Subject: Re: object newbie
Posted by davidf on Thu, 10 Aug 2000 07:00:00 GMT
View Forum Message <> Reply to Message

Chip Sample (sample@idcomm.com) writes:

> I must admit that based on comments from this list, I have experimented with
> the object features of IDL for the past week or so, and have implemented
> them in a few places in a fairly big widget code I have written.
>
> My initial observations are that there seems to be less of a temptation to
> use common blocks when objects are used. On the other hand my first
> impression was that an object is a structure whose fields can not be
> accessed until you write additional "methods" to get at each and every damn
> one of them. So my object was littered with about 25 "methods" just so I
> could pry the data out of the object.

Well, we typically write a GetProperty method and a SetProperty
method to get at things, but this will work too. :-)

> I eventually came on a work around to write a "proto_object" with a method
> allowing you to pass a string containing a tag name which returns the
> contents of the field with that tag name. This "proto_object" is inherited
> by all other objects I create just so I can use this method. Along the way
> I found that the TAG_NAMES function in IDL doesn't work for objects so I had
> to create one. It basically copies the object structure into a regular
> structure so the TAG_NAMES can be used.
>
> Am I making this too hard?

No. You are becoming a righteous IDL programmer! :-)

Cheers,

David

P.S. There is still time to get your public service
requirement out of the way before the final IDL EPA
Membership Committee meeting in September. It wouldn't
take much more work than publishing this method. :-)

--
David Fanning, Ph.D.
Fanning Software Consulting
Phone: 970-221-0438 E-Mail: davidf@dfanning.com
Coyote's Guide to IDL Programming: http://www.dfanning.com/
Toll-Free IDL Book Orders: 1-888-461-0155

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 */

Subject: Re: object newbie
Posted by Martin Schultz on Fri, 11 Aug 2000 07:00:00 GMT
View Forum Message <> Reply to Message

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

--
[[[[[[[[[[
[[Dr. Martin Schultz   Max-Planck-Institut fuer Meteorologie   [[
[[Bundesstr. 55, 20146 Hamburg   [[
[[phone: +49 40 41173-308   [[
[[fax: +49 40 41173-298   [[
[[martin.schultz@dkrz.de   [[
[[[[[[[[[[

Subject: Re: object newbie
Posted by Mark Hadfield on Fri, 11 Aug 2000 07:00:00 GMT
View Forum Message <> Reply to Message

"David Fanning" <davidf@dfanning.com> wrote in message
news:MPG.13fd20f4e909bd8c989bba@news.frii.com...
> Chip Sample (sample@idcomm.com) writes:
>
>> I eventually came on a work around to write a "proto_object" with a
method
>> allowing you to pass a string containing a tag name which returns the
>> contents of the field with that tag name. This "proto_object" is
inherited
>> by all other objects I create just so I can use this method. Along the
way

>> I found that the TAG_NAMES function in IDL doesn't work for objects so I had
>> to create one. It basically copies the object structure into a regular
>> structure so the TAG_NAMES can be used.
>>
>> Am I making this too hard?
>
> No. You are becoming a righteous IDL programmer! :-)

I don't quite agree with David. There is a conventional way of accessing object properties, i.e. via keywords to the SetProperty and GetProperty methods. I suggest you explore this convention and get a feel for its strengths and weaknesses before you try anything else. It's similar to what you've developed, but it uses keywords rather than strings. This means that it can take advantage of keyword inheritance and allows abbreviation of property names. 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.

Mark Hadfield
m.hadfield@niwa.cri.nz http://katipo.niwa.cri.nz/~hadfield/
National Institute for Water and Atmospheric Research
PO Box 14-901, Wellington, New Zealand

Subject: Re: object newbie
Posted by Martin Schultz on Mon, 14 Aug 2000 07:00:00 GMT
View Forum Message <> Reply to Message

"J.D. Smith" wrote:
>
> Martin Schultz wrote:
>>
>>
>>
>> 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
>

> And J.D. Smith commented:
>
>
> 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.
>

I fully agree, but ... ;-)

I am currently reworking a netcdf file object (based in turn on a
geenric datafile object, but that''s a detail), and I am thinking about
how to best achieve the two conflicting goals:
* allow flexible access to subportions of the data
* minimize memory usage

What I envision is a method that would allow something like
  thefile->GetData, variables=['o3','co'], /include_dimensions,
lon=,lat=
which should return an array containing only the selected data. But this
means that I have to copy the data rather than pass a pointer back,
because otherwise the next access with a different range would overwrite
the data stored in the object and thus render the first pointer
"invalid" in the sense that it would point to something different now.
The problem arises, because I do not want to read in the whole field
first and then extract what I need but I want to take advantage of
netcdf's capabilitity to extract a subset of the data directly from the
file. So, the data will be loaded dynamically only when requested. On
the other hand it will also be possible to retrieve the complete
variable data from the file and do the extraction in memory, which may
be considerably faster if you need frequent access to various portions
of the data. For example, the dimension variables
    (for non-netcdf'ers: these are variables holding the values of the
dimensions of other variables.
    Example: a model ozone field may have dimensions LON, LAT, and LEV,
then LON will be a variable
    named LON which contains the longitude values, i.e. 2.5, 7.5, 12.5,
... 357.5 on a rather coarse grid)
will always be loaded into memory and accessed from there. Makes things
a little more convoluted if I want to support write access to the netcdf
file as well (which I do want), but it's a nice challenge, and
thanks to the beauty of objects, the user doesn't have to care a dent.

As to pointer freeing: I remember that it took me three or four attempts

(about 15 years ago) before I had a vague idea what a pointer was and how one could use one. Well, this was Pascal (maybe Borland 3 or so), and things have certainly changed since then (my little daughters learned about pointers at age 2 already: "Where is pointer, where is pointer? Here I am..."). But the key to not loosing pointers nor data is that you should make it clear to yourself where your data actually resides. What often helps me is the concept of a "master pointer", i.e. the one pointer that got the data passed first. And then I usually only allow the master pointer to free the data. All other pointers are considered local variables (they are just unsigned longs in a sense), and I don't need to care about memory allocation or freeing for these. Example:

```
   master = Ptr_New(data)   ; this is the master pointer
   other = master           ; this is only a local pointer
   (*other) = (*other)*0.5  ; this operation affects the data
which is conceptually stored in master
   Ptr_Free, master         ; free memory
```

So, you don't have to worry about the fate of other at all (just that one should of course check if it is a valid pointer before using it).

[excerpt from b2p2 = beginners' guide to pointers, unpublished material, 2000, copyright = left]

Cheers,
Martin

--
[[[[[[[[[[
[[Dr. Martin Schultz   Max-Planck-Institut fuer Meteorologie   [[
[[Bundesstr. 55, 20146 Hamburg   [[
[[phone: +49 40 41173-308   [[
[[fax: +49 40 41173-298   [[
[[martin.schultz@dkrz.de   [[
[[[[[[[[[[