Unspecified Sizes in Definitions of Arrays, v2

Author: Martin Uecker
Number: N3427
Date: 2025-01-04

Changes for version 2 (N3295)

- various wording improvements
- address comments from SC22WG14.26614.
- rationale for using the composite type
- rationale for using `*' as wildcard size
- additional example

Introduction:

Array types generally know their size and since C99 there are also arrays of dynamic size. The size can be used for bounds checking or recovered using **sizeof** or **_Lengthof.**

```
char a[10];

// C89
char (*x)[10] = &a;
size_t N1 = sizeof(*x);

// C99
int N2 = 10;
char (*y)[N2] = &a;
```

In C23 (or with preceding language extensions such as __auto_type) it is possible to assign such a type to a variable declared with **auto**, which retains the static or dynamic size information as part of the type.

```
// C23
char b[N2] = { };
auto z = &b;
size_t N3 = sizeof(*z);
```

In principle, this is very useful, because it helps to avoid mistakes related to inconsistent sizes (which compilers are still bad at detecting in many contexts – despite some progress). One downside of **auto** is that it removes the type information from the view of the programmer and the type checker

Proposal

It is proposed to allow declarations for arrays with unspecified sizes where the size is then transferred from the initializer, while the other parts of the type have to match according to the usual type compatibility rules.

```
// C2Y
char (*w)[*] = &b;
size_t N4 = sizeof(*w);
```

The size is transferred during initialization making the object a regular C object with known sizes, possibly with a variably modified type. Semantically, for a variably modified type this then corresponds to

```
char (*w)[_Lengthof(b)] = &b;
size_t N4 = sizeof(*w);
```

This is the same as for **auto**, except that it would be a constraint violation when the type of the initializer and the type of the declaration are not compatible. In terms of specification, the usual rules for composite types already provide the machinery for inserting the sizes at the right place.

Rationale

Why use the composite type?

The composite type is the most specific type that is formed from two compatible types and merges all information from both types. The standard already has a mechanism that replaces unspecified sizes with specified or known-constant sizes. In a C compiler this mechanism can be directly used. The composite type is used in a similar situation, i.e. for merging information of multiple declarations for external definitions (6.9.3). (It is also used for the conditional operator, but there the use of a most specific type is dubious as the conditional operator should arguably not produce the most specific type, e.g. combining attributes.)

Why use `*’ as wildcard type?

The mechanisms for replacing unspecified sizes with more specified ones already exists and works when forming composite types with function arguments that make use of unspecified sizes. The syntax is intuitive as `*’ is widely used as a wildcard symbol. The use of empty brackets as in arrays of unknown size seems problematic, as it would mix it up with the notion of incomplete types.

Proposed Wording (rel. to N3301)

6.7 Declarations
6.7.1 General

Constraints

7 An underspecified declaration shall have an initializer.[x)]

[x)] **This is further specified for declarations that contain unspecified sizes in 6.7.11§5 and is implied for type inference as described in 6.7.10§2.**

Semantics

12 A declaration such that the declaration specifiers contain no type specifier or that is declared with **constexpr, or that contain an array declarator with unspecified sizes in its nested sequence of declarators,** is ~~said to be~~ *underspecified*. If such a declaration is not a definition, if it declares no or more than one ordinary identifier, if the declared identifier already has a declaration in the same scope, if the declared entity is not an object, or if anywhere within the sequence of tokens making up the declaration identifiers that are not ordinary are declared, the behavior is implementation-defined.[122)]

6.7.7.3 Array declarators

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 **for a parameter declaration or for an underspecified definition of an object** as part of the nested sequence of declarators or abstract declarators ~~for a parameter declaration~~, 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) They can **not** be used ~~only~~ in function declarations that are ~~not~~ **also function** definitions (see 6.7.7.4 and 6.9.2).

6.7.11 Initialization

Constraints

5 The initializer for an object with an underspecified definition that contains array declarators with unspecified size in its nested sequence of declarators shall be a single expression, optionally enclosed in braces. The unqualified type of the expression shall be compatible with the unqualified type of the object to be initialized, and no unspecified sizes shall remain when forming a composite type of the declaration and the type of the expression, except possibly as part of nested declarations of function parameters. The object after initialization has the appropriately qualified version of a composite type. The same type constraints and conversions as for simple assignment apply, taking the type of the scalar to be the unqualified version of the composite type.

Example 15 After initialization, the pointer p has type pointer to array of char of length 10.

```
char b[] = “something”;
char (*p)[*] = &b;
```

Example 16 After initialization, the pointer p has type pointer to variable length array with the same length as the array ‘b’.

```
void f(int n)
{
	char b[n] = { };
	char (*p)[*] = &b;
}
```

Acknowledgment: Joseph Myers for comments on a previous version of the paper and Jens Gustedt for suggestions to improve the proposed wording.