Unspecified Sizes in Definitions of Arrays

Author: Martin Uecker
Number: N3295
Date: 2024-06-23

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 programatically using **sizeof.**

```
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);
```

One downside of **auto** is that it removes the type information from the view of the programmer and the type checker. Thus, it is suggested 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

```
const size_t __SIZE1 = sizeof b / sizeof b[0];
char (*w)[__SIZE1] = &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 initialize 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.

Proposed Wording

6.7. Declarations

12 A declaration such that the declaration specifiers contain no type specifier, or that is declared with **constexpr,** **or that contains specifiers for array types of unspecified size that are not part of a declaration of a function parameter,** is said to be underspecified.

6.7.6.2 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 in declarations or type names with function prototype scope174)**, or in an underspecified definition**; 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 need not support; see 6.10.9.3.)

174) Thus, * can **not** be used ~~only~~ in function ~~declarations that are not~~ definitions (see 6.7.6.3)

6.7. Declarations

Constraints

6 An underspecified declaration declared with a variably modified type shall be a definition for an object with a pointer type and shall have an initializer.

6.7.10 Initialization

Constraints

5 The initializer for an object declared using an underspecified declaration with a variably modified type 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 the 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 the 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 size 10.

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