Subject: Re: Object Data and pointer assignments
Posted by John-David T. Smith on Thu, 09 Mar 2000 08:00:00 GMT
View Forum Message <> Reply to Message

David Fanning wrote:
>
> J.D. Smith (jdsmith@astro.cornell.edu) writes:
>
>> Just to be clear... you are free to free self.inarray, and point it somewhere
>> else, at any time. This can be useful if you have a list which is either empty
>> (NULL pointer a.k.a. a dangling reference), or not (pointer to a list of finite
>> size). If the list changes size, and becomes empty again, you can simply free
>> it, which indicates its emptiness. If it then grows again, simply use ptr_new()
>> to get another heap variable for it. So, while it might be easiest in some
>> cases only to call ptr_new() once, in other cases it is useful to let a single
>> member variable like self.inarray point to different heap variables over its
>> life.
>
> Lord knows I need more excitement in my life if I'm quibbling with
> quibbles, but let me make one suggestion:
>
> If I want to point to an "empty" variable, I prefer to
> use a pointer to an undefined variable. The advantage
> to me is that this is a VALID pointer, in contrast
> to the NULL pointer, which is an invalid pointer.
>
> Note:
>
> IDL> a = Ptr_New()
> IDL> Print, Ptr_Valid(a)
> 0
> IDL> *a = 5
> % Unable to dereference NULL pointer: A.
>
> IDL> b = Ptr_New(/Allocate_Heap)
> IDL> Print, Ptr_Valid(b)
> 1
> IDL> *b = 5
>
> I like this because it fits into the programming style
> I've developed. For example:
>
> IF N_Elements(color) EQ 0 THEN color = 5
> IF N_Elements(*b) EQ 0 THEN *b = 5
>
> But again, you must *initialize* this pointer to an
> undefined variable in the INIT method, NOT in the __DEFINE
> module.

>

That's a nice idea. I hadn't thought of doing it that way. In my method, the validity of the pointer is what indicates an empty vs. non-empty list. In your method, whether the variable pointed to by the pointer is defined provides the same distinction. With your method, you save yourself tests like:

 if ptr_valid(ptr) n_elem=0 else n_elem=n_elements(ptr)

(of which I have *many*) in favor of:

 n_elem=n_elements(*ptr)

This is very clean. To pay for that, though, each time your list (or whatever) reaches 0 size, you must do a:

ptr_free,ptr
ptr=ptr_new(/ALLOC)

the latter line not being required in my method (a consequence of the indistinguishability of null pointers and dangling pointers). I think this trade is well worth it, though, and I will consider using your method in the future.

Thanks for the tip!

JD

--
 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 |*|

Subject: Re: Object Data and pointer assignments
Posted by John-David T. Smith on Thu, 09 Mar 2000 08:00:00 GMT
View Forum Message <> Reply to Message

Ben Tupper wrote:
>
> "J.D. Smith" wrote:
>
>>
>> The only reason I quibble is to dispel the notion that the __define in class
>> definitions merely suggests or defines the class data members, as it does for
>> fields of structures, and that you must "fill out" the skeleton of class data in
>> the Init method. In the context of object creation, obj_new *does* in fact

>> implicitly assign values to them all: namely null (0,'',null pointer, etc.).
>> You can think of the first step of obj_new being something like
>> self={MY_CLASS}. So by the time you get to the Init Method, you do have a
>> *real* pointer, namely a null pointer.
>>
>>
>
> Hello,
>
> This doesn't sound like a quibble to a newbie. So, the filling out of the
> skeleton may or may not occur in the INIT function, but not in the __DEFINE
> procedure.
>
> You have brought up the the issue of SELF ... which is another source of
> confusion. I have just written the SetProperty and GetProperty methods. They
> work just fine (so far I can get and set what I need). However, it feels a little
> bit like living in Flatland where things pop in and out of my two dimensions from
> some unkown third dimension. Magically, SELF appears as if out of thin air; how
> does it get there if there is no SELF argument in the procedure... and how come I
> don't have to call the structure BLAH (rather than self) if it is a named
> structure?

Think of self as an invisible final argument, passed by reference to every single method of the class. This is most certainly how RSI even implemented it. All object oriented languages have some similar way to access easily the "member data" of an object, whether it be called "self", "this", "here", etc. So, it's not too much magic, just a little bit of argument hiding. Just as a regular variable can hold a named structure... i.e. a={THIS_STRUCT,data1:1}, the specially named variable "self", which implicitly exists in all methods, can hold a named class "structure" (really it's a full fledged object... the distinction being that I couldn't do a->Print or some such on the above definition). The "self" is "a good thing", and provides part of the object oriented solution.

