The Allocation of Storage for Arrays
In Algol-60

By

K. Sattley  P. Z. Ingerman

K. Sattley
Laboratory for Applied Sciences
Institute for Air Weapons Research
University of Chicago

P. Z. Ingerman
Office of Computer Research and Education
University of Pennsylvania

November, 1960
The Allocation of Storage for Arrays in Algol-60

By

Kirk Sattley
Laboratory for Applied Sciences
Institute for Air Weapons Research
University of Chicago

Peter Zilahy Ingerman
Office of Computer Research and Education
University of Pennsylvania

Contract No. AF-49(638)-951

November, 1960

UNIVERSITY OF PENNSYLVANIA

OFFICE OF COMPUTER RESEARCH AND EDUCATION
ABSTRACT

One of the more challenging features of ALGOL 60 is the possibility of allowing the dimensions of both own and non-own arrays to be defined by variables which take on their values only dynamically, so that no fixed amount of storage in the computer can be reserved by the compiler at compilation time. The purpose of this paper is to demonstrate the realizability of this feature of the language by presenting a model of such an allocator. The basis of the approach used here is that of allowing the translator to include in the coding it puts out copies of those of its own functions which must, because of the specification of the problem, be carried out at run time. The mechanism described in this paper is independent of the possible presence of recursive procedures in the program.
THE ALLOCATION OF STORAGE FOR ARRAYS IN ALGOL 60
KIRK HATTLEY*

One of the more challenging features of ALGOL 60 is the possibility of allowing the dimensions of an array to be defined by variables which take on their values only dynamically, so that no fixed amount of storage in the computer can be reserved by the compiler at compilation time. This requires a mechanism for dynamically determining the location and size of an array at the time the program is run.

The purpose of this paper is to demonstrate the realizability of this feature of the language by presenting a model of such an allocator. This model is by no means a solution to the general storage allocation problem, since it makes essential use of the block structure of ALGOL, and since it does not consider the problem of the best action to take when insufficient space remains to fill a request for storage assignment (although it recognizes the condition). On the other hand, this model does not assume an infinite memory -- it will, at any time, never have space reserved for information which can no longer be referenced.

The basis of the approach used here is that of allowing the translator to include in the coding it puts out copies of those of its own functions which cannot be carried out at compilation time. This is analogous to the action of most present compilers in translating references to arrays: if all the subscripts of a particular subscripted variable are numerical constants, the translator itself will apply the storage-mapping function to the subscripts, and output the absolute address of the element; however, if some of the subscripts depend on variables, the translator puts out the coding necessary to evaluate the storage-mapping function at run time.
Thus, all of the sample programs in this paper should be regarded as either parts of the translator, to be executed during compilation if possible, or as parts of a "caretaker package" included with the running program when the data required to execute them will not be available until the program is run.

**Working Assumptions**

The sample programs will be presented in ALGOL -- with a certain amount of literary license -- since that is the best mechanism available at present for communicating algorithms. One feature used here needs some explanation; since it is often necessary to refer to memory addresses and the contents of memory cells in this discussion, the almost-legitimate device of the "Memory vector" will be used. This is done by assuming the entire memory of the machine in question to constitute a single one-dimensional vector, named "Memory". Given this, any integer expression can be used as a machine address, by referring to "Memory[I]". Thus, for example, "Memory [365]" is the number found in cell 365; the value of the subscripted variable

"Memory [ Memory [A+3] - 12 ]"

is the contents of that cell whose address is 12 less than the number found in the cell whose address is 3 greater than the current value of the (integer) variable A.

For definiteness in the discussion, certain arbitrary but not implausible assumptions will be made about the action of the translator: (A) Storage will be allocated for program, constants, and the like, at the low end of memory; this will be called the "fixed storage". This does not mean that all of the program must be in the memory at one time, but rather that, if segmenting and overlaying are used, the translator will have determined the
largest amount of program, etc., which will be in memory at any one time, and reserved this amount as "fixed storage". The remainder of the machine's memory will be available to the allocator function. (B) Corresponding to each identifier in the program, the translator will assign a single memory cell in the fixed storage area. The single cell corresponding to an array identifier will be expected to contain at any time the correct current location of the array -- thus, when the allocator allocates a segment of storage to an array, it places the first address of this segment in that cell; the address of this cell will be called the "name" of the array. (C) Stored along with the elements of an array will be a "dope vector" containing the information necessary to reference the array (the parameters of the storage-mapping function). This dope vector will immediately precede the array in storage, and it is the address of the zero-th word of this dope vector which is delivered to the array "name" cell. The dope vector for an N-dimensional array STUFF will consist of the following $2N + 1$ words:

\[
\begin{align*}
\text{Memory }[\text{STUFF} + 0] &= N \text{ (the number of subscripts)} \\
+1 &\quad \text{size of first subscript position} \\
+2 &\quad \text{lower bound of first subscript position} \\
\vdots &\quad \vdots \\
+2N - 1 &\quad \text{size of N'th subscript position} \\
+2N &\quad \text{lower bound of N'th subscript position}
\end{align*}
\]

