Name

n3325 − New nelementsof() operator (v3)

Category

Feature (keyword; operator).

Author

Alejandro Colomar Andres; maintainer of the Linux man−pages project.

Cc

GNU Compiler Collection
Martin Uecker
Xavier Del Campo Romero
Joseph Myers
Gabriel Ravier
Jakub Jelinek
Kees Cook
Qing Zhao
Jens Gustedt
David Brown
Florian Weimer
Andreas Schwab
Timm Baeder
"A. Jiang"
Eugene Zelenko
Aaron Ballman
Paul Koning
Daniel Lundin
Nikolaos Strimpas
Fernando Borretti
JeanHeyd Meneide
Jonathan Protzenko
Chris Bazley
Ville Voutilainen
Alex Celeste
Jakub Åukasiewicz

History

n2529 v1; 2020-06-04; authored by Xavier.

New pointer-proof keyword to determine array length

n3313 v2; 2024-08-15.

New elementsof() operator (v2)

- Provide an implementation for GCC.
- Rename _Lengthof => elementsof.
- Clarify when it should result in an integer constant expression.
- Require parentheses.
- Document prior art.
- Document backwards compatibility.
- Document reasons for having this operator beyond pointer safety (which is already solved with complex macros and/or diagnostics).
- Add specific proposed changes to the draft document (based on n3220).

n3325 v3; 202?-09-01.

New nelementsof() operator (v3)

- Rename elementsof => nelementsof.
- Propose an alternative shorter name: neltsof.
- Rebase on n3301.
- Document performance problem of sizeof division.
- Fix support for VLAs in example of NITEMS(). This needs GNU C's *__builtin_types_compatible_p*().
- Fix typos, and improve wording.

Synopsis

This operator yields the number of elements of an array.

Problem description

Portability

It is possible to write a macro that yields the number of elements of an array. However, it is impossible to reject pointer arguments portably. Here's an implementation using GNU C:

```
#define is_same_type(a, b)    __builtin_types_compatible_p(a, b)
#define is_same_typeof(a, b)  is_same_type(typeof(a), typeof(b))
#define decay(a)              (&*(a))
#define is_array(a)           (!is_same_typeof(a, decay(a)))
#define must_be(e)                                          \
(                                                           \
    0 * (int) sizeof(                                       \
        struct {                                            \
            static_assert(e);                               \
            int ISO_C_forbids_a_struct_with_no_members;     \
        }                                                   \
    )                                                       \
)
#define sizeof_array(a)    (sizeof(a) + must_be(is_array(a)))
#define NITEMS(a)          (sizeof_array(a) / sizeof((a)[0]))
```

While diagnostics could be better, with good helper-macro names, they are decent.

Type names

This *NITEMS*() macro is not ideal, since it only works with expressions but not with type names. However, for most use cases that's enough.

constexpr

The usual *sizeof* division evaluates the operand and results in a run-time value in cases where it wouldn't be necessary. If the number of elements of the top-level array is determined by an integer constant expression, but an internal array is a VLA, *sizeof* must evaluate:

```
int  a[7][n];
int  (*p)[7][n];

p = &a;
NITEMS(*p++);
```

With a *nelementsof* operator, this would result in an integer constant expression of value 7.

Double evaluation

With the *sizeof*-based implementation from above, the example above causes double evaluation of **p++*. It's possible to write a macro that is free of double-evaluation problems using a GNU statement expression and *typeof*(), but then the macro cannot be used at file scope.

Diagnostics

Having more constant expressions would allow for increased diagnostics, which would result in safer code. For example:

```
$ cat f.c
#define NITEMS(a)  (sizeof(a) / sizeof(*(a)))

void f(char (*a)[3][*], int (*b)[nelementsof(*a)]);
void g(char (*a)[3][*], int (*b)[NITEMS(*a)]);

int
main(void)
{
      int   i5[5];
      char  c35[3][5];

      f(&c35, &i5);
      g(&c35, &i5);
}

$ /opt/local/gnu/gcc/nelementsof/bin/gcc f.c
f.c: In function âmainâ:
f.c:12:17: error: passing argument 2 of âfâ from incompatible pointer type [-Win
   12 |         f(&c35, &i5);
      |                 ^~~
      |                 |
      |                 int (*)[5]
f.c:3:31: note: expected âint (*)[3]â but argument is of type âint (*)[5]â
    3 | void f(char (*a)[3][*], int (*b)[nelementsof(*a)]);
      |                         ~~~~~~~~~~~~~~~~~~~~~~~~~
```

Performance

In cases where sizeof evaluates to a run-time value, the division must be performed at run time. A new operator would yield the value directly, exposing information that the compiler already has internally, without needing a division.

