Face-off with *Sizeof()*

In C programming, the unary operator *sizeof()* returns the size of its operand in bytes. The *sizeof()* operator is discussed in detail in this article, along with illustrative examples of code.

Sizeof() is used extensively in the C programming language. It is a compile-time unary operator, which can be used to compute the size (in bytes) of any data type of its operands. The operand may be an actual type-specifier (such as *char* or *ints*) or any valid expression. The *sizeof()* operator returns the total memory allocated for a particular object in terms of bytes. The resultant type of the *sizeof()* operator is *size_t*. *Sizeof()* can be applied both for primitive data types (such as *char, ints, floats*, etc) including pointers and also for the compound data types (such as structures, unions, etc).

Usage

The *sizeof()* operator can be used in two different cases depending upon the operand types. Let me elaborate this with syntax, examples and sample codes.

Case 1: When the operand is a type-name

When *sizeof()* is used with 'type-name' as the operand (such as *char, int, float, double*, etc), it returns the amount of memory that will be used by an object of that type. In this case, the operand should be enclosed within parenthesis. The *sizeof* operator is used when the operand is a type-name as shown in Table 1.

Illustration: The sample demo code is shown below (Code 1), and the output of the code when I run it in my system is shown in Figure 1.

Code 1

```
#include <stdio.h>
#include <stddef.h>

int main()
{
        printf("Sizeof(char) : %lu\n", (long unsigned )
sizeof(char));
        printf("Sizeof(int) : %lu\n", (long unsigned )
sizeof(int));
        printf("Sizeof(float) : %lu\n", (long unsigned )
sizeof(float));
        printf("Sizeof(double) : %lu\n", (long unsigned )
sizeof(double));
```

Table 1: Syntax with some commonly used examples

Syntax	Examples
Sizeof (type-name)	1. sizeof (char) 2. sizeof (int) 3. sizeof (float) 4. sizeof (double) 5. sizeof (12.5) 6. sizeof (‘A’) etc...

Table 2: Syntax with some commonly used examples

Syntax	Examples
Sizeof (expression) *OR* Sizeof expression	1. sizeof (i++) 2. sizeof (a+b) etc...

```
        return 0;
}
```

Note 1: The return value of the *sizeof()* operator is implementation-defined, and its type (an unsigned integer type) is *size_t,* defined in <*stddef.h*>.

Note 2: C99 has included *%zu* as a type specifier for size_t. But for older compilers, %z will fail. Hence, use *%lu* or *%llu* along with a typecasting to achieve portability of the code across various platforms.

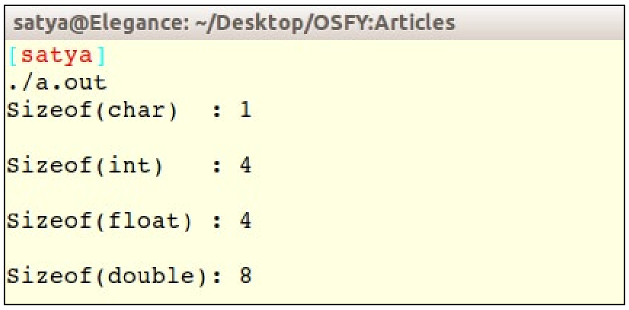
Case 2: When the operand is an expression

When *sizeof()* is used with expression as an operand, the operand can be enclosed with or without parenthesis. The syntax of how the *sizeof()* operator is used preceding the expression is shown in Table 2 along with some examples.

Illustration: The sample demo code is shown in the Code 2 snippet, and the output of the code when I run it in my system is shown in Figure 2.

