

Re: qsort semantics

Source: http://coding.derkeiler.com/Archive/C_CPP/comp.lang.c/2007-09/msg03443.html

- *From:* gauss010@xxxxxxxxxx
 - *Date:* Thu, 20 Sep 2007 21:39:09 -0700
-

On Sep 20, 10:27 pm, Barry Schwarz <schwa...@xxxxxxxxxx> wrote:

On Thu, 20 Sep 2007 00:09:12 -0700, gauss...@xxxxxxxxxx wrote:

Suppose I have an object A of type `char[M][N]`. Each `A[i]` is a buffer containing a string, and I want to sort the M strings of A using the `strcmp` function. The description of the `qsort` function says that I must pass a pointer to the first object of the array to be sorted. This leads me to write the following:

```
char A[M][N];
int cmp(const void *a, const void *b) {
    int v = strcmp(*(char (*)[N])a, *(char (*)[N])b);
```

You cast a to "pointer to an array of N char". You then dereference this expression which yields an array of N char. This expression does not meet one of the three exceptions and is therefore converted to the address of the first char in the array with type pointer to char. This is what you want to pass to `strcmp` but you don't really care about the size of the array. You could achieve the same with less typing with a simple `(char*)a`.

But as you note below, even that is not necessary.

```
if (v < 0) return -1;
else if (v == 0) return 0;
else return 1;
```

This is not necessary. You can just return `v` since the requirement is for negative, 0, or positive but not for exactly `-1` or `+1`.

My mistake. I incorrectly recalled that the comparison function needs

Re: qsort semantics

to return either -1 , 0 , or 1 .

```
}  
void sortA(void) {  
    qsort(&A[0], sizeof A / sizeof *A, sizeof *A, cmp);  
}
```

But I also want to be able to use my comparison function to sort other arrays of strings whose second array dimension is of a different size. For example, I may want to use it to do the same sort on an object of type `char[2*M][2*N]`. Then I think of rewriting it like this:

```
char A[M][N], B[2*M][2*N];  
int cmp(const void *a, const void *b) {  
    int v = strcmp(a,b);  
    if (v < 0) return -1;  
    else if (v == 0) return 0;  
    else return 1;  
}  
void sortA(void) {  
    qsort(&A[0][0], sizeof A / sizeof *A, sizeof *A, cmp);  
}  
void sortB(void) {  
    qsort(&B[0][0], sizeof B / sizeof *B, sizeof *B, cmp);  
}
```

But now I'm not strictly following the description of the `qsort` function. I'm now not *really* passing a pointer to the first object of the array to be sorted, rather I'm passing a pointer to the first byte of the first object of the array to be sorted.

Sure you are. Since the `qsort` prototype will force the first argument to be converted to `void*`, the type of the pointer in the calling statement is irrelevant. While the expressions `A`, `&A`, `A[0]`, `&A[0]`, and `&A[0][0]` all have different types~, the address they evaluate to points to the same byte, namely the first byte of `A`. There is no way for `qsort` to know which one you actually used in your calling statement.

Ah, this gets at the heart of my problem. I'm unsure when switching between pointer types like this, and that's the reason why I used the

Re: qsort semantics

weird cast to char (*)[N] — treating qsort as a magic black box that I should receive pointers from as the same type I passed in seemed more correct.

Suppose I have an array defined as `T x[2][2]`, where `T` is any type. The expression `&x` then has type `pointer-to-array-of-2-array-of-2-T`, and the expression `&x` would be said to point to the object named `x`, correct? But is this only because of the type of `&x` (`&x` could be considered to point to both `x[0]` and `x[0][0]` as well)? Is it always OK then, to write expressions such as:

- `(T *)&x` or `(T *)&x[0]` to produce a pointer to `x[0][0]`?
- `(T *)((char *)&x + sizeof x[0])` to produce a pointer to `x[1][0]`?
- any similarly contrived expression to point to a subobject of `x`?

These are intuitively correct to me, but I have trouble tracking the reasoning through the standard as to why they should behave properly.

~Yes, I know `A` and `&A[0]` have the same type as do `A[0]` and `&A[0][0]`.

Nitpick: they don't have the same type, but are equivalent when not used as the operand of `sizeof` nor `&`.

[snip]

.