

Author: Javier A. Múgica

Date: 2026 - oct - 3

Array ICE subscript out of bounds, r. 1

This proposal was on the agenda for the August 2026 meeting, with number 3791. It depends on “discarded”, that was discussed at that same meeting but was not even voted. Therefore the present proposal was only briefly presented and, it seems, no discussion took place.

We take advantage of this fact to include some feedback on the implementation of the required diagnostics, that in some corner cases is not as trivial as it seems at first. This feedback did not arrive in time to be included in the paper presented at the August meeting.

The proposed wording is exactly the same as the one from **N3791**.

The problem

In code like

```
int a[3];
int b[n];
a[-1];    a[8];    b[-1];
```

it is obvious that the expressions `a[-1]`, `a[8]` and `b[-1]` are wrong. Their being wrong follows from the value of integer constant expressions, and thus can be known to the translator.

The reason why the standard did not mandate diagnoses for these constructions is that they were defined as equivalent to

```
*((a)+(-1))    *((a)+(8))    *((b)+(-1))
```

and these are not, in general, invalid pointer arithmetic expressions. Therefore, a diagnosis for `a[-1]`, etc. would need to step over this equivalence.

Last year the definition of array subscripting was changed and it is no longer based on pointer arithmetic. Therefore a diagnosis for these constructions can be introduced.

However, these constructions need not always be wrong, as in the two examples below

```
#define GET_IF_INARR(a, n) (((n) < _Countof(a)) ? a[n] : 0)
int a[3], b;
b = GET_IF_INARR(a,8);

#define GET_POS(a, n) ((n)>=0 ? (a)[n] : (a)[-n])
int a[3], b;
b = GET_POS(-1);
```

After the macros are expanded and `_Countof` is evaluated the result is, removing some parentheses:

```
(8 < 3) ? a[8] : 0          and
-1>=0 ? a[-1] : a[-(-1)]
```

We'd like to make `arr[i]` be a constraint violation when the index is known to be out of bounds and the expression is not discarded, but allow it when the expression is discarded. When **N3360** was written (the paper that changed the definition of array subscripting not to be based on conversion to pointer), discarded did not yet exist. Without this concept WG14 cut with another knife: negative subscripts require a constraint (even in discarded code), positive, too large ones, do not (even in not discarded code).

In this paper we propose to fix this and require the diagnostic or not according as to whether the expression is not discarded or it is.

Semantic changes implied

N3360 already implied a change from C23. The following box shows the semantics of each of the four different situations according to C23, N3360 and this proposal:

int arr[3];

	C23	N3360	Proposal
arr[-1]	dUB	C	C
0 && arr[-1]	✓	C	✓
arr[8]	dUB	dUB	C
0 && arr[8]	✓	✓	✓

Here «dUB» stands for «deterministic undefined behaviour»: the instruction will cause undefined behaviour whenever executed. «C» stands for «Constraint violation», and ✓ stands for «allowed (no constraint) and will never be evaluated».

It can be seen that the present proposal keeps the unproblematic situations allowed and mandates a constraint for the ones that result deterministically in undefined behaviour.

Implementation

In most cases the implementation is trivial: if the array is fixed, the subscript is an integer constant expression not within the allowed range and the context is not discarded, issue an error or warning message. M. Uecker noted that there are cases where the compiler cannot know whether the array subscripting expression will be discarded or not at the time the expression is first reached by the translator. Here is an example where this can happen:

```
int a[2];
int B[4][n];
sizeof(a[3]+B); //Runtime sizeof. n*sizeof(int)
```

Here the argument to **sizeof()** is not discarded, but it may be that the translator cannot know this the first time it reaches **a[3]**. It depends on how it translates the code. If it first absorbs all the argument of **sizeof**, determines its type and processes it the way it needs, this problem will not arise. It also cannot arise if the implementation does not support variable length arrays.