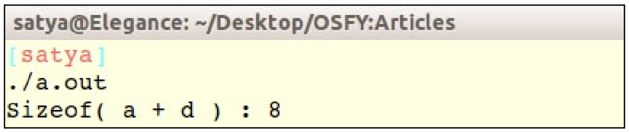
Code 2

```
#include <stdio.h>
#include <stddef.h>

int main()
{
        int a = 10;

        double d = 12.34;

        printf(“Sizeof( a + d ): %lu\n”, (long unsigned)
sizeof(a + d));

        return 0;
```



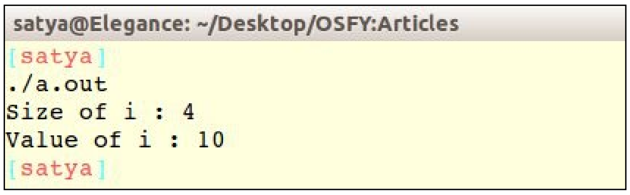
```
satya@Elegance: ~/Desktop/OSFY:Articles
[satya]
./a.out
Sizeof(char)  : 1

Sizeof(int)   : 4

Sizeof(float) : 4

Sizeof(double): 8
```

Figure 1: Output of the demo program shown in Code 1



```
satya@Elegance: ~/Desktop/OSFY:Articles
[satya]
./a.out
Sizeof( a + d ) : 8
```

Figure 2: Output of the demo program shown in Code 2



```
satya@Elegance: ~/Desktop/OSFY:Articles
[satya]
./a.out
Size of i : 4
Value of i : 10
[satya]
```

Figure 3: Output of the demo program shown in Code 3

```
}
```

From the above demo code, it is very clear that when the *sizeof()* operator is applied to an expression, it yields a result that is the same as if it had been applied to the type-name of the resultant of the expression. Since, at compile time the compiler analyses the expression to determine its type, but it will never evaluate the expression which takes place at runtime.

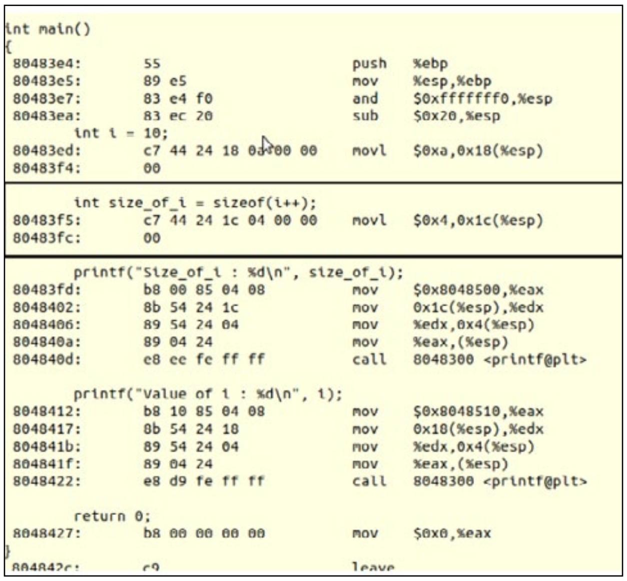
In the example shown in Code 2, ‘a’ is of *int* type and‘d’ is of double type. When type conversion is applied, as usual, the lower rank data type is promoted to a higher rank data type and the resultant data type is nothing but a double in our case; hence, *sizeof(a + d)* yields *sizeof(double)* which is 8 bytes as shown in Figure 2. In general, if the operand contains the operators that perform type conversions, the compiler considers these conversions in determining the type of the expression.

Behaviour

The *sizeof()* operator behaves differently in comparison with other operators. In this article, let me point out the uniqueness of this operator by taking two real-time programming examples. The first is about compile-time behaviour and the second one is about runtime behaviour.

