DYNAMIC ARRAYS VIA A MODIFIED RATFOR PREPROCESSOR

by

W. T. Nye

Memorandum No. UCB/ERL M80/23

4 June 1980

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

DYNAMIC ARRAYS VIA A MODIFIED RATFOR PREPROCESSOR

by

W. T. Nye

I. Introduction.

In many programming situations, there are needs for arrays whose sizes are not fixed, i.e., not known at compile time. A simple scheme is presented here in which not only is the memory allocation transparent to the user, but the dynamic arrays used in the program appear and are used just like ordinary Fortran arrays. These features exist in a modified Ratfor [1] preprocessor which was originally obtained from the CSAM department at Lawrence Berkeley Laboratory (who obtained it from "Software Tools"). This means that only Ratfor source programs can enjoy the dynamic array features described in this paper. However, the Ratfor language enjoys the same portability as Fortran since (1) the Ratfor preprocessor generates portable Fortran from Ratfor source, and (2) the Ratfor preprocessor itself is quite portable, subject to the implementation of a very few low level primitives such as ones for file opening and closing.

The following simple example shows the use of dynamic arrays in a Ratfor program:

```
## main program.

    integer mem(1000)             # one big memory array.
    common mem

    call setmem (mem, 1000, 5)    # initialize dynamic
                                  # memory; don't keep free
```

```
                                   # blocks smaller than 5.
call easy
call harder
stop
end

subroutine easy

dynamic real xary
integer i

call getmem (50,xary)    # allocate dynamic
                         # array xary.
do i = 1 , 50
   xary(i) = 0.0
 .
 .
 .
call clrmem (xary)       # clear (eliminate)
                         # dynamic array xary.
return
end

## harder - allocates 3 dynamic arrays then keeps
## reading in an unknown quantity of triples
## until reaching the end-of-file.

subroutine harder

dynamic real xary, yary
dynamic integer iary
integer i, sizmem

call getmem (20,xary)
call getmem (20,yary)
call getmem (20,iary)

i = 0

repeat  {
   i = i + 1
   if ( i > sizmem(xary) )  {    # then extend
      call extmem (10,xary)      # dynamic arrays.
      call extmem (10,yary)
      call extmem (10,iary)
      }
   read (5,1,end=2)  xary(i), yary(i), iary(i)
 1 format (2f5.2, i5)
   }

2 continue
```

```
          .
          .
          .
          call clrmem (xary)
          call clrmem (yary)
          call clrmem (iary)

          return
          end
```

The preprocessor output for subroutine easy above follows:

```
          subroutine easy
          integer xary
          integer izi(1)
          real rzr(1)
          double precision dzd(1)
          complex czc(1)
          common izi
          equivalence (izi(1),rzr(1),dzd(1),czc(1))
          integer i
          call getmem(50,xary,1)
          do 23000 i=1,50
          rzr(xary+i)=0.0
    23000 continue
          call clrmem(xary,1)
          return
          end
```

This example shows that dynamic arrays can be allocated and extended to any size that fits in the large array provided in the main program, at any time in the program. Then, after an array is cleared, it is completely gone and no longer uses up space in the large array. The 'dynamic real' statement tells the Ratfor preprocessor that xary and yary are dynamic arrays and of type real, as opposed to integer, complex, etc. As discussed in the next section, this statement also has some other effects on the Fortran output of the preprocessor. Note that the simple transformation shown here does not allow the feature of dynamic arrays with more than one subscripting dimension.

A textual substitution by a preprocessor similar to the one presented here was recently suggested as a way of resolving the problem of using several work space arrays in a Fortran program library for optimization [4]. Partitioning a single user provided array into several work arrays by keeping track of base subscript pointers "can lead to a lack of clarity in the resulting code, since the names of the conceptual arrays of the original algorithm do not appear" [4, p. 269]. Their goal is for program clarity rather than for arrays whose sizes vary dynamically during exection and it appears that they have not attempted to write such a preprocessor.

