The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

6.7.5.2 Array declarators

Constraints

In addition to optional type qualifiers and the keyword **static**, the **[** and **]** may delimit an expression or *. 1564

Commentary

Saying in words what is specified in the syntax (and in a constraint!)

C90

*The expression delimited by **[** and **]** (which specifies the size of an array) shall be an integral constant expression that has a value greater than zero.*

Support for the optional type qualifiers, the keyword **static**, the expression not having to be constant, and support for * between **[** and **]** in a declarator is new in C99.

C++

Support for the optional type qualifiers, the keyword **static**, the expression not having to be constant, and * between **[** and **]** in a declarator is new in C99 and is not specified in the C++ Standard.

If they delimit an expression (which specifies the size of an array), the expression shall have an integer type. 1565

Commentary

The expression is usually thought of, by developers, in terms of specifying the number of elements, not the size.

Other Languages

Many languages require the value of the expression to be known at translation time, and some (e.g., Ada) allow execution time evaluation of the array bounds, while a few (e.g., APL, Perl, and Common Lisp) support dynamically resizeable arrays. In some languages (e.g., Fortran) there is an implied lower bound of one. The value given in the array declaration is the upper bound and equals the number of elements. Languages in the Pascal family require that a type be given. The minimum and maximum values of this type specifying the lower and upper bounds of the array. This type also denotes the type of the expression that must be used to index that array. For instance, in:

```
C_A :Array[Char] of Integer;
```

the array C_A can only be indexed with an expression having type **char**.

Coding Guidelines

At the time of this writing there is insufficient experience with use of this construct to known whether any guideline recommendation (e.g., on the appearance of side effects) might be worthwhile.

array declaration size greater than zero

If the expression is a constant expression, it shall have a value greater than zero. 1566

Commentary

C does not support the creation of named objects occupying zero bytes of storage.

Note that this constraint is applied before any implicit conversions of array types to pointer types (e.g., even although the eventual type of a in `extern int f(int a[-1]);` is pointer to **int**, a constraint violation still occurs).

Other Languages

Very few languages support the declaration of zero sized objects (Algol 68 allows the declaration of a flexible array variable to have zero size, assigning an array to such a variable also resulting in the new bounds being assigned). Languages in the Pascal family support array bounds less than one. For instance, the declaration:

```
1   C_A :Array[-7..0] of Integer;
```

creates an array object, C_A, that must be indexed with values between -7 and 0 (inclusive).

1567 The element type shall not be an incomplete or function type.

array element not incomplete type
array element not function type

Commentary

The benefits of allowing developers to declare arrays having element types that are incomplete types was not considered, by the C committee, to be worth the complications (i.e., cost) created for translator writers (the C language definition is intended to translatable in a single pass). C does not support treating functions as data, so while pointers to functions are permitted arrays of function types are not.

implementation single pass

C90

In C90 this wording did not appear within a Constraints clause. The requirement for the element type to be an object type appeared in the description of array types, which made violation of this requirement to be undefined behavior. The undefined behavior of all known implementations was to issue a diagnostic, so no actual difference in behavior between C90 and C99 is likely to occur.

array contiguously allocated set of objects

C++

8.3.4p1

T is called the array element type; this type shall not be a reference type, the (possibly cv-qualified) type **void***, a function type or an abstract class type.*

The C++ Standard does not disallow incomplete element types (apart from the type **void**). This difference permits a program, written using only C constructs, to be acceptable to a conforming C++ implementation but not be acceptable to a C implementation.

Other Languages

Few languages support arrays of functions (or pointers to functions).

1568 The optional type qualifiers and the keyword **static** shall appear only in a declaration of a function parameter with an array type, and then only in the outermost array type derivation.

array parameter qualifier only in outermost

Commentary

The functionality provided by the appearance of type qualifiers and the keyword **static** in this context is not needed for declarations involving array types in other contexts (because function parameters are the only context where the declaration of an object having an array type is implicitly converted to a pointer type).

array converted to pointer

C90

Support for use of type qualifiers and the keyword **static** in this context is new in C99.

C++

Support for use of type qualifiers and the keyword **static** in this context is new in C99 is not supported in C++.