Exponential macro expansions

Macros that perform type checks on the arguments need to expand those several times. When such macros are nested, the number of expansions grows exponentially, making compilation slower. See this <LWN.net> article.

Proposal description

Add a new keyword named *nelementsof* which evaluates to the number of elements of an array operand, that is, the number of elements in the array. The syntax should be similar to sizeof.

The operand must be a parenthesized complete array type or an expression of such a type. It is a constraint violation to pass something else. For example:

```
int  a[n];

elementsof(a);           // returns n
elementsof(int [7][3]);  // returns 7

elementsof(int);         // constraint violation
elementsof(n);           // constraint violation
```

The result of this operator is an integer constant expression, unless the top-level array is a variable-length array. The operand is only evaluated if the top-level array is a variable-length array. For example:

```
elementsof(int [7][n++]);  // integer constant expression
elementsof(int [n++][7]);  // run-time value; n++ is evaluated
```

Design choices

Prior art

C

It is common in C programs to get the number of elements of an array via the usual sizeof division and wrap it in a macro. Common names include:

- NITEMS()
- NELEM()
- NELEMS()
- NELTS()
- elementsof()
- lengthof()
- ARRAY_SIZE()

We can extract some patterns from these macros:

- The name derives from one of

  number of elements

  size — But there's a proposal to remove that term from the standard due to ambiguity with the number of bytes of an array (*sizeof(a)*).

  length — This is also ambiguous in the context of strings, where length means the number of non-zero characters.

- The name either ends in "of", to denote it being an operator-like macro, or it is in upper-case, to denote it being a "magic" macro.

C++

In C++, there are several standard features to determine the number of elements of an array:

std::size() (since C++17)
std::ssize() (since C++20)

The usage of these is the same as the usual C macros named above.

It's a bit different, since it's a general purpose sizing template, which works on non-array types too, with different semantics.

But when applied to an array, it has the same semantics as the macros above.

std::extent (since C++23)

The syntax of this is quite different. It uses a numeric index as a second parameter to determine the dimension in which the number of elements should be counted.

C arrays are much simpler than C++'s many array-like types, and I don't see a reason why we would need something as complex as *std::extent* in C. Certainly, existing projects have not developed such a macro, even if it is technically possible:

```
#define DEREFERENCE_n(a, n)  DEREFERENCE_ ## n (a, c)
#define DEREFERENCE_9(a)     (*********(a))
#define DEREFERENCE_8(a)     (********(a))
#define DEREFERENCE_7(a)     (*******(a))
#define DEREFERENCE_6(a)     (******(a))
#define DEREFERENCE_5(a)     (*****(a))
#define DEREFERENCE_4(a)     (****(a))
#define DEREFERENCE_3(a)     (***(a))
#define DEREFERENCE_2(a)     (**(a))
#define DEREFERENCE_1(a)     (*(a))
#define DEREFERENCE_0(a)     ((a))
#define extent(a, n)         NITEMS(DEREFERENCE(a, n))
```

If any project needs that syntax, they can implement their own trivial wrapper macro, as demonstrated above.

Existing prior art in C seems to favour a design that follows the syntax of other operators like *sizeof*.

Naming

It is tradition in C to name operators (and operator-like macros) with an *of termination, and in lower case:

- sizeof
- alignof
- typeof
- offsetof

It seems reasonable to use a similar syntax to indicate users that they can expect similar syntax and semantics from such an operator.

n3187 attempts to standardize the term length to refer to the number of elements in an array. However, length might generate confusion: there's the length of a string (number of non-zero characters) and the length of an array (the total number of elements in the array), and both a string and an array often coexist. It is common to use 'n' for a variable that holds the number of elements of an array and 'len' for a variable that holds the length of a string. Also, the main precedent of length in C is in the term VLA, which ironically refers to arrays of variable size. In C, there's actually negligible precedent of using the term "length" for referring to the number of elements of an array.