II. Gramatical Rules.

The key transformation performed by the modified Ratfor preprocessor is the conversion of

'xary(i)' into 'rzr(xary+i)'

where the variable xary is actually an integer subscript which points to the beginning of the dynamic array xary in array rzr, which has been equivalenced to the blank common array provided in the main program. This is the same technique used in the circuit analysis program SPICE2 [2] except that the dynamic arrays (called 'tables' in [2]) appear in the program like the substituted string above, i.e., 'rzr(ltab+i)' appears directly in the Fortran programs ('rzr' is 'value' in SPICE2). Another difference is that the Fortran code discussed below also appears directly in the SPICE2 source code.

To be precise, the first 'dynamic' statement encountered causes the following Fortran code to be output by the modified Ratfor preprocessor:

```
        integer izi(1)
        real    rzr(1)
        double precision dzd(1)
        complex czc(1)
        common izi
        equivalence (izi(1), rzr(1), dzd(1), czc(1))
```

Also, the Ratfor source line 'dynamic real a,b,c,...' causes the line 'integer a,b,c,...' to be output, so that the dynamic array variables are actually array subscripts as mentioned above. The equivalence statement above causes dynamic arrays of any type (integer, real, etc.) to all occupy memory space in the same large common array. Figure I shows a possible layout of memory for two dynamic arrays, IARY, real, size 5, and RARY, real, size 3. The values shown for the array subscript pointers are arbitrary and simply shows that the dynamic arrays could be anywhere in the large array (not overlapping, of course).

To ensure that the scope of dynamic arrays only extends to the end of the current subroutine (in a file of, say, several subroutines), the Fortran 'end' statement causes the preprocessor to eliminate all dynamic array names from its symbol table. Also, a flag is reset so that the above code is output again upon encountering the first 'dynamic' statement in the next routine.

The present preprocessor allows the types shown in the following examples:

```
dynamic integer a,b, ...
dynamic real    a,b, ...
dynamic double  a,b, ...
dynamic complex a,b, ...
```

The preprocessor keeps track of the type, so as to output for 'a(i)' one of 'izi(a+i)', 'rzr(a+i)', 'dzd(a+i)', or 'czc(a+i)'. This substitution occurs only when a dynamic array name is encountered, followed by a left parenthesis: if x is dynamic real then 'x(...' in the Ratfor source becomes 'rzr(x+...' in the preprocessor output. There is one special case which is considered by the preprocessor, and that is when the left parenthesis is followed by a minus sign: 'x(-k...)' is substituted by 'rzr(x-k...)' to avoid the consecutive operators in 'rzr(x+-k...)' (here, k is presumed to be negative).

One serious implication of this scheme concerns passing dynamic arrays to subroutines. Since the dynamic array variable name is simply an integer, either the array subscript value itself or the actual array can be passed. To pass the actual array, it must be subscripted as 'rary(1)' for example, in order to achieve the desired substitution from the left parenthesis. But arrays passed in this way cannot be extended in the subroutine. If the name is passed alone (not followed by a left parenthesis), then the corresponding subroutine argument can be declared as dynamic and the array can be extended in the subroutine. The following examples compare these two cases:

```
        dynamic double dary                  dynamic double dary

        call foo (dary(1), n, ...)           call foo (dary, ...)

        end                                  end

        subroutine foo (dv, n, ...)          subroutine foo (dv, ...)

        double precision dv(n)               dynamic double dv

        dv(1) = ...                          dv(1) = ...
                                             if ( i > sizmem(dv) )
                                                call extmem (50,dv)

        return                               return
        end                                  end
```

The case on the left above would be desirable if subroutine foo made no calls to any dynamic memory array subroutines, since it avoids the extra addition for each array access, i.e., it would use 'dv(1)' instead of 'dzd(dv+1)'.

An additional peculiarity concerning dynamic array names followed by right parenthesis is discussed next.

