The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

6.5.3.4 The sizeof operator

Constraints

sizeof constraints

The **sizeof** operator shall not be applied to an expression that has function type or an incomplete type, to the parenthesized name of such a type, or to an expression that designates a bit-field member. 1118

Commentary

In the C90 Standard the result of the **sizeof** operator was a constant known at translation time. While there are some applications where being able to find out the size of a function would be useful (one specification might be the number of bytes in its generated machine code), this information is not of general utility. The C90 constraint was kept.

If the **sizeof** operator accepted a bit-field as an operand, it would have to return a value measured in bits for all its operands. sizeof result of 1119

C++

The C++ Standard contains a requirement that does not exist in C.

5.3.3p5 *Types shall not be defined in a* ***sizeof*** *expression.*

A C source file that defines a type within a **sizeof** expression is likely to cause a C++ translator to issue a diagnostic. Defining a type within a **sizeof** expression is rarely seen in C source.

```
1  int glob = sizeof(enum {E1, E2}); /* does not affect the conformance status of the program */
2                                    // ill-formed
```

Other Languages

A few other languages provide an explicit mechanism (e.g., an operator or a keyword) for obtaining the number of bytes occupied by an object. Such a mechanism is also a common extension in implementations of languages that do not specify one. Other languages obtain the size implicitly in those contexts where it is needed (i.e., the operand in a call to **new** memory allocation function in Pascal).

Table 1118.1: Occurrence of the **sizeof** operator having particular operand types (as a percentage of all occurrences of this operator). Based on the translated form of this book's benchmark programs.

Type	%	Type	%
struct	48.2	**unsigned short**	2.7
[ ]	12.2	**struct ***	2.6
int	11.6	**char**	2.0
other-types	4.7	**unsigned char**	1.5
long	3.8	**char ***	1.5
unsigned int	3.6	**signed int**	1.2
unsigned long	3.4	**union**	1.1

Semantics

sizeof result of

The **sizeof** operator yields the size (in bytes) of its operand, which may be an expression or the parenthesized name of a type. 1119

Commentary

The operand referred to is the execution-time value of the operand. In the case of string literals, escape sequences will have been converted to a single or multibyte character. In these cases the value returned by translation phase 5

the **sizeof** operator does not correspond to the number of characters visible in the source code. Most of the uses of the result of this operator work at the byte, not the bit, level; for instance, the argument of a memory-allocation function, which operates in units of bytes. Having to divide the result by CHAR_BIT, for most uses, would not be worth the benefit of being able to accept bit-field members.

Other Languages

The **SIZE** attribute in Ada returns the number of bits allocated to hold the object, or type. The **BIT_SIZE** intrinsic in Fortran 90 returns the number of bits in its integer argument; the **SIZE** intrinsic returns the number of elements in an array.

Common Implementations

A few vendors have extended the **sizeof** operator. For instance, Diab Data[1] supports a second argument to the parenthesized form of the **sizeof** operator. The value of this argument changes the information returned (e.g., if the value of the second argument is 1 the alignment of the type is returned, if it is 2 a unique value denoting the actual type is returned).

Coding Guidelines

The size of an object, or type, is representation information and the guideline recommendation dealing with the use of representation information might be thought to be applicable. However, in some contexts many uses of the **sizeof** operator are symbolic. The contexts in which the size of an operand is often used include the following:

?? representation information using symbolic name

- A call to a storage allocation function requires the number of bytes to allocate.
- When copying the representation of an object, either to another object or to a binary file, the number of bytes to be copied is required.
- When an object is being overlaid over the same storage as another object (using a union or pointer to object type), the sizes in the two types need to agree.
- When calculating the range of values representable by the operand (based on the number of bits it contains).

In some of the uses in these contexts the result of the **sizeof** operator is treated as a symbolic value— the size of its operand, with no interest in its numeric properties. While in others the result is manipulated as an arithmetic value; it is an intermediate value used in the calculation of the final value. However, a strong case can be made for claiming that certain kinds of arithmetic operations are essentially symbolic in nature:

- Multiplication of the result (e.g, to calculate the size of an array of objects)
- Division of the result (e.g., to calculate how many objects will fit in a given amount of storage)
- Subtracting from the result (e.g., to calculate the offset of the character that is third from the end of a string literal.
- Adding to the result (e.g., calculating the size of an array needed to hold several strings)

Dev ??

The **sizeof** operator may be used provided the only operators applied to its result (and the result of these operations) are divide and multiple.

Dev ??

The **sizeof** operator whose operand has an array type may be used provided the only operators applied to its result (and the result of these operations) are divide, multiply, addition, and subtraction.

