Flexible Dynamic Array Allocation

C's static zero-based arrays are not representative of the actual data to be stored by many systems. Here's a more flexible dynamic array allocator that can be used to supplement or replace static arrays.

Not every embedded problem can be described in definite terms at the outset. Frequently, a certain amount of flexibility is needed. The C language doesn’t help us much in this regard; it's pretty much a static language, with dynamic capabilities defined in libraries as an after-thought. The simplest way to gain flexibility, up to a certain limit, is to define an array of values or structures with a fixed dimension. As long as the attempted allocation of slots in the array does not exceed the dimensional limit, then everything is okay. This, however, uses a fixed, and almost always larger than necessary, block of memory. If several of these allocations are used in the several steps of a lengthy calculation, a lot more memory gets used up than necessary. If memory could be allocated/deallocated for each step, a smaller, more flexible program would result. The usual allocation functions—malloc(), realloc(), and so on—are useful in many cases, but they return an address of a block of memory that does not impose any structure on what is to be stored there. I, and every other programmer, wrestle with this issue whether creating embedded systems or any other type of software. I created the following scheme because I wanted some way to specify structural characteristics at the outset that would make using dynamically allocated memory much easier.

**Dynamic array allocator**

What are the most desirable dynamic memory allocation characteristics and what is the most flexible API with which to implement them? First, the allocator has to allocate for an arbitrary data type, not just the standard C types, that is, for anything on which you can use the sizeof() function. This includes structures and typedefs.

Second, the allocator has to allocate as a multidimensional array for easy access, with the sizes and starting subscripts—which can be negative numbers—of each dimension specifiable. Static C arrays allow the extent of each dimension to be specified, but not the starting subscript, because they are always zero-based. Adding this additional wrinkle allows FORTRAN-like subscripting from 1 or plus/minus subscripting from a negative subscript to a positive one, for example -5 to +5.
Third, and most important, I want the allocation of the multidimensional structure to be specified as a nested pointer to pointer to ... array, as opposed to a static array which is a pointer to a block of memory upon which subscript calculations locate the individual array elements. All ANSI C compilers understand the difference. The reason for this requirement is to allow the array to be passed sequentially to any depth of subroutine, be modified there, and be seen in modified form in any of the callers. This is not possible with static multidimensional arrays. Circumvention of this limitation results in the declaration of a lot of global data. These types of arrays have been traditionally implemented with multiple malloc(). A malloc() is done to allocate storage for one or more levels of linked pointer arrays, and finally malloc()s are done for the data rows at the end of the pointer links. For both embedded and non-embedded environments, this situation presents several difficulties.

Repeated calls to malloc() will allocate at non-local heap space locations. For a workstation environment this can result in multiple page faults during array access. In an embedded environment paging will not be a problem, but fragmentation of heap space memory might be. This also results in a structure that is inherently difficult to free. The pointer structure must either be traversed and the mallocOed blocks freed recursively, or a list of mallocOed block pointers must be saved for freeing, which is awkward. Lastly, we’re faced with the problem of cleaning up after a failed malloc(). The structure will be only partially allocated, which makes freeing more awkward still. For all these reasons I want to allocate the entire structure with only one malloc(). This provides locality of reference, makes freeing simple, does not fragment the heap, and avoids multiple malloc() calls.

Usage
The result of all this is the implementation defined by daa.c (dynamic array allocator). I have found these routines to be incredibly useful over the years. Two implementations exist: the first, daa() (and its associated das()) is for embedded systems; the second, daav(), is for a workstation environment (I use Suns mostly), where the page-aligned library allocation routine valloc() is available. Let’s examine a simple example of each from the test suite in daa.c.

