Subject: Re: IDL object copying
Posted by Wout De Nolf on Thu, 28 Jul 2011 09:20:34 GMT
View Forum Message <> Reply to Message

On Wed, 27 Jul 2011 22:47:17 -0700 (PDT), Beaker
<mattjamesfrancis@gmail.com> wrote:

> IDL seems to have an unexpected behaviour when dealing with custom
> objects.
>
> If I have an object, say obj1 and I then say:
>
> obj2 = obj1
>
> I would expect to create a copy of obj1. Any changes to members of
> obj2 I made through its methods should not change obj1, but that it
> not what I find. In fact any changes made to one changes the other.
>
> Looking at the details of the two variables we have (note that the
> class of the object is called DATETIME)
>
> OBJ1 OBJREF = <ObjHeapVar829(DATETIME)>
> OBJ2 OBJREF = <ObjHeapVar829(DATETIME)>
>
> IDL has decided that both of these are in fact pointers to the same
> heap memory, rather than being different instances of the same class,
> without their own heap memory. This is bad design; it is not the
> expected behaviour of the assignment operator, and IDL provides no
> mechanism to distinguish copy construction from assignment!
>
> Can anyone suggest a workaround? I am in the midst of re-writing an
> IDL code using custom objects (I am C++ developer by preference) and
> am starting to realise the severe limitations of IDL's object
> implementation!

You are right, IDL classes don't have copy constructors and you can't
overload the assignment operator '='. What you can do is add a "clone"
or "copy" method and call it explicitly:
http://www.idlcoyote.com/tips/copy_objects.html

Subject: Re: IDL object copying
Posted by Wout De Nolf on Thu, 28 Jul 2011 10:19:44 GMT
View Forum Message <> Reply to Message

On Wed, 27 Jul 2011 22:47:17 -0700 (PDT), Beaker
<mattjamesfrancis@gmail.com> wrote:

> Can anyone suggest a workaround? I am in the midst of re-writing an
> IDL code using custom objects (I am C++ developer by preference) and
> am starting to realise the severe limitations of IDL's object
> implementation!

Maybe you can make a base class that handles object copying and derive
all your classes from it. I didn't test it but check out 'cloneclass'
below. Suppose we derived 'myclass' from 'cloneclass', then we could
make a copy like this

```
pro myclass__define
class={myclass,...,INHERITS cloneclass}
end

pro test
o1=obj_new('myclass',...)
o2=o1->Clone()
stop,'Check objects'
end
```

It should work for all kinds of members (didn't test it) except for
objects that aren't derived from 'cloneclass' (makes a new object
without copying members and prints a warning). Here is the base class:

```
pro cloneclass::HeapCopy,p
case size(p,/type) of
8: begin ; structure
 for i=0l,n_tags(p)-1 do begin
  s=size(p.(i),/type)
  if s eq 8 or s eq 10 or s eq 11 then begin
   tmp=p.(i)
   self->HeapCopy,tmp
   p.(i)=temporary(tmp)
  endif
 endfor
 endcase
10: begin ; pointer
 ind=where(ptr_valid(p),ct)
 for i=0l,ct-1 do begin
  j=ind[i]
  p[j]=ptr_new(*p[j])
  self->HeapCopy,*p[j]
 endfor
 endcase
11: begin ; object
 ind=where(obj_valid(p),ct)
```

```
 for i=0l,ct-1 do begin
  j=ind[i]

  ind=where(obj_class(p[j],/superclass) eq
'CLONECLASS',ct)
  if ct ne 0 then begin
   p[j]=p[j]->Clone()
  endif else begin
   class=obj_class(p[j])
   p[j]=obj_new(class)
   print,'WARNING: object members not copied
',p[j]
  endelse

 endfor
 endcase
else: ; static field doesn't need copying
endcase
end;pro cloneclass::HeapCopy


pro cloneclass::AssignMember,field,value
self->HeapCopy,value
self.(field)=value
end;cloneclass::AssignMember


function cloneclass::Clone

out=obj_new(obj_class(self))
for i=0l,n_tags(self)-1 do $
 out->AssignMember,i,self.(i)

return,out
end;function cloneclass::Clone


pro cloneclass__define
 class={cloneclass,dummy:0b}
end;pro cloneclass__define
```