Comments on Array Sizes

Document: n3428
Author: Martin Uecker
Date: 2025-01-10

Introduction

In this paper, we discuss the use of arrays in C. It also serves as a response to the enhancements to _Generic proposed in N3441.

C has arrays with different status with respect to its size.

```
int a[4]; // array of known constant length

int a[]; // array of unknown length

int n = 3;
int a[n]; // variable length array

int a[*]; // array of unspecified size

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

There is an important conceptual distinction between incomplete arrays and array of unspecified size. In both case the size is not known at compile-time, but their purpose is different which will be explained in the following.

Unknown Size vs. Unspecified Size

Incomplete arrays - as any other incomplete types - are restricted from being used in any case where the size is required. Examples are as struct members or elements of other arrays.

```
struct {
    int a[];    // constraint violation
    int b;
};

int a[3][];     // constraint violation
```

In contrast, arrays of unspecified size can be used where complete types are required.

```
int f(int, int, char a[*][*]);  // ok
```

Because the size is not specified, arrays of unspecified size can not actually be used at run-time where the size is required. Consequently, arrays of unspecified sizes can be used only unevaluated contexts. Currently, they are restricted to function prototypes, where an array of unspecified size can be used in place of

a array of variable size. In fact, when not part of a function definition, variable sizes in function parameters are automatically treated as unspecified (and never evaluated). Hence, the following two declarations are equivalent from the point of the standard with the only difference that the one with the sizes can enable better warnings.

```
int f(int, int, char a[*][*]);  // ok
int f(int n, int m, char a[n][m]);
```

This example also illustrates why it is important that arrays of unspecified size can be used in scenarios where incomplete types are forbidden: The unspecified size is a placeholder for a variable size which is declared later.

Vice versa, an array of unknown length can be used where variably modified types are forbidden:

```
struct foo {
    char (*buf)[];      // ok
};
```

In contrast, an array of unspecified size is forbidden in structures because it is a variably modified type.

```
int f(struct foo { char (*buf)[*]; });  // constraint violation
```

In summary, while both concepts relate to missing information, they serve different and orthogonal roles in the standard. So far, they are used consistently in the ISO C standard.

Potential Future Use of Arrays of Unspecified Size

Currently, arrays of unspecified size are only allowed in function prototypes. But they could consistently be used in any other unevaluated contexts, e.g. where they are replaced by specified sizes before use. This would make them ideal choice as a wildcard size in the type name of _Generic associations (which are also never evaluated) as proposed in N3348.

```
_Generic(p, int (*)[*][*]: 1);
```

This works perfectly already with all existing rules for type compatibility, which can be shown by artificially placing the array in a function type as an argument type. In fact, simply removing the error in GCC already makes this feature work, demonstrating that this extension is very natural.

Another potential use case would be on the right hand side of underspecified declarations as proposed in N3427.

```
int (*p)[3] = malloc(sizeof *p);
int (*q)[*] = p;
auto q2 = p;    // possible in C23 but hides information
```

Arrays Sizes in Generic (A Response to N3441)

N3441 invents more complicated rules for _Generic. I am skeptical about the proposed rules for the reasons explained in the following. N3441 start with this basic example:

```
int main() {
    int arr[10] = {};
    int result = _Generic(typeof(arr),
        int[10]: 0,
        int[11]: 1,
        default: 3
    );
    return result;
}
```

This correctly selects the right branch with length '10' and this would not change with any proposal. It gets more interesting with VLAs:

```
int main() {
    int n = 20;
    int arr[n] = {};
    int result = _Generic(typeof(arr),
        int[10]: 0,
        int[11]: 1,
        int[20]: 2,
        default: 3
    );
    return result;
}
```

This is currently rejected at compile time as a constraint violation. N3441 proposes to make this valid by making the VLA case match the `default` branch. In my opinion, rejecting this seems preferable and the safer choice. With the rules proposed in N3441, the `n == 20` case would *not* be handled by the `int[20]` but by the default branch. This would introduce a subtle difference in semantics depending on whether the array is a VLA or not. Consider the following example which changes `N` to a constant (it could also use `constexpr`).

```
int main() {
    enum { N = 20 };
    int arr[N] = {};
    int result = _Generic(typeof(arr),
        int[10]: 0,
        int[11]: 1,
        int[20]: 2,
        default: 3
    );
    return result;
}
```

This is a minor and seemingly harmless change in the program that currently does not cause a change in run-time semantics in other scenarios. With N3441,

the modifies program would now return ‘2’ instead of ‘3’ - introducing a new risk for introducing errors!

In fact, we should honor the general principle that *changing a variable to a constant should not break a valid program or cause a change in behavior.* While exceptions to this rule could be made, those should be very explicit and not retrospectively added to existing language features.[1]

N3348 would allow the use of VLA in _Generic associated branches. N3441 considers the following example.