The first example, Test 1, uses daa() to allocate a four-dimensional double array with non-zero subscripting and without initialization. The second example—Test 2, using daa()—allocates the same array with initialization:

```c
int err_code = 0;
char *free_ptr;
char *base_ptr;
int d[4] = {3, 5, 4, 2};
int st[4] = {-1, -5, 10, 0};
double init = 123.;
double ****array;

/* Test 1 */
if ( (array = (double ****)
daa(sizeof(double), 4, d,
&err_code, &free_ptr, NULL)) == NULL )
{
    printf("daa: error on dynamic allocation. %s\n",
da_errs[err_code]);
}

/* Test 2 */
asize = das(sizeof(double), 4, d,
&err_code);
base_ptr = (char *)malloc(asize);

if ( (array = (double ****)
daav(sizeof(double), 4, d, st,
&err_code, base_ptr, (char *)&init)) == NULL )
{
    printf("daav: error on dynamic allocation. %s\n",
da_errs[err_code]);
}
```

The first argument to daav() and daa() is the size of the basic object being addressed in the array—almost always calculated with the sizeof() operator. The second argument is the maximum dimension. The third and fourth arguments are the dimensional extent and start subscript of each dimension in order, from left to right. The fifth argument is the returned error code. This will be set, like errno, only if an error occurs and indexes the daa_errs[] array available globally from daa.h.

The sixth argument is, for daav(), a free pointer used to deallocate the array. Do not use free(array) to free the allocation. This will result in an error, as the array pointer does not point to the start of the vallocOed memory area. For daa(), a base pointer to the caller-allocated memory is passed for daa() to initialize with linked pointers. The caller deallocates with free(base_ptr).

Initialization can either be NULL for none, or the cast-to-pointer address of a type the same as that used as an argument to sizeof() in the first argument. In the case of daa() it helps to know how much space to allocate in the caller. This is calculated by a call to das() (dynamic array size) and then passed to malloc(). Notice in both
cases the (double *****) cast. This is required to match the assigned to pointer from the void pointer returned by \texttt{daa()} and \texttt{daav()}.

The valid subscript values for each array dimension are determined by taking the starting subscript, as defined by the \texttt{st[3]} array, and adding the corresponding dimension extent, as defined by the \texttt{d[3]} array, and subtracting 1. Thus, for the previous arrays, the valid subscript ranges are –1 to +1, –5 to –1, 10 to 13, and 0 to 1. If you address elements of the array with any subscripts outside these limits, the result is just as wrong as a similar addressing mistake in conventional C arrays. If the subrating mistake is only off by a few elements, the error may silently slide by; however, more serious mistakes often result in segmentation faults, at least on workstations.

If the application is a fixed-size array, the call to \texttt{das()} before \texttt{daa()} would then be removed and a \texttt{#define} substituted with the known size. Put another way, it is clearly unnecessary to call \texttt{das()} repeatedly only to return the same size. In a dynamic environment the \texttt{das()}/\texttt{malloc()}/\texttt{daa()} sequence will be needed. I often use \texttt{daa()} or \texttt{daav()}, even with fixed-size arrays, to be able to modify the array in a subroutine and still be seen in the caller.

The extreme case of 10-dimension- al allocation will result in an unusual cast. For example, let’s assume an array of doubles. The following code would result:

```c
double **********array;
array = (double **********) daa();
array[i][j][k][l][m][n][o][p][q][r] = 1.0;
```

Test 4 of the test suite does exactly this with varying dimensional extents and starting subscripts.

The locality of reference and the non-zero subrating are certainly useful. However, the most useful feature by far is the ability to pass such arrays to subroutines and get visibility of subroutine modifications to the array without having to do anything special such as taking addresses or using global data. In fact, I have used these routines in applications where I allocated a multidimensional array and passed only a slice of the allocated array to subroutines for modification. This is incredibly convenient. Assuming the allocation of a four-dimensional array, this is demonstrated by:

```c
upkg(bpd, array[i][j][k], &err_code);
```

Here the \texttt{bpd} structure contains bit-packed data that is unpacked in \texttt{upkg()} into the right-most two subscripts of \texttt{array[3][3][3][3]} and passed back as the two-dimensional sub-array of \texttt{array[i][j][k]}.