Coding Guidelines

Support for these constructs is new in C99 and insufficient experience has been gained with their usage to know if any guideline recommendations are worthwhile.

variable modified only scope

~~Only an~~ An ordinary identifier (as defined in 6.2.3) ~~with both block scope or function prototype scope and no linkage shall have a variably modified type.~~ that has a variably modified type shall have either block scope and no linkage or function prototype scope. 1569

Commentary

ordinary identifiers

Ordinary identifiers do not include members of structure or union types. These members are prevented from having a variably modified type for practical implementation reasons. For instance, requirement on the relative ordering of storage allocated to members of a structure cannot be met without reserving impractically large amounts of storage for any member having a variably modified type (variably modified types are usually implemented via a pointer to the actual, variable sized, storage allocated).

structure members later compare later

VLA lifetime starts/ends

static storage duration when initialized

Storage for objects having external or internal linkage is only allocated once and it makes no sense for such objects to have a variably modified type.

This wording was changed by the response to DR #320.

C90

Support for variably modified types is new in C99.

C++

Support for variably modified types is new in C99 and is not specified in the C++ Standard.

If an identifier is declared to be an object with static storage duration, it shall not have a variable length array type. 1570

Commentary

static storage duration when initialized

The standard requires that storage for all objects having static storage duration be allocated during program startup. Thus the amount of storage to allocate must be known prior to the start of program execution, ruling out the possibility of execution time specification of the length of an array.

Semantics

qualified array of

If, in the declaration "**T D1**", **D1** has one of the forms: 1571

```
D[ type-qualifier-list_opt assignment-expression_opt ]
D[ static type-qualifier-list_opt assignment-expression ]
D[ type-qualifier-list static assignment-expression ]
D[ type-qualifier-list_opt * ]
```

and the type specified for *ident* in the declaration "**T D**" is "*derived-declarator-type-list T*", then the type specified for *ident* is "*derived-declarator-type-list* array of *T*".[121)]

Commentary

derived-declarator-type-list

Phrases such as *derived-declarator-type-list* are rarely used outside of the C standard, even amongst the Committee.

C90

Support for forms other than:

```
D[ constant-expression_opt ]
```

is new in C99.

C++

The C++ Standard only supports the form:

```
D[ constant-expression_opt ]
```

A C++ translator will issue a diagnostic if it encounters anything other than a constant expression between the [and] tokens.

The type of the array is also slightly different in C++, which include the number of elements in the type:

> *. . . the array has N elements numbered 0 to N-1, and the type of the identifier of D is "derived-declarator-type-list array of N T."* 8.3.4p1

Other Languages

Some languages (e.g., the Pascal family of languages) require both a lower and upper bound to be specified. Although BCPL uses the tokens [ ] to denote an array access (and also a function call), these tokens are not used in an array declaration. For instance, the following creates an array of 10 elements (lower bound of zero, upper bound of nine) and assigns a pointer to the first element to N:

```
1   let N = vec 9
```

1572 (See 6.7.5.3 for the meaning of the optional type qualifiers and the keyword **static**.)

Commentary

See elsewhere for commentary on optional type specifiers and the keyword **static**.

array type
adjust to pointer to
function declarator
static
array
incomplete type

1573 If the size is not present, the array type is an incomplete type.

Commentary

This is stated is a different way elsewhere. As well as appearing between [] tokens, the size may also be specified, in the definition of an object, via the contents of an initializer.

array
unknown size
array of unknown size
initialized

C++

The C++ Standard classifies all compound types as object types. It uses the term *incomplete object type* to refer to this kind of type.

object types

> *If the constant expression is omitted, the type of the identifier of D is "derived-declarator-type-list array of unknown bound of T," an incomplete object type.* 8.3.4p1

Coding Guidelines

The following are some of the contexts in which the size might not be present:

