sizeof

sizeof is a unary operator in the programming languages C and C++. It generates the storage size of an expression or a data type, measured in the number of *char*-sized units. Consequently, the construct *sizeof (char)* is guaranteed to be *1*. The actual number of bits of type char is specified by the preprocessor macro `CHAR_BIT`, defined in the standard include file limits.h. On most modern computing platforms this is eight bits. The result of *sizeof* has an unsigned integer type that is usually denoted by size_t.

The operator has a single operand, which is either an expression or the cast of a data type, which is a data type enclosed in parentheses. Data types may not only be primitive types, such as integer and floating-point types, but also pointer types, and compound datatypes (unions, structs, and C++ classes).

Purpose

Many programs must know the storage size of a particular datatype. Though for any given implementation of C or C++ the size of a particular datatype is constant, the sizes of even primitive types in C and C++ may be defined differently for different platforms of implementation. For example, runtime allocation of array space may use the following code, in which the sizeof operator is applied to the cast of the type *int*:

```
int *pointer =  malloc(10 * sizeof (int));
```

In this example, function *malloc* allocates memory and returns a pointer to the memory block. The size of the block allocated is equal to the number of bytes for a single object of type *int* multiplied by 10, providing space for ten integers.

It is generally not safe to assume the size of any datatype. For example, even though most implementations of C and C++ on 32-bit systems define type *int* to be four octets, this size may change when code is ported to a different system, breaking the code. The exception to this is the data type *char*, which always has the size *1* in any standards-compliant C implementation. In addition, it is frequently difficult to predict the sizes of compound datatypes such as a *struct* or *union*, due to padding. The use of *sizeof* enhances readability, since it avoids unnamed numeric constants (magic numbers).

An equivalent syntax for allocating the same array space results from using the dereferenced form of the pointer to the storage address, this time applying the operator to a pointer variable:

```
int *pointer =  malloc(10 * sizeof *pointer);
```

Use

The operator *sizeof* produces the required memory storage space of its operand when the code is compiled. The operand is written following the keyword *sizeof* and may be the symbol of a storage space, e.g., a variable, an expression, or a type name. Parentheses for the operand are optional, except when specifying a

type name. The result of the operator is the size of the operand in bytes, or the size of the memory storage requirement. For expressions, it evaluates to the representation size for the type that would result from evaluation of the expression, which is not performed.

For example, since *sizeof (char)* is defined to be 1[1] and assuming the integer type is four bytes long, the following code fragment prints `1,4` :

```
char c;
printf ("%zu,%zu\n", sizeof c, sizeof (int));
```

Certain standard header files, such as *stddef.h*, define *size_t* to denote the unsigned integral type of the result of a *sizeof* expression. The *printf* width specifier *z* is intended to format that type.

sizeof cannot be used in C preprocessor expressions, such as `#if`, because it is an element of the programming language, not of the preprocessor syntax, which has no data types.

The following example in C++ uses the operator *sizeof* with variadic templates.

```
template <typename... Args>
std::size_t GetSize (Args&&... args)
{
    /* Get size of parameter pack.*/
    std::size_t Count = sizeof... (Args);
    return Count;
}
```

sizeof can be used with variadic templates in C++11 and above on a parameter pack to determine the number of arguments.

Application to arrays

When *sizeof* is applied to the name of an array, the result is the number of bytes required to store the entire array. This is one of the few exceptions to the rule that the name of an array is converted to a pointer to the first element of the array, and is possible just because the actual array size is fixed and known at compile time, when the *sizeof* operator is evaluated. The following program uses *sizeof* to determine the size of a declared array, avoiding a buffer overflow when copying characters:

```
1  #include <stdio.h>
2
3  int main(int argc, char **argv)
4  {
5    char buffer[10]; /* Array of 10 chars */
6
7    /* Copy at most 9 characters from argv[1] into buffer,
8     * null-terminate the buffer. */
9    snprintf(buffer, sizeof buffer, "%s", argv[1]);
10
11   return 0;
12 }
```

Here, `sizeof buffer` is equivalent to `10 * sizeof buffer [0]`, which evaluates to 10, because the size of the type *char* is defined as 1.

C99 adds support for flexible array members to structures. This form of array declaration is allowed as the last element in structures only, and differs from normal arrays in that no length is specified to the compiler. For a structure named *s* containing a flexible array member named *a*, `sizeof s` is therefore equivalent to `offsetof (s, a)`:

```c
#include <stdio.h>

struct flexarray {
    char val;
    int array[];  /* Flexible array member; must be last element of struct */
};

int main(int argc, char **argv)
{
    printf("sizeof (struct flexarray) == %zu\n", sizeof (struct flexarray));
    return 0;
}
```

In this case the *sizeof* operator returns the size of the structure, including any padding, but without any storage allowed for the array. Most platforms produce the following output:

```
sizeof (struct flexarray) == 4
```