```
int main() {
    int n = 20;
    int vla[n] = {};
    int result = _Generic(typeof(vla),
        int[10]: 0,
        int[11]: 1,
        int[ n]: 2, // VLA matches?
        default: 3
    );
    return result;
}
```

According to the existing rules and also still with the extension from N3348 this would be a constraint violation. N3441 considers it a problem that this is currently constraint violation. From my position, this is not a problem but prevents the same issue already discussed above.

Another example considered in N3441 is the following.

```
int main() {
    int n = 10;
    int arr[n] = {}; // Variable-length array
    int result = _Generic(typeof(arr),
        int[11] : 0, // this matches
        default: 1
    );
    return result;
}
```

This example is undefined behavior at runtime in C23. This is a valid concern, but N3441 does not seem to propose a fix to this problem as it falls back to existing rules in this case. In this context, it is worth noting that mismatches in size expression are not specific to _Generic. In C23 even simple assignment of arrays with inconsistent sizes is undefined behavior.

```
int n = 10;
int arr[n];
int (*p)[11] = &arr;
```

[1] A separate language feature that allows the user to distinguish between integer constant expressions and non-constants could be considered, e.g. something providing the functionality of the Linux kernel’s ___is_constexpr macro (link: Github).

A Realistic Example on How Generic and VLAs Interact

We want to point out why the current model in C where VLA types are compatible with fixed sized arrays is useful. In fact, it is a crucial - but apparently not well understood - property of the type system that a VLA and fixed-size array of the same type can be used together! Consider the following example.

```
void matmul(int N, double out[N][N],
                   const double in1[N][N],
                   const double in2[N][N]);

double out[3][3];
const double in1[3][3] = { /* ... */ };
const double in2[3][3] = { /* ... */ };

matmul(3, out, in1, in2);
```

Despite the parameters having VLA types, fixed-size arrays can be assigned to them. If this would not work, this feature would be much less useful.

Now, let's consider a type-generic version of matmul as a more realistic example on how _Generic could be used together with VLAs.

```
#define matmul(N, o, i1, i2) \
    _Generic(typeof(o), \
        float[N][N]: matmul_float, \
        double[N][N]: matmul_double)(N, o, i1, i2)
```

We can now use this generic macro and apply it in different scenarios. Just like for the regular C function, we argue that all three examples should work for the generic matmul function, and should work identically (Link: Godbolt Compiler Explorer).

```
float o[3][3];
const float i1[3][3] = { };
const float i2[3][3] = { };
matmul(3, o, i1, i2);

int N = 3;
float o[N][N];
const float i1[N][N] = { };
const float i2[N][N] = { };
matmul(3, o, i1, i2);

float o[N][N];
const float i1[N][N] = { };
const float i2[N][N] = { };
matmul(N, o, i1, i2);
```

Currently, the first two indeed do work, matching fixed-size arrays to fixed-size arrays or VLAs to fixed-size arrays, respectively. This shows how important it is that fixed-size array match VLAs in _Generic! The last case does not work in C23 and N3348 proposes to remove this asymmetry. This would be a

huge improvement in usability, but is rejected and labelled 'poor behavior' in N3441! In fact, the key assumption of N3441 seems to be that the seamless interoperability of VLAs and fixed size arrays demonstrated in all three cases above is somehow a misfeature and the user would rather want to distinguish between them. But in our experience programming numerical code using VLAs - and as these examples hopefully demonstrate - it is the exact opposite: We need to treat VLAs and fixed-size arrays coherently as much as possible, with seamless handover from fixed-size arrays to length-generic functions and back. **This means that VLAs have to match fixed size arrays and vice versa!**

We acknowledge that this comes at price: When sizes do not match when converting from a VLA to another array type, then there is run-time UB. But this is not inherently different to, for example, converting to a void pointers and back: If the user gets this wrong, there can be UB at run-time. In contrast to converting void pointers, where the UB is difficult to detect (it requires tracking effective type for memory locations), the UB here can easily be checked at run-time with neglectable overhead[2] and often also diagnosed at compile-time.

As a final example, we enhance the macro to not only select based on the first argument, but then check all three for consistency (type checking is one of the more common uses of _Generic).

```
#define matmul(N, o, i1, i2) \
    _Generic(typeof(o), float[N][N]: matmul_float, \
        double[N][N]: matmul_double)(N, o, \
            _Generic(typeof(i1), typeof(o): i1, const typeof(o): i1), \
            _Generic(typeof(i2), typeof(o): i2, const typeof(o): i2))
```

Link: Godbolt Compiler Explorer

Here, also the 'b' case above is rejected in C23 because the 'typeof(o)' used for checking the input arguments is a variably-modified type. Note, that this macro improves type safety relative to the non-generic functions, but we can not currently use it in C (without first adopting N3348) except by using the prototype hack (Link: Goldbolt Compiler Explorer).

In summary, I explained why the type compatibility rules for variably modified types are designed as they are, and why we believe that _Generic - as mechanism to build generic functions via overloading - should not have special rules that diverge from how type compatibility is used for parameters of regular functions with respect to variably modified type, but should remain aligned with the general language rules governing function calls. In conclusion, in the interest of simplicity, safety, and good language semantics, I recommened to accept N3348 instead of N3441.

[2] A patch to GCC that does exactly this can be found here: Link