WG14 N 3431

Title:

Author, affiliation: Robert C. Seacord, Woven by Toyota
rcseacord@gmail.com

Date: 2024-11-10

Proposal category: Informative

Target audience: Implementers, users

Abstract: Document the argument for using the term extent instead of array length

Prior art: C23

Size, length, and extent

Reply-to: Robert C. Seacord (rcseacord@gmail.com)

Document No: **N 3431**

Reference Document: **N 3301**

Date: 2024-2-20

Document the argument for using the term extent instead of array length. This paper will not receive agenda time.

Change Log

2024-11-09:

- Initial version

Table of Contents

1 Problem Description

The C Standard uses terms like “size” and “length” inconsistently and haphazardly. This paper seeks to clarify the use of the terms in the C Standard and, more generally, in C language programming. For example, the following text appears in n3301, Subclause 6.2.7, paragraph 3:

A composite type can be constructed from two types that are compatible. If both types are the same type, the composite type is this type. Otherwise, it is a type that is compatible with both and satisfies the following conditions:

— If both types are structure types or both types are union types, the composite type is determined recursively by forming the composite types of their members.

— If both types are array types, the following rules are applied:

- If one type is an array of known constant size, the composite type is an array of that size.
- Otherwise, if one type is a variable length array whose size is specified by an expression that is not evaluated, the behavior is undefined.
- Otherwise, if one type is a variable length array whose size is specified, the composite type is a variable length array of that size.
- Otherwise, if one type is a variable length array of unspecified size, the composite type is a variable length array of unspecified size.
- Otherwise, both types are arrays of unknown size and the composite type is an array of unknown size.

The element type of the composite type is the composite type of the two element types.

It is not always easy to change the meaning of these terms, but there are different degrees of difficulty. The most difficult are the names that are present in the source code as keywords, function names, or type names. These frequently cannot be changed without breaking existing code, which violates the C Standard charter (https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3280.htm).

Terms may also show up in diagnostic messages issued by compilers and documentation. The bigger issue is that the names may be used in compiler interfaces, things like **`-Wvla`** (definitely exists in some compilers) or **`-fvariable-length-array`** (may or may not exist). Changing these will take a long time to accomplish because the flags can't simply be renamed without breaking existing build systems), Consequently, a deprecation period is needed to phase the old names out.

These can also be difficult to change, as implementations might decide to change these or not. It is also possible that software exists that reads and interprets the contents of these messages.

Finally, terms may only appear in the standard and in education material for the language, such as Effective C, 2nd Edition (https://nostarch.com/effective-c-2nd-edition) and Modern C (https://gustedt.gitlabpages.inria.fr/modern-c/).

Consequently, we must prioritize the existing definitions of these terms based on how difficult they would be to change. In deciding to change the definition of a term we have to decide if making the change will improve both the short term and long term understandability of the language.

This paper is incomplete and only intended to determine if there is a naming approach that could gain consensus in the committee.

1.1 Size

The term “size” is defined in the language and library and basically means the number of bytes in a storage instance.

Subclause 7.21.1 defines the following type:

`size_t`

which is the unsigned integer type of the result of the **`sizeof`** operator;

Here we have two places in the language, the **`size_t`** type and the **`sizeof`** operator where size is defined to be the number of bytes.

This definition of “size” would be extremely difficult to change because it is exposed in the syntax of the language.

1.2 Length

The term “length” is defined in the C Standard library as the number of characters preceding a null termination character in a string.

For example, Subclause 7.26.6.4 states:

The **`strlen`** function returns the number of characters that precede the terminating null character.

The function definition clearly defines “len” (which is short for length) as the number of characters before the terminating null character in a string.

Similarly, in Subclause 7.31.4.7.1 The **`wcslen`** function states:

The **`wcslen`** function returns the number of wide characters that precede the terminating null wide character.

The **`wchar_t`** type is compiler dependent and may be 8, 16, or 32 bits wide (on modern machines), signed or unsigned. Consequently, the length of a string may be significantly less than the number of bytes in the backing array.

In Subclause 7.31.6.5.2 the **`mbsrtowcs`** function also used the identifier **`len`** to indicate the number of characters, e.g., "**`len`** wide characters". Note that although this parameter is of type **`size_`**t, the use of the parameter identifier **`len`** indicates that this parameter does not represent the number of bytes but the number of characters.

Length could conceivably be defined as the number of bytes that have been initialized and applied to arrays. However, this becomes problematic for several reasons. First, how would you define the length if the array was partially initialized from the back going forward? What if every other element in the array was initialized? There is also a problem with arrays that contain a string. Because the null termination character of the array is initialized, the length of the string is always one less than the length of the backing array.

1.3 Bound, extent, or count

The C Standard lacks a consistent term for the number of elements in an array. I have heard this term referred to as the bound, but this word only appears in the C Standard 3 times and never with this meaning.

Similarly, the word "extent" appears 27 times in the C Standard but with the same meaning as "degree".

C++, on the other hand, uses **`std::extent`** to indicate the number of elements along the Nth dimension of the array.

The C Standard does not use the term count to refer to the number of elements of an array. Some implementations provide a **`countof`** macro that returns the number of elements in the array, expressed as a **`size_t`**.

