Subject: Re: Why IDL needs Garbage Collection
Posted by David Fanning on Wed, 20 Jul 2005 22:07:03 GMT
View Forum Message <> Reply to Message

JD Smith writes:

> This is why RSI needs to implement a simple but effective garbage
> collection paradigm in the next version of IDL. Anyone agree?

There is no question this becomes a problem in even
moderately sized object applications. We have written
a very simple reference counting mechanism in Catalyst
that works well, but I agree the memory management would
be a nightmare without it.

But if we are already starting our annual Top Ten List,
I am becoming convinced that no one will be writing large
object applications until it is possible to inherit from
multiple objects without running into structure field
name restrictions. It is possible to work around it, but
only by duplicating code and stepping into that same
code maintenance quagmire object programming is suppose
to get you out of.

Cheers,

David
--
David Fanning, Ph.D.
Fanning Software Consulting, Inc.
Coyote's Guide to IDL Programming: http://www.dfanning.com/

Subject: Re: Why IDL needs Garbage Collection
Posted by Antonio Santiago on Thu, 21 Jul 2005 06:29:10 GMT
View Forum Message <> Reply to Message

I was thinking on the same problem a couple of days ago
(http://www.grahi.upc.edu/santiago/?p=149).

The implementation of IDL objects is pretty similary to the GObject
system.
GObject is a library written in C (as part of GTK+ project) that realizes
some object oriented compiler functions.

Both have an 'initializer' and a 'finalizer' object methods,
althought GObject has an 'initializer/finalizer' for the entery class.
Both use structure inheritance and only gives simple inheritance and

method overriding, but in GObject every time you create/destroy an object
the "system" controls the refcount of the object and only destryis it when
is 0.

Another point that surprise me, as C programmer, is the use of
"conventional memory" and "heap memory":
p=PTR_NEW('hello')

'p' is in "conventional memory" an is controlled by GC and 'hello' is in
heap memory and is my responsability free it.

a=10
p=PTR_NEW(a)

a and p are in "conventional memory" but 'p' points to a copy of 'a' at
heap memory. What?? I want a reference to the real 'a' !!!
Once you are familiarized with this it is no problem but I dont understant
the utility of this. I suposse it is for problems in object oriented
implementation and the pre-implemented "conventional memory".

Bye.

On Wed, 20 Jul 2005 14:24:43 -0700, JD Smith wrote:

>
> IDL pointers are great. We all use them to tuck things inside of
> structures, or pass around heavyweight data without penalty. IDL
> objects are great too, encapsulating data and functionality, enabling
> reasonably hassle-free GUI programming, and more. What is not great
> is the inflexibility that IDL's manual resource management imposes.
> Sure, object's have their Cleanup method, and that can be used to
> effectively free the object's heap data when the object is explicitly
> destroyed. Very useful. But, and this is the catch, that requires
> someone or something to continuously keep track of that object, and
> free it at the right time. Consider the simple case:
>
> IDL> a=obj_new('Foo')
> IDL> a=obj_new('Bar')
>
> Well, that's a memory leak right there. No one know about the 'Foo'
> object anymore. This is easy enough to avoid, but now imagine a
> system for passing around many many pointers and objects. For a
> concrete example, let's imagine a pointer pointing to a big pile of
> data called BOB. To keep from using too much memory, you don't want
> to replicate BOB in every corner of a set of applications that need to
> use it, so you allow different routines to share the BOB pointer.

> Fine. Well, what happens when a new BOB pointer gets sent in to
> occupy the same slot? Whose job is it to free the original BOB? How
> do you know someone else isn't still making use of the data being
> pointed to?
>
> Because of these types of issues, I find myself passing around lots of
> back-channel information like "make sure to free this pointer when you
> are done, but not this one, because I'll still be using that here,
> probably". Ugly. You can of course invent your own form of garbage
> collection (e.g. reference counting), but why shouldn't IDL, which
> clearly can keep track of heap data which is no longer being pointed
> to (vz. HEAP_GC,/VERBOSE), do the dirty work for you? Then, whether a
> pointer or object is shared across 10 different programs for the
> duration of an IDL session, or simply created, used once, and then
> discarded, you wouldn't need any additional logic to decide if and
> when to free a given resource. And no, I don't consider putting
> HEAP_GC in your event callback effective garbage collection.
>
> This is why RSI needs to implement a simple but effective garbage
> collection paradigm in the next version of IDL. Anyone agree?
>
> JD