Heavy duty magic makes use of the fact that self is more or less just a passed in, by reference variable, which allows you to switch, in place, the self object (just as you could overwrite any other variable passed in by reference). For instance, in one of my applications, a simple "restore from disk" menu option running *within a method* on a given object performs such a voodoo transmogrification to undo all changes made since the last save without tediously setting all of the (many) data members. So my advice: have fun with your self.

JD

--
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 |*|

Subject: Re: Object Data and pointer assignments
Posted by davidf on Thu, 09 Mar 2000 08:00:00 GMT
View Forum Message <> Reply to Message

J.D. Smith (jdsmith@astro.cornell.edu) writes:

> Just to be clear... you are free to free self.inarray, and point it somewhere
> else, at any time. This can be useful if you have a list which is either empty
> (NULL pointer a.k.a. a dangling reference), or not (pointer to a list of finite
> size). If the list changes size, and becomes empty again, you can simply free
> it, which indicates its emptiness. If it then grows again, simply use ptr_new()
> to get another heap variable for it. So, while it might be easiest in some
> cases only to call ptr_new() once, in other cases it is useful to let a single
> member variable like self.inarray point to different heap variables over its
> life.

Lord knows I need more excitement in my life if I'm quibbling with
quibbles, but let me make one suggestion:

If I want to point to an "empty" variable, I prefer to
use a pointer to an undefined variable. The advantage
to me is that this is a VALID pointer, in contrast
to the NULL pointer, which is an invalid pointer.

Note:

```
IDL> a = Ptr_New()
IDL> Print, Ptr_Valid(a)
      0
IDL> *a = 5
   % Unable to dereference NULL pointer: A.

IDL> b = Ptr_New(/Allocate_Heap)
IDL> Print, Ptr_Valid(b)
      1
IDL> *b = 5
```

I like this because it fits into the programming style
I've developed. For example:

```
IF N_Elements(color) EQ 0 THEN color = 5
IF N_Elements(*b) EQ 0 THEN *b = 5
```

But again, you must *initialize* this pointer to an

undefined variable in the INIT method, NOT in the __DEFINE
module.

Cheers,

David

--
David Fanning, Ph.D.
Fanning Software Consulting
Phone: 970-221-0438 E-Mail: davidf@dfanning.com
Coyote's Guide to IDL Programming: http://www.dfanning.com/
Toll-Free IDL Book Orders: 1-888-461-0155

Subject: Re: Object Data and pointer assignments
Posted by Ben Tupper on Thu, 09 Mar 2000 08:00:00 GMT
View Forum Message <> Reply to Message

"J.D. Smith" wrote:

>
> The only reason I quibble is to dispel the notion that the __define in class
> definitions merely suggests or defines the class data members, as it does for
> fields of structures, and that you must "fill out" the skeleton of class data in
> the Init method. In the context of object creation, obj_new *does* in fact
> implicitly assign values to them all: namely null (0,",null pointer, etc.).
> You can think of the first step of obj_new being something like
> self={MY_CLASS}. So by the time you get to the Init Method, you do have a
> *real* pointer, namely a null pointer.
>
>

Hello,

This doesn't sound like a quibble to a newbie. So, the filling out of the skeleton may or may not occur in the INIT function, but not in the __DEFINE procedure.

You have brought up the the issue of SELF ... which is another source of confusion. I have just written the SetProperty and GetProperty methods. They work just fine (so far I can get and set what I need). However, it feels a little bit like living in Flatland where things pop in and out of my two dimensions from some unkown third dimension. Magically, SELF appears as if out of thin air; how does it get there if there is no SELF argument in the procedure... and how come I don't have to call the structure BLAH (rather than self) if it is a named structure?

