Subject: Important object lesson
Posted by Phillip &amp; Suzanne on Tue, 23 Jun 1998 07:00:00 GMT
View Forum Message <> Reply to Message

I've been doing some more playing around with objects, and learned that there is a HUGE difference between creating an object with a null constructor and destructor and creating an object with no constructor or destructor. I'll show a sample snippet of each, then explain.

```
----  file test1__define.pro  ----

pro test1::cleanup
end

function test1::init
   return, 1     ; success
end

pro test1__define{
   struct = {TEST1, NULL:0b}
}

----  end test1__define.pro  ----

----  file test2__define.pro  ----

pro test2__define{
   struct = {TEST2, NULL:0b}
}

----  end test1__define.pro  ----
```

When I perform obj1 = Obj_New('test1'), it takes virtually no time. obj2 = Obj_New('test2') takes about 3 1/2 seconds (on a Pentium Pro 200 running Windows NT 4.0 and IDL 5.1). Similarly, calling Obj_Destroy, obj1 takes virtually not time, but Obj_Destroy, obj2 takes about 3 1/2 seconds as well. When I actually timed these two methods, I found that there was a factor of about 200,000 times between the two. WOW! I had time to create and destroy 200,000 test1 objects for every test2 object I created.

The moral of this story is: ALWAYS define your constructors and destructors when defining IDL objects.

Phillip David
IDL Tool Developer
XonTech, Inc

Subject: Re: Important object lesson
Posted by davidf on Mon, 29 Jun 1998 07:00:00 GMT
View Forum Message <> Reply to Message

Mark Hadfield (m.hadfield@niwa.cri.nz) writes:

> Interestingly enough, this does not seem to apply to subclasses. If one
> creates a class
> MyClass that inherits from SomeOtherClass, then IDL attempts to resolve
> calls to
> obj_new('MyClass') by first searching for MyClass::Init & if it can't find
> that it settles for SomeOtherClass:Init. But as far as I can tell it does
> this once only, either at the time MyClass is first defined or (more likely)
> the first time it creates a MyClass. After that the rule "when initialising
> MyClass call SomeOtherClass::Init" seems to be lodged in its memory banks,
> with the result that if you later compile a MyClass::Init it will be
> ignored.

Yes, it must be that when the first object definition is made that
the two lifecycle methods are "registered" for the object. Probably
much the way keywords are registered for procedures and functions
when they are compiled. If you inadvertently forget to define an INIT
or CLEANUP method, and you have already made an object of that class,
then you must exit IDL and start over for those INIT and CLEANUP
methods to be recognized. This does NOT apply to other methods,
however, which you can add in the same way you add other procedures
and functions to programs. As far as I know, this behavior is not
documented anywhere.

> This explains some confusing experiences I have had with IDL objects.

Indeed. I am beginning to wonder if it is not one source
for the slowness of objects in general, apart from the
problems of just manipulating this huge 3D space.

> It also suggests a solution (though not a very elegant one): make all your
> objects subclasses of something, if only a dummy class, and make sure one of
> the superclasses has explicit Init and Cleanup methods.

I feel certain you are going to try this, Mark. Could you
fill us in on the result? :-)

Cheers,

David

--
David Fanning, Ph.D.
Fanning Software Consulting

E-Mail: davidf@dfanning.com
Phone: 970-221-0438
Coyote's Guide to IDL Programming: http://www.dfanning.com/

[A copy of this news article was also sent to the author.]

---

## Subject: Re: Important object lesson
Posted by Phillip &amp; Suzanne on Mon, 29 Jun 1998 07:00:00 GMT
View Forum Message <> Reply to Message

It appears that you're right, David. I do have a rather involved path. At
home, where I only have a simple path, the speedup is only 4,000 times instead
of 200,000 times. I don't mind that IDL follows its rules for locating the
routine anywhere along the path. However, once IDL determines that the
routine cannot be found, wouldn't it make more sense to compile a "dummy"
empty routine than to continue searching for each new object of the same type defined?

Since RSI didn't want to implement this that way, I still hold that the
solution is to ALWAYS define your INIT (constructor in C++ terminology --
sorry!) and CLEANUP (destructor) functions in IDL. This avoids the problem of
writing code for users with involved paths who don't know it.