One thing that must be known by subroutine getmem is the type of the dynamic array being allocated. This is important because getmem allocates arrays internally in terms of the number of integers requested and on some computers, a real might take up the same memory as two integers. In such a case, 'call getmem (50,r)', where r is dynamic real, would actually allocate 100 integers from the large 'mem' array provided to subroutine setmem. Another consideration is that the number of integers per variable, for each variable type, must be easily changed for portability reasons. For example, on the VAX 11-780, one real and one integer are both one 32 bit word while on other computers, an integer might be one 16 bit word and a real two 16 bit words (this latter case is shown in Figure I where rary(3) is "wider" than iary(5)). In both the modified Ratfor preprocessor and in the recent version of SPICE2, the number of integers per variable, for each variable type, is easily changed: in the former by means of 'define's and in the latter in a DATA statement.

One approach to providing the type information to getmem is that taken in the most recent version of SPICE2 [3] in which different getmem routines are provided for each dynamic array type, i.e., 'getm4' for integers, 'getm8' for reals, and 'getm16' for double precision. This has the pitfall of requiring possibly several changes to a program to change a dynamic array from type real to type double, for example. (It is our belief that the declaration of a variable's type should occur only once, i.e., in the 'dynamic' statement, and not be intertwined into the program.)

The approach taken here is to append the number of integers per variable as another subroutine argument following the array name. Thus 'call getmem (20,d)' would be substituted by the preprocessor by 'call getmem (20,d,2)' if d had been declared as 'dynamic double' and there were 2 integers per double precision variable. The portability of the modified Ratfor preprocessor in this regard is discussed in appendix A. Also, another execution time reason for appending this argument is discussed in appendix

B.

This substitution occurs when a dynamic array name is followed by a right parenthesis: '...x)' becomes '...x,2)' if there are 2 integers per variable for the type of dynamic array x. Of course, this substituion can occur for all subroutine calls, not just to getmem. For the other dynamic memory allocation subroutines, this number is used to locate the dynamic array quickly (in constant time) in the large array. This is discussed further in appendix B.

This substitution can be dangerous if the programmer forgets about this extra calling argument since it might sabotage the subroutine linkage by having a different number of actual arguments than formal arguments in the subroutine. If, for example, the value of a dynamic array variable x was to be printed by a subroutine (which makes no sense!) then instead of 'call printi (x)' we could write 'call printi (x+0)' to stop the right parenthesis substitution. But this technique must not be used to pass a single dynamic array variable to a subroutine because 'x+0' might produce a temporary variable and the whole allocation system is based on an assumed call-by-address. The actual name x must be passed (more on this in appendix A). All one can do is plan on the extra argument by having one more formal (dummy) argument in the subroutine.

III. The Dynamic Memory Array Routines.

The dynamic memory allocation subroutines and calling conventions are shown in the following list. (It should be noted that the subroutine names are almost identical to those of SPICE2 [2]. However, the arguments are reversed in most cases due to the right parenthesis convention.)

routine	description
setmem (big,nvar,min)	initializes dynamic memory routines with large integer array 'big' of dimension 'nvar'. Blocks smaller than 'min' never appear on the linked list of free blocks (see section IV.).
getmem (nvar,ary)	allocates dynamic array 'ary' of size 'nvar'.
relmem (nvar,ary)	(releases) reduces size of dynamic array 'ary' by 'nvar' variables.
extmem (nvar,ary)	extends size of dynamic array 'ary' by 'nvar' variables.
clrmem (ary)	clears (eliminates) dynamic array 'ary'.
ptrmem (newary,oldary)	changes the dynamic array name of 'oldary' to 'newary'. (Both must of the same type.)
sizmem (ary)	integer function which returns the current size of dynamic array 'ary'.
memptr (ary)	logical function which returns true if 'ary' is currently an allocated dynamic array.

The user must declare sizmem as integer since it is an integer function but does not start with i, j, k, l, m, or n. Note that the dynamic array names above are always followed by a right parenthesis causing the number-integers-per-variable argument to be appended.

IV. Memory Allocation Techniques.

This section presents a brief discussion of the actual memory allocation techniques used in the routines listed in the previous section. The basic problem of programming with arrays whose size varies at run-time may be tackled in many ways.

Many programmers create one large array and then allocate new arrays on the 'top' of (at one end of) the large array ... a stack allocation technique. This permits only the array on the top to be extended dynamically. If extension of the dynamic array below the top is desired, then the top array must first be eliminated -- the arrays must be eliminated in the reverse of the order in which they were allocated.