Thanks again,

Ben

--
Ben Tupper

Bigelow Laboratory for Ocean Science
tupper@seadas.bigelow.org

pemaquidriver@tidewater.net

Subject: Re: Object Data and pointer assignments
Posted by John-David T. Smith on Thu, 09 Mar 2000 08:00:00 GMT
View Forum Message <> Reply to Message

Ben Tupper wrote:
>
> David Fanning wrote:
>
>> You don't leak any memory because IDL is managing this
>> whole process for you. (Remember, these pointers are
>> not real pointers in the C sense. They are really
>> glorified variables in the IDL sense.) This is the
>> bestest feature of IDL pointers. :-)
>>
>
> Thanks for the tips. It's probably a good thing that I don't know much about
> C (no bad habits, eh?)
>
>>
>> If you overwrite the pointer like this:
>>
>> self.InArray = Ptr_New(newStruct)
>>
>> you *will* leak memory because now you destroyed the
>> only reference to that pointer area of memory. You could
>> do this:
>>
>>
>
> So, if I am following your instruction correctly, I should only see ...
>
> self.InArray = Ptr_New(newStruct)

>
> once in my code in the INIT function. Thereafter (in SetProperty for
> example) it is simply derefence....
>
> *self.inarray = newStruct
>

Just to be clear... you are free to free self.inarray, and point it somewhere else, at any time. This can be useful if you have a list which is either empty (NULL pointer a.k.a. a dangling reference), or not (pointer to a list of finite size). If the list changes size, and becomes empty again, you can simply free it, which indicates its emptiness. If it then grows again, simply use ptr_new() to get another heap variable for it. So, while it might be easiest in some cases only to call ptr_new() once, in other cases it is useful to let a single member variable like self.inarray point to different heap variables over its life.

Good Luck,

JD

--
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 |*|

Subject: Re: Object Data and pointer assignments
Posted by John-David T. Smith on Thu, 09 Mar 2000 08:00:00 GMT
View Forum Message <> Reply to Message

David Fanning wrote:
>
> Ben Tupper (tupper@seadas.bigelow.org) writes:
>
>> I am in the middle of wrtting my first object from scratch. Scratch is
>> a good word since I'm doing a lot of that on my head. I'm hoping to get
>> some advice on organization of data. I need 4 pieces of data (one 2d
>> arrays and two structures that vary in size according to the size of the
>> arrays) plus six keywords that I need to get/set. Currently, I have
>> defined each of the 3 bits of data as null pointers in the BLAH__DEFINE
>> procedure.
>>
>> In the BLAH::INIT function, the user passes one of the two arrays as an
>> argument. At that point I reassign one of the pointers to...
>>

>>   Self.InArray = Ptr_New(InArray).
>>
>> I think I understand why I can reassign the structure field when going
>> from a null pointer to a filled pointer. On second thought, I don't
>> understand it but I can accept that it works. It's the next step I need
>> help on.
>
> The reason you need to use an actual pointer (Ptr_New) here,
> is that you *don't* have a pointer from the BLAH__DEFINE
> module. What you have done in that module is said that the
> *definition* of the InArray field *will be* a pointer. In other
> words, the BLAH__DEFINE module only *defines* the object and
> its fields, it doesn't assign anything to the self object. This
> is what must be done by the INIT method.

This is true. But keep in mind that all structure defines (e.g.
a={BLAH_STRUCT}) zero the structure members upon creation. In the case of
arrays, it fills them with zeroes. In the case of strings, it makes them zero
length. In the case of pointers and objects, it makes them null
pointers/objects. These *are* real pointers, but ones which cannot be
dereferenced, as they point nowhere! They are "dangling references" at birth.
Note that you can say:

IDL> a=ptr_new(1)
IDL> b=a
IDL> ptr_free(a)

and b will be, for all practical purposes, a NULL pointer! It points to a heap
variable which does not exist. It has the same ptr_valid() properties, etc.
Only printing it can reveal the difference.

The only reason I quibble is to dispel the notion that the __define in class
definitions merely suggests or defines the class data members, as it does for
fields of structures, and that you must "fill out" the skeleton of class data in
the Init method. In the context of object creation, obj_new *does* in fact
implicitly assign values to them all: namely null (0,'',null pointer, etc.).
You can think of the first step of obj_new being something like
self={MY_CLASS}. So by the time you get to the Init Method, you do have a
real pointer, namely a null pointer.

The moral is: if null class data is what you want, you can skip the Init (though
of course there are other advantages to Init'ing), or skip irrelevant
assignments in Init (e.g. self.var=0).

One curious by-product of structure/class definition/instantiation peculiarities
is shown in the following example:

pro foo__define

```
   foo={FOO, $
        ptr: ptr_new(fltarr(100))}
end


IDL>  a={FOO}
% Compiled module: FOO__DEFINE.
IDL> print,a
{<NullPointer>}
IDL> help,/heap
Heap Variables:
   # Pointer: 1
   # Object : 0

<PtrHeapVar1>   FLOAT     = Array[100]
```

An IDL-provided memory leak! Yay. The moral of that is, *never* use anything but the bare ptr_new() and obj_new() inside of your __defines.

