---
title: Types and sizes
author: Jens Gustedt, INRIA, France
self_contained: yes
include-in-header: diff.html
toc: true

| | | |

------------------------|---|---|---|---|
| org: | ISO/IEC JCT1/SC22/WG14 | document: | N2820
| | | | |
| target: | IS 9899:2023, TS 6010:2023 | version: | 0
| | | | |
| date: | 2021-9-19 | license: | [CC
BY](https://creativecommons.org/licenses/by/4.0/ "Distributed under a Creative
Commons Attribution 4.0 International License") | | | | |

Introduction

In 6.5.2 of the C standard, sizes are primarily defined for types.
Although this is not stated explicitly, it is commonly assumed that
such sizes cannot exceed `SIZE_MAX`{.C}. Sizes of *objects* (in
contrast to *storage instances* as of TS 6010) are only a deduced
property that is in most cases defined through the type that an object
has. This proposal attempts to make this approach consistent
throughout the standard, and to reduce the number of marginal cases
where the interpretation of `sizeof`{.C} is different between C and
C++.

For the latter, observe that code as in
``` {.C .number-lines }
int const n = 23;
int const m = 24;
double A[n][m];
int j = 0;
printf("sizeof is %zu\n", sizeof A[++j]);
```

is interpreted much differently in C and C++ since both languages have
quite different definitions for integer constant expressions. For both
the declaration of `A`{.C} is valid, but for C++ it is an array with
compile-time fixed lengths, whereas for C it is a VLA. Therefore the
`sizeof` operator may evaluate the increment operator in C, but not in
C++.

Changes

Change in 6.2.5 p25 {#size}

> <ins>A complete type shall have a size that is less than or equal
> to `SIZE_MAX`{.C}. </ins>A type has known constant size if
> ~~the type is not incomplete~~ <ins>it is complete</ins> and
> is not a variable length array type.

Rationale:

In view of our recent discussion about overflow in
[`calloc`{.C}](http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2800.pdf

"Seacord, calloc overflow handling"), we did some search into existing implementations and asked on the WG14 reflector and some other media if there could be objects defined that with a size that exceeds `SIZE_MAX`{.C}. It turned out that all interpret the current standard that huge objects make the behavior implicitly undefined. This change here makes that explicit. In particular, it makes it explicit that even requesting such a huge object has no defined behavior.

Impact:

This change is only a clarification and should not have an impact on existing programs or implementations.

Change in 6.5.3.4 p2 {#evaluation}

> The `sizeof`{.C} operator yields the size (in bytes) of its
> operand, which may be an expression or the parenthesized name of a
> type. The size is determined from the type of the operand. The
> result is an integer. If ~~the type of~~ the operand is
> <ins>the declarator of</ins> a variable length array type<ins>
> (possibly enclosed in a nested set of `typeof`{.C}
> declarators)</ins>, the operand is
> evaluated;<ins>^{FNT1}</ins> otherwise, the operand is not
> evaluated<ins>.</ins>~~and the~~ <ins>The</ins> result is an
> integer constant <ins>if the type has a known constant size, see
> 6.2.5.</ins>.

> <ins>^{FNT1} The evaluation of such declarators is specified
> in 6.7.6.2, below</ins>

Rationale:

This changes the specification when the operand of `sizeof`{.C} is not a declarator but an lvalue of VLA type. Such an object cannot be declared as a compound literal, so it must have a declaration that preceeds the `sizeof`{.C} expression. Thus, the type of such an object and by that its size is already fixed before the execution reads the `sizeof`{.C} expression.

So evaluation should be restricted to the case where the operand is a *type name* for which the evaluation makes a difference in the type, namely to the case of the *type name* of a VLA that either occurs directly in the `sizeof`{.C} or with intermediate `typeof`{.C}.

Impact:

Consider the the following code snippet

``` {.C .number-lines }
double A[n][m];
int j = 0;
int i = 0;
printf("sizeof is %zu\n", sizeof A[++j][++i]);
printf("now j is %d, i is %d\n", j, i);
printf("sizeof is %zu\n", sizeof A[++j]);
printf("now j is %d\n", j);                 // what is printed here?
```

where `n`{.C} and `m`{.C} are supposed to be some integer variables with values greater than `2`{.C}. In a non-representative survey among C and C++ enthusiasts we asked for the value that is printed for `j`{.C}. Their answers were distributed as follows:

`j`	%
`0`	37.8
`1`	7.8
`2`	54.3

So over 90% had a wrong answer. For us this clearly shows the need reform that marginal property of the `sizeof`{.C} operator.

With the current standard, the first two increment operators are not evaluated because the expression has type `double`{.C} (so `i`{.C} stays `0`{.C}, for example) but would increment the last because the expression has type `double[*]`{.C}, a VLA. As a consequence, `j`{.C} is `1`{.C} for the `printf`{.C} in line 7. With the proposed change, none of the increment operators would be evaluated.

By that, this proposal changes the status of some `sizeof`{.C} expressions, namely those that concern lvalues of VLA type.

This change also has an impact on the new `typeof`{.C} feature ([N2724](http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2724.htm)) which follows the same strategie as currently for `sizeof`{.C}. If this change here is agreed, we will coordinate for a similar change for `typeof`{.C} with the author.

Questions to WG14

#. Shall we integrate [Change 2.1](#size) into C23?

#. Shall we integrate [Change 2.2](#evaluation) into C23?

#. Shall we integrate the same changes into TS 6010?

Acknowledgements

The idea for [Change 2.2](#evaluation) was brought to my attention by Tomasz Stanislawski.