- An automatically generated initializer, used in an object definition, may be written to a file that is **#include**d at the point of use (while the generation of the members of the initializer may be straightforward, the cost of automatically modifying the C source code, so that it contains an explicit value for the number of elements may be not be considered worth the benefit).
- A typedef name denoting an array type of unknown size is a useful way of giving a name to an array of some element type (the size being specified later when an object, of that named type, is defined).
- The type of a parameter in a function declaration. While this usage may seem natural to developers who have recently moved to C from some other languages, it looks unusual to experienced C developers (the parameter is not treated as an array at all, its type is converted to a pointer to the element type).
- The type of an object declared in a header, where there is a desire to minimize the amount of visible information. In this case a pointer could serve the same purpose. The only differences are in how the storage for the object is allocated (and initialized) and the machine code generated for accesses

array of unknown size
initialized
array
converted to pointer

(a pointer requires an indirect load, a minor efficiency penalty; the use of an array can simplify an optimizers attempt to deduce which objects are not aliased, potentially leading to higher-quality machine code. The introduction of the keyword **restrict** in C99 has potentially removed this advantage).

While not having the number of elements explicitly specified in the declaration of an array type may be surprising to users of other languages (experienced users of other languages, sometimes promoted to a level whether they no longer write code and are not experienced with C, are surprisingly often involved in the creation of a company's coding guidelines) the usage is either cost effective or the alternatives offer no additional benefits. For these reasons no guideline recommendation is given here.

Example

In the following:

```
1  extern void f(int *p1, int *p2);
2  extern void f(int p1[], int p2[3]);
```

the first parameter is not an array of known size in either declaration, but the types are compatible. Possible composite types are:

```
1  extern void f(int *p1, int p2[3]);
```

and

```
1  extern void f(int p1[], int p2[3]);
```

In the following:

```
1  extern int g(int p3[3]);
2  extern int g(int p3[5]);
```

the parameter types are compatible (because both are first converted to pointers before checking for compatibility). Possible composite types are:

```
1  extern int g(int p3[3]);
```

and

```
1  extern int g(int p3[5]);
```

In the following:

```
1  extern int h(int n, int p4[2]);
2  extern int h(int n, int p4[n]);
```

The composite type is:

```
1  extern int h(int p4[2]);
```

variable length array specified by *

If the size is * instead of being an expression, the array type is a variable length array type of unspecified size, which can only be used in declarations with function prototype scope;[122)] 1574

Commentary

It is possible that implementations will want to use different argument passing conventions for variable length array types than for other array types. The * notation allows a variable length array type to be specified without having to specify the expression denoting its size, removing the need for the identifiers used in the size expression to be visible at the point in the source the function prototype is declared. If an expression is given in function prototype scope, it is treated as if it were replaced by *.

1581 VLA size treated as *

Syntactically the size can be specified using * in any context that an array declarator can appear. If this context is not in function prototype scope the behavior is undefined.

C90

Support for a size specified using the * token is new in C99.

C++

Specifying a size using the * token is new in C99 and is not available in C++.

Other Languages

Several languages (e.g., Fortran) use * to denote an array having an unknown number of elements.

1575 such arrays are nonetheless complete types.

Commentary

The size of a parameters type in function prototype scope is not needed by a translator. Specifying that arrays declared using the * notation are complete types enables pointers to variable length arrays, and arrays having more than one variable length dimension, to be declared as parameters, e.g., `int f(int p_1[*][*], int (*p_2)[*])`.

1576 If the size is an integer constant expression and the element type has a known constant size, the array type is not a variable length array type;

variable length array type

Commentary

This defines a variable length array type in terms of what it is not. In:

known constant size

```
1  extern int n,
2             m;
3  int a_1[5][n]; /* Size is not an integer constant expression. */
4  int a_2[m][6]; /* Element type is not a known constant size.  */
```

C90

Support for specifying a size that is not an integer constant expression is new in C99.

C++

Support for specifying a size that is not an integer constant expression is new in C99 and is not specified in the C++ Standard.

1577 121) When several "array of" specifications are adjacent, a multidimensional array is declared.

footnote 121

Commentary

Although not notated as such, this is the definition of the term *multidimensional array*.