**New and improved**

Ten years ago, I wrote an article on an earlier version of this code. The basic allocation scheme and the functional intent is the same. However, I have made several enhancements:

- The old \texttt{daa()}, now called \texttt{daav()}, was originally not designed for embedded use. A new embedded version, the new \texttt{daa()}, has been added
- The test suite has been expanded. All tests are now independent and both \texttt{daa()} and \texttt{daav()} are tested
- The code has been made compliant with the ANSI C standard
- The data and pointer areas have been reversed to allow the heap space library routine to allocate space at the most stringent boundary for data storage
- The code to align the pointer space on a pointer alignment boundary has been added—Sun’s \texttt{cc} will fix any misalignment, but gcc will not

**Internals**

Only four error returns are possible, all resulting in a NULL return value. These are:

- A check that the number of dimensions is greater than zero and less than or equal to \texttt{MAX_DIM}
- A check that the data size of the first argument is greater than zero
- A check that each dimensional extent is greater than zero
- A check of the \texttt{malloc()} return value

The first three checks are applicable to \texttt{das()} and \texttt{daa()} while all four apply to \texttt{daav()}. Error-free operation returns a non-NULL pointer to \texttt{void} that should be cast to the type of the intended array and does not set \texttt{err_code}.

There is no algorithmic reason why the maximum dimension has to be 10. \texttt{MAX_DIM} is used internally for storing information about the dimensional start and extent. If, for some reason, you want more dimensions, just increase \texttt{MAX_DIM} and recompile. Ten seemed like a good round number that most applications would be unlikely to exceed. Decreasing or increasing the limit carries no performance benefit or penalty.

The total memory size of the allocation is the sum of two parts. The first part is the actual data, which is immediately followed by the space of pointers to pointers to... that point, eventually, to the last layer of pointers, and then to the data. The size of these two parts is calculated by: number of data elements * size of data element + number of pointers * size of pointer (plus a few bytes up to \texttt{sizeof(char *)} to align the pointer area). On Unix systems, do a \texttt{pagesize} command to see how much space you have before the allocation runs into a second page or how many total pages.
altogether will be needed. The `valloc()` will ensure that allocation starts on a page boundary.

Data initialization—done only if the last argument pointer to initialize element is non-NULL, and identical for both `daa()` and `daav()`—is done just before pointer setup. The initialization element is copied repeatedly to fill the data area. I chose not to use a `memmove()` or any other standard copy routine because I wanted to remain as independent of any library requirements as possible.

The pointer area setup is the last and most complex step. This is done by `ptr_init()` and two other recursive routines it calls—`off()` and `doff()`. The `ptr_init()` routine repeatedly calls itself with each new invocation descending to a lower level in the pointer array hierarchy. The first argument, `level`, is incremented by 1 for each successive array dimension. For a three-dimensional array, the level would go from 0 to 2. The `dim_ind[]` (dimension index) array tells each recursive invocation of `ptr_init()` exactly where within each level the pointers returned from the next level are to be stored. The `dim_ind[]` array will have, in some recursive call (except the last), every combination of array values. The last level is special because its pointers point to data and not to other pointers. Initially `dim_ind[]` will be zeroed. Successive calls to `ptr_init()` will increment each dimensional level by one from zero to the maximum subscript. Thus, a three-dimensional array dimensioned 5 x 10 x 15 will see `dim_ind[0]` go from 0 to 4, `dim_ind[1]` go from 0 to 9, and `dim_ind[2]` stay 0.

Every pointer is calculated from three parts: the constant base pointer; the offset of the arrays of pointers within each level, calculated by `off()`; and the data offset of the arrays of pointers within each level, calculated by `doff()`. Again, the last level is different because it’s the data element level, which has a different basic data unit element size, and because no further levels are called. To accommodate non-zero starting subscripts, a few adjustments to the pointers are made. Each recursive call to `ptr_init()` has its return value adjusted by an amount based on that level’s desired starting subscript obtained from the `st[]` array. The passed-back pointer for one dimensional arrays, which do not need a pointer structure, is adjusted separately.