In order to handle the more general case of several arrays being created or extended at the same time, and in an arbitrary order, a more sophisticated memory allocation strategy is needed. The Boundary Tag Method [5] [6] is a heap allocation technique which starts out with the large array provided to setmem being considered as one big free block. When a dynamic array is requested, it becomes an allocated block in this large free block. When arrays are cleared, 'free' blocks are left behind. These free blocks are added to a doubly linked list of such blocks, used to facilitate the location of a free block for subsequent getmem's. Following [6], a First-Fit strategy is used when searching down the list of free blocks for one whose size is greater than or equal to the requested size. In fact, the routines used internal to the memory allocation routines listed in section III are closely patterned after procedures 'allocate' and 'free' in [6]. When no free block can be found large enough to meet the size of the requested dynamic array, an internal subroutine, 'crunch', is called to compress all the allocated blocks together leaving one large free block. If this block is still not large enough, then an out-of-memory message is printed and the program halts.

The current method of handling dynamic array size extensions via 'extmem' is important to understand so as to not create cpu-time inefficiency. The algorithm may be stated briefly as:

```
Input:  ary, n     (dynamic array name and extend size)

Convert n from number of variables to number
of integers using the augmented number-integers-
per-variable argument.

Try to allocate a new dynamic array of size
equal to the new total size, i.e., oldsize+n.
If successful, copy the old array into the
new, clear the old, and return.

The attempt was unsucessful so call
crunch to compact all the allocated
blocks to the lower subscript end of
the large array and try to allocate
the new total size again.

If not successful, perform a double-swap-flop
to move the dynamic array being extended right
below the one free block left after a crunch.
Then, it may be extended into this free block
even though there is not enough room for
the new total size.
```

Under any conditions, the least amount of work to extend an array is to copy the entire array one time. This implies that extending a large array can be costly and thus the number of extends should be minimized. This is achieved by extending with n greater than one, as shown in the example subroutine harder in section I.

V. Possible Inefficiencies.

There are a number of inefficiencies which this scheme can create, most of which are due to the fact that 'what you see is not quite what you get'. One simple example is shown below, along with its translated Fortran output:

```
          dynamic integer k               integer k

          do i = 1 , 100                  do 23000 i=1,100
             k(i) = 0                     izi(k+i)=0
                                    23000 continue
```

To avoid the extra addition for every loop iteration, this code may be rewritten as:

```
          dynamic integer k

          i1 = k+1
          i2 = k+100
          do i = i1 , i2
             izi(i) = 0
```

This modification, using 'izi', is considered by us to be a gross misuse of the knowledge of what the modified Ratfor preprocessor produces, and is strongly discouraged. Also, Fortran compilers with code optimization which removes loop-invariant computation from loops or performs induction variable modification, as shown in [7], will make the original code no less efficient than the rewritten code by actually generating code which mimics the latter.

VI. Conclusions.

The modified Ratfor preprocessor has been in existence for about 8 months and has been used extensively by the author on several large programming projects. One, in particular, is a LALR(1) [7] compiler-compiler called RACC which has about 15 dynamic arrays. During the processing for an average size grammar, the program executes a few crunches without snags. As the amount of available array space comes close to being exhausted, the occurence of crunches becomes very frequent. This implies, as suggested by Knuth in [5], that execution should halt after the total amount of free space falls below a certain amount. This would avoid the cpu time of the last several crunches before a likely exhaustion of space anyway.

Using the dynamic array subroutines mentioned in this program requires caution in regard to subscripting arrays out of bounds. This is because certain internal information such as links, requested size, allocated size, etc., are stored directly around the dynamic array in the large equivalenced array: if xary is of size n then xary(0) and xary(n+1) must not be destroyed. If they are, the dynamic memory routines are certain to 'crash'. Also, Fortran compiler generated run-time array subscripting checks would not catch such subscripting violations.

