

Re: [Q] about free(ptr)

Source: http://coding.derkeiler.com/Archive/C_CPP/comp.lang.c/2004-12/0537.html

From: Chris Torek (*nospam_at_torek.net*)

Date: 12/05/04

Date: 5 Dec 2004 19:51:11 GMT

>>> `my_free(&p);`

>Keith Thompson <*kst-u@mib.org*> wrote in message

>news:<*lnpt1p7k8r.fsf@nuthaus.mib.org*>...

>> *It's invalid. You're passing an argument of type `int**` to a function*

>> *that expects a `void**`. There's an implicit conversion from `int*` to*

>> *`void*`, but not from `int**` to `void**`.*

In article <*9dcfb2af.0412050508.d7de9a7@posting.google.com*>

subnet <*sunglo@katamail.com*> wrote:

>*You are right of course, haste makes waste. I wrongly extended the ""*

>*behavior to ""*.

>

>*I think that line should look*

>

>`my_free((void **) &p);`

>

>*(correct me if I'm wrong) but it's ugly anyway :-)*

That shuts up the compiler, but the code remains wrong.

Suppose we replace `my_free()` with a similar function whose only job is to set a "double" variable to 0.0:

```
void clear_it_out(double *dp) {
    /* free(dp) - does not make sense */
    *dp = 0.0;
}
```

This is very similar to what we are doing with the "void **" in the original `my_free`, which, to reiterate, looks like this:

```
void my_free(void **vpp) {
    free(*vp);
    *vp = NULL;
}
```

But now we have an ordinary "int":

```
int x;
```

and at some point, we want to clear it out:

```
clear_it_out(&x); /* WRONG; draws a diagnostic, given a prototype */
```

We can shut up the compiler by inserting a cast:

```
clear_it_out((double *)&x); /* still WRONG, but no diagnostic */
```

but (from long use of this compiler on this machine) we already know that an "int" is 4 bytes and a "double" is eight bytes. This is going to write 8 bytes of 0.0 into the four-byte "int" named "x". What happens to the other four bytes making up 0.0? The answer is: they clobber some other, possibly very important, data.

What we *can* do, if we want to clear out "x", is copy it to a "double", clear out the "double", and then copy the result back:

```
int x;
double temp;
...
temp = x; /* make a big 8-byte version */
clear_it_out(&temp); /* give clear_it_out 8 bytes to clobber */
x = temp; /* copy the cleared value back */
```

Likewise, if we want to use `my_free()` to clear out "p", which has type "int *" and is thus quite possibly smaller than a "void *" (perhaps "int *" is 4 bytes and "void *" is eight, just like the int-and-double situation), we have to copy it to a temporary:

```
int *p;
void *temp;
...
temp = p; /* make a big 8-byte version */
my_free(&temp); /* give my_free 8 bytes to clobber */
p = temp; /* copy the cleared value back */
```

Now, at last, the code is completely valid and guaranteed to work on all implementations, regardless of differences between `sizeof(void *)` and `sizeof(int *)` (ancient PRIMES), or any change in bit patterns for byte pointers vs word pointers (ancient Data General Eclipse machines), or whatever. This is *guaranteed* to work.

There is just one problem. It is also utterly stupid.

We *know* what `clear_it_out()` does to `&temp`, and using the sequence:

```
temp = x;
clear_it_out(&temp);
x = temp;
```

is just totally bonkers. All we have to do is write:

```
x = 0; /* well, that was easy */
```

which is WAY INCREDIBLY EASIER.

Likewise, we *know* what my_free() does to &temp. There is one extra step my_free() has that clear_it_out() lacks, though, and that is a call to free(). So we can expand my_free() in-line, by adding its argument as a local variable and doing assignments:

```
int *p;
void *temp;
...
temp = p;
{
    void **vpp = &temp;
    free(*vpp);
    *vpp = NULL;
}
p = NULL; /* well, that was slightly easier than "p = temp" */
```

This time, it may not be immediately obvious what all the vpp nonsense is about -- but if we note that vpp is set to &temp, and think about it for half a second, we should realize that *vpp is just another name for "temp". So we can get rid of *vpp by substituting "temp":

```
...
temp = p;
{
    void **vpp = &temp;
    free(temp);
    temp = NULL;
}
p = NULL;
```

and now it is clear that vpp is not really any use either, so we can get rid of it too:

```
...
temp = p;
free(temp);
temp = NULL;
p = NULL;
```

Now that just leaves one more moment of thought required. What good is "temp"? In the call to free(), it passes a value of the correct type ("void *" -- remember, p is "int *"). But we are allowed to assign any data-pointer type to any variable of type "void *"; it is this very property that lets us write "temp = p" in the first place. And of course, any prototyped function call passes parameters in just the same way as assignment -- so we do not have to copy "p" to a temp variable just to call free():

```
temp = p;
free(p); /* instead of free(temp) */
temp = NULL;
p = NULL;
```

But now "temp" is obviously useless, and we should get rid of it entirely. This gives:

```
free(p);
p = NULL;
```

which is of course what we should have written in the first place. That dinky little my_free() function was worse than useless -- it forced us to invent a temp variable, just so that we could copy the cleared-out value back to the variable we *really* wanted to clear out. It is just as bad as that stupid "clear_it_out" function that clears out a double.

If we need to clear out an int -- or even a double -- we can just do it ourselves. It only takes one line of code.

Likewise, if we need to clear out an int-star -- or even a void-star -- we can just do it ourselves. It only takes one line of code.

If you dislike the fact that we now have to use two lines to squeeze in the call to free() as well, just use a comma-expression to cram it into one line:

```
free(p), p = NULL;
```

or even:

```
#define FREE_VARIANT1(x) (free(x), (void *)NULL)
p = FREE_VARIANT1(p);
```

or:

```
#define FREE_VARIANT2(x) (free(x), (x) = NULL)
FREE_VARIANT2(p);
```

(I dislike the macro versions, in general; just writing the two statements, or joining them into a simple comma-expression, is simpler. The macro versions may be simpler in special cases, where

comp.lang.c: Re: [Q] about free(ptr)

they have names that suggest some useful invariant property of the code, and/or provide some sort of user-defined abstract type interface. In this case, the macro(s) probably should not be named "FREE" anyway.)

--

In-Real-Life: Chris Torek, Wind River Systems
Salt Lake City, UT, USA (40°39.22'N, 111°50.29'W) +1 801 277 2603
email: forget about it <http://web.torek.net/torek/index.html>
Reading email is like searching for food in the garbage, thanks to spammers.