, C2)
```

where C1 and C2 are scaling factors applied to the contents of ID1 and ID2, respectively, before the result is placed in ID3. CHOPER defines the actual operation; here it is addition.

2. A Fortran 90 approach

Modern programming increasingly involves the few who write libraries and the many who use them. That is how packages such as HBOOK arose and became so successful. Imagine that our library writer is working in the near future with a Fortran 90[3] compiler and run-time system available, and wants to reinvent this particular wheel. How would he or she go about it? To start off with the last operation above, it would be very convenient and expressive if the addition of two histograms could simply be written as

```
H(k) = c1*H(i) + c2*H(j)        ! Scale and add two histograms
```

where *H(i)* etc. are elements in an array of objects of type histogram. Well, this can be done, as long as we have a definition of a one-dimensional histogram type available. In Fortran 90 it might look like this:

```
TYPE hist_1
   INTEGER          :: id              ! Identifier
   CHARACTER*80     :: title           ! Name
   INTEGER          :: n_channel       ! Number of channels
   REAL             :: lower_edge, upper_edge
   INTEGER, DIMENSION(100) :: bins     ! Histogram contents
   REAL, DIMENSION(100)    :: weights  ! Bin weights
END TYPE hist_1
```

(This definition sweeps a few problems under the carpet, and is just to get us going.) All that is now needed to make the statement above apparently correct is to define the array variable *H* to be of type *hist_1*:

```
TYPE(hist_1), DIMENSION(20) :: H
```

However, since *H* is a derived-data type, not an intrinsic one with its predefined operations, the statement finally becomes valid only if we define (overload) also the * operator in terms of 'real times histogram' arithmetic, and the + operator in terms of 'histogram plus histogram' arithmetic. (The assignment operator, =, has an assumed and obvious definition, but we have the possibility, optionally, to define that too.)

The way we define these operators is to write a function with two operands that performs the required operations and whose value is the final result. Before looking at an example, we recall that the way an individual component of a derived-data type is referenced is by use of the % qualifier: to reference the *j*th element of *bins* in

element *i* of *H*, we write

```
H(i)%bins(j)
```

Thus, a function to multiply a histogram by a real value might be

```
FUNCTION c_times_h(c, h)
   TYPE(hist_1) c_times_h, h
   REAL c
   c_times_h%bins = c*h%bins      ! Array operation and assignment
END FUNCTION c_times_h
```

and the compiler has to be informed that this operation is available by placing the statement

```
MODULE PROCEDURE c_times_h
```

inside an *interface block* specifying the association between the operator and the function:

```
INTERFACE OPERATOR(*)
   MODULE PROCEDURE c_times_h
END INTERFACE
```

We note here that it is possible to define our own named operators for use with derived types; they have the form *.name.* . However, extending the intrinsic operators in natural ways has the advantage of retaining the familiar rules of precedence in the evaluation of expressions. Thus, in the example, both 'multiplications' will be carried out before the 'addition', without having to force this order of evaluation through the use of parentheses.

The MODULE PROCEDURE statement gives the clue as to how all this is packaged. The data type definition and all the associated interface blocks and operation definitions are placed by the library writer inside a module:

```
MODULE hbook
   TYPE hist_1
      INTEGER          :: id              ! Identifier
      CHARACTER*80     :: title           ! Name
      INTEGER          :: n_channel       ! Number of channels
      REAL             :: lower_edge, upper_edge
      INTEGER, DIMENSION(100) :: bins     ! Histogram contents
      REAL, DIMENSION(100)    :: weights ! Bin weights
   END TYPE hist_1
   INTERFACE OPERATOR(*)
      MODULE PROCEDURE c_times_h
   END INTERFACE
CONTAINS
   TYPE(hist_1) FUNCTION c_times_h(c, h) OPERATOR(*)
      REAL C
      TYPE(hist_1) h
      c_times_h%bins = c*h%bins      ! Array operation and assignment
   END FUNCTION c_times_h
END MODULE hbook
```

and in any subroutine requiring access to the histogram data type the user simply adds

```
USE hbook
```

to the specifications, and the type and its operations are made available. Note that within the module the subprograms defining operations (and assignment where appropriate) follow a CONTAINS statement. They are called *module procedures* and can reference one another and can be referenced as shown in the example above.

Just as in the current HBOOK there are some undocumented entries used internally in the package, so in a module we can write a private operation or procedure accessible only from within the module. Thus, for a private operator *.private_op.*, the library writer just adds, for example,