For simplicity the deviation wording permits some unintended uses of representation information. For instance, the deviations permit both of the expressions `sizeof(array_of_int)-5` and `sizeof(array_of_char)-5`. There is a difference between the two in that in the former case the developer is either making use of representation information or forgot to write `sizeof(array_of_int)-5*sizeof(int)` (these guideline recommendations are not intended to recommend against constructs that are faults). Character types are special in that `sizeof(char)` is required to be 1, so it is accepted practice to omit the multiplication for these types.

guidelines not faults
sizeof char defined to be 1 1124

The size is determined from the type of the operand. 1120

Commentary

If an object appears as the operand, its declared type is used, not its effective type. The operand is also a special case in that some implicit conversions do not occur in this context.

lvalue converted to value
array converted to pointer
function designator converted to type

For floating-types, any extra precision maintained by an implementation is not included in the number of bytes returned. For instance, `sizeof(a_double * b_double)` always returns the size of the type specified by the C semantics, not the size of the representation used by the implementation when multiplying two objects of type **double**.

FLT_EVAL_METHOD

C++

5.3.3p1 *The **sizeof** operator yields the number of bytes in the object representation of its operand.*

Coding Guidelines

Developers sometimes write code that uses an operand's size to deduce the range of values it can represent (applies to integer types only). Information on the range of values representable in a type is provided by the contents of the header **<limits.h>**. However, when writing a macro having an argument that can be one of several types, there is no mechanism for deducing the type actually passed. It is not possible to use any of the macros provided by the header **<limits.h>**. The **sizeof** operator provides a solution that works on the majority of processors.

A size determined from the type of the operand need not provide an accurate indication of the range of values representable in that operand type (it provides an upper bound on the range of values that can be stored in an object of that type). A type may contain padding bytes, which will be included in its size. In the case of floating-point types, it is also possible that an expression is evaluated to a greater precision than implied by its type. Using the **sizeof** operator for this purpose is covered by the guideline recommendation dealing with the use of representation information.

FLT_EVAL_METHOD
representation information using ??

The result is an integer. 1121

Commentary

To be exact, the result has an integer type, `size_t`.

C90

In C90 the result was always an integer constant. The C99 contexts in which the result is not an integer constant all involve constructs that are new in C99.

C++

Like C90, the C++ Standard specifies that the result is a constant. The cases where the result is not a constant require the use of types that are not supported by C++.

---

sizeof operand evaluated

If the type of the operand is a variable length array type, the operand is evaluated; 1122

Commentary

The number of elements in the variable length array is not known until its index expression is evaluated. This evaluation may cause side effects. The requirement specified in this C sentence is weakened by a later sentence in the standard. It is possible that the operand may only be partially evaluated.

sizeof VLA unspecified evaluation

C90

Support for variable length array types is new in C99.

C++

Variable length array types are new in C99 and are not available in C++.

Coding Guidelines

The issue of side effects in VLA's is discussed elsewhere.

sizeof VLA unspecified evaluation

Example

```
1  extern int glob;
2
3  void f(void)
4  {
5  int loc = sizeof(int [glob++]); /* glob is incremented. */
6
7  /*
8   * To obtain the size of the type ++glob need not be evaluated in
9   * the first case, but must be evaluated in the second.
10  */
11 loc=sizeof((int *)[++glob]);
12 loc=sizeof( int * [++glob]);
13 }
```

1123 otherwise, the operand is not evaluated and the result is an integer constant.

sizeof operand not evaluated

Commentary

A full expression having a **sizeof** operator as its top-level operator, with such an operand, can occur anywhere that an integer constant can occur. The size is obtained from the type of the operand. This information is available during translation. (There is no need to generate any machine code to evaluate the operand, and this requirement prohibits such generation.) Although the operand is not evaluated, any operators that appear in it will still cause the integer promotions and usual arithmetic conversions to be performed.

integer promotions usual arithmetic conversions

Coding Guidelines

Some coding guideline documents recommend that the operand of the **sizeof** operator should not contain any side effects. In practice such usage is very rarely seen and no such guideline recommendation is given here.

Example

```
1  extern int i;
2  extern unsigned char uc;
3
4  void f(void)
5  {
6  int loc_i = sizeof(i++);    /* i is not incremented. */
7  int loc_uc = sizeof(uc);    /* sizeof an unsigned char. */
8  int loc_i_uc = sizeof(+uc); /* Operand promoted first. */
9  }
```

sizeof char defined to be 1

When applied to an operand that has type **char**, **unsigned char**, or **signed char**, (or a qualified version thereof) the result is 1. 1124

Commentary

The number of bits in the representation of a character type is irrelevant. By definition the number of bytes in a character type is one.

byte addressable unit

Coding Guidelines

Developers sometimes associate a byte as always containing eight bits. On hosts where the character type is 16 bits, this can lead to the incorrect assumption that applying **sizeof** to a character type will return the value 2. These issues are discussed elsewhere.

CHAR_BIT macro

When applied to an operand that has array type, the result is the total number of bytes in the array.[85)] 1125

Commentary

array converted to pointer

In this case an array is not converted to a pointer to its first element.