"Number of elements of an array" is an expression commonly used in the standard. Thus, it is a term that programmers are already familiar with.

A contraction of the proposed name would also make sense. *neltsof* is unused in the wild, so we could claim the name easily. It also has a name length similar to other existing operators. There's prior art in contracting names for operators, such as *alignof*, which stands for "alignment of".

Backwards compatibility

A code search on large online platforms revealed that *nelementsof* is in use in a single project (that we could find), and it is semantically compatible with our proposal, by yielding the number of elements of an array.

lengthof is in use with incompatible semantics, so it would be more difficult to own that name.

Also, while projects already use names like *nelts* for variable names, they don't use names ending in *of* for variable names. That's more reason to use a name ending in *of* which is commonly used only for operator-like macros and functions.

Parentheses

alignof requires that the operand is a type name. However, some compilers allow passing an expression as an extension, and they don't require parentheses, just like with *sizeof*. For example:

```
$ cat s.c
#include <stdalign.h>

int
main(void)
{
      int  *x;

      return alignof *x;
}
$ gcc -Wall -Wextra s.c
$ ./a.out; echo $?
4
```

Some compilers may want to require parentheses for simplicity. It is left as a quality-of-implementation detail if an implementation allows unparenthesized expressions. In GCC, not requiring parentheses resulted in a simpler implementation.

We recommend that ISO C deprecates unparenthesized expressions from *sizeof* if that is not wanted in newer operators. That would result in a simpler language. However, that's out-of-scope for this proposal.

Uglification

C23 seems to have shifted away from uglified keywords. This proposal defaults to providing the keyword directly, since it's semantically compatible with existing code.

Future directions

nelementsof could be extended to support function parameters declared with array notation. Here's an example borrowing notation from n3188:

```
wchar_t *
wmemset(wchar_t wcs[.n], wchar_t wc, const size_t n)
{
    for (size_t i = 0; i < nelementsof(wcs); i++)
        wcs[i] = wc;

    return wcs;
}
```

Questions

- Should this new keyword accept an expression without parentheses (like *sizeof* does)? Or should it require parentheses?
- What name should we use for it?
- Should we use an uglyfied name plus a header providing a macro? Or just the nice name directly?

Proposed wording

6.3.3.1 Lvalues, arrays, and function designators

p3

```
 Except when it is the operand of the sizeof operator,
+or the nelementsof operator,
 or typeof operators,
 or the unary & operator,
 or is a string literal used to initialize an array,
 an expression that has type "array of type"
 is converted to an expression with type "pointer to type"
 that points to the initial element of the array object
 and is not an lvalue.
```

Forward references

```
 prefix increment and decrement operators (6.5.4.2),
-the sizeof and alignof operators (6.5.4.5),
+the sizeof, nelementsof, and alignof operators (6.5.4.5),
 structure and union members (6.5.3.4).
```

6.4.2 Keywords

Syntax (p1)

```
 long
+nelementsof
 nullptr
```

6.5.4 Unary operators

Syntax (p1)

```
 unary-expression:
       postfix-expression
       ++ unary-expression
       -- unary-expression
```

```
       unary-operator cast-expression
       sizeof unary-expression
       sizeof ( type-name )
+      nelementsof ( type-name )
       alignof ( type-name )
```

6.5.4.5 The sizeof and alignof operators

Title

```
−The sizeof and alignof operators
+The sizeof, nelementsof, and alignof operators
```

Constraints (p1)

```
 or to an expression that designates a bit-field member.
+The nelementsof operator shall not be applied to an expression that
+has an incomplete type or
+does not have array type,
+or to the parenthesized name of such a type.
 The alignof operator shall not be applied to
 a function type or an incomplete type.
```

Semantics (pX; insert as p2)

```
+The nelementsof operator yields the number of elements
+of its operand.
+The number of elements is determined from the type of the operand.
+The result is an integer.
+If the number of elements of the array type is variable,
+the operand is evaluated;
+otherwise,
+the operand is not evaluated
+and the result is an integer constant expression.
```