```
PRIVATE OPERATOR(.private_op.)
```

to the specification statements in the module, and the corresponding operation is accessible only from within the module. This is a better level of protection than we have now, as an undocumented entry in a current package can still be accessed by a user, whereas that is not the case for operations or procedures declared to be PRIVATE in a module.

So far, we have looked only at how a histogram data type might be specified and used as a whole entity. We need, of course, to go back a level and consider how the user actually fills the histogram. The simplest way would be something like

```
H(i)%bins(j) = H(i)%bins(j) + 1        ! Increment bin count
```

and so on for the weights. However, these operations clearly need to be packaged as well, as we would otherwise be worse off than we are with the existing interface. The module itself must calculate the bin number *j* and increment everything in a correct and consistent way, and the data must be hidden from the user. This hiding we do by adding a PRIVATE statement to the data type:

```
TYPE hist_1
   PRIVATE
   :
   :
END TYPE hist_1
```

(Fortran 90 allows us to make even the type itself private, but that is useful only for a data type whose use is exclusively within the module.) For a derived-data type as inhomogeneous as *hist_1*, the most sensible approach is probably to retain a subroutine call as a way to increment an entry:

```
CALL increment(H(i), x, w)
```

However, we note that in a simpler case another approach would be to use a defined operator, as before. We illustrate this by an example from interval arithmetic: given the type

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

and a statement such as

```
a_interval = b_interval + c_real    ! Interval plus real arithmetic
```

we can write an interface block and add a function to perform the necessary operations:

```
INTERFACE OPERATOR(+)
   MODULE PROCEDURE add_interval_real
END INTERFACE
:
FUNCTION add_interval_real(a, b)
   TYPE(interval) add_interval_real, a
   REAL           b
   add_interval_real%lower = a%lower + b
   add_interval_real%upper = b%upper + b
END FUNCTION add_interval_real
```

Indeed, we could adopt this sort of solution in the case of HBOOK. By defining, for instance, a new entry to be a type:

```
TYPE entry
   REAL point, weight
END TYPE entry
```

we could develop an appropriate add function. In order not to abuse the + symbol, we might, in this case, prefer to use a named operator. The interface then becomes

```
MODULE PROCEDURE OPERATOR(.increment.)
```

and a statement to update a histogram becomes

```
H(i) = H(i).increment.entry(x, w)        ! Method 1
```

By defining a suitable assignment subroutine, we could even write the confusing statement

```
H(i) = entry(x, w)                       ! Method 2
```

In both cases, the name of the type, *entry*, is used to form a *structure constructor* corresponding to the type. This is rather like a template for the type, and is the way in which constants of a derived type are formed:

```
TYPE(entry) thing
thing = entry(2., 3.)
```

The function *increment* would contain the same basic code as the existing subroutine HFILL. The disadvantage of the first method is that the histogram in question has to be referenced twice in the same statement, and a new name, here *entry*, is added to the name space. The second method involves overloading assignment to perform

both addition and assignment which, although perfectly valid, some might find surprising.

Both methods have the advantage of a protected interface in terms of the number, type and intent of the operands (whether they are input, output, or both). However, this protected interface can anyway be provided by the library writer by making the interface of the subroutine visible, or *explicit*, to the user. If subroutine *increment* is in the *hbook* module, this is automatic. If, for organizational reasons, it is a separate module, the *hbook* module has to contain just the interface of *increment*. This would look like

```
INTERFACE
   SUBROUTINE increment(H, point, weight)
      TYPE(hist_1), INTENT(IN) :: H
      REAL, INTENT(IN)         :: point, weight
   END SUBROUTINE increment
END INTERFACE
```

If the weight assigned to a point could have the default value of 1.0, *weight* could additionally be declared to be optional:

```
      OPTIONAL :: weight
```

and its presence in a particular call, for instance,

```
      CALL increment(H(i), x)
```

established by a call to the intrinsic function PRESENT:

```
      IF (PRESENT(weight)) THEN
```

Which ever way it is done, never again need programs fail because library code is called with incorrect arguments – this error is detected at compile-time.

3. Dynamic storage

Before rolling back in the sequence of events one further step, to booking a histogram, we must note a restriction that we have so far placed upon ourselves but ignored: the definition of the *hist_1* structure contains a fixed maximum number of bins, and the variable *n_channel* could just as well be a named (symbolic) constant. Our storage is static, and fixed for all the histograms declared to be of type *hist_1*. So we do slightly better by specifying *n_channel* outside the type definition,

```
   INTEGER, PARAMETER :: n_channel=100
```

and defining the number of bins etc. in terms of this value:

```
   INTEGER, DIMENSION(n_channel) :: bins
```

Still all histograms have the same maximum size.

Another possibility, to specify types with a variable parameter:

```
TYPE(hist_1(no_of_bins_required)) H(20)      ! Invalid
```

was removed from Fortran 90 as part of the great 'compromise' back in 1986, so that method of specifying sets of histograms, each set of a different size, is not available, although one imagines some vendors providing this as an extension (which is why it is mentioned here).

What we really want is the completely dynamic structure that HBOOK obtains now by using the memory manager, data structuring and I/O subroutine package ZEBRA[2]. The key to this is the POINTER attribute and the dynamic ALLOCATE statement, the latter giving access to the run-time heap storage which a computer system must provide under Fortran 90. The following section briefly describes these features.

3.1 The POINTER attribute

Pointers have been added to Fortran in the form of an *attribute*, (rather than as a separate data type). The specification of an object with the pointer attribute designates a descriptor of the object, rather than the object itself. No storage is reserved – it has to be allocated explicitly.

A pointer may reference any dynamic object, such as an array whose actual bounds are defined at run-time, or any static object defined with the TARGET attribute. The use of the name of the pointer is, in most contexts, such as expressions or I/O lists, interpreted as a reference to the contents of the target. (This implies that pointers themselves cannot be written and read – their targets become the list items.) This use is known as *automatic dereferencing, i.e.* no explicit notation using a pointer symbol is required. This has the advantage that existing code can readily be modified to use pointers, as only the specification statements need to be changed. Simple examples are:

```
INTEGER, POINTER              :: count  ! May point to an integer scalar
INTEGER, DIMENSION(:), POINTER :: bins ! May point to a 1D integer array
:
ALLOCATE(bins(100))                     ! Allocate storage to bins
:                                       ! Define parts of bins
bins(1) = bins(1) + 1                   ! The target values are used
```

Pointers are strongly typed, that is, they may point only at objects of the same type and type parameters as the ones with which they themselves are defined.

Sometimes, of course, it is necessary to change the target of a pointer. For this case, a *pointer assignment* is foreseen. An example is shown in

```
REAL, DIMENSION(:,:), POINTER :: a, b, c
:
ALLOCATE (a(10,10), b(10,10)) ! Give actual space to a and b
:                             ! Define a and b
c => a                        ! c now points to the same array as a
DO i = 1, 2
   IF (i == 2) c => b         ! Change target of c, no copy of data
:                             ! On first pass, c refers to a;
:                             ! on second pass to b.
:                             ! Operations on c
END DO
:
```

Pointers may be components of a structure and of the same type as the structure. In other words, a recursive definition is allowed which enables structures representing lists, trees and graphs to be defined. An example is

```
TYPE pixel
   REAL x, y
   CHARACTER*10 colour
   TYPE (pixel), POINTER :: next
END TYPE pixel
```

Given a variable *list* of type *pixel*, we can write an indefinitely long sequence like

```
list%colour = 'red'
ALLOCATE(list%next)
list%next%colour = 'blue'
ALLOCATE(list%next%next)
list%next%next%colour = 'green'
:
```

For long lists it is, of course, simpler to keep a pointer to the current entry and traverse the list using a DO construct.

Pointers become associated with a target by execution of a pointer assignment or of an ALLOCATE statement. They become disassociated by, in general, execution of a NULLIFY statement. Storage occupied by a target may be returned to the processor by execution of a DEALLOCATE statement; this can result in a pointer being left 'dangling', and programmers must guard against their further use until they are newly associated. An intrinsic inquiry function, ASSOCIATED, indicates whether an association exists between a pointer and its target, or whether a pointer is associated at all.

Pointers may be used for static objects. In order that this should not inhibit optimization, any static object which might become a pointer target must be declared with the TARGET attribute. The compiler then knows which assumptions it can and cannot make about references to static objects.

3.2 Pointers for hbook

Given this information, we can now see how to construct a truly dynamic histogram entry. The definition of *hist_1* has to be modified such that those components of variable length are defined as pointers to an area of dynamic storage which is defined when it is needed. Before that definition, they are just dope vectors pointing nowhere, *i.e.*, undefined:

```
TYPE hist_1
   PRIVATE
   INTEGER          :: id              ! Identifier
   CHARACTER*80     :: title           ! Name
   INTEGER          :: n_channel       ! Number of channels
   REAL             :: lower_edge, upper_edge
   INTEGER, DIMENSION(:), POINTER :: bins    ! Histogram contents
   REAL,    DIMENSION(:), POINTER :: weights ! Bin weights
END TYPE hist_1
```

Thus equipped, the booking of a histogram might look much as it does now:

```
CALL hbook1(id, chtitl)
```

although with the improvement that trailing arguments for which defaults are provided and accepted are omitted. Inside *hbook1* we would expect to find the allocation of the actual storage for *bins* and *weights*:

```
ALLOCATE(H(i)%bins(n_channel), H(i)%weights(n_channel), STAT=status)
```

3.2.1 Managing the histograms: method I

Finally, we need a way by which a set of *n_hist* histograms can be managed. In particular, the mapping of the IDs of individual histograms onto their serial numbers is a task that must be undertaken by the library writer and not by the user. (An ID is not limited to the range 1 to *n_hist* – it may assume any legal integer value.) One way to do this would be to have a an array of pointers, one element to each histogram, but arrays of pointers are not allowed in Fortran 90. However, we can circumvent this difficulty by defining an array of objects of a data type whose only component is a pointer:

```
TYPE h_pointer
   TYPE(hist_1), POINTER :: ptr
END TYPE h_pointer
```

and the initial code to set up these pointers to dynamic storage might be

```
TYPE(hist_1),    DIMENSION(:), ALLOCATABLE, TARGET :: H
TYPE(h_pointer), DIMENSION(:), ALLOCATABLE         :: pointers
READ (*, *) n_hist     ! Obtain the number of histgrams for this run
ALLOCATE(H(n_hist), pointers(n_hist), STAT=status) ! Note error code
```

where the arrays *pointers* and *H* are specified with the ALLOCATABLE attribute to allow their storage to be defined dynamically. (This attribute provides a mechanism to define dynamic storage in simple situations where pointers are inappropri-

ate.) In order to make the *i*th pointer point to the corresponding histogram, the library writer can simply write

```
pointers(i)%ptr => H(i)
```

and any sorting, for instance, of the histograms can be carried out just by reordering the values of these pointers:

```
ptemp       = pointers(i)        ! ptemp of type h_pointer
pointers(i) = pointers(j)
pointers(j) = ptemp
```

where we note that ordinary, not pointer, assignment is specified because structures, not pointers, are being manipulated. The fact that the only component involved is a pointer causes pointer assignment to occur at that level. An example of an algorithm to solve exactly this sorting problem is given on pp. 280-282 of [4]. As histograms are booked, a check must be kept as to whether they are in order. If not then, before an operation which requires them to be manipulated in ID rather than serial order, the sort algorithm would have to be invoked. This sort is very fast, as only pointers, not data, are manipulated.

3.2.2 Managing the histograms: method II

The method just described works very well if we know in advance the number of histograms we have to deal with: *n_hist*. This is, of course, not always the case, and is anyway a poor model for more general data types. What we really would like is the ability to add new histograms to an existing set. This can be achieved by storing the histograms as members of a linked list. As each histogram is booked, a new member of the list is created, new storage is allocated for the number of bins requested, and it is linked in as the last member of the list. Outline code to do this is modelled on an example given in Annex C of DIS 1539, the draft Fortran 90 standard:

```
TYPE hist_                              ! Define a recursive type
   :
   INTEGER, POINTER, DIMENSION(:)  :: bins
   TYPE(hist_1), POINTER           :: next_hist
END TYPE hist_1