Phillip

---

## Subject: Re: Important object lesson
Posted by Mark Hadfield on Tue, 30 Jun 1998 07:00:00 GMT
View Forum Message <> Reply to Message

David Fanning wrote in message ...
>> Much deleted
>
> Yes, it must be that when the first object definition is made that
> the two lifecycle methods are "registered" for the object. Probably
> much the way keywords are registered for procedures and functions
> when they are compiled. If you inadvertently forget to define an INIT
> or CLEANUP method, and you have already made an object of that class,
> then you must exit IDL and start over for those INIT and CLEANUP
> methods to be recognized. This does NOT apply to other methods,
> however, which you can add in the same way you add other procedures
> and functions to programs.

This is my current working hypothesis for how all this works, based on a
little experimentation, a modest amoung of logic, generous conjecture & a
minimal scanning of the documentation...

If IDL encounters

   MyClass->MyMethod

the three situations are:

 1. IDL finds a MyClass::Method in memory and uses it. (In the normal course
of events the method will have been included in the myclass__define.pro,
before the myclass__define procedure, so it will have been compiled the
first time an instance of the class was created.) If MyClass::MyMethod is
recompiled, the modifications are recognised.

2. Not finding MyClass::Method, IDL searches up the inheritance tree, finds
a ASuperClass::Method in memory and uses it for the remainder of the
session. If MyClass::MyMethod is recompiled, the modifications are not
recognised, because this method is never called. Which is confusing.

3. Failing 1 & 2, IDL searches the !path for myclass__mymethod.pro (and
maybe then for similar files for all superclasses). This can take a while.
For ordinary methods, failure to find it results in an error. Obj_new and
obj_destroy look for an Init and Cleanup respectively, but if they fail to
find them, they just skip that step--until the next time & the next time &
...etc.

> Indeed. I am beginning to wonder if it is not one source
> for the slowness of objects in general, apart from the
> problems of just manipulating this huge 3D space.

I don't think objects themselves are all that slow. Objects are just
references to named structures on the heap, which are pretty lightweight
things, much like pointers. You can have a few tens of thousands of the
things before there is any slowdown. Binding of methods to objects is much
like resolving procedures, and not all that slow, except in situations where
methods can't be found, as above.

Object Graphics are slow because they hold so much more information than
graphics windows--when you draw a 10,000-point plot to a Direct Graphics
window you end up with a bunch of pixels; when you add it to a model & a
view & a window, you still have all those points in 3D space.

>>  It also suggests a solution (though not a very elegant one): make all
your
>>  objects subclasses of something, if only a dummy class, and make sure one
of
>>  the superclasses has explicit Init and Cleanup methods.
>
> I feel certain you are going to try this, Mark. Could you
> fill us in on the result? :-)

Are you suggesting that I am a software fiddler? I'm afraid it's true. I did
think a while back about having a general class called Object with genreally
useful behaviours that all my other classes could descend from, as in Java.
Trouble is, I couldn't think of anything very useful for Object to do and it
still wouldn't be available for IDL built-in classes. Re the present
situation, it's debatable whether my suggestion is more or less trouble than
adding non-functional Inits & Cleanups to all classes.

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

## Subject: Re: Important object lesson
Posted by Mark Hadfield on Tue, 30 Jun 1998 07:00:00 GMT
View Forum Message <> Reply to Message

Sorry for all the traffic!

Please ignore my first post in this thread, the one submitted 11:24. It is
incorrect and I never meant to post it. (Didn't know I had until I saw it
there!)

The following post--the one I'm following up here, submitted 11:52--is quite
possibly incorrect too, but it does represent what I really meant to say.

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

## Subject: Re: Important object lesson
Posted by Mark Hadfield on Tue, 30 Jun 1998 07:00:00 GMT
View Forum Message <> Reply to Message

David Fanning wrote in message ...
> Phillip David (pdavid@earthling.net) writes:
>
>>  I've been doing some more playing around with objects, and learned that
there
>>  is a HUGE difference between creating an object with a null constructor
and
>>  destructor and creating an object with no constructor or destructor.
>>  ...

> ...
>
> Actually, I think this is a function of how many directories you have
> on your path. If the INIT and CLEANUP methods are not compiled when
> IDL has to use them, it looks for a file named myclass__init.pro or
> myclass__cleanup.pro in the directories specified by the !PATH
> system variable. If you have a lot of files there, it can take
> a long time!

I'm sure this is the correct explanation.

Interestingly enough, this does not seem to apply to subclasses. If one
creates a class
MyClass that inherits from SomeOtherClass, then IDL attempts to resolve
calls to
obj_new('MyClass') by first searching for MyClass::Init & if it can't find
that it settles for SomeOtherClass:Init. But as far as I can tell it does
this once only, either at the time MyClass is first defined or (more likely)
the first time it creates a MyClass. After that the rule "when initialising
MyClass call SomeOtherClass::Init" seems to be lodged in its memory banks,
with the result that if you later compile a MyClass::Init it will be
ignored.

