**Matching of Multi-Dimensional Arrays in Generic Selection Expressions (updates N3290)**

Author: Martin Uecker

Number: n3348
Date: 2024-09-14

Changes since n3290: updated wording

**Introduction:** In function declarations that are not definitions, we allow the use of `[*]` syntax to indicate size expressions that are only later specified in the definition of the function.

```
int foo(int, char (*buf)[*]);

int foo(int N, char (*buf)[N])
{
     return sizeof *buf;
}
```

The syntax is not too useful (except in rare cases where it is needed), but otherwise intuitive as it resembles a wild card in different context. Here, it is proposed to adopt it for generic selection expressions, where it would be much more useful, simply by using the same rules for type compatibility C already has. With generic selection expressions it is possible to match types at compile time. This was improved with N3260 which allows direct use of type names instead of a controlling expression (we use this in the examples for simplicity, but they would also work equally with an expression).

Generic selection expressions can be used to match specific sizes for array types.

```
     _Generic(int[3], int[3]: 3, int[2]: 2)
```

When the size is irrelevant or dynamic, we can also match incomplete arrays (since N3260):

```
     _Generic(int[3], int[]: 1, char[]: 0)
     _Generic(int[n], int[]: 1, char[]: 0)
```

Unfortunately, this does not work for for multi-dimensional arrays, because incomplete element types are not allowed. The following example is rejected.

```
     _Generic(int[3][2], int[][]: 1)
```

**Proposal:** The proposal is to allow the star notation to be used here to allow matching of multi-dimensional arrays types:

```
     _Generic(int[3][2], int[*][*]: 1)
     _Generic(int[3][2], int[3][*]: 1)
     _Generic(int[3][2], int[*][2]: 1)
```

Which would then also work naturally in the case of variably modified types:

```
     _Generic(int[3][n], int[*][*]: 1)
```

This does not cause any new problems or fundamentally new rules as **this matching essentially already works in C23**, simply by moving the type name in a syntactic context where this is already allowed: Into function prototype scope.

```
_Generic(void(*)(int(*)[3][n]),
         void(*)(int(*)[3][*]): 1);
```

https://godbolt.org/z/bqhrEcGn4

Note that the second indirection via a pointer in the parameter types is to avoid the effect of adjustment that would transform the array into a pointer, losing the size information. Of course, having to use this workaround is annoying and not obvious to programmers.

**Variably Modified Types in Generic Associations**

For generic selection expressions, it is currently a constraint violation if the type name in a generic association is variably modified type, which – in addition to another constraint – also would prevent the use of [*] for this purpose. Removing this constraint does not seem to introduce any new problems or complications, as the controlling expression is already allowed to be variably modified and not evaluated. The following example is allowed at compile time but is run-time UB for n != 3:

```
typedef int vmt[n];
_Generic(vmt, int[3]: 1);
```

https://godbolt.org/z/To1W346r8

Therefore, there is no reason **_Generic** with swapped roles could not also be allowed.

```
_Generic(int[3], vmt: 1)
```

All possible problems which could potentially be avoided by excluding variably modified types at this point can also arise in the first version. On the other hand, there are examples where the constraint is surprising and may lead to a programmer to work around in a way that then leads to run-time UB, *i.e.* by inserting incorrect bounds to make the code compile.

```
int (*vmt)[n];
_Generic(vmt, int (*)[n]: 1);      // error!
_Generic(vmt, typeof(vmt): 1);     // error!

_Generic(vmt, int (*)[1]: 1);      // compiles, run-time UB
```

As arrays with unspecified sizes are variably modified types, removing this constraint and also allowing [*] in this syntactic context is sufficient to enable this feature. In fact, all the existing rules for type compatibility will then kick in and automatically give the desired semantics as the example using function prototypes above demonstrates. Alternatively, one could add a new constraint that carves out an exception specifically for [*], but the former is simpler and seems preferable. Consequently, the following wording simply allows variably modified types in associations and clarifies that those type names are then not evaluated (if not even the controlling expression is evaluated, we should not evaluate the type names in associations either).

**Proposed Wording (relative to N3301)**

Changes since N3290: Updated wording to N3301 and added change to 6.7.7.3p5 as requested in SC22WG14.26570.

**6.7.7.3 Array declarators**

**Semantics**

4 If the size is not present, the array type is an incomplete type. 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 as part of the nested sequence of declarators or abstract declarators for a parameter declaration **or as part of such a sequence in a type name of a generic association,** not including anything inside an array size expression in one of those declarators;159) such arrays are nonetheless complete types. 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; otherwise, the array type is a variable length array type. (Variable length arrays with automatic storage duration are a conditional feature that implementations may support; see 6.10.10.4.)

159) Thus, * can **not** be used ~~**only**~~ in function declarations that are ~~**not**~~ definitions (see 6.7.7.4 and 6.9.2).

5 If the size is an expression that is not an integer constant expression: if it occurs in a declaration at function prototype scope **or in a type name of a generic association (as described above)**, it is treated as if it were replaced by *; otherwise, each time it is evaluated it shall have a value greater than zero.

**6.5.2.1 Generic selection**

**Constraints**

2 A generic selection shall have no more than one default generic association. ~~**The type name in a generic association shall specify a type other than a variably modified type.**~~ No two generic associations in the same generic selection shall specify compatible types. If the generic controlling operand is an assignment expression, the controlling type of the generic selection expression is the type of the assignment expression as if it had undergone an lvalue conversion,88) array to pointer conversion, or function to pointer conversion. Otherwise, the controlling type of the generic selection expression is the type designated by the type name. The controlling type shall be compatible with at most one of the types named in the generic association list. If a generic selection has no default generic association, its controlling type shall have type compatible with exactly one of the types named in its generic association list

**Semantics**

3 The generic controlling operand ~~**is**~~ **and size expressions and typeof operators contained in the type names of generic associations are** not evaluated. If a generic selection has a generic association with a type name that is compatible with the type of the controlling type, then the result expression of the generic selection is the expression in that generic association. Otherwise, the result expression of the generic selection is the expression in the default generic association. None of the expressions from any other generic association of the generic selection is evaluated.

EXAMPLE: The following generic selection expressions are valid and all evaluate to 1.

```
void foo(int n, int m)
{
     _Generic(int[3][2], int[3][*]: 1, int[2][*]: 0);
     _Generic(int[3][2], int[*][2]: 1, int[*][3]: 0);
     _Generic(int[3][n], int[3][*]: 1, int[2][*]: 0);
     _Generic(int[n][m], int[*][*]: 1, char[*][*]: 0);
     _Generic(int(*)[2], int(*)[*]: 1);
}
```