Case 1: Compile-time behaviour

To start with, let us consider the simple code as shown in the Code 3 snippet.



```
int main()
{
 80483e4:       55                      push   %ebp
 80483e5:       89 e5                   mov    %esp,%ebp
 80483e7:       83 e4 f0                and    $0xfffffff0,%esp
 80483ea:       83 ec 20                sub    $0x20,%esp
        int i = 10;
 80483ed:       c7 44 24 18 0a 00 00    movl   $0xa,0x18(%esp)
 80483f4:       00

        int size_of_i = sizeof(i++);
 80483f5:       c7 44 24 1c 04 00 00    movl   $0x4,0x1c(%esp)
 80483fc:       00

        printf("Size_of_i : %d\n", size_of_i);
 80483fd:       b8 00 85 04 08          mov    $0x8048500,%eax
 8048402:       8b 54 24 1c             mov    0x1c(%esp),%edx
 8048406:       89 54 24 04             mov    %edx,0x4(%esp)
 804840a:       89 04 24                mov    %eax,(%esp)
 804840d:       e8 ee fe ff ff          call   8048300 <printf@plt>

        printf("Value of i : %d\n", i);
 8048412:       b8 10 85 04 08          mov    $0x8048510,%eax
 8048417:       8b 54 24 18             mov    0x18(%esp),%edx
 804841b:       89 54 24 04             mov    %edx,0x4(%esp)
 804841f:       89 04 24                mov    %eax,(%esp)
 8048422:       e8 d9 fe ff ff          call   8048300 <printf@plt>

        return 0;
 8048427:       b8 00 00 00 00          mov    $0x0,%eax
}
 804842c:       c9                      leave
```

Figure 4: Assembly code generated by the compiler for Code 3

Code 3

```
#include <stdio.h>
#include <stddef.h>

int main()
{
        int i = 10;

        size_t size = sizeof(i++);

       printf("Size of i : %lu\n", (long unsigned ) size);

        printf("value of i : %d\n", i);

        return 0;
}
```

Can you guess what the output of the above mentioned program will be? At first glance, anybody would say it is 4 (assuming the *sizeof(int)* is 4 bytes) and 11. But, when I run the program in my system, it shows 4 and 10 (refer Figure 3 for output).

Why are we getting the value of variable 'i' as 10 instead of 11? Here is the reason.

The *sizeof* operator is the only one in C, which is evaluated at compile time, where *sizeof(i++)* in our example is replaced by the value 4 during compile time itself. We can validate this by referring to Figure 4, which contains the assembly code equivalent to the *sizeof(i++)* in C.

Note: To obtain the assembly code as shown in Figure 4, follow the steps shown below:
+ *gcc -g filename.c* (in our case, the file name is *sizeof_run.c*)
+ *objdump -S output_file* (in our case, the output file is *a.out*)

From Figure 4, we can see that *sizeof()* is completely evaluated at compile time (the exception is *gcc*, which supports zero-sized structures as a GNU extension, which is evaluated at the runtime). And the whole *sizeof(i++)* is replaced by the constant value 4, which is highlighted in the box. Hence, there are no assembly instructions for *i++* at all, which is supposed to be evaluated at the runtime.

Case 2: Runtime behaviour

As mentioned earlier, *sizeof()* is the only operator in C, which is evaluated at the compile time. But, there is an exception for this in C99 standards, for variable length arrays.

To start with, let us consider the following code (Code 4):

Code 4

```
#include <stdio.h>
#include <stddef.h>

int main()
{
        unsigned int size;
        size_t array_size;

        printf("Enter the size:");
        scanf("%u", &size);

        //Declaring the variable length array
        int array[size];

        //Finding the size of array
        array_size = sizeof(array);

        printf("Size of array : %lu\n", (long unsigned ) array_size);

        return 0;
}
```

Let us see the output, when the above code is compiled and run (shown in Figure 5).

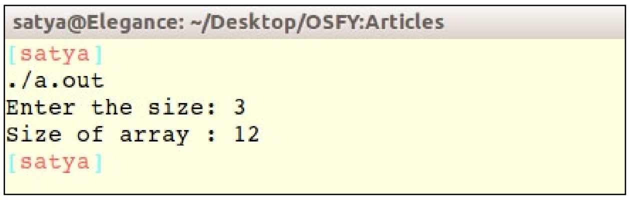
From the above output it is very clear that the *sizeof()* operator is evaluated at runtime. We can observe the equivalent assembly code generated by the compiler as shown in Figure 6.

Also, note the difference between the assembly code in Figures 4 and 6.