--

Antonio Santiago Pï¿½rez
(email: santiago<<at>>grahi.upc.edu)
(www: http://www.grahi.upc.edu/santiago)
(www: http://asantiago.blogsite.org)

GRAHI - Grup de Recerca Aplicada en Hidrometeorologia
Universitat Politï¿½cnica de Catalunya

Subject: Re: Why IDL needs Garbage Collection
Posted by JD Smith on Thu, 21 Jul 2005 17:05:24 GMT
View Forum Message <> Reply to Message

Creating a reference counting super-class for all objects to inherit
from isn't hard, and I suspect this is what David has done in
Catalyst. The real problem is pointers, which have no such semantics.

One obvious option would be to make a composite pointer type which is
actually an object, let's call it oPtr, which does nothing other than
deliver the value of the contained pointer, and keep a reference count
for it. A few problems with this:

1. There still is no automatic way to increment the reference count; programs would have to do this themselves whenever they store a copy of the oPtr object. If a program forgets, then the reference count is wrong and the pointer may be freed at the wrong time. Compare this to a system in which the program doesn't even have to think about reference counts, but instead just overwrite its pointers with impunity.

2. Programs still have to explicitly free all oPtr's when they are through with them (which doesn't always correspond to when they are through running). They can free things with impunity without worrying about stepping on someone else's toes, but they still have to actually free it. I.e., this type of system gaurds allows run-time flexibility, but not accommodate simple coding mistakes. Also, OBJ_DESTROY can't be used (since by the time the Cleanup code is called, it's too late). You'd have to introduce something like oPtr->Destroy (and make sure everyone uses that).

3. It's fairly heavy weight, and introduces an awkward syntax. Instead of *self.myptr, you'd need self.myoPtr->GetValue(). This creates a copy of the heap data, whereas *self.myptr simply references in the in-memory copy of the heap data. For small things, no big deal, for big things, potentially a very big deal.

So while this may appear to be a partial solution, I think coding garbage collection at a much lower level (essentially optimizing HEAP_GC, and having it run automatically) would be far superior. There are a wide variety of interesting GC technologies to choose among, from simple mark-and-sweep, reference counting, etc. to parallel algorithms which don't requiring stopping everything to do their work. Since IDL is now multi-threaded, it should be a manageable operation.

JD

Subject: Re: Why IDL needs Garbage Collection
Posted by Dick Jackson on Thu, 21 Jul 2005 18:40:36 GMT
View Forum Message <> Reply to Message

"JD Smith" <jdsmith@as.arizona.edu> wrote in message news:pan.2005.07.20.21.24.42.658685@as.arizona.edu...
>
> IDL pointers are great. [...]
>
> This is why RSI needs to implement a simple but effective garbage
> collection paradigm in the next version of IDL. Anyone agree?

(Just to clarify, the issue here is "Why IDL needs *Automatic* Garbage
Collection", as we do have manual Heap_GC as you point out)

Absolutely. I did a lot of work using LISP some years ago, and I recall how
often I could breeze through some programming tasks without having to worry
about the cleanup. It makes for more carefree (not careless!) work.

I'm with you on this, JD.

For anyone who wants to read a nice article on the issue:
http://en.wikipedia.org/wiki/Garbage_collection_%28computer_ science%29

Cheers,
--
-Dick

Dick Jackson / dick@d-jackson.com
D-Jackson Software Consulting / http://www.d-jackson.com
Calgary, Alberta, Canada / +1-403-242-7398 / Fax: 241-7392

Subject: Re: Why IDL needs Garbage Collection
Posted by marc schellens[1] on Mon, 25 Jul 2005 10:35:36 GMT
View Forum Message <> Reply to Message

Full acknowledge.
While in some languages such as C++ performance is
crucial and so manual cleanup is the proper choice here
(and is even much better supported due to the cleanup
of automatic variables), for a language like IDL automatic GC is the
natural thing to provide.
At least with reference counting and copy on write I guess
performance of most programs could be improved.
marc

Subject: Re: Why IDL needs Garbage Collection
Posted by Michael Wallace on Mon, 25 Jul 2005 14:15:14 GMT
View Forum Message <> Reply to Message

> This is why RSI needs to implement a simple but effective garbage
> collection paradigm in the next version of IDL. Anyone agree?

Yes. Wholeheartedly.

-Mike