Remaining for future work is to measure the amount of time spent in the dynamic memory allocation routines to find bottlenecks which might be made more efficient. Preliminary feelings are that the whole system is quite efficient and that the extra computer time to extend arrays, etc., is negligible compared to the benefits of the availability of dynamic arrays. Certainly if, without dynamic arrays, all program arrays were dimensioned to the size needed by the largest possible input program, then either the program size might be too large to fit in memory or the program might create tremendous paging overhead in a virtual memory environment.

VII. Acknowledgement.

Research sponsored by the National Science Foundation under Grants ECS-79-13148/(Rann) PFR79-08261 and Joint Services Electronics Program Contract F49620-79-C-0178.

Appendix A. Machine Independence of Routines.

The machine independence or portability of the dynamic memory routines described in this paper is based on a few 'define's in the file 'machdep':

```
define (NUMIPERI,1)    # number of integers per integer var.
define (NUMIPERR,1)    # number of integers per real var.
define (NUMIPERD,2)    # number of integers per double
                       # precision var.
```

```
     define (NUMIPERC,2)     # number of integers per complex var.
     define (LCMULT,2)       # least common multiple of above.
```

Also, the two machine dependent routines getadr and setadr must be implemented. These get the address of the argument variable and set the variable at the argument address. The way that the dynamic memory routines use these two subroutines makes it essential that the Fortran be implemented using strictly call-by-address. Most implementations use this approach. Also, Fortran 77 now states explicitly that call-by-address is part of the standard.

Appendix B. Another Reason for Appending the Number of Integers per Variable.

In section II the right-parenthesis convention was presented in which an additional argument, the number of integers per variable, is appended after the dynamic array name. It was stated that this argument is necessary for a getmem since the actual allocation is internally in terms of the number of integers. Additionally, this argument is used to determine the location of the dynamic array in the large integer array provided to setmem. For example, if there are 2 integers per real, then dynamic real array ary with subscript pointer ary=50 would actually begin at subscript 100+1 in the large integer array. This immediate access to the actual array location gives rapid access to the surrounding information about the array, i.e., the requested size, the allocated size, the subscript pointer address, etc. In particular, function sizmem excutes and returns the size in constant time, independent of the number of dynamic arrays. This makes the use of function sizmem ok in an inner loop as shown in subroutine harder in section I.

This is not the case in the memory manager in SPICE2, however, due to the fact that determination of the actual location of a dynamic array requires a search through a list of addresses for one which matches the address of the routine's table pointer argument. This search may be as long as the number of allocated dynamic arrays.

Appendix C. EECS-SESM VAX UNIX File Locations.

On the University of California EECS-SESM VAX, the dynamic memory routines are located in directory /usr/eecs/nye/mem and consist of files memae.r, memfr.r, and memsz.r . To load a Fortran program that uses these routines, one may append '/usr/eecs/nye/mem/mem*.o' to the 'ld' command.

The modified Ratfor preprocessor is located in directory /usr/eecs/nye/ratfor . There is a shell script /usr/eecs/nye/rat4 which, when invoked using 'rat4 abc', both Ratfor preprocesses and Fortran compiles a Ratfor source file named abc.r .

References.

[1] Kernighan, B. W. [1975]. "RATFOR - a Preprocessor for a Rational Fortran," Software--Practice and Experience, 5:4, 395-406.

[2] Cohen, E. [1976]. "Program Reference for SPICE2,"

Electronics Research Laboratory Report No. ERL-M592, University of California, Berkeley, Ca.

[3] Vladimirescu, A., Newton, A. R., Pederson, D. O. [1980]. "SPICE Version 2F.0 User's Guide," Electronics Research Laboratory, University of California, Berkeley, Ca.

[4] Gill, P., Murray, W., Picken, S., and Wright, M. [Sept, 1979]. "The Design and Structure of a Fortran Program Library for Optimization," ACM TOMS, Vol 5, No. 3, 259-293.

[5] Knuth, D. E. [1968]. The Art of Computer Programming, Vol I: Fundamental Algorithms, Addison-Wesley, Reading, Mass.

[6] Horowitz, E. and Sahni, S. [1976]. Fundamentals of Data Structures, Computer Science Press, Woodland Hills, Calif.