(the "size" of a subscript position is the number of values allowable for a subscript in that position = (upper bound of subscript position) - (lower bound) + 1. ) Thus, the programming required
to locate the memory cell corresponding to the subscripted variable

\[ \text{STUFF}[S1, S2, \ldots, SN] \]

will be equivalent to:

\[
\begin{align*}
N & := \text{Memory[STUFF]} ; \\
\text{Base} & := \text{STUFF} + 2*N ; \\
\text{Index} & := 0 ; \\
\text{for } I & := 1 \text{ step } 1 \text{ until } N \text{ do} \\
\text{begin } J & := \text{STUFF} + 2*I ; \\
\text{Coord} & := \{\text{The value of the I'th subscript}\} \text{ expression, SI } ; \\
\text{Incr} & := \text{Coord} - \text{Memory}[J] ; \\
\text{if Incr} & < 0 \text{ or Incr} > \text{Memory}[J-1] \text{ then go to OUT OF BOUNDS} ; \\
\text{Index} & := \text{Index} + \text{Incr} ; \\
\text{if } I \neq N \text{ then } \text{Index} & := \text{Index} * \text{Memory}[J+1] \\
\text{end} ; \\
\{\text{address}\} & := \text{Base} + \text{Index} ;
\end{align*}
\]

The dope vector for an array is constructed when the declaration of the array is encountered. The programming for constructing the dope vector will be equivalent to the following (assuming again that \( N \) is the number of subscript positions -- i.e., the translator encounters \( N \) bound-pair expressions in the declaration):

\[
\begin{align*}
\text{Length} & := 1 ; \\
\text{for } I & := 1 \text{ step } 1 \text{ until } N \text{ do} \\
\text{begin } \text{Lobnd} & := \{\text{value of lower-bound expression in the} \\
& \quad \text{I'th bound pair}\} ; \\
\text{Upbnd} & := \{\text{value of upper-bound expression in the} \\
& \quad \text{I'th bound pair}\} ; \\
\text{Dopevec}[2*I] & := \text{Lobnd} ; \\
\text{Dopevec}[2*I - 1] & := \text{Dimsize} := \text{Upbnd} - \text{Lobnd} + 1 ; \\
\text{Length} & := \text{Length} * \text{Dimsize} \quad \text{end} ; \\
\text{Length} & := \text{Length} + 2*N + 1 ; \\
\text{Dopevec}[0] & := N ;
\end{align*}
\]
This dope vector is then presented to the storage allocator along with the "name" of the array (the address of the cell in which the actual array location is to be stored) and Length, which is the total amount of space which must be reserved (including the dope vector). Actually, since a number of arrays may be declared at once in a single "array segment" (if they are of the same type and have the same bound pair expressions), the allocator is invoked only once for such a segment, and is presented with a list of the names declared in the segment; this list will be called "Namevec" and, if \( M \) arrays are declared, will consist of \( M + 1 \) items, the zero-th being \( M \) itself. If these \( M \) arrays are declared _own_, the zero-th item will be negative, \(-M\).

**Allocation of Storage for Non-own Arrays.**

The block structure of ALGOL 60 makes the allocation of storage to non-own arrays quite straightforward, since it imposes a strict last-in-first-out ordering on the assignment and releasing of storage. The arrays declared in the outermost block of the program may be placed at the beginning (low-address end) of the available storage. When the first sub-block is entered, its arrays are placed immediately following (higher addresses) those of the first block, and this process is continued as long as successively lower-level blocks are entered. When control exits from a block, the space reserved for the arrays declared in that block can be released as available -- the array declaration is rescinded by exiting from the block in which it occurred. And when this happens, as a result of the tree structure of the blocks, the storage about to be released must necessarily be the storage most recently assigned.

To mechanize this process, a global identifier (memory cell) \( \text{FUS} \) is assumed, whose value (contents) at any time will be the address of the _First Usable Storage_ cell; hence, the initial value of
FUS is the first address beyond the fixed storage area. Now, when a segment of non-own arrays is declared, the allocator (henceforth to be called by its proper name, FUSBUDGET), for each array in the segment: places a copy of the dope vector at FUS (= in storage, starting at the cell whose address is the present value of FUS), places the current value of FUS in the cell addressed by the "name" of the array, and augments FUS by the total length of the array.