Other Languages

The process of repeating array declarators to create multidimensional arrays is used in many languages. Some languages (e.g., those in the Algol family) support the notational short cut of allowing information on all dimensions to occur within one pair of brackets (when this short cut is used it is not usually possible to access slices of the array). For instance, a multidimensional array declaration in Pascal could be written using either of the following forms:

```
1  Arr_1 : Array[1..4] of Array[0..3] of Integer;
2  Arr_2 : Array[1..4, 0..3] of Integer;
```

while in Fortran it would be written:

```
1  INTEGER Arr_2(10,20)
```

Common Implementations

To improve performance, when accessing multidimensional arrays, Larsen, Witchel, and Amarasinghe[1] found that it was sometimes worthwhile to pad the lowest dimension array to ensure the next higher dimension started at a particular alignment (whole program analysis was used to verify that there was no danger of changing the behavior of the program; multidimensional arrays are required to have any no padding between dimensions). For instance, if the lowest dimension was specified to contain three elements, it might be worthwhile to increase this to four.

---

footnote 122

122) Thus, * can be used only in function declarations that are not definitions (see 6.7.5.3). 1578

Commentary

object type complete by end

The size of all parameters in a function definition are required to be known (i.e., they have complete types), even if the parameter is only passed as an argument in a call to another function.

---

array variable length

otherwise, the array type is a variable length array type. 1579

Commentary

This is true by definition.

If the size is an expression that is not an integer constant expression: 1580

Commentary

integer constant expression

That is, the expression does not have a form that a translator is required to be able to evaluate to an integer constant during translation.

C90

Support for non integer constant expressions in this context is new in C99.

C++

Support for non integer constant expressions in this context is new in C99 and is not available in C++.

VLA size treated as *

if it occurs in a declaration at function prototype scope, it is treated as if it were replaced by *; 1581

Commentary

The implication of this wording is that the expression is not evaluated. A size expression occurring in a declaration at function prototype scope serves no purpose other than to indicate that the parameter type is a variably modified array type.

variable length array specified by * 1574

Other Languages

Languages that support execution time evaluation of an arrays size face the same problems as C. Few checks can be made on the type during translation and the expression essentially has to be treated as evaluating to some unknown value.

Coding Guidelines

The expression might be given for documentation purposes. It provides another means for developers to obtain information about the expected size of the array passed as an argument. Whether such usage is more likely to be kept in sync with the actual definition than information given in a comment is not known. A guideline recommendation that the expression be identical to that in the function definition is not given for the same reason that none is given for keeping comments up-to-date with the code.

comment disadvantages

1582 otherwise, each time it is evaluated it shall have a value greater than zero.

Commentary

This specification is consistent with that given when the expression is a constant. The expression is evaluated each time the function is called, even if the call is a recursive one.

1566 array declaration size greater than zero

Example

```
#include <stddef.h>

size_t fib(size_t n_elems, char arr[static n_elems],
           char p_a[static (n_elems == 1) ? 1 : (n_elems + fib(n_elems-1, arr, arr))])
{
return sizeof(p_a);
}
```

1583 The size of each instance of a variable length array type does not change during its lifetime.

Commentary

The implication is that subsequent changes to the values of objects appearing in the expression do not affect to size of the VLA. The size expression is evaluated only when the declarator containing it is encountered during program execution. The number of elements in the array type is then fixed throughout its lifetime. In the following the change in the value of `glob` does not array the value of `sizeof(arr)`.

object initializer evaluated when function entry parameter type evaluated

```
extern int glob;

void f(void)
{
int arr[glob];

glob++;
sizeof(arr);
}
```

A variable length array type need not denote a named object. For instance, in:

```
void f(int n)
{
static char (*p)[n++];
}
```

the pointer `p` has a variably modified type, but is not itself a VLA. Even although storage for `p` is allocated at program startup its type is not known until the array declarator is evaluated, each time `f` is called (any assignment to `p` has to be compatible with this type).

Other Languages

A few languages (e.g., APL and Perl) support arrays that change their size on an as-needed basis.

Coding Guidelines

It is possible that the expression denoting the size of the VLA array will also appear elsewhere in the function, e.g., to perform some array bounds test. Any modification of the value of an object appearing in this expression is likely to cause the value of the two expressions to differ. Until more experience is gained with the use of VLA types it is not possible to evaluate the cost/benefit of any potential guideline recommendations (e.g., recommending against the modification of objects appearing in the expression denoting the size of a VLA).

sizeof VLA unspecified evaluation

Where a size expression is part of the operand of a **sizeof** operator and changing the value of the size expression would not affect the result of the operator, it is unspecified whether or not the size expression is evaluated. 1584

Commentary

The rather poor argument (in your authors opinion) put forward, by some Committee members, for this unspecified behavior, was that there were existing implementations that did not evaluate sub-operands that did not affect the value returned by the **sizeof** operator.