C99 also allows variable length arrays that have the length specified at runtime,[2] although the feature is considered an optional implementation in later versions of the C standard. In such cases, the *sizeof* operator is evaluated in part at runtime to determine the storage occupied by the array.

```c
#include <stddef.h>

size_t flexsize(int n)
{
   char b[n + 3];   /* Variable length array */
   return sizeof b;  /* Execution time sizeof */
}

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

sizeof can be used to determine the number of elements in an array, by dividing the size of the entire array by the size of a single element. This should be used with caution; When passing an array to another function, it will "decay" to a pointer type. At this point, sizeof will return the size of the pointer, not the total size of the array. As an example with a proper array:

```c
int main (void)
{
   int tab [10];
   printf ("Number of elements in the array: %zu\n", sizeof tab / sizeof tab [0]); /* yields
10 */
   return 0;
}
```

Incomplete types

sizeof can only be applied to "completely" defined types. With arrays, this means that the dimensions of the array must be present in its declaration, and that the type of the elements must be completely defined. For *struct*s and *union*s, this means that there must be a member list of completely defined types. For example, consider the following two source files:

```
/* file1.c */
int arr [10];
struct x {int one; int two;};
/* more code */

/* file2.c */
extern int arr [];
struct x;
/* more code */
```

Both files are perfectly legal C, and code in `file1.c` can apply *sizeof* to *arr* and `struct x`. However, it is illegal for code in `file2.c` to do this, because the definitions in `file2.c` are not complete. In the case of *arr*, the code does not specify the dimension of the array; without this information, the compiler has no way of knowing how many elements are in the array, and cannot calculate the array's overall size. Likewise, the compiler cannot calculate the size of `struct x` because it does not know what members it is made up of, and therefore cannot calculate the sum of the sizes of the structure's members (and padding). If the programmer provided the size of the array in its declaration in `file2.c`, or completed the definition of `struct x` by supplying a member list, this would allow the application of *sizeof* to *arr* or `struct x` in that source file.

Object members

C++11 introduced the possibility to apply the *sizeof* parameter to specific members of a class without the necessity to instantiate the object to achieve this.[3] The following example for instance yields `4` and `8` on most platforms.

```
#include <iostream>

struct foo {
  int a;
  int b;
};

int main ()
{
  std::cout << sizeof foo::a << "\n" << sizeof (foo) << "\n";
}
```

Variadic template packs

C++11 introduced variadic templates; the keyword *sizeof* followed by ellipsis returns the number of elements in a parameter pack.

```
template <typename... Args>
void print_size (Args... args)
{
  std::cout << sizeof... (args) << "\n";
}

int main ()
```

```
{
  print_size (); // outputs 0
  print_size ("Is the answer", 42, true); // outputs 3
}
```

Implementation

When applied to a fixed-length datatype or variable, expressions with the operator *sizeof* are evaluated during program compilation; they are replaced by constant result-values. The C99 standard introduced variable-length arrays (VLAs), which required evaluation for such expressions during program execution. In many cases, the implementation specifics may be documented in an application binary interface (ABI) document for the platform, specifying formats, padding, and alignment for the data types, to which the compiler must conform.

Structure padding

When calculating the size of any object type, the compiler must take into account any required data structure alignment to meet efficiency or architectural constraints. Many computer architectures do not support multiple-byte access starting at any byte address that is not a multiple of the word size, and even when the architecture allows it, usually the processor can fetch a word-aligned object faster than it can fetch an object that straddles multiple words in memory.[4] Therefore, compilers usually align data structures to at least a word boundary, and also align individual members to their respective boundaries. In the following example, the structure *student* is likely to be aligned on a word boundary, which is also where the member *grade* begins, and the member *age* is likely to start at the next word address. The compiler accomplishes the latter by inserting padding bytes between members as needed to satisfy the alignment requirements. There may also be padding at the end of a structure to ensure proper alignment in case the structure is used as an element of an array.

Thus, the aggregate size of a structure in C can be greater than the sum of the sizes of its individual members. For example, on many systems the following code prints `8` :

```
struct student {
  char grade; /* char is 1 byte long */
  int age; /* int is 4 bytes long */
};

printf ("%zu", sizeof (struct student));
```

See also

- typeof
- offsetof

References

1. "C99 standard (ISO/IEC9899)" (http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1256.pdf) (PDF). ISO/IEC. 7 September 2007. 6.5.3.4.3, p. 80. Retrieved 31 October 2010.
2. "WG14/N1124 Committee Draft ISO/IEC 9899" (http://www.open-std.org/JTC1/SC22/wg14/www/docs/n1124.pdf) (PDF). 6 May 2005. 6 May 2005. 6.5.3.4 The *sizeof* operator.

3. "N2253 Extending sizeof to apply to non-static data members without an object (Revision 1)" (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2253.html).
4. Rentzsch, Jonathan (8 February 2005). "Data alignment: Straighten up and fly right" (http://www.ibm.com/developerworks/library/pa-dalign/). IBM. Retrieved 29 September 2014.
