Subject: Re: object newbie
Posted by John-David T. Smith on Fri, 11 Aug 2000 07:00:00 GMT
View Forum Message <> Reply to Message

Martin Schultz wrote:
>
> Mark Hadfield wrote:
>>
> [...]
>
>> Also object properties are not strictly tied to the class
>> structure: GetProperty/SetProperty keywords can represent tags in the class
>> structure or they can be dynamically interpreted, thus hiding the
>> implementation details.
>>
>
> ... to elaborate (because Chip calls himself an object newbie): I have
> an example where I store among other things an array of structures
> (actually objects of the same type) in the object (and I am in fact
> using a container for this). Say this structure array is named
> "campaigns". In my GetProperty method I then have keywords to access
> - the complete array as objects campaigns=campaigns
> - the campaigns data as structrues struc_campaigns=struc_campaigns
> - only the campaign names cnames=cnames
> - only the campaign dates cdates=cdates
> etc.
>
> Specific retrieval methods (usually functions then) enhance the
> flexibility of the access. E.g.
> GetCampaignDates(name=name) can be used to retrieve the dates for
> campaigns selected by name
> (including a pattern match) -- this is
> something I would consider
> "value added feature" when you use
> objects.
>
> The GetProperty method may in fact use the special retrieval methods to
> extract things. As long as you store only small amounts of data, it
> won't matter if you access the campaigns structure array several times
> and pass parts of it between methods. If you envision huge amounts of
> data you may want to think more carefully how often these arrays must be
> copied. I haven't dealt with these issues yet, but I would be happy to
> hear comments.
>
> Cheers,
> Martin

It should be pointed out that the GetProperty procedures outlined in the prior

postings are fully functional only when the _REF_EXTRA keyword is employed (in fact it was created by RSI due to this failing). This allows proper keyword inheritance when chaining up to the properties of superclasses -- the only other option is enumerating in the keyword list all superclass properties, which of course is an egregious violation of encapsulation... But of course, all this still does not excuse the inflexible encapsulation functionality IDL's object framework presents us.

Remember to give those properties unique names -- if you follow the data member naming convention as Martin does (e.g. cnames=cnames) you'll avoid risking namespace conflicts an additional time, since you have to worry about it with INHERIT'ing anyway. This leaves only the computed ("dynamically interpreted") property names to worry over.

As far as the array copying issue, for dealing with those properties which are truly large, the only way to pass by reference out of your GetProperty method is to use pointers... which is actually good: imagine the confusion of having otherwise unremarkable variables as silent referents to object data members. Pointers are your friends.

An additional side note: I think a basic concern people have with pointer usage is memory management -- forgetting to free pointers at the right time, or the awkwardness of having to free them. When you have various nested levels of pointers to structures with pointers to arrays of pointers, etc., it can get ugly. There are a few tricks to make freeing pointers at cleanup (object or otherwise) less cumbersome. They rely on a few convenient properties of ptr_free:

1. The fact that you may free a null pointer with impunity.
2. Arguments (of which there may be any number), are freed from first to last.
3. Pointers in arrays can be freed all at once by passing the array.

Consider this statement:

```
if ptr_valid(self.Recs) then $
   ptr_free,(*self.Recs).Ints,(*self.Recs).Time,self.Recs
```

self.Recs is a pointer to a struct containing various other pointers (like Ints and Time), which may or may not be defined (i.e. they might be null pointers). Rather than pedantically test all of the cases before freeing each field and then, and only then, free the higher level pointer, I rely on properties #1 & #2. I can free self.Recs ptr field members and the pointer itself all in one go. No worry about the chicken before the egg scenario, or undefined pointers.

Consider a pointer to an array of pointers: self.parr. Freeing it is as simple as:

if ptr_valid(self.parr) then ptr_free,*self.parr,self.parr

This works even if any or all of the pointers are null -- a general assumption for which I always allow. With these few properties of ptr_free you can really reduce the hassle of cleaning up after yourself. Allowing any pointer to be a potential null pointer also allows you to prune various things from your data structure before cleanup, to save them for other uses. Simply do something like:

```
kept_ptr=self.saveme
self.saveme=ptr_new()
```

This technique has a myriad of uses -- but that's another article.

Good Luck,

JD

--

J.D. Smith /*\ WORK: (607) 255-6263
Cornell University Dept. of Astronomy */ (607) 255-5842
304 Space Sciences Bldg. /*\ FAX: (607) 255-5875
Ithaca, NY 14853 */