The C Standard most commonly uses the term "size" to indicate the extent of an array. However, this is a poor choice because it conflicts with other, harder to change uses of the term in the C Standard, as documented in section 1.1 of this proposal.

Because of the WG14 commitment, documented in our charter, of compatibility with C++, I would recommend the use of the term "extent" for consistency with C++. "Bounds" and "count" would be equally valid. A compound term like "array length" may also be acceptable, but would be harder to implement.

1.4 Variable Length Array

Variable length array (VLA) is a horrible name. VLAs have an extent and not a length. They do not have a “variable length” in that they do not change length after being created. However, this term is well established and referenced in compiler option flags, diagnostic messages, the standard, and numerous educational materials. This proposal recommends leaving this term unchanged.

2 **Proposed Text**

3.1 Proposed Wording

Text in green is added to the N3301 working draft. ~~Text in red~~ that has been struck through is removed from the N3301 working draft.

Subclause 6.2.7, paragraph 3 (wording proposal #1)

Change the text from subclause 6.2.7, paragraph 3 as follows:

— If both types are array types, the following rules are applied:

• If one type is an array of known constant ~~size~~ extent, the composite type is an array of that ~~size~~ extent.

• Otherwise, if one type is a variable length array of known constant extent, the composite type is a variable length array of that constant extent.

• Otherwise, if one type is a variable length array whose ~~size~~ extent is specified by an expression that is not evaluated, the behavior is undefined.

• Otherwise, if one type is a variable length array whose ~~size~~ extent is specified, the composite type is a variable length array of that ~~size~~ extent.

• Otherwise, if one type is a variable length array of unspecified ~~size~~ extent, the composite type is a variable length array of unspecified ~~size~~ extent.

• Otherwise, both types are arrays of unknown ~~size~~ extent and the composite type is an array of unknown ~~size~~ extent.

Subclause 6.2.7, paragraph 3 (wording proposal #2)

Replace the text from subclause 6.2.7, paragraph 3 with the following text:

— If both types are array types, the following rules are applied:

• If one type is an array of known *constant* extent, the composite type is an array of that extent.

• Otherwise, if one type is an array whose extent is specified by an unevaluated expression, the behavior is undefined.

• Otherwise, if one type is an array of *unspecified* extent, the composite type is an array of unspecified extent.

• Otherwise, both types are arrays of *unknown* extent and the composite type is an array of unknown extent.

If both types are arrays of known constant size, the composite type is an array of known constant size; otherwise, the composite type is a variable length array.

The text from subclause 6.2.7, paragraph 28 is unchanged:

A complete type shall have a size that is less than or equal to **`SIZE_MAX`**. A type has known constant size if it is complete and is not a variable length array type.

Subclause 6.7.7.3, “Array declarators” paragraph 1

Change the text from subclause 6.7.7.3, paragraph 1 as follows:

In addition to optional type qualifiers and the keyword static, the [and] can delimit an expression or *. If they delimit an expression (which specifies the ~~size~~ extent of an array), the expression shall have an integer type. If the expression is a constant expression, it shall have a value greater than zero. The element type shall not be an incomplete or function type. 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. In addition to optional type qualifiers and the keyword static, the [and] may delimit an expression or *. If they delimit an expression (which specifies the ~~size~~ extent of an array), the expression shall have an integer type. If the expression is an integer constant expression, it shall have a value greater than zero. The element type shall not be an incomplete or function type. 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.

Subclause 6.7.7.3, paragraphs 4-6

Change the text from subclause 6.7.7.3, paragraphs 4-6 as follows:

If ~~the size is not~~ neither an expression nor * are present, the array type is an incomplete type. If ~~the size is~~ a * is present ~~instead of being an expression~~, the array type is a variable length array type of unspecified ~~size~~ extent, which can only be used as part of the nested sequence of declarators or abstract declarators for a parameter declaration, not including anything inside an array ~~size~~ extent expression in one of those declarators;159) such arrays are nonetheless complete types. If the ~~size~~ extent 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.)

If the ~~size~~ extent ~~is an expression that~~ is not an integer constant expression: if it occurs in a declaration at function prototype scope, it is treated as if it were replaced by *; otherwise, each time it is evaluated it shall have a value greater than zero. The size of each instance of a variable length array type does not change during its lifetime. Where an ~~size~~ extent expression is part of the operand of a **`typeof`** or **`sizeof`** operator and changing the value of the ~~size~~ extent expression would not affect the result of the operator, it is unspecified whether or not the ~~size~~ extent expression is evaluated. Where an ~~size~~ extent expression is part of the operand of an **`alignof`** operator, that expression is not evaluated.

For two array types to be compatible, both shall have compatible element types, and if both ~~size specifiers are present, and~~ extents are integer constant expressions, then both ~~size specifiers~~ extents shall have the same constant value. If the two array types are used in a context which requires them to be compatible, the behavior is undefined if the two ~~size specifiers~~ extent expressions evaluate to ~~unequal~~ different values.

3 Acknowledgements

We would like to recognize the following people for their help with this work: Aaron Ballman, Joseph S. Myers, Jens Gustedt, Martin Uecker, Anthony Williams, Tyler Kowalis, Vincent Mailhol, and Caleb McGary.