This explains some confusing experiences I have had with IDL objects.

The reason for this is clear in the light of Phillip's experience. Some IDL
objects, eg graphics objects, pick up lots of behaviour from their parents,
and if IDL had to resolve every method all the way up the inheritance tree
every time it was called, searching the !path at every step, performance
would be affected severely.

It also suggests a solution (though not a very elegant one): make all your
objects subclasses of something, if only a dummy class, and make sure one of
the superclasses has explicit Init and Cleanup methods.

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

---

Subject: Re: Important object lesson
Posted by Mark Hadfield on Tue, 30 Jun 1998 07:00:00 GMT
View Forum Message <> Reply to Message

David Fanning wrote in message ...
> Phillip David (pdavid@earthling.net) writes:
>

>>  I've been doing some more playing around with objects, and learned that there
>>  is a HUGE difference between creating an object with a null constructor and
>>  destructor and creating an object with no constructor or destructor.
>>  ...
>  ...
>
> Actually, I think this is a function of how many directories you have
> on your path. If the INIT and CLEANUP methods are not compiled when
> IDL has to use them, it looks for a file named myclass__init.pro or
> myclass__cleanup.pro in the directories specified by the !PATH
> system variable. If you have a lot of files there, it can take
> a long time!

I imagine that this applies also to the Draw method of objects that are inherited from IDL graphics classes.

Whenever one calls

oDest->Draw, oView

(where oDest is a destination object & oView a view object) , IDL scans through the models & atoms inside the view and for each one (say oAxis) it calls

oAxis->Draw, oDest, oView.

So if oAxis is a member of a class MyAxis, inherited from IDLgrAxis, IDL presumably looks first for MyAxis::Draw, and only if it can't find it anywhere in the search path does it settle for IDLgrAxis::Draw. (I haven't verified that this is the case, but it seems to follow from the behaviour reported by Phillip for Init & Cleanup.)

So perhaps, for performance reasons, one should include an explicit Draw method in any derived graphcis class. But it isn't even documented (except for IDLgrModel). What about all the other methods that we want to inherit

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

>
>>   [Code with and without constructors and destructors snipped.]
>
>>  When I actually timed these two methods, I found that there was a factor

of
>>  about 200,000 times between the two.  WOW!  I had time to create and
destroy
>>  200,000 test1 objects for every test2 object I created.
>>
>>  The moral of this story is:  ALWAYS define your constructors and
destructors
>>  when defining IDL objects.
>
> Cheers,
>
> David
> ___
> David Fanning, Ph.D.
> Fanning Software Consulting
> E-Mail: davidf@dfanning.com
> Phone: 970-221-0438
> Coyote's Guide to IDL Programming: http://www.dfanning.com/

---

## Subject: Re: Important object lesson
## Posted by mirko_vukovic on Thu, 02 Jul 1998 07:00:00 GMT

View Forum Message <> Reply to Message

In article <6nabkb$p5j$1@clam.niwa.cri.nz>,
  "Mark Hadfield" <m.hadfield@niwa.cri.nz> wrote:
>
much deleted
>>>  It also suggests a solution (though not a very elegant one): make all
>  your
>>>  objects subclasses of something, if only a dummy class, and make sure one
>  of
>>>  the superclasses has explicit Init and Cleanup methods.

I would vote against it.  This may only confuse your code and invite errors
further down the line.  Faster CPU's are allways around the corner.

>
>  Are you suggesting that I am a software fiddler? I'm afraid it's true. I did
>  think a while back about having a general class called Object with genreally
>  useful behaviours that all my other classes could descend from, as in Java.
>  Trouble is, I couldn't think of anything very useful for Object to do and it
>  still wouldn't be available for IDL built-in classes. Re the present
Actually I do have a class OBJ which almost any object inherits.  It has three
main methods:

pro obj::debug -- just stops and allows the user to examine the object
pro obj::property -- sets a property (but must be completely typed)

function obj::property -- retrieves a property (same restriction as above)

It in addition has a usefull field like self.version, and I hope to add in
a generic ::read and ::write methods for file access (but I have not thought
that one through completely).

oh, and btw, I've generated up to 0.5MO (mega objects) and IDL was performing
fine.  These were instances of incorrect cleanups, but I was still glad
nothing got corrupted.

Mirko