[7] Aho, A. V. and Ullman, J. D. [1977]. Principles of Compiler Design, Addison Wesley, Reading, Mass.

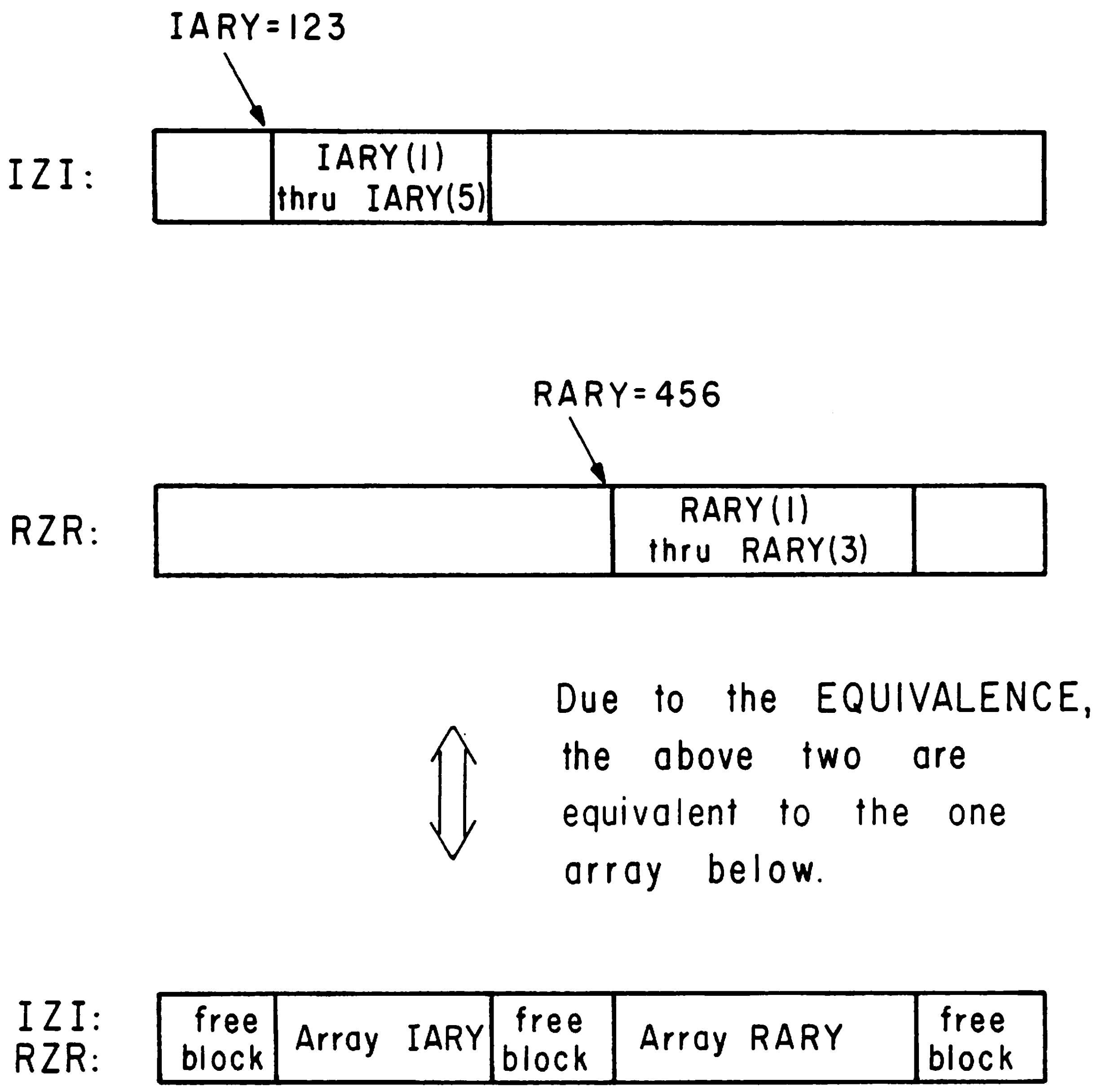

Figure I. Memory layout for two dynamic arrays.