Subject: Re: pointers--avoiding a memory leak
Posted by David Fanning on Wed, 19 May 2004 19:34:57 GMT
View Forum Message <> Reply to Message

M. Katz writes:

> Here's a simple Pointers 101 question for the pointer gurus.
>
> Suppose you have a structure with a pointer field
> s = {a:10, b:ptr_new(10)}
>
> Somewhere down the line you want to update the value of *s.b making it
> equal to the value contained in a another pointer, say *q = 20. After
> the assignment, you'll no longer need the q pointer.
>
> So which is a better strategy?
>
> #1)
> ptr_free, s.b
> s.b = q
>
> #2)
> *s.b = *q
> ptr_free, q
>
> #3)
> s.b = q ;--- what becomes of the old s.b in this case?
>
> I can see how #1 is memory-efficient because only the pointer is
> passed. I can see that #2 is memory inefficient because the values are
> swapped. This could be slower if the value is a large array. I can see
> how #3 might result in a memory leak, since the old s.b value could be
> stranded in memory with no pointer pointing to it. Am I right about
> these? What else should I be thinking about in the above situation?

I think you pretty much understand the situation. You definitely
leak memory in #3. I quibble a little bit with you conclusion that
#2 is memory inefficient, since I think internally C pointers are
moving around, not actual data. But other than that, I think you
can start handling the pointer guru questions from now on. :-)

Cheers,

David
--
David Fanning, Ph.D.
Fanning Software Consulting, Inc.
Coyote's Guide to IDL Programming: http://www.dfanning.com/

Subject: Re: pointers--avoiding a memory leak
Posted by JD Smith on Wed, 19 May 2004 20:54:27 GMT
View Forum Message <> Reply to Message

On Wed, 19 May 2004 13:34:57 -0600, David Fanning wrote:

> M. Katz writes:
>
>> Here's a simple Pointers 101 question for the pointer gurus.
>>
>> Suppose you have a structure with a pointer field
>> s = {a:10, b:ptr_new(10)}
>>
>> Somewhere down the line you want to update the value of *s.b making it
>> equal to the value contained in a another pointer, say *q = 20. After
>> the assignment, you'll no longer need the q pointer.
>>
>> So which is a better strategy?
>>
>> #1)
>> ptr_free, s.b
>> s.b = q
>>
>> #2)
>> *s.b = *q
>> ptr_free, q
>>
>> #3)
>> s.b = q ;--- what becomes of the old s.b in this case?
>>
>> I can see how #1 is memory-efficient because only the pointer is
>> passed. I can see that #2 is memory inefficient because the values are
>> swapped. This could be slower if the value is a large array. I can see
>> how #3 might result in a memory leak, since the old s.b value could be
>> stranded in memory with no pointer pointing to it. Am I right about
>> these? What else should I be thinking about in the above situation?
>
> I think you pretty much understand the situation. You definitely
> leak memory in #3. I quibble a little bit with you conclusion that
> #2 is memory inefficient, since I think internally C pointers are
> moving around, not actual data. But other than that, I think you
> can start handling the pointer guru questions from now on. :-)

Actually, I have to agree with M. Katz that #2 is inefficient. The basic difference between #1 and #2 is that, in #2, you have two copies of *q after the assignment.

Let's have a look:

```
IDL> s={a:10, b:ptr_new(bytarr(1024,1024,100),/NO_COPY)}
IDL> q=ptr_new(make_array(VALUE=5b,1024,1024,100),/NO_COPY)
IDL> help,/memory
heap memory used:  210667841, max:  210667860, gets:     6933, frees:     6614
IDL> *s.b=*q
IDL> help,/memory
heap memory used:  210667870, max:  210667889, gets:     6937, frees:     6616
```

We used no extra memory in the assignment. Fine then, no problem, right? Well, actually, this is not a memory-intensive operation, but it is *slow*, since it requires copying over all of that data. Let's do a fun, bigger example, with 1/2 GB of memory in each:

```
IDL> s={a:10, b:ptr_new(bytarr(1024,1024,512),/NO_COPY)}
IDL> q=ptr_new(make_array(VALUE=5b,1024,1024,512),/NO_COPY)
IDL> t=systime(1) & *s.b=*q & print,systime(1)-t ; method 2
       40.175291
IDL> t=systime(1) & ptr_free,s.b & s.b=q & print,systime(1)-t ; method 1
     0.089432001
```

Oh man, we really hit the disk there (this machine has 1GB of memory, so two 1/2GB arrays was enough to send it to the swap). The lesson is that straight assignment does copy the data, and can be slow for large data sizes. If you think about it, this had to be the case: after assignment both s.b and q needed to point at *different* arrays. If we did *s.b=*q and then modified *s.b, wouldn't you be surprised if *q changed as well? This is why another copy has to be made.

But wait, we have another trick up our sleeve:

```
IDL> s={a:10, b:ptr_new(bytarr(1024,1024,512),/NO_COPY)}
IDL> q=ptr_new(make_array(VALUE=5b,1024,1024,512),/NO_COPY)
IDL> t=systime(1) & *s.b=temporary(*q) & print,systime(1)-t
     0.059531927
```

Now *this* is your "C pointers moving" example. Since we declare that we don't need *q anymore (with TEMPORARY), the pile of data the IDL variable *q pointed to is just linked up to *s.b. So really this method is functionally equivalent to the IDL pointer method (#1), except instead of using IDL pointers to do a lightweight re-assignment, you've used C pointers internally (which, as you can see, is actually a little faster, since the whole IDL variable doesn't need to get re-created, just its internal data pointer). My preference would be method #1, since no "allocated but undefined" pointers are left dangling around.

By the way, all of this discussion applies to regular variables too,
since as we know pointers are just a clever way of accessing regular
old IDL variables.

JD

---

Subject: Re: pointers--avoiding a memory leak
Posted by MKatz843 on Thu, 20 May 2004 03:04:06 GMT
View Forum Message <> Reply to Message

This newsgroup is a gold mine of helpful information. Many thanks to David and JD!
M.

---