The program for this simple, non-own FUSBUDGET would look like this:

```pascal
Nrnames := Namevec [0] ;
Nrdopes := 2 * Dopevec [0] ;
for J := 1 step 1 until Nrnames do
  begin Memory [Namevec [J]] := FUS ;
    for I := 0 step 1 until Nrdopes do
      Memory [FUS + I] := Dopevec [I] ;
    FUS := FUS + Length end ;
```

Using the FUS mechanism, releasing storage upon exiting from a block is accomplished merely by resetting FUS to the value it had at the time the block was entered; this requires that that value have been saved at block-entry time and be available at block-exit time. An appropriate technique to apply to a recursive situation like this is that of linking. For this, a second global parameter is needed, called POINT, whose value at any instant will be the value that FUS had at the most recent block-entry time. When control enters a block, the program will call upon a short procedure PREFUS to accomplish this setting of POINT. The function of PREFUS will be to store the present value of POINT in the cell at FUS, thus recording the link back to the preceding block-entry point, set POINT equal to the current value of FUS, and augment FUS by one, to protect the
linking cell; thus POINT, at any time, contains the address of the cell in which its previous value is stored.

procedure PREFUS ;

begin Memory [ FUS ] := POINT ;
POINT := FUS ;
FUS := FUS + 1 end

When control exits from a block, the program must call upon the complementary procedure UNFUS to restore conditions as they were when the block was entered. A checking feature is also included in UNFUS, which is probably superfluous in a sufficiently discriminating compiler, to verify that UNFUS has not been called more times than PREFUS was, thus avoiding the possibility of turning over part of the fixed storage area as "available". This checking requires that the last cell of fixed storage contain its own address, and that the initial value of POINT be the address of that cell. The operation of UNFUS, then, is, after checking that the contents of the cell at POINT are different from POINT, to set FUS equal to the present value of POINT (which was the value FUS had when the block was entered), set POINT equal to the contents of the cell at this address (which is the value that POINT had upon entry to this block) and exit.

procedure UNFUS ;

begin if Memory [ POINT ] = POINT then OUT OF PHASE ;
FUS := POINT ; POINT := Memory [ POINT ] end

Own arrays.

The difference between own and non-own identifiers lies in the requirement that the value of an own quantity must remain unchanged during the time between exit from the block in which it is declared and the subsequent re-entry into that block. For simple variables declared own this is simply attained by having the transla-
tor assign to the variable a memory cell in a segment of fixed storage which is never overlaid or re-initialized. The own-array storage mechanism will assume the same to be true of the "name" cell of an own array -- i.e., that this cell will be available even if the block of program in which the own array was declared is not in the memory.

Since the values of the elements of an own array must be retained, the storage assigned to the array may not be released when control exits from the block which declares the array, and so this space must be kept out of the way of the FUS mechanism. The most out-of-the-way location would seem to be the other end of memory, and so own arrays will be placed in the high-address end of the memory, and as each own array is first declared, a FUS-mechanism-in-reverse is activated to place it in the currently highest available memory locations. (This does not mean, of course, that the order of the elements in the array is reversed, but just that, e.g., the second own array to be declared will occupy a segment of storage immediately below (in the sense of machine addresses) the first one).

To implement this allocation, one more global parameter is needed, called UFUS (for Upper First Unavailable Storage), whose value, at any instant, will be the lowest address currently used for own-array storage. The operation of FUSBUDGET then, when assigning storage for an own array the first time its declaration is encountered is: decrement UFUS by the amount of storage to be reserved for the array; transmit this new value of UFUS to the "name" cell as the location of the array, and copy the dope vector at this location (see the complete FUSBUDGET program at the end of this paper).

This leaves, then, only the problem of handling own arrays when the block which declares them is re-entered. Evidently, if the subscript-bound expressions in the declaration contain no variables, or if, even containing variables, the expressions have the
same value as they had at the last time the declaration was processed, nothing more need be done -- the array as it was left satisfies the present declaration. However, if any of the subscript-bound expressions has a different value, the elements of the array must be rearranged to satisfy the new declaration. It is not sufficient merely to make a new reservation of the appropriate amount of space and release the space previously used, since the ALGOL 60 report specifies that any subscripted variable whose subscripts are defined both under the old declaration and the new must have the same value it had before. Mr. Ingerman has addressed himself to this general question of mapping one array structure into another of arbitrarily different parameters (requiring only that the number of dimensions remain constant) while preserving the values of all elements which occur in both structures, and he describes a procedure for doing this in an accompanying paper. For present purposes it is sufficient to know that the procedure, called ARSHIFT, requires as inputs the locations of two dope vectors, one corresponding to the old declaration and one to the new. The procedure tests for identity of the two dope vectors, and, if they are not the same, transmits those elements of the old array which are still defined to their proper locations relative to the dope vector of the new array. "ARSHIFT" appears as a boolean function designator; it returns the value true if it found the dope vectors different and performed the mapping, and returns the value false if the dope vectors were identical and the array did not have to be moved.