TYPE(hist_1), TARGET   :: head
TYPE(hist_1), POINTER, :: current, temp       ! Declare pointers
INTEGER :: ieom

current => head                         ! current points to head of list
NULLIFY(current%next_hist)              ! Pointer to rest of empty list
                                        ! is disassociated
DO
   READ (*, *, IOSTAT=ieom) n_channel ! Read number of bins, if any
   IF (ieom /= 0 ) EXIT
   ALLOCATE(temp)                       ! Create new cell each iteration
   ALLOCATE(temp%bins(1:n_channel)) ! Create new bins
   temp%bins = 0                        ! Set all bins to zero
   NULLIFY(temp%next_hist)              ! Set status to disassociated
   current%next_hist => temp            ! Attach new cell to list
   current => temp                     ! current points to new end of list
END DO
```

We now have a list whose last member contains a disassociated pointer. Both the length of the list and the size of the contents of individual entries are dynamic. We can traverse the list with a loop, for instance to find the size of each histogram:

```
current => head
DO
   IF(.NOT.ASSOCIATED(current%next_hist)) EXIT
   current => current%next_hist
   WRITE(*, *) current%n_channel
END DO
```

and the value of *n_channel* would have to be used inside defined operator and assignment procedures, where appropriate, to ensure that no inadvertent attempt is made to manipulate histograms of unequal length.

Now we have arrived at the full generality enjoyed by the present HBOOK for simple, 1-dimensional histograms, but without recourse to a memory management package – the run-time system does it for us. In addition, the library writer can write all the code to manipulate the components of the histograms using meaningful identifiers, rather than in terms of obscure quantities such as Q(L+N).

3.3 Input/Output

A major advantage of ZEBRA is, of course, its very sophisticated I/O capability, in particular its ability to read and write completely linked structures, even between two different computers. This cannot be directly compared to the features of Fortran 90 for two reasons. Firstly, as pointers are defined as an attribute rather than a data type, any reference to a pointer in an I/O list is an implicit reference to the target and not to the pointer itself. Secondly, language standards do not deal

with the thorny topic of machine-independent I/O. So, at least for this second reason, some subset of ZEBRA would still be required.

As far as I/O of a data type such as *hist_1* is concerned, part of the *hbook* module would have to contain the code necessary to perform the I/O of individual histograms. By using the *id* as an index[1] into a direct-access file, the system becomes responsible for storing and retrieving individual histograms:

```
WRITE(unit, NREC=H(i)%id) H(i)%title, H(i)%n_channel, ...,    &
                          H(i)%bins(1:maximum), ...
```

where we note the fact that the number of bins written each time must be the maximum possible rather than the actual number booked.

4. Conclusions

The purpose of this paper has been to try to demonstrate that Fortran 90 provides a usable derived-data type facility, using as an example a non-trivial data type which is readily understood in the world of HEP data processing. There has been no intention to propose that the existing HBOOK be replaced at some future time in this way, and indeed the only two changes in the existing interface which I would propose would be to provide interface blocks to all callable entries, so that all calls can be checked at compile-time, and to reintroduce optional arguments wherever possible.

It is quite possible that, if Fortran 90 is successfully introduced, a large library of derived-data types for use in physics analysis could be built up, and it will be very important to have a set a guidelines and models on how this could be achieved. This paper is just another step, following those of Wampler [5] and Schonfelder and Morgan[6].

Acknowledgements

René Brun first suggested that I investigate this subject; I thank F. Carminati and D. Myers for their careful reading of drafts of this paper, and J.L. Schonfelder for some helpful suggestions.

---

[1] In practice, the value of *id* might have to hashed into a limited range.

Bibliography

1. R. Brun and D. Lienart, *HBOOK User Guide,* CERN Program Library, Y250, 1987.

2. R. Brun and J. Zoll, *ZEBRA User Guide,* CERN Program Library, Q100, 1987.

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

4. W.S. Brainerd, C.H. Goldberg and J.C. Adams, *Programmer's Guide to Fortran 90* (McGraw-Hill, New York, 1990).

5. K.D. Wampler, *Computers in Physics,* **4**, 4, July/August, 1990, 385-389.

6. J.L. Schonfelder and J.S. Morgan, *Dynamic strings in Fortran 90*, to appear in *Software – Practice and Experience,* 1990.

Imprimé au CERN. Prix : 0.70 Fr.