Name

n???? − New lengthof keyword to determine array length (v2)

Category

Feature (keyword).

Author

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

Cc

GNU Compiler Collection
Martin Uecker
Xi Ruoyao
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

History

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

n???? v2; 2024-??-??;

Problem description

Portability

Prior to C23 it was impossible to do this, but since C23 it is possible to portably write a macro that determines the length of an array, that is, the number of elements in the array.

```
#define must_be(e)                                          \
(                                                           \
    0 * (int) sizeof(                                       \
        struct {                                            \
            static_assert(e);                               \
            int ISO_C_forbids_a_struct_with_no_members;     \
        }                                                   \
    )                                                       \
)
#define is_array(a)                                         \
(                                                           \
    _Generic(&(a),                                          \
        typeof((a)[0]) **:  0,                              \
        default:            1                               \
    )                                                       \
)
#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 top-level array length is determined by a 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 *lengthof* operator, this would result in an integer constant expression of value 7.

Having more constant expressions would allow for better diagnostics.

Double evaluation

With the *sizeof*-based implementation from above, the example from above causes double evaluation of **p++*.

Proposal description

Add a new keyword named *lengthof* which evaluates to the length of an array operand, that is, the number of elements in the array. The syntax should be identical 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];

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

lengthof(int);         // constraint violation
lengthof 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:

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

Syntax

A question has been raised: Should this new keyword accept an expression without parentheses (like *sizeof* does)? Or should it require 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
```

For consistency with existing operators, unparenthesized expressions should be allowed. If that syntax was not wanted, WG14 should consider deprecating it from all such operators at once, but not for new operators only.

Keeping the same syntax of existing operators allows keeping the implementation simpler.

Future directions

lengthof 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, size_t n)
{
    for (size_t i = 0; i < lengthof(wcs); i++)
        wcs[i] = wc;

    return wcs;
}
```

Proposed wording

6.3.2.1 Lvalues, arrays, and function designators

p3

```
 Except when it is the operand of the sizeof operator,
+or the lengthof operator,
 or the 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 (6.5.4.1),
−the sizeof and alignof operators (6.5.4.4),
+the sizeof, lengthof, and alignof operators (6.5.4.4),
 structure and union members (6.5.3.4).
```

6.4.1 Keywords

Syntax (p1)

```
 int
+lengthof
 long
```

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 )
+      lengthof unary-expression
+      lengthof ( type-name )
        alignof ( type-name )
```

6.5.4.4 The sizeof and alignof operators

Title

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

Constraints (p1)

```
 or to an expression that designates a bit-field member.
+The lengthof operator shall not be applied to an expression that
+has an incomplete type or
+does not have array type,
+to the parenthesized names 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 lengthof operator yields the length
+(in number of elements)
+of its operand.
+The length  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.
```

EXAMPLE 2 (p7)

−Another use of the sizeof operator is +A use of the lengthof operator is

to compute the number of elements in an array − sizeof array / sizeof array[0]

+ lengthof array

6.6 Constant expressions

Semantics (p8)

```
 An integer constant expression117) shall have integer type
 and shall only have operands that are
 integer constants,
 named and compound literal constants of integer type,
 character constants,
−sizeof expressions whose results are integer constants,
+sizeof or lengthof expressions whose results are integer constants,
 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,
+lengthof operator,
 or alignof operator.
```

Footnote 115)

```
 The operand of a
 typeof (6.7.3.6),
 sizeof,
+lengthof,
 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 constants,
floating constants,
named or compound literal constants of arithmetic type,
character constants,
−sizeof expressions whose results are integer constants,
+sizeof or lengthof expressions whose results are integer constants,
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,
+lengthof operator,
or alignof operator.
```

6.7.2 Storage-class specifiers

Footnote 128)

```
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,
+lengthof,
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 lengthof 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;
+•  part of the operand of a lengthof operator
     whose result is an integer constant;
 •  part of the operand of an alignof operator
     whose result is an integer constant;
```

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, lengthof, 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.191)
```

6.10.2 Conditional inclusion

EXAMPLE 5 (p22)

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

EXAMPLE 4 (p19)

```
−     const size_t f_size = sizeof(embed_data);
+     const size_t f_n = lengthof(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(lengthof(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(lengthof(sound_signature) == 4);
```

6.10.4.4 prefix parameter

EXAMPLE (p4)

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

A.2.2 Keywords

(6.4.1)

```
          int
+         lengthof
          long
```

A.3.1 Expressions

(6.5.4)

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

J.2 Undefined behavior

(52)

```
          An expression that is required to be an integer constant expression
          does not have an integer type;
          has operands that are not integer constants,
          named constants,
          compound literal constants,
          enumeration constants,
          character constants,
          predefined constants,
-         sizeof expressions
```

```
+sizeof or lengthof expressions
 whose results are integer constants,
 alignof expressions,
 or immediately-cast floating constants;
 or contains casts
−(outside operands to sizeof and alignof operators)
+(outside operands to sizeof, lengthof, and alignof operators)
 other than conversions of arithmetic types to integer types (6.6).
```

(54)

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

J.6.3 Particular identifiers or keywords

p2

```
 ldiv_t
+lengthof
 lgamma
```

K.3.5.3.3 The fscanf_s function

EXAMPLE 2 (p8)

```
−     n = fscanf_s(stdin, "%s", s, sizeof s);
+     n = fscanf_s(stdin, "%s", s, lengthof 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 = lengthof(str1);
+     rsize_t max2 = lengthof(str2);
```

K.3.9.4.1.2 The wcrtomb_s function

Description (p4)

```
−     wcrtomb_s(&retval, buf, sizeof buf, L , ps)
+     wcrtomb_s(&retval, buf, lengthof buf, L , ps)
```

See also

The discussion of a patch set implementing a *__lengthof__* operator in GCC.