Subject: Re: Restored Object Method Compilation
Posted by J.D. Smith on Tue, 06 Oct 1998 07:00:00 GMT
View Forum Message <> Reply to Message

J.D. Smith wrote:
>
> Object Programmers,
>
> I have recently discovered a problem with the object method compilation
> technique discussed earlier in this newsgroup by David Fanning and
> myself. This discussion is archived on David's IDL page
> (http://www.dfanning.com/tips/saved_objects.html). The technique
> discussed basically allows for the compilation of uncompiled methods for
> objects restored from file when the entire set of methods resides in the
> file class__define.pro. The alternative (unacceptable in my opinion) is
> to give each method its own file of the form class__method.pro
>
> The problem relates to superclasses. The normal IDL implicit
> compilation of an object's methods (contained in class__define.pro), as
> when invoking obj_new('class'), proceeds up the inheritance tree,
> compiling and defining all of the superclasses which are not yet
> defined. Our technique does not compile superclasses, and so our
> methods are broken if they fail to override, or chain up to a superclass
> method.
>
> One might think that all you would need to do is call the class__define
> procedure for the class of the restored object, and all superclasses
> would be defined and compiled as encountered. This works in any context
> but this one. The reason why is that a restored object contains
> implicitly in its definition the class structure definitions of all its
> superclasses. Therefore, when class__define is called, it doesn't call
> or compile any of the superclass__define definitions. As far as IDL is
> concerned, "superclass" is already a valid class (struct) and nothing
> further need be done.
>
> The only solution is to proceed up the inheritance tree compiling by
> hand. A procedure, resolve_obj, which accomplishes this, is attached.
> Note that the recursion must be breadth first as coded... that is, a
> class must be compiled before its superclasses (because how would we
> know which superclasses it has otherwise). The way to call this routine
> on a restored object is:
>
> resolve_obj,obj
>
> Alternatively, if you have the name of a class rather than an object
> instance, you could do:
>
> resolve_obj,CLASS=class

>
> and the recursion would proceed in the same way. To reiterate, this
> procedure is necessary only when an class has been *defined* but not
> *compiled*.
>
> JD
>
<resolve_obj snipped>

A bit more....

OK, I think I've finally got all the bugs sorted out. While my resolve_obj procedure works fine called after an object is restored, it doesn't do exactly what I want in the case of a *changing* class definition. This is what I was doing:

```
restore,file, RESTORED_OBJECTS=obj,/RELAXED_STRUCTURE_ASSIGNMENT
resolve_obj,obj[0]
```

This works just fine, compiling the methods for the object's class. However, if I had upgraded the class definition since the file was saved, perhaps adding another tag to it, this updated version of the class would be shadowed by the version restored from file, unless the class definition was already made in the session. That is, when the object is restored, the class definition is restored along with it, and IDL sees no need to call class__define. The new definition doesn't get used.

Well obviously, we need the new version of the class to be defined *before* the object is restored from file to prevent this unwanted shadowing. To do this requires modifying the above to be:

```
resolve_obj,CLASS='class'
restore,file, RESTORED_OBJECTS=obj,/RELAXED_STRUCTURE_ASSIGNMENT
```

where resolve_obj is now changed slightly. The new resolve_obj is attached below, and differs from the previous one simply by the line:

```
call_procedure,defpro
```

instead of

```
resolve_routine,defpro
```

Basically, this compiles the class methods *and* defines the class, if it is not defined. All this is still done *only* if class__define is not compiled.

As an example, I create a class, CLASS1. I save an object instance of that class. In the meantime I update that class. Several weeks later, I start IDL and restore the saved object. If I haven't used resolve_obj yet, the *old* CLASS1 definition overrides my new definition. If I had called

resolve_obj, CLASS='class1',

it would have defined and compiled my *new* CLASS1, and the old class would be "relaxed" restored to fit in my new class definition.

One wrinkle. If for some reason, CLASS1 is already defined before all this with a conflicting structure to what is in the class1__define file, this will fail. There is yet an alternative: replacing the call_procedure with a command:

```
 null=execute('null={'+class[i]+'}')
```

which will only call class1__define if the {CLASS1} structure is not already defined. Since I hate execute statements, I've avoided this method, but the danger persists that someone does a:

dumm={class1, dumm:5}

before you get to this and it'll crash. Of course, in this danger always exists, and we must live with it, and choose unusual class names. The execute way out only saves you if someone on the command line has made a struct/class very similar and compatible with yours, which would be rare.

One other caution: save does not just save the class definition for the object you've specified and it's superclasses... it follows all pointers and data members and saves the structure definition of any thing connected to the data saved. I have come up with a method to avoid this, which has some positive side effects:

Commonly, a class will contain data, along with widget and interface type information... similar to the "Info" struct in non OOP. Not only is this unecessary information, it often changes, and should not be saved along with the object. A perfect mechnism to avoid this is to make a class member a pointer to the Info Structure:

```
st={Class1,
 data:findgen(1000)
 wInfo:ptr_new()
}

st={wInfo, base:0L, button:0L, .....}
```

And then, when saving, simply "disconnect" the pointer momentarily...

```
isav=self.wInfo
self.wInfo=ptr_new()
save,self
self.wInfo=isav
```

Now, none of the wInfo struct will be saved to disk. This technique becomes critical when you need to avoid saving member data which contain implicit links to objects. Otherwise these class definitions will be saved, and you'll have to compile *all* of the associated methods on restore or you'll have the same problems with these classes which prompted this thread!

The moral is, object saving is extremely powerful, but be warned that it can be tricky.

JD

P.S. Sorry for the longwindedness.

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

```
pro resolve_obj,obj,CLASS=class,ROUTINE_INFO=ri
  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

  if n_elements(ri) eq 0 then ri=routine_info()

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

File Attachments

1) resolve_obj.pro, downloaded 77 times