C90

The operand of **sizeof** was not evaluated in C90. With the introduction of variable length arrays it is possible that the operand will need to be evaluated in C99.

Common Implementations

As part of the process of simplifying and folding expressions some translators only analyze as much of their intermediate representation as is needed. Such translators are unlikely to generate machine code to evaluate any operands of the **sizeof** operator that are not needed to obtain the value of an expression.

Coding Guidelines

Evaluation of the operands of **sizeof** only becomes an issue when it may appear to generate a side effect. However, no guideline recommendation is given here for the same reason that none was given for the non-VLA case.

sizeof operand not evaluated

Example

```
1  extern int glob;
2
3  void f(int n,
4         int arr[++n]) /* Expression evaluated. */
5  {
6  int (*p)[glob++];    /* Expression evaluated. */
7  /*
8   * To obtain the size of the type ++n must be evaluated,
9   * but the value of ++glob does not affect the final size.
10  */
11 n=sizeof(int * (int * [++glob])[++n]);
12 }
```

array type to be compatible

For two array types to be compatible, both shall have compatible element types, and if both size specifiers are present, and are integer constant expressions, then both size specifiers shall have the same constant value. 1585

Commentary

structural compatibility

While the compatibility rules for structure and union types are based on the names of the types, array type compatibility is based on what is sometimes called *structural compatibility*. The components of the type, rather than just its name, is compared. This requirement can be verified at translation time. If either size specifier is not present then the array types are always compatible (provide their element types are).

structural compatibility 1585

compatible type if

C++

compatible type if

The C++ Standard does not define the concept of compatible type, it requires types to be the same.

Other Languages

Languages in the Pascal family usually compare the names of array types, rather than their components. This is sometimes considered too inconvenient when passing arguments having an array type and special rules for argument/parameter array type compatibility are sometimes created (e.g., conformant arrays in Pascal).

Some languages (e.g., CHILL) use *structural compatibility* type checking rules. When the lower bound of an array is specified by the developer it is necessary to check that both the lower and upper bounds, for each dimension of the two arrays, have the same value.

1586 If the two array types are used in a context which requires them to be compatible, it is undefined behavior if the two size specifiers evaluate to unequal values.

Commentary

In most contexts expressions having an array type are converted to a pointer to their element type. Contexts where the size specifiers need to be considered include the following:

array converted to pointer

- an argument whose corresponding parameter has an array type that includes the type qualifier **`static`**, and

function declarator static

- assignment to an object having type pointer to array of type.

assignment-expression syntax

C90

The number of contexts in which array types can be incompatible has increased in C99, but the behavior is intended to be the same.

C++

There are no cases where the C++ Standard discusses a requirement that two arrays have the same number of elements.

Common Implementations

In practice the unexpected (undefined) behavior may not occur after the point of assignment. For instance, at the point where an object having the array type is accessed. For arrays having a single dimension, provided the access is within the bounds of the object, an implementation is likely to behave the same way as when the two array types have equal size specifiers. For multidimensional arrays the number of elements in each array declarator, except the last one, is used as a multiplier for calculating the index value. This means that even if the object, having an array type, is larger than required the incorrect element will still be accessed.

```
1  static int glob = 9;
2
3  void f(char p_arr[static 4][static 6])
4  {
5  /*
6   * The following will access location p_arr+(2+3*4) had the sizes
7   * been the same location p_arr+(2+3*9) would have been accessed.
8   */
9  p_arr[2][3] = 9;
10 }
11
12 void g(void)
13 {
14 char arr[glob][4];
15
16 f(arr);
17 }
```

Coding Guidelines

Developers may intentionally write code where the two array specifiers have different values, relying on the behavior being defined when only allocated storage is accessed. Support for VLA types are new in C99 and there is not yet sufficient experience to be able to judge the cost/benefit of recommending against this usage. Developers may also accidentally write code where two array specifiers have different values. However, these coding guidelines are not intended to recommend against the use of constructs that are obviously faults.

guidelines not faults

EXAMPLE
array of pointers

EXAMPLE 1 1587

```
float fa[11], *afp[17];
```

declares an array of **float** numbers and an array of pointers to **float** numbers.