EXAMPLE 2 (p7)

```
−Another use of the sizeof operator is
+A use of the nelementsof operator is
 to compute the number of elements in an array
−     sizeof array / sizeof array[0]
+     nelementsof(array)
```

6.6 Constant expressions

Semantics (p8)

```
 An integer constant expression115) shall have integer type
 and shall only have operands that are
 integer literals,
 named and compound literal constants of integer type,
 character constants,
−sizeof expressions
+sizeof or nelementsof expressions
 whose results are integer constant expressions,
 alignof expressions,
 and floating, named, or compound literal constants of arithmetic type
 that are the immediate operands of casts.
 Cast operators in an integer constant expression
 shall only convert arithmetic types to integer types,
 except as part of an operand to the typeof operators,
 sizeof operator,
```

```
+nelementsof operator,
 or alignof operator.
```

Footnote 113)

```
 The operand of a
 typeof (6.7.3.6),
 sizeof,
+nelementsof,
 or alignof operator
 is usually not evaluated (6.5.4.4).
```

Semantics (p10)

```
 An arithmetic constant expression
 shall have arithmetic type
 and shall only have operands that are
 integer literals,
 floating literals,
 named or compound literal constants of arithmetic type,
 character literals,
−sizeof expressions
+sizeof or nelementsof expressions
 whose results are integer constant expressions,
 and alignof expressions.
 Cast operators in an arithmetic constant expression
 shall only convert arithmetic types to arithmetic types,
 except as part of an operand to the typeof operators,
 sizeof operator,
+nelementsof operator,
 or alignof operator.
```

6.7.2 Storage-class specifiers

Footnote 127)

```
 The implementation can treat any register declaration simply
 as an auto declaration.
 However,
 whether or not addressable storage is used,
 the address of
 any part of an object declared with storage-class specifier register
 cannot be computed,
 either explicitly
 (by use of the unary & operator as discussed in 6.5.4.2)
 or implicitly
 (by converting an array name to a pointer as discussed in 6.3.2.1).
 Thus,
−the only operator
+the only operators
 that can be applied to
 an array declared with storage-class specifier register
−is sizeof
+are sizeof,
+nelementsof,
 and the typeof operators.
```

6.7.7.3 Array declarators

Semantics (p5)

```
 Where a size expression is part of
 the operand of a typeof or sizeof operator
 and changing the value of the size expression
 would not affect the result of the operator,
 it is unspecified whether or not the size expression is evaluated.
+Where a size expression is part of
+the operand of a nelementsof operator
+and changing the value of the size expression
+would not affect the result of the operator,
+the size expression is not evaluated.
 Where a size expression is part of
 the operand of an alignof operator,
 that expression is not evaluated.
```

6.9.1 General

Constraints (p3)

```
 •  part of the operand of a sizeof operator
     whose result is an integer constant expression;
+•  part of the operand of a nelementsof operator
+    whose result is an integer constant expression;
 •  part of the operand of an alignof operator
     whose result is an integer constant expression;
```

Semantics (p5)

```
 An external definition is
 an external declaration
 that is also a definition of a function
 (other than an inline definition)
 or an object.
 If an identifier declared with external linkage
 is used in an expression
 (other than as
 part of the operand of a typeof operator
 whose result is not a variably modified type,
 part of the controlling expression of a generic selection,
 part of the expression in a generic association
 that is not the result expression of its generic selection,
−or part of a sizeof or alignof operator
+or part of a sizeof, nelementsof, or alignof operator
 whose result is an integer constant expression),
 somewhere in the entire program
 there shall be exactly one external definition for the identifier;
 otherwise, there shall be no more than one.190)
```

6.10.2 Conditional inclusion

EXAMPLE 5 (p22)

```
−     return (int)(meow[0] + meow[(sizeof(meow) / sizeof(*meow)) − 1]);
+     return (int)(meow[0] + meow[nelementsof(meow) − 1]);
```

6.10.4.1 #embed preprocessing directive

EXAMPLE 1 (p16)

```
−     have_you_any_wool(baa_baa, sizeof(baa_baa));
+     have_you_any_wool(baa_baa, nelementsof(baa_baa));
```

