Author: Javier A. Múgica

Date: 2026 - jan - 24

Array ICE subscript out of bounds

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.

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 furhter 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.