Subject: Re: Object Madness or Restoring Nightmares
Posted by JD Smith on Wed, 03 Mar 2004 17:24:10 GMT
View Forum Message <> Reply to Message

On Wed, 03 Mar 2004 02:22:37 -0600, Craig Markwardt wrote:

>
> David Fanning <david@dfanning.com> writes:
>
>> IDL realizes this, thinks it is being helpful, and saves every
>> object in sight! And although I can't prove it, I think it
>> saves two backup copies as well, because the entire save file
>> tops out at a hefty 10 MBytes.
>
> Craigbot says: I think IDL has a cycle counting bug. If your objects
> are doubly- (or morely-) linked, then I'm guessing that IDL is trying
> to resolve the cycles, but fails. I bet if you try a simpler data
> structure, one without cycles, it will save fine.
>
> But robots don't guess or bet, so I must have a logic error.
>
> Craig-bot

I think it's actually simpler than that. I suspect that if you follow the train of objects containing pointers to objects with pointers etc., you'll find that some object somewhere beneath your "theStudy" object actually has a pointer or object reference to the top-level application object in it. This is very common (just keeping track of that object to consult it later on).

If this is the case, here's what happens: IDL very dutifully follows all of these downward-linking object/pointer chains, collecting and saving everything it finds on the way. This is the correct thing to do, since, as far as it knows, to have a valid "theStudy" object on disk requires all of its various holdings. Now, if at some point down the chain, IDL runs into an object which is just a convenience reference to the top level application object, it will dutifully jump right to the top of the heap and start saving the whole thing. It does know better than to save heap variables twice and how to avoid circular references; you can test this with:

```
IDL> b=obj_new('IDL_CONTAINER')
IDL> b->Add,b
IDL> save,b,FILENAME='~/b.sav'
  < new session >
IDL> help,/HEAP &  restore,'~/b.sav',RESTORED_OBJECTS=r & help,/HEAP
Heap Variables:
   # Pointer: 0
```

```
   # Object : 0
Heap Variables:
   # Pointer: 1
   # Object : 1

<ObjHeapVar1>   STRUCT    = -> IDL_CONTAINER Array[1]
<PtrHeapVar2>   STRUCT    = -> IDL_CONTAINER_NODE Array[1]
IDL> print,r
<ObjHeapVar1(IDL_CONTAINER)>
```

but simply including a convenience copy of the top level object will end up including the whole thing.

This is a problem. It's actually a bigger problem than you think, because (see the various articles on your site describing it), any object which is saved has implicit in it its class definition, so if you accidentally save 10 extra objects of different classes along with the one you're really interested in, when you restore them, any updates to any of the class definition files (class__define.pro) will never be consulted, since IDL thinks it already knows all about them. The much-discussed solution is to explicitly resolve the class *before* restoring the object. You can find my latest incarnation of my routine which automates this here:

http://turtle.as.arizona.edu/idl/restore_object.pro

It provides some help to make sure you get the actual object you're after, and that all "helper" classes get compiled too. Sadly, you can't just inspect the objects which pop out and resolve their methods after the fact, since any changes to the actual class structure in your class__define.pro will not be noticed. This "read-once" nature of IDL classes is perhaps the biggest failing of IDL's OOP methodology. Imagine if classes could be extended at run-time? None of these restore issues would exist, and object development would be sped up immensely.

So, how do you avoid this situation? What I do is "detach" all the irrelevant data from my object before saving it. I've talked about this before, but the basic idea is (in your terms):

```
theStudy = self.currentStudy
theStudy->Save,'somename.sav'
```

with

```
pro theStudy::Save,filename
   saved_ptr=self.BigAndUselessDataPtr ; detach
   self.BigAndUselessDataPtr=ptr_new() ; a null pointer
   save, self,FILENAME=filename
```

```
    self.BigAndUselessDataPtr=saved_ptr  ; reattach
end
```

and to restore it:

```
theStudy=restore_object(file,'theStudy')
if obj_valid(theStudy) then begin
   if NOT obj_isa(theStudy,'theStudy') then $
      message,'Error restoring Study file: '+file
   ;; The study is valid
   obj_destroy,self.currentStudy
   self.currentStudy=theStudy
endif
```

This requires, of course, that you plan ahead and group all of the data that isn't necessary to include in the save file in some conveniently detachable object or pointer (or perhaps a few of them). Aside from convenience object references, widget data is a good candidate for detachment. Detaching an object reference works just the same, but with "obj_new()" instead of "ptr_new()".

Good luck,

JD

---