Subject: Re: IDLgrLegend broken
Posted by John-David T. Smith on Thu, 07 Dec 2000 08:00:00 GMT
View Forum Message <> Reply to Message

Mark Hadfield wrote:

> David:
>> But before I did this, I'd have a closer read of this
>> article, where JD and I (and probably Mark) discussed
>> this restore object problem and came up with a
>> "sorta" solution:
>>
>> http://www.dfanning.com/tips/saved_objects.html
>
> Hey nice one David. I don't know that I can claim any of the credit or blame
> for this article. I have read it before and I should have remembered its
> existence before posting very similar material.
>
> A comment/question on the RESOLVE_OBJ routine that's shown at the above
> link:
>
> The following code snippet ensures that each object's __define procedure is
> called only if it has not already been compiled. (The array ri holds a list
> of currently compiled routines, generated by a call to ROUTINE_INFO.)
>
> if (where(ri eq defpro))[0] eq -1 then begin
> ;; Compile and define the class.
> call_procedure,defpro
> endif
>
> My comment is: why bother? Once a __define method has been called once,
> further calls have no effect.

Well, I suppose I should offer my two bits, since I am primarily reponsible for the method on David's page (though david hit on the original idea of using class__define in some way).

There are two subtleties in this method. At first blush, you might consider using resolve_routine to have all the methods compiled (as David first thought). This works, but does not deal with the issue of changing class definitions (e.g. adding another data member). The only solution is to define your class prior to restoring the object (by *calling* class__define). Then, relaxed restoration will (usually) do what you want. This provides upward compatibility, but forces you to specify the class name in advance. For me this is often not a problem, since usually the class of the object I'm restoring is the same as the class from which I'm restoring it (say that 10 times fast), so there is no ambiguity.

AN IMPORTANT NOTE: saved objects contain in them implicit definitions of their own class, all their superclasses, and the class+superclasses of any other objects they contain!

But you can control this behavior:

To make life easier, you can extirpate the most rapidly changing class data (or any data, such as unwanted objects, for that matter), that doesn't really need to be saved. For instance, I detach all widget interface specifications (ala the ancient revered "state" structure), before saving an object. I can then feel free to redo the interface entirely. This is done with pointers (or object references): simply replace one of these with a "stub" -- ptr_new() or obj_new() -- if you don't want it in the file. I have covered the simple method for doing this many times, but I'll repeat it:

```
wSav=self.wInfo  ; detach all the widget state stuff.
self.wInfo=ptr_new()
; do whatever to save the object... please do some error checking
if no_error then self.wInfo=wSav ; reattach
```

That way, I can detach all parts which aren't central to my data structure, and be free to develop those as I like. The core components which are vital to the representation of the data are treated with more care.

This method also enables other really cool features. For instance, one of my applications has a "restore from disk" feature which simply updates, in place, an object's self variable (which workds because it's implicitly passed by *reference* to all methods) -- self-transmogrification. Besides sounding cool, it's a very useful technique.

With a bit of care, saved objects become a valid and simple tool with which to store very complex data structures.

JD

P.S.

As far as slugging around the routine_info() results, aside from some efficiency gain for large inheritance trees, this was a holdover from when compiling was done with resolve_routine, rather than (somewhat underhandedly) with call_procedure. Obviously, resolve_routine,'class__define' compiles all methods each and everytime it's called (which wouldn't be pretty if you had 100 objects to restore). Since call_procedure compiles nothing if class__define has

already been defined (a feature), the overhead of making the repeated calls is probably minimal. An update would look like:

```
pro resolve_obj, obj, CLASS=class
  if n_params() ne 0 then begin
    if NOT obj_valid(obj) then begin
      message,'Object is not valid.'
    endif
    class=obj_class(obj)
  endif

  for i=0,n_elements(class)-1 do begin
    defpro=class[i]+'__DEFINE'
    ;; (maybe) compile and define the class.
    call_procedure,defpro
    supers=obj_class(class[i],/SUPERCLASS,COUNT=cnt)
    if cnt gt 0 then resolve_obj,CLASS=supers
  endfor
end
```

But I don't expect it to be faster, and occasionally be slower (haven't tested).