The `level` and `dim_ind[]` arguments are the only ones modified recursively. The others are all statically set up before `ptr_init()` is called and remain unchanged. In my original version of this code, these were all file global statics. I eliminated them in favor of passed arguments in order to
make the routines fully reentrant. I use them in a Solaris MT/MP/RT environment in which full reentrancy allows me to avoid protecting the code with mutexes.

Alignment of the data area is the alignment of the first argument sizeof(), while the alignment of the pointer area is sizeof(char *). The data comes first, and the assumption here is that the valloc() (for daav()) and whatever allocation routine is used between das() and daa() will align at the most stringent boundary, thus accommodating the data area alignment. The pointer area comes second and may, depending on the total size of the data area, need to be aligned on a sizeof(char *) boundary. The base pointer of the pointer area is tested for alignment in the main routine before pointer array calculations, and its alignment adjusted if necessary.

The easiest way to see how the pointer levels are being calculated is to put a printf() at the start of the ptr_init() routine and print the level and dim_ind[] array. This gives you an idea as to how the arrays of pointers at each level are being positioned. Additionally, you could print out the pointer values themselves and walk through them to verify that the pointer to pointer to ... to data structure is being properly implemented.

Test suite
The test suite in daa.c, which is compiled with the DAA_TEST define set, has 15 tests that exercise the allocation routines extensively. I have tried to vary the tests over a range of dynamic allocations that most users might want to use. Each test examines the test variable for the DAA or DAAV constants and calls the corresponding allocation routine. To switch from testing one to the other, change the test variable assignment at the start.

Tests 1 and 2 modify the allocated array in a subroutine and print out assigned values in the caller. Test 4, a 10-dimensional double array, is the largest. It takes up the most CPU time of any of
the tests by far and produces the largest array. It uses non-zero subscripting over several dimensions of varying extent. After allocation, I set every element of the array with incremented-by-one values. I only print out the first and last elements for verification. An entire print-out would be too long. The first element should be 1 and the last element should be the product of all the dimensional extents. Subsequent tests allocate structures, enums, and unions, as well as varying the dimensional extents and start subscripts. All of the tests are completely independent of each other. That is, each should be removable and usable elsewhere without modification. Hopefully, these tests will serve as examples and prototypes for the user’s code.

This code has been compiled and tested with both the SunPro(cc) and Gnu(gcc) compilers. The following commands were used:

```
cc -Xc -g -DDAA_TEST -o daa daa.c -lm
gcc -g -ansi -pedantic -Wall
    -DDAA_TEST -o daa daa.c -lm
```

For an excellent reference on this type of array access see *Numerical Recipes in C*.

### A detailed example

A more graphical and explicit numerical example is helpful. I define my example array as a two-dimensional integer array with some non-zero starting subscripts. I have run three variations on this array to illustrate the effect of different non-zero subscripts, as shown in Figure 1. The 2 x 3 array has starting subscripts of: A. \{0, -1\}, B. \{-1, -1\}, and C. \{-25, -1\}. I use the same notation as in the previous section: the dimension array is \texttt{d}[2] and the starting subscript array is \texttt{st}[2]. The location and contents of each integer word are given exactly as printed out in hex by my example program with the following loop:

```c
for ( i=0 ; i<36 ; i+=4 )
{
    printf(“0x%08x 0x%08x\n”,
            free_ptr+i, *((int *)
                      (free_ptr+i)));
}
```

The allocated space pointed to by \texttt{free_ptr} begins the data area at 0x00021F50. The data areas are all 2x3x4=24 bytes long, and the pointer areas are all 12 bytes long for a total allocated space of 36 bytes. I assigned a value to each data element equal to the sum of the two subscripts.