To accommodate this possibility, FUSBUDGET must be modified as follows: when presented with a segment of own arrays, for each array, it must determine whether the array has been previously declared. If it has not been, space is allocated for it below UFUS, as discussed before. If it has been previously declared, FUSBUDGET tentatively decrements UFUS (sets a local variable "Nufus" to UFUS minus the amount of storage to be reserved), copies
the dope vector for the new declaration at this location (Nufus), and then presents to ARSHIFT the location of this dope vector together with the location of the dope vector accompanying the previous version of the array. If ARSHIFT responds that it has moved the array, it will have stored the proper location (Nufus) in the appropriate "name" cell, and the tentative new value of UFUS is confirmed; if ARSHIFT responds that it did not have to move the array, UFUS is left at its current value, and, since the "name" cell already contains the correct location, nothing more need be done for this array.

Some additional structure of own-array storage is required, however, to enable FUSBUDGET to determine whether an own array is being declared for the first time, or has been declared before. Also, after an array has been ARSHIFTed from its previous location inside the compact own storage area, the space it previously occupied can be released for re-use. At this point, any sophisticated process for making available detached segments of storage might be applied; for simplicity, the process presented here is a simple collapsing of own storage back into a compact space at the high end of memory, moving each array which has "empty" space above it up against the next-highest array. (For time efficiency, this collapsing procedure should be applied only when FUS and UFUS overlap, and action must be taken to release more available space. However, for conceptual neatness, we have included it here within FUSBUDGET, to be executed (if required) after processing each segment of own arrays.)

The additional structure in the own storage area consists of a two-word "packet" for each array: the first word contains the location of the array -- this allows a rapid linked-search through the own arrays, since the packet for the next (lower) own array will be found at this location minus one; the second word of the packet contains the address of the "name" cell in charge of this array -- thus,
when an _own_ array is moved, either by ARSHIFT or for collapsing
_own_ storage, the contents of this cell tells the moving routine where to
record the new location of the array. If ARSHIFT moves an array, it
will set this "name" word of the packet to zero in the old copy of the
array, to indicate that its space is now "empty".

**Relation to Recursive Procedures.**

The mechanism described in this paper is independent of the
possible presence of recursive procedures in the program. If a block
is entered a second time without having been exited, by virtue of being
within the body of a recursive procedure, the declarations of the block
are interpreted anew, thus making new copies (perhaps of different
size) of the non-_own_ arrays, and, correspondingly, either re-mapping
or leaving unchanged the _own_ arrays which are re-declared. The pro-
cess of exiting from a recursive nest is accomplished one level at a
time (see the accompanying paper by Messrs. Feurzeig and Irons on
recursive procedures), so that the multiple copies of non-_own_ arrays
are obliterated in the correct order, and the _own_ arrays (which are
ignored by UNFUS) will be left in the condition they last assumed.
This accords entirely with the interpretation given to the behavior of
all _own_ and non-_own_ identifiers under recursion.

**Summary.**

The sample programs presented in this paper exemplify a
principle of storage allocation for arrays of variable size in ALGOL 60.
Up to the point at which no more memory space is available in the
machine, storage is allocated efficiently at little cost in time and space.
The requirements of the mechanism are: (A) Three reserved cells:
FUS, POINT, and UFUS (these are assumed global so that other parts
of the "caretaker package" -- e.g., the routines handling recursive
procedures -- can use available space for their own purposes). (B) The
last cell of fixed storage contains its own address, the initial value of
POINT is the address of this cell, the initial value of FUS is one greater
than the address of this cell, and the initial value of UFUS is the high-
est memory location addressable by the program; (C) when control
enters a section of program corresponding to a block in the ALGOL
formulation, the procedure PREFUS must be called; (D) when control
encounters (the run-time output from) an array-declaration segment,
FUSBUDGET is called upon, with the requisite information in hand
(list of names of arrays to be declared, dope vector common to these
arrays, length of array, and indication if arrays are own); (E) when
control exits from a block, the procedure UNFUS must be called.