When applied to an operand that has structure or union type, the result is the total number of bytes in such an object, including internal and trailing padding. 1126

Commentary

structure trailing padding

All of the bytes in the object representation need to be included because of the kinds of use to which the result of the **sizeof** operator is put (e.g., allocating storage and copying objects). Trailing padding needs to be taken into account because more than one object of that type may be allocated or copied (e.g., an array of types having a given size, or a structure containing a member having a flexible array type). The standard requires that there be no leading padding.

structure unnamed padding

---

sizeof result type

The value of the result is implementation-defined, and its type (an unsigned integer type) is **size_t**, defined in **<stddef.h>** (and other headers). 1127

Commentary

The implementation-defined value will be less than or equal to the value of the `SIZE_MAX` macro. There is no requirement that `SIZE_MAX == PTRDIFF_MAX`. When the operand of **sizeof** contains more bytes than can be represented in the type `size_t` (e.g., `char x[SIZE_MAX/2][SIZE_MAX/2];`). The response to DR #266 stated:

DR #266

> *The committee has deliberated and decided that more than one interpretation is reasonable.*

There is no requirement on implementations to provide a definition of the type `size_t` that is capable of representing the number of bytes in any object that the implementation is capable of allocating storage for. It is the implementation's responsibility to ensure that the type it uses for `size_t` internally is the same as the typedef definition of `size_t` in the supplied header, **<stddef.h>**. If these types differ, the implementation is not conforming.

A developer can define a typedef whose name is `size_t` (subject to the constraints covering declarations of identifiers). Such a declaration does not affect the type used by a translator as its result type for the **sizeof** operator.

C++

5.3.3p1

. . . ; the result of ***sizeof*** *applied to any other fundamental type (3.9.1) is implementation-defined.*

The C++ Standard does not explicitly specify any behavior when the operand of **sizeof** has a derived type. A C++ implementation need not document how the result of the **sizeof** operator applied to a derived type is calculated.

Coding Guidelines

Use of the **sizeof** operator can sometimes produce results that surprise developers. The root cause of the surprising behavior is usually that the developer forgot that the result of the **sizeof** has an unsigned type (which causes the type of the other operand, of a binary operator, to be converted to an unsigned type). Developers forgetting about the unsignedness of the result of a **sizeof** is not something that can be addressed by a guideline recommendation.

1128 EXAMPLE 1 A principal use of the **sizeof** operator is in communication with routines such as storage allocators and I/O systems. A storage-allocation function might accept a size (in bytes) of an object to allocate and return a pointer to **void**. For example:

```
extern void *alloc(size_t);
double *dp = alloc(sizeof *dp);
```

The implementation of the **alloc** function should ensure that its return value is aligned suitably for conversion to a pointer to **double**.

Commentary

Measurements of existing source (see Table **??**) shows that this usage represents at most 14% of all uses of the **sizeof** operator.

1129 EXAMPLE 2 Another use of the **sizeof** operator is to compute the number of elements in an array:

```
sizeof array / sizeof array[0]
```

Commentary

The declaration of an object having an array type may not contain an explicit value for the size, but obtain it from the number of elements in an associated initializer.

Other Languages

Some languages provide built-in support for obtaining the bounds or the number of elements in an array. For instance, Fortran has the intrinsic functions **LBOUND** and **UBOUND**; Ada specifies the attributes **first** and **last** to return the lower and upper bounds of array, respectively.

1130 EXAMPLE 3 In this example, the size of a variable length array is computed and returned from a function:

```
#include <stddef.h>

size_t fsize3(int n)
{
        char b[n+3];       // variable length array
        return sizeof b;   // execution time sizeof
}

int main()
{
        size_t size;
        size = fsize3(10); // fsize3 returns 13
        return 0;
}
```

Commentary

In this example the result of the **sizeof** operator is not known at translation time.

C90

This example, and support for variable length arrays, is new in C99.

---

footnote 85

85) When applied to a parameter declared to have array or function type, the **sizeof** operator yields the size of the adjusted (pointer) type (see 6.9.1). 1131

Commentary

array type adjust to pointer to array converted to pointer

The more specific reference is 6.7.5.3. The parameter type is converted before the **sizeof** operator operates on it. There is no array type for the operand exception.

C++

This observation is not made in the C++ Standard.

Other Languages

Converting an array parameter to a pointer type is unique to C (and C++).

Coding Guidelines

Traditionally, the reason for declaring a parameter to have an array type is to create an association in the developer's mind and provide hints to static analysis tools. (Functionality added in C99 provides a mechanism for specifying additional semantics with this usage.) In most contexts it does not matter whether readers of the code treat the parameter as having an array or pointer type. However, in the context of an operand to the **sizeof** operator, there is an important difference in behavior.

qualified array of

Example

In the following the array `b` will be declared to have an upper bound of `sizeof(int *)`, not the number of bytes in the array `a`.

```
1  void f(int a[3])
2  {
3  unsigned char b[sizeof(a)];
4  }
```

Forward references: common definitions <**stddef.h**> (7.17), declarations (6.7), structure and union specifiers (6.7.2.1), type names (6.7.6), array declarators (6.7.5.2). 1132

References

1. Diab Data. *D-CC & D-C++ Compiler Suites User's Guide*. Diab Data, Inc, www.ddi.com, 4.3 edition, June 1999.