For implementations where the problem does arise, it is of little importance. The present proposal requires a diagnostic message when the offending expression is not discarded, but does not forbid it elsewhere. When the implementation is processing the argument of **sizeof** and similar operands, as long as it cannot tell whether the argument will be discarded or not, it may suppose for the purposes of diagnostic issuing that it will not be (these are the cases that require the diagnostic). A spurious message from the compiler may arise, therefore, in cases where the operand is discarded:

```
sizeof(a[3])
```

If the implementation supports VLAs and if it has not programmed deferring the error message until all the argument of **sizeof** has been read it may issue a warning like «Constant array subscript out of bounds in a possibly not discarded expression», or something similar. This «possibly erroneous construct» warning is already issued by some compilers when, e.g., a variable is initialized only in some branches of a switch statement and is used after the switch, or in other branches of the switch statement. In many cases, the code is likely to be right. For the case being considered now, this kind of warning would be much less common because users don't write this sort of code: when they want to know the size of the element of an array, they use the subscript zero: **sizeof(a[0])**. Further, the present situation, reflected by the middle column in the table above, already *mandates* the spurious warning if the subscript is negative.

The situation is not new

The current draft includes:

There shall be no more than one external definition for each identifier declared with internal linkage in a translation unit. Moreover, if an identifier declared with internal linkage is used in an expression there shall be exactly one external definition for the identifier in the translation unit, unless it is:

- part of the operand of a **sizeof** operator which is an integer constant expression;
- part of the operand of a **_Countof** operator which is an integer constant expression;
- ...

And there is a similar wording (with the same intended semantics) in C23.

Thus, a translator encountering such an identifier with no definition as part of the argument of **sizeof**, needs to know, for the determining if the constraint is broken or not, whether the sizeof expression will or will not be an integer constant expressions, which corresponds identically with being discarded or not.

This problem (or no-problem) has existed since C99. This proposal is not introducing it.

The security of programs is an increasing concern. With the advent of artificial-intelligence assisted programming, security vulnerabilities in code are more easily spotted, in particular in someone else's code. In view of this we believe that a diagnostic when the subscript is known to be out of bounds, and in particular when it is positive (negative constant subscripts arise much less often), is a must-have in future compilations, even more so by the time that many projects will switch to C2y. (This AI assisted programming also makes easier the proper programming of the diagnostic).

Furthermore, if the constraint as proposed in this paper is eventually adopted but only in a future version of C, compilers supporting past standards will need to implement an annoying exception for C2y: Up to C23 the situation is the one reflected in the first column of the table above; for C3x onwards, say, it will be as in the last column of the table, and C2y will stay with the unique constraint requirements from the middle column.

Proposed wording

Blue text is to be added; grey text to be removed.

6.5.3.2 Array subscripting

Constraints

- 2 One of the operands shall have type “pointer to complete object *type*” or “array of *type*”, the other operand, called the *subscript*, shall have integer type, and the result has type “*type*”. If one of the two operands has array type and the subscript is an integer constant expression, the value of the subscript shall not be negative.
- 3 If one of the two operands has array type, the expression is not discarded and the subscript is an integer constant expression, the value of the subscript shall not be negative. If further the length of the array is fixed known the subscript shall be less than or equal to the length of the array.

Further restriction

If the change is accepted, we propose a further restriction on the subscript, so that it may only equal the length of the array if the expression is the operand of the unary & operator or decays to pointer, allowing for further subscript selections with the subscript equal to zero, as in

```
int a[4][4];
&a[4][0];
```

There are other situations where the subscript equals the length of the array and shall be exempt from the constraint, as `sizeof(a[4])`. These are already exempt from the constraint because of being discarded.

With this further change, the paragraph added becomes

- 3 If one of the two operands has array type, the expression is not discarded and the subscript is an integer constant expression, the value of the subscript shall not be negative. If further the length of the array is fixed known the subscript shall be less than or equal to the length of the array. If further the subscript equals the length of the array, the `[]` operator shall be followed by zero or more `[]` operators where each subscript either is not an integer constant expression or has value zero, and the resulting postfix expression shall be the operand of the unary `&` operator or be converted to an expression with pointer type as described in 6.3.3.1.

For comparison, we reproduce here the current text in **Semantics** that parallels this one:

(The subscript) shall not be negative and shall be less than the length of the array or equal to it; it shall only equal the length of the array if the `[]` operator is followed by zero or more `[]` operators with subscripts equal to zero and the resulting postfix expression is the operand of the unary `&` operator or is converted to an expression with pointer type as described in 6.3.3.1.

In order to understand why the wording for the constraint is more involved than the wording in the **Semantics** section, consider the following example:

```
int n=5;
float A[3][3];

A[3][n];
A[3][5];
```

Currently, there is undefined behaviour in both cases. But we cannot require a diagnostic message in the first case, with a non-constant `n`.