**Examine A**

I first give the location and contents of the \texttt{arr} variable. This pointer is allocated prior to \texttt{daa()} and is assigned...
the return value of \texttt{daa()}. It contains the pointer, that when adjusted by the initial dimension first subscript—0, -1, and -25 in our three cases—points to the first pointer inside the \texttt{daa()} allocated block of pointers to pointers. \texttt{arr} points to \texttt{&arr[0](0x00021f68)}, which contains pointer \texttt{arr[0](0x00021f54)} pointing to the second element in the data area, \texttt{&arr[0][0](0x00021f54)}, which is the middle of the first subarray of data subscripted by the second subscript (running from \(-1\) to 1). Since the second subscript begins with \(-1\), indexing one element previous gives the location \texttt{&arr[0][-1](0x00021f50)} which contains \(-1\) \((0 + \(-1\) = \(-1\)). Since an element is four bytes, the addresses are adjusted by four bytes for every change of one in subscript. The second element of the first subscript, \texttt{arr[1]}, stored at \texttt{&arr[1](0x00021f6c)}, points to \texttt{&arr[1][0](0x00021f60)}, the fifth element of the data area, the middle of the second subarray of data subscripted by the second subscript. The decimal values of the data are in parenthesis at the right. Why does the location \texttt{0x00021f70} have a zero in it? I mentioned earlier that if the data area ended on a boundary that was not a pointer boundary, the addresses of the pointer area would have to be adjusted to word-align all pointers. In this case, the pointers did not need different alignment, so the extra allocated space shows up at the end. Stare at this for awhile, but not too long, then examine B.

Examine B

This case is only different in the starting subscript of the first dimension—it begins at \(-1\) instead of 0. \texttt{arr} now points to \texttt{&arr[0](0x00021f6c)}, which when indexed by \(-1\), gives the address \texttt{&arr[-1](0x00021f68)}, the same place as last time. From here on the pointers are identical to case A. The trick is where \texttt{arr} points to after being adjusted by the starting subscript for the first dimension. Since the basic pointer adjust unit is one integer, adjusting by \(-1\) means subtracting four bytes \((0x00021f6c - 4 = 0x00021f68)\).

Examine C

Again, this case is only different in the starting subscript, but with a more extreme -25 offset. \texttt{arr} now points to \texttt{&arr[0](0x00021f6c)} which is entirely outside the allocated space. When this allocated space is indexed by -25, it points to \texttt{&arr[-25](0x00021f68)}, the same location as the other cases, and leading to the same values. That is, \(-25\times4 = -100\) (d) = \(-64\) (h), and \(0x00021f6c - 64 = 0x00021f68\). No memory violation should occur if valid
first-dimension subscripts are used, –25 and –24, since these will adjust arr back into the allocated space.

If you want to generate some serious piles of dandruff, then try a three-dimensional array of a large struct with odd starting subscripts. At some point you just trust that the recursive routines are putting stuff where it should be. The acid test is to fill the entire array with known data and then read it back, comparing with stored values and making sure that no memory accesses were outside the allocated space.

Closing observations
These routines were designed and used in a 32-bit environment. That is not to say that they can’t be ported to other environments. However, 8- or 16-bit machines might seem to have more restrictive environments with regards to pointer usage than the 32-bit environments. Machines with 64 bits are something else. I’m sure that a port to these would not be difficult; however, such applications are still rare.

If you have ported this code to a new machine/compiler environment, I would take a few steps to ensure valid results at the outset. The first would be to run all the test suite examples and verify the expected results. The second would be to run through every dimensional extent from low to high and write/read/verify the data. The third step would be to ensure address closure, that is, print out the high and low addresses of the allocated memory and test that all pointers to pointers and pointers to data point inside these limits.

Last of all is the matter of optimization. I have used this code with both debug and optimized compiles. I have not done any detailed analysis of the difference, but it is hard to imagine how optimization will improve the traversing of the pointers that much. Try it, and let me know.

The source is available for download at www.embedded.com/code.htm.

Richard Hogaboom works for COMSYS ITS at MIT Lincoln Laboratory generating serious piles of dandruff on NASA/FAA research projects mostly in Solaris MT/MP/RT environments. He can be reached at hogaboom@ll.mit.edu.

References