Variable-Size Arrays in C

Dennis M. Ritchie

Bell Laboratories
Murray Hill, NJ, USA

[This is a version of a paper published in Journal of C Language Translation, vol 2 number 2, September 1990.]

Both the original C language and ANSI C handle vectors of indefinite length, but neither caters for multidimensional arrays whose bounds are determined during execution. Such a facility is especially useful for numerical library routines, but is currently painful to program. Many of the issues, and some suggested language changes, were discussed by Tom MacDonald in *J. C Language Translation* **1:3** (December, 1989, pp. 215-233). Some compilers, for example the GCC compiler from the Free Software Foundation, already have extensions in this area. This note discusses problems in these approaches, and offers a differing proposal for an extension.

Previous thoughts (including my own) have fixed on the idea of allowing general expressions, and not merely constants, as bounds for automatic arrays. Although this extension, with careful restrictions, seems to fit into the structure of the existing language, and is useful by itself, it presents implementation difficulties in some run-time systems. Moreover, I argue that this proposal alone does not properly resolve the question of function parameters that involve varying arrays. The rules for both the GCC and MacDonald schemes are difficult to use and comprehend, and are difficult to formalize even to the level of the current ANSI standard; in particular, the type calculus for variable-sized arrays is murky for both. In the existing ANSI C language, the type and value of an object `p` suffice to determine the evaluation of operations on it. In particular, if `p` is a pointer, the code generated for expressions like `p[i]` and `p[i][j]` depend only on its type, because any necessary array bounds are part of the type of `p`. In the MacDonald and GCC extensions, the values of non-constant array bounds are not tied firmly to its type.

My proposal, by contrast, is to avoid allowing variable-sized arrays as declared objects, but to provide pointers to such arrays; the pointers carry the array bounds inside themselves. This will preserve the property that pointer arithmetic and dereferencing depend only on the type and value of the pointer. Allocation of varying-size arrays can be done by already existing facilities.

I intend that the proposed amendments to the Standard do not change any currently conforming programs.

New Rules

To simplify the description, let us call the types 'array of *T*,' 'array of array of *T*,' and so forth *matrix types*.

The first extension permits array declarators to be written

```
D [ ? ]
```

which imputes to the declarator `D` the type 'adjustable array.' Adjustable arrays are heavily restricted: no object, or structure or union member, may be a matrix type with any adjustable bound. However, pointers to matrices, some of whose bounds are adjustable, are permitted.

The second extension permits any integral expression to appear as a bound in an array declarator that occurs in a cast (but nowhere else). Such a cast may appear only in a function, and the expression is evaluated in the scope in which the cast appears. The type of the resulting array is the same as if the bound had been the constant expression of equal value. (The implied constraints here are not statically checkable.)

The final extension specifies that the `sizeof` operator, when applied to an expression involving a pointer to an adjustable matrix, need not yield a constant; there is a corresponding constraint that the value of such a `sizeof` expression is undefined unless the value of the pointer is defined. (This implies that the argument of `sizeof` must be evaluated when its type involves an adjustable array.)

Pointers to adjustable matrices (that is, matrices at least one bound of which is adjustable) are ‘fat pointers’ whose representation will include the space to store the adjustable bounds at run-time. It is crucial to arrange the rules so that these bounds can always be filled in properly. Therefore, the following restrictions are necessary.

The type of an adjustable array of an element type is compatible only with another adjustable array, and only if the element types are compatible. In particular, an adjustable array is not compatible with an ordinary array or an incomplete array.

Pointers to adjustable matrices may be assigned (or initialized or passed as arguments). If `p` is a pointer to an adjustable matrix, then `p = q` is permitted if the types are compatible, or if `q` fails to be compatible with `p` only because one matrix has a known array bound where the other has an adjustable bound. If the LHS `p` has a known bound at a particular position and `q` has an adjustable bound, it is necessary (though not statically checkable) that the actual bound for `q` at that position be the equal to the known bound of `p`.

Although the ANSI standard made `void *` pointers a generic type for pointers to objects, it exempted function pointers from this universality. Similarly, this proposal forbids automatic coercion of `void *` pointers to pointers to adjustable matrices; a cast specifying the bounds must be used.

These rules for casts and assignments are not simple generalizations of existing rules; they are formulated so that no pointer to an adjustable matrix can be constructed unless there is a way of finding the appropriate bounds at run time.