The need for *sizeof()*

Case 1: Auto determination of the number of elements in an array

To compute the number of elements of the array automatically, depending on the data-type of the element, the *sizeof()* operator comes in handy.



```
satya@Elegance: ~/Desktop/OSFY:Articles
[satya]
./a.out
Enter the size: 3
Size of array : 12
[satya]
```

Figure 5: Output of the *sizeof_run.c*

For an explanation, let us consider the code snippet given below:

```
1 #include <stdio.h>
 2 #include <stddef.h>
 3
 4 int main()
 5 {
 6         int array[] = {10, 20, 30, 40, 50};
 7
 8         size_t i;
 9
 10        for(i = 0; i < sizeof(array) / sizeof(array[0]);
i++)
 11        {
 12                //some code
 13        }
 14 }
```

In the example shown above, at line number 10, by using the *sizeof()* operator the number of elements is automatically computed.

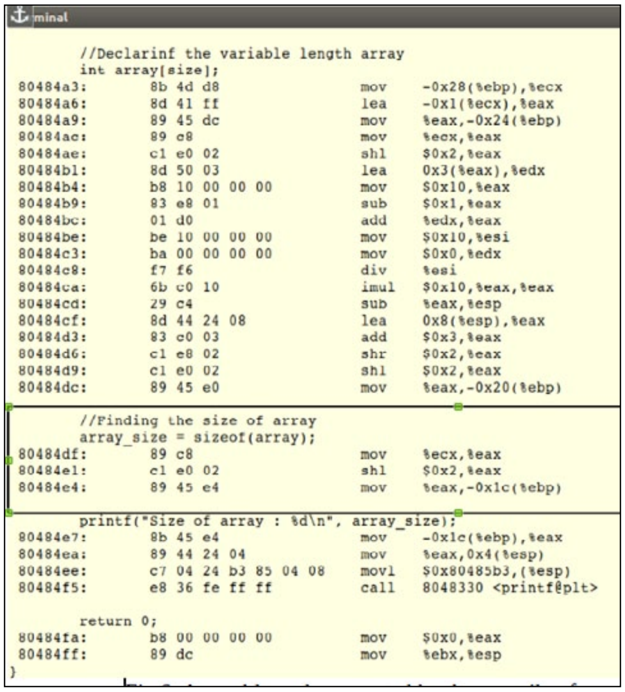
The sizes of primitive data types in C are implementation defined. For example, the *sizeof(long)* on 32-bit architecture may vary from that on 64-bit architecture. So, when we decide statically the *sizeof(long)* as 4 bytes wide on 32-bit architecture, and when the same code is ported to 64-bit architecture, the results may go wrong. So, in order to avoid the portability issue, it is a best practice to use *sizeof()* to compute the *sizeof* variables or arrays, depending on the exact size of a particular data type.

Case 2: To allocate a block of memory dynamically of a particular data type

In case of dynamic memory allocation of an array, *sizeof()* plays an important role. For example, if we want to allocate a block of memory that is big enough to hold '5' integers in an array, *sizeof* comes in handy and is a great help, since we do not know the exact sizeof(int) to dynamically allocate the memory using malloc function for a particular architecture.

```
 int *iptr = malloc( 5* sizeof(int));
```