Commentary

The algorithm for reading declarations involving both array and pointer types is to:

1. start at the identifier and work right through each array size, until a semicolon or closing parenthesis is reached,
2. then restart at the identifier and work left through the type qualifiers and *'s,
3. if a closing parenthesis is reached on step 1, it the opening parenthesis will eventually be reached on step 2. This parentheses bracketed sequence is then treated as an identifier and the process repeated from step 1.

EXAMPLE
function returning pointer to

A similar rule can be applied to reading function types.

Coding Guidelines

symbolic name

The discussion on using symbolic names rather than integer constants is applicable here.

Example

```
int *   (*   (*glob[1]      [2])     [3])     [4];
/*
                    array of
                             array of
                pointer to
                                      array of
          pointer to
                                               array of
     pointer to

or in linear form:

array of array of pointer to array of pointer to array of pointer to int
*/
```

EXAMPLE
array not pointer

EXAMPLE 2 Note the distinction between the declarations 1588

```
extern int *x;
extern int y[];
```

The first declares **x** to be a pointer to **int**; the second declares **y** to be an array of **int** of unspecified size (an incomplete type), the storage for which is defined elsewhere.

Commentary

declarator
syntax

The issue of the same object being declared to have both pointer and array types, one in each of two translation units, is discussed elsewhere.

1589 EXAMPLE 3 The following declarations demonstrate the compatibility rules for variably modified types.

```
extern int n;
extern int m;
void fcompat(void)
{
        int a[n][6][m];
        int (*p)[4][n+1];
        int c[n][n][6][m];
        int (*r)[n][n][n+1];
        p = a;        // invalid: not compatible because 4 != 6
        r = c;        // compatible, but defined behavior only if
                      // n == 6 and m == n+1
}
```

Commentary

Array types having more than two dimensions are rarely seen in practice (see Table **??**).

Other Languages

In some languages (e.g., APL, Perl) assigning a value may change the type of the object assigned to (to be that of the value assigned).

1590 EXAMPLE 4 All declarations of variably modified (VM) types have to be at either block scope or function prototype scope. Array objects declared with the **static** or **extern** storage-class specifier cannot have a variable length array (VLA) type. However, an object declared with the **static** storage-class specifier can have a VM type (that is, a pointer to a VLA type). Finally, all identifiers declared with a VM type have to be ordinary identifiers and cannot, therefore, be members of structures or unions.

```
extern int n;
int A[n];                               // invalid: file scope VLA
extern int (*p2)[n];                    // invalid: file scope VM
int B[100];                             // valid: file scope but not VM

void fvla(int m, int C[m][m]);          // valid: VLA with prototype scope

void fvla(int m, int C[m][m])           // valid: adjusted to auto pointer to VLA
{
        typedef int VLA[m][m];          // valid: block scope typedef VLA

        struct tag {
                int (*y)[n];            // invalid: y not ordinary identifier
                int z[n];               // invalid: z not ordinary identifier
        };
        int D[m];                       // valid: auto VLA
        static int E[m];                // invalid: static block scope VLA
        extern int F[m];                // invalid: F has linkage and is VLA
        int (*s)[m];                    // valid: auto pointer to VLA
        extern int (*r)[m];             // invalid: r has linkage and points to VLA
        static int (*q)[m] = &B;        // valid: q is a static block pointer to VLA
}
```

Commentary

In this example `m` is being used in the same way as a symbolic name might be used in the declaration of non-VM array types.

1591 **Forward references:** function declarators (6.7.5.3), function definitions (6.9.1), initialization (6.7.8).

References

1. S. Larsen, E. Witchel, and S. Amarasinghe. Techniques for increasing and detecting memory alignment. Technical Report MIT-LCS-TM-621, MIT, USA, Nov. 2001.