EXAMPLE 4 (p19)

```
−     const size_t f_size = sizeof(embed_data);
+     const size_t f_n = nelementsof(embed_data);
−     unsigned char f_data[f_size];
+     unsigned char f_data[f_n];
      FILE* f_source = fopen("data.dat", "rb");
      if (f_source == nullptr)
            return 1;
      char* f_ptr = (char*)&f_data[0];
−     if (fread(f_ptr, 1, f_size, f_source) != f_size) {
+     if (fread(f_ptr, 1, f_n, f_source) != f_n) {
            fclose(f_source);
            return 1;
      }
      fclose(f_source);

−     int is_same = memcmp(&embed_data[0], f_ptr, f_size);
+     int is_same = memcmp(&embed_data[0], f_ptr, f_n);
```

6.10.4.2 limit parameter

EXAMPLE 1 (p5)

```
−     static_assert((sizeof(sound_signature) / sizeof(*sound_signature)) == 4,
−           "There should only be 4 elements in this array.");
+     static_assert(nelementsof(sound_signature) == 4);
```

EXAMPLE 2 (p6)

```
−     static_assert((sizeof(sound_signature) / sizeof(*sound_signature)) == 4,
−           "There should only be 4 elements in this array.");
+     static_assert(nelementsof(sound_signature) == 4);
```

6.10.4.4 prefix parameter

EXAMPLE (p4)

```
−     int is_good = (sizeof(whl) == 1 && whl[0] == ' ')
+     int is_good = (nelementsof(whl) == 1 && whl[0] == ' ')
      || (whl[0] == '\xEF' && whl[1] == '\xBB'
−     && whl[2] == '\xBF' && whl[sizeof(whl) − 1] == ' ');
+     && whl[2] == '\xBF' && whl[nelementsof(whl) − 1] == ' ');
```

A.2.2 Keywords

(6.4.1)

```
 long
+nelementsof
 nullptr
```

A.3.1 Expressions

(6.5.4.1)

```
 unary-expression:
       postfix-expression
       ++ unary-expression
       −− unary-expression
       unary-operator cast-expression
       sizeof unary-expression
       sizeof ( type-name )
+      nelementsof ( type-name )
       alignof ( type-name )
```

J.2 Undefined behavior

(50)

```
 An expression that is required to be an integer constant expression
 does not have an integer type;
 has operands that are not integer literals,
 named constants,
 compound literal constants,
 enumeration constants,
 character literals,
 predefined constants,
−sizeof expressions
+sizeof or nelementsof expressions
 whose results are integer constant expression,
 alignof expressions,
 or immediately-cast floating literals;
 or contains casts
−(outside operands to sizeof and alignof operators)
+(outside operands to sizeof, nelementsof, and alignof operators)
 other than conversions of arithmetic types to integer types (6.6).
```

(52)

```
 An arithmetic constant expression does not have arithmetic type;
 has operands that are not integer literals,
 floating literals,
 named and compound literal constants of arithmetic type,
 character literals,
 predefined constants,
−sizeof expressions
+sizeof or nelementsof expressions
 whose results are integer constant expressions,
 or alignof expressions;
 or contains casts
−(outside operands to sizeof or alignof operators)
+(outside operands to sizeof, nelementsof, or alignof operators)
 other than conversions of arithmetic types to arithmetic types (6.6).
```

J.6.3 Particular identifiers or keywords

p2

```
 negative_sign
+nelementsof
 nextafterd128
```

K.3.5.4.3 The fscanf_s function

EXAMPLE 2 (p8)

```
−     n = fscanf_s(stdin, "%s", s, sizeof s);
+     n = fscanf_s(stdin, "%s", s, nelementsof(s));
```

K.3.7.4.1 The strtok_s function

EXAMPLE (p10)

```
−     rsize_t max1 = sizeof(str1);
−     rsize_t max2 = sizeof(str2);
+     rsize_t max1 = nelementsof(str1);
+     rsize_t max2 = nelementsof(str2);
```

K.3.9.4.1.2 The wcrtomb_s function

Description (p4)

```
−     wcrtomb_s(&retval, buf, sizeof buf, Lâ â, ps)
+     wcrtomb_s(&retval, buf, nelementsof(buf), Lâ â, ps)
```

See also

The discussion of a patch set implementing an *__nelementsof__* operator in GCC. It also discusses drafts of this paper.