In the above example, we are *mallocing* the block of memory, which is equal to the number of bytes of type



```
minal
        //Declarinf the variable length array
        int array[size];
 80484a3:       8b 4d d8                mov    -0x28(%ebp),%ecx
 80484a6:       8d 41 ff                lea    -0x1(%ecx),%eax
 80484a9:       89 45 dc                mov    %eax,-0x24(%ebp)
 80484ac:       89 c8                   mov    %ecx,%eax
 80484ae:       c1 e0 02                shl    $0x2,%eax
 80484b1:       8d 50 03                lea    0x3(%eax),%edx
 80484b4:       b8 10 00 00 00          mov    $0x10,%eax
 80484b9:       83 e8 01                sub    $0x1,%eax
 80484bc:       01 d0                   add    %edx,%eax
 80484be:       be 10 00 00 00          mov    $0x10,%esi
 80484c3:       ba 00 00 00 00          mov    $0x0,%edx
 80484c8:       f7 f6                   div    %esi
 80484ca:       6b c0 10                imul   $0x10,%eax,%eax
 80484cd:       29 c4                   sub    %eax,%esp
 80484cf:       8d 44 24 08             lea    0x8(%esp),%eax
 80484d3:       83 c0 03                add    $0x3,%eax
 80484d6:       c1 e8 02                shr    $0x2,%eax
 80484d9:       c1 e0 02                shl    $0x2,%eax
 80484dc:       89 45 e0                mov    %eax,-0x20(%ebp)

        //Finding the size of array
        array_size = sizeof(array);
 80484df:       89 c8                   mov    %ecx,%eax
 80484e1:       c1 e0 02                shl    $0x2,%eax
 80484e4:       89 45 e4                mov    %eax,-0x1c(%ebp)

        printf("Size of array : %d\n", array_size);
 80484e7:       8b 45 e4                mov    -0x1c(%ebp),%eax
 80484ea:       89 44 24 04             mov    %eax,0x4(%esp)
 80484ee:       c7 04 24 b3 85 04 08    movl   $0x80485b3,(%esp)
 80484f5:       e8 36 fe ff ff          call   8048330 <printf@plt>

        return 0;
 80484fa:       b8 00 00 00 00          mov    $0x0,%eax
 80484ff:       89 dc                   mov    %ebx,%esp
}
```

Figure 6: Assembly code generated by the compiler, for variable length array

int, multiplied by 5, ensuring sufficient space for all five *ints* is allocated.

Case 3: To determine the *sizeof* compound data types

Sometimes, it is very difficult to predict the sizes of compound data types such as structures, due to structure padding, and to predict the size of unions. *Sizeof()* is of great use here.

Cases when *sizeof()* will not work

The *sizeof* operator will not work when applied to the following cases:

- A bit field
- A function type
- An incomplete type (such as void)
- Zero-sized array (except in GCC, which supports zero-sized structures as a GNU extension).

sizeof() and incomplete data types

An incomplete type in C is one that describes an identifier, but lacks the information needed to determine the size of the identifier.

Examples of incompletely defined types are:

1. An array type whose dimensions have not yet been specified
2. A structure type whose members have not yet been specified

Illustration: An array whose dimensions are not specified

File1.c	File2.c
int array[10];	extern int array;

In the above example, for the code in *file1.c*, *sizeof()* can be applied to find the size of the array, as it is completely defined in *file1.c*. But, in *file2.c*, the *sizeof()* operator will not work since the dimensions of the array are missing. Without this information, the compiler has no knowledge of how many elements are in the array and cannot calculate the *sizeof* of the array.

How *sizeof()* is different from a function call

Let us consider the following code to understand how *sizeof()* is different from a function call:

```
#include <stdio.h>
#include <stddef.h>

int main()
{
        int x = 5;

  printf("%lu:%lu:%lu\n", (long unsigned )sizeof(int),
 (long unsigned )sizeof x, (long     unsigned )sizeof 5);

        return 0;
}
```

In the above example, *sizeof()* will work even if the braces are not present across operands, whereas in functions, braces are a must. So, here are three reasons why *sizeof* is not a function:

1. It can be applied for any type of operand.
2. It can also be used, when type is an operand.
3. No brackets needed across operands

You can implement your own *SIZEOF()* macro, which should work like a *sizeof()* operator.

According to the C99 standards, the *sizeof()* operator yields the size (in integer bytes) of its operand, which may be an expression or the parenthesised name of a type. If the type of the operand is a variable length array type, the operand is evaluated at runtime; otherwise, the operand is not evaluated and the result is an integer constant, during the compile time itself. END

By: Satyanarayana Sampangi

The author is a member of the embedded software team at Emertxe Information Technology (P) Ltd *(http://www.emertxe.com).* His areas of interest are embedded C programming combined with data structures and microcontrollers. He can be reached at *satya@emertxe.com*