Examples

A function with a general two-dimensional array as a parameter might be declared

```
void f1(int (*a)[?][?]);
```

and defined as

```
void f1(int (*a)[?][?]) {
     int i, j;
     ... (*a)[i][j] ...
}
```

The type of the argument is ‘pointer to adjustable array of adjustable array of `int`.’ This formulation has the advantage that both array bounds are automatically available to the function; the first and second bounds are respectively

```
sizeof(*a) / sizeof((*a)[0])
sizeof((*a)[0]) / sizeof((*a)[0][0])
```

This suggests that macros be provided:

```
#define UPB1(x)  (sizeof(*x) / sizeof((*x)[0]))
#define UPB2(x)  (sizeof((*x)[0]) / sizeof((*x)[0][0]))
```

If the function needs a temporary array of the same size and shape as the argument is needed, it can be allocated:

```
int (*b)[?][?] = (int (*)[UPB1(a)][UPB2(a)])malloc(sizeof(*a));
```

In order to call the function `f`, an ordinary array can be used:

```
int aa[100][100];

f1(&aa);
```

because the pointer to the constant-bound, complete array `aa` is converted to the argument’s type.

The ugliest aspect of this formulation is that uses of the argument need to be explicitly dereferenced; we must write `(*a)[i][j]` inside of `f1`. One way of ameliorating the effect, within the function, is to generate a pointer to the first row of the array:

```
void f1(int (*a)[?][?]) {
     int (*ap)[?] = *a;
     ... ap[i][j] ...
}
```

This example was explicit in passing a pointer to the two-dimensional array. It can also be written in the more usual style, in which the pointer's explicitness is somewhat suppressed; this gives an alternate way of simplifying the references.

```
void f2(int a[][?]) { ... a[i][j] ... }
```

or equivalently,

```
void f2(int (*a)[?]) { ... a[i][j] ... }
```

Here the type of the argument is 'pointer to adjustable array of `int`,' and what is passed is a pointer to the first row of the two-dimensional array, instead of a pointer to the entire array. This function could be called

```
int aa[100][100];

f2(aa);
```

This formulation is in some ways more natural, but just as in ANSI C, the function loses the ability to compute the first bound of the array.

Discussion

Previous attempts to generalize C to adjustable arrays have taken the apparently straightforward approach of allowing array bounds (in carefully chosen circumstances) to be arbitrary expressions, instead of constants. I suggest that the straightforwardness is only apparent, and that the necessary rules are in fact quite complicated; neither the FSF nor MacDonald have really worked out their consequences. For example, the GCC language extension envisions function definitions like

```
int f(int n, int m, int (*a)[n][m]) { ... }
```

in which the bounds are passed explicitly to the function (somewhat in the style of Fortran). Such proposals can possibly be made to work, but manifest unpleasant difficulties.

For example, what is the type of this function `f`, and how does one write a prototype for it? In GCC, one says

```
void f(int, int, int (*)[][]);
```

while MacDonald makes the adjustability indicator more syntactic:

```
void f(int, int, int [*][*]);
```

The type `int (*)[][]` here, if one has only non-constant arrays, is strange; it is a notation that has to be introduced only for function declarations, and is of no use elsewhere. More seriously, one cannot determine, simply by looking at the declaration and at a call to the function, whether the call is type-correct, because there is no language-visible way to determine where the array bounds come from.

Other problems occur because of the way the function definition is written. In the example,

```
void f(int n, int m, int (*a)[n][m]) { ... }
```

the bounds come lexically first, before the array that depends on them. The arguments really have to be processed by the compiler left-to-right, in order that `n` and `m` be defined before the appearance of `a`. If the user wrote

```
void f(int (*a)[n][m], int n, int m) { ... }
```

then, in GCC, `n` or `m` would either be undefined, or would be captured by an outside declaration. (This is, in part, an unfortunate aspect of the ANSI function prototype syntax; the problem wouldn't occur with the old syntax.)

MacDonald suggests rules that cause all the arguments be processed simultaneously, or in two passes, but the rules would be messy, and dissimilar to the approach taken in the rest of the language. One example of the problems is arguments that depend on each other:

```
void g((*p)[sizeof(*q)],  (*q)[sizeof(*p)]) { ... }
```

Thus this approach seems fraught with difficulties. Either there are strange rules about the order in which parameters are mentioned, or there are strange rules about how parameter lists are parsed.

Conclusion

This paper proposes to extend C by allowing pointers to adjustable arrays and arranging that the pointers contain the array bounds necessary to do subscript calculations and compute sizes. In this way, the problem of handling variable-size array references is solved in a way that can be made consistent with the rest of the language. By contrast, the apparently most obvious language generalization (that is, allowing general expressions instead of constants in array bounds) complicates the type structure of the language, causes difficulties with the calculus of datatypes, and causes nonuniformity in discovering the sizes of arrays.