Tip 23: the limits of `sizeof`

Ben Klemens

16 November 2011

level: obscure
purpose: recognize bad advice about `sizeof`

This tip is about the `sizeof` operator, and one difference where the statement *arrays and pointers are identical in C* is false.

I marked this entry as *obscure* because you can live a long life without using the `sizeof` operator except inside allocations like

```
element_type * newlist = malloc(listlen * sizeof(element_type));
```

I even wouldn't fault you if you wrapped it in a macro like

```
#define Allocate(element_type, listname, listlength) \
    element_type * listname = malloc(listlength * sizeof(element_type));
```

and never typed the word `sizeof` again. [You'd need a realloc macro too. Exercise for the reader.]

To summarize today's tip: the above form is a safe use of `sizeof`, and not much else is.

We start the story under the hood. The C compiler is famously ignorant of metadata, knowing only a few facts about your data:

1. It knows the base location of your data (so you can always point to it).

2. It knows the size of one unit of your data. Recall that C relies heavily on a base-plus-offset system (Entry #066) for pulling elements of arrays and structs, so it needs to know how many bytes to step when you write `base + 3` or `mystruct.third_elmt`.

3. For static and automatic memory, it needs to know the total size that has to be automatically freed at the end of the function or at the end of the program.

That's about it.

[That is as much as the system needs for its own operation, but ¿why doesn't C provide more, like a consistent method to query the size of a block of manually allocated memory? Letting implementers of `malloc` pick their favorite means of recording the block size provided the sort of freedom that delights the system programmers, and thanks to this non-policy there are a lot of different implementations of `malloc`[1]. It's annoying, but your computer is faster for it.]

Which brings us to the `sizeof` operator. You might think `sizeof` is just another function, but it's a keyword built into the compiler, because it has to have exceptional knowledge about structure internals and is the only non-macro chance you have to operate on a type. It is a window into item #3 in the list.

Here's a trick that's often thrown around[2]: you can get the size of an automatic or static array by dividing its total size by the size of one element. This is usually via a form like

```
//This is not reliable:
#define arraysize(list) sizeof(list)/sizeof(list[0])
```

The denominator of the expression depends on #2 above: the system has to know the size of one element.

The numerator really depends on #3, and is where the distinction between automatic versus manually-allocated data will trip you up. What is the size of the data that C will have to free when the variable goes out of scope? For an automatic array like `double list[100]`, the compiler had to allocate a hundred `double`s, and will have to free that much space at the end of scope. For manually-allocated memory, all the system has to do at the end of the scope is destroy the pointer—freeing the data itself is your problem. So: `sizeof` will probably return 200 in the case of the auto array, and will probably return one in the case of the manual array.

Some cats, when you point to a toy, will go and inspect the toy; some cats will sniff your finger.

Here's some sample code, so you can see what your own system returns when dealing with automatic and manual memory.

```
#include <stdio.h>

#define peval(cmd) printf(#cmd ": %g\n", cmd);

int main(){
    double *liszt = (double[]){1, 2, 3};
    double list[] = {1, 2, 3};
    peval(sizeof(liszt)/(sizeof(double)+0.0));
    peval(sizeof(list)/(sizeof(double)+0.0));
}
```

You'll recognize the add-zero trick from the last tip (Entry #072). The initialization of `liszt` may be a form unfamiliar to you. For now, rest assured that it works in appropriate conditions; I'll get to it in a few tips, so think of it as foreshadowing.

When you run the program, you get two different values. The first variable is a pointer, and the second an array. The size of the first is the size of one pointer (which is appropriately half of a `double`); the size of the second is three `double`s long.

[1] `http://en.wikipedia.org/wiki/Malloc#Implementations`
[2] `http://c-faq.com/aryptr/arraynels.html`

The only reason the system cares about the total size of your data is for the purposes of freeing it when leaving scope, and that's the size you're going to get when you use `sizeof`. But that is pretty much always a different purpose than what you had in mind.

[Formally, the `sizeof` operator isn't really tied to the end-of-scope freeing, but it coincides so darn well that I'm comfortable recommending it here as a functional mental model.]

This break in purposes makes `sizeof` largely useless outside of calls to `malloc`. We'd like to write a function `manipulate\_array(double in_array[])` and use `sizeof` to get the size of the input array, rather than wasting the user's time asking for the length. But that won't work because the user may send either a pointer or an array, and we won't know which. [It can also fail for other reasons that aren't worth getting into.]