The sample program for FUSBUDGET which follows is not
represented to be the only, or the most efficient way of realizing the
mechanism discussed in this paper; it is included in order to give a
detailed exemplification of the principles involved, for those who might
wish to study it.

ACKNOWLEDGMENTS

This paper, like the accompanying papers by Ingerman,
Feurzeig, and Irons, is a product of the "Rump Group" seminar held
in Chicago in August, 1960, during which the problems of writing
translators for ALGOL 60 were explored; a previous version of this
paper was presented at the meeting of the (then) ALGOL Working
Group in Milwaukee, the day before the ACM National Conference.
The first ideas for this mechanism arose in the general discussion
among the Rump Group members, and the basis of the mechanism was
worked out in discussions with P. Z. Ingerman, while the others were
concentrating on other problems.

*Laboratories for Applied Sciences, University of Chicago. This work
was sponsored in part by Wright Air Development Division, Air Research
and Development Command, United States Air Force.
procedure FUSBUDGET (Namevec, Dopevec, Length);

value Length; integer Length; integer array Namevec, Dopevec;

begin integer Nrnames, Nrdopes, Nufus, Test, Name; boolean Collapse;

Nrnames := abs(Namevec[0]);
Nrdopes := 2 * Dopevec[0];

if Namevec[0] < 0 then go to Own arrays;
for J := 1 step 1 until Nrnames do
    begin Memory[Namevec[J]] := comment Store location of array in "name" cell;
        for I := 0 step 1 until Nrdopes do
            Memory[FUS+I] := Dopevec[I]; comment Move dope vector to new location
    end;

if FUS > UFUS then PANIC else go to Exit;

Own arrays: Collapse := false;
for J := 1 step 1 until Nrnames do
    begin Nufus := UFUS - Length - 2; comment Tentative new UFUS;
        if Nufus < FUS then PANIC;
        for I := 0 step 1 until Nrdopes do
            Memory[FUS+I] := Dopevec[I];
        Name := Namevec[J];
        Test := 2^15; comment Means "highest addressable memory cell";
    end;

Namesearch: if Test = UFUS then
    begin Memory[UFUS-1] := Name;
        UFUS := Memory[Name] := Memory[UFUS-2] := Nufus;
        comment This alternative executed only if array name not found in own storage
    end
else if Name ≠ Memory[Test] then
    begin Test := Memory[Test-1] - 1; go to Namesearch;
        comment This branch executed each time array name fails to match Test name
    end
else if ARSHIFT (Memory [Test-1], Nufus) then
    begin Collapse := true ; UFUS := Nufus ;
        comment This branch executed when name is found; and ARSHIFT had to move the array
    end
else comment Control here if name found, but ARSHIFT didn't move

end of J-loop (Names) for own arrays

if Collapse then COLLAPSE OWN STORAGE; comment execute here the collapse procedure which follows, or equivalent;

Exit: end FUSBUDGET

NOTE: The no-parameter procedure PANIC used in FUSBUDGET is called into play when no space apparently remains available for assignment. It would presumably first try to collapse own storage, and then proceed to dumping routines, etc.

procedure COLLAPSE OWN STORAGE ;
begin integer To, From, Next, Much ;
    To := 2|15 ; comment Highest address in memory;
Towhere: if To = UFUS - 1 then go to Quit ;
    if Memory [To] ≠ 0 then
        begin To := Memory [To-1] - 1 ; go to Towhere ;
            comment Looking for first empty array space end ;
    From := Memory [To-1] - 1 ; comment Having found empty space, start looking for array to move into it;
Fromwhere: if From = UFUS - 1 then
        begin UFUS := To + 1 ; go to Quit end ;
    if Memory [From] = 0 then
        begin From := Memory [From - 1] - 1 ; go to Fromwhere end ;
    Next := Memory [From-1] - 1 ; comment Having found an array to move, prepare to move it ;
    Much := From - Next - 1 ; comment Much+1 is amount of storage to move ;
for I := 0 step 1 until Much do
  Memory [To-1] := Memory [From-1]; comment Move the array;
  Memory [Memory[To]] := Memory [To-1] := To - Much;
  comment New location of array put in first packet cell and "name" cell;
  Memory[To] := 0; comment Make old copy of array look empty;

To := To - Much - 1; comment Just below newly-shifted array is next place to move to;

From := Next; comment Just below old copy of array is next place to try moving from (if not empty);

go to Fromwhere;

Quit: end of COLLAPSE OWN STORAGE