JD

--
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 |*|

Subject: Re: Object Data and pointer assignments
Posted by Ben Tupper on Thu, 09 Mar 2000 08:00:00 GMT
View Forum Message <> Reply to Message

David Fanning wrote:

> You don't leak any memory because IDL is managing this
> whole process for you. (Remember, these pointers are
> not real pointers in the C sense. They are really
> glorified variables in the IDL sense.) This is the
> bestest feature of IDL pointers. :-)
>

Thanks for the tips. It's probably a good thing that I don't know much about C (no bad habits, eh?)

>
> If you overwrite the pointer like this:

>
>   self.InArray = Ptr_New(newStruct)
>
> you *will* leak memory because now you destroyed the
> only reference to that pointer area of memory. You could
> do this:
>
>

So, if I am following your instruction correctly, I should only see ...

    self.InArray = Ptr_New(newStruct)

once in my code in the INIT function. Thereafter (in SetProperty for example) it is simply derefence....

   *self.inarray = newStruct

Thanks again,

Ben

--
Ben Tupper

Bigelow Laboratory for Ocean Science
tupper@seadas.bigelow.org

pemaquidriver@tidewater.net

Subject: Re: Object Data and pointer assignments
Posted by John-David T. Smith on Thu, 09 Mar 2000 08:00:00 GMT
View Forum Message <> Reply to Message

Ben Tupper wrote:
>
> Hello,
>
> I am in the middle of wrtting my first object from scratch. Scratch is
> a good word since I'm doing a lot of that on my head. I'm hoping to get
> some advice on organization of data. I need 4 pieces of data (one 2d
> arrays and two structures that vary in size according to the size of the
> arrays) plus six keywords that I need to get/set.   Currently, I have
> defined each of the 3 bits of data as null pointers in the BLAH__DEFINE
> procedure.

>
> In the BLAH::INIT function, the user passes one of the two arrays as an
> argument. At that point I reassign one of the pointers to...
>
> Self.InArray = Ptr_New(InArray).
>
> I think I understand why I can reassign the structure field when going
> from a null pointer to a filled pointer. On second thought, I don't
> understand it but I can accept that it works. It's the next step I need
> help on.

A pointer is a pointer is a pointer, whether to nothing (a NULL pointer), or to an array of 1 million images. Think of a pointer as just some number, like <1185>, and it won't be as confusiong. In your case the structure field just contains one of these special "numbers". In a machine-level language, the number would be a hardware address. In IDL, the number is some arbitrary lookup in an internal table of pointer heap data.

>
> I would like to change the contents of this field later to some other
> value (a differently sized array.) Here's where the ice under me gets
> very very thin and my eyes get misty. In the BLAH::SETPROPERTY method,
> I don't know if I should free this pointer before reassigning (and does
> that leave the structure field undefined?), or if I should simply
> overwrite it as I did in the INIT function. If I reassign the filed
> to a new pointer, what happens to the previously occupied heap space?
> Have I sprung a leak?

Two possibilities:

*ptr=newarr

or

ptr_free,ptr
ptr=ptr_new(newarr) ; with optional /NO_COPY keyword -- faster but makes newarr undefined

In the first case, think of it just like reassigning a regular variable. If you say:

IDL> a=[1,2,3]
IDL> a=[4,5,6,7,8,9]

you don't worry that the space allocated for the [1,2,3] is lost... no memory leak occurs here. IDL makes sure of that (or they are supposed to!). Likewise, when changing the value of the pointer heap data (which in essence is just a specially accessed and globally persistent form of a regular variable like the

"a" above), the exact same rules apply.

The only way you will have memory leaks is if you reassign your ptr elsewhere without freeing the heap data. E.g.

```
IDL> a=ptr_new(findgen(100,100))
IDL> help,/heap
Heap Variables:
   # Pointer: 1
   # Object : 0

<PtrHeapVar1>   FLOAT     = Array[100, 100]
IDL> a=ptr_new(5)
IDL> help,/heap
Heap Variables:
   # Pointer: 2
   # Object : 0

<PtrHeapVar1>   FLOAT     = Array[100, 100]
<PtrHeapVar2>   INT       =        5
IDL> help
% At  $MAIN$
A               POINTER   = <PtrHeapVar2>
```

You see we have two "heap" variables, PtrHeapVar1 and 2, and and only variable ("a") pointing to #2. The data in PtrHeapVar1 is not referenced by anyone at all. What can be done about this?

1. The first and best answer is: don't let it happen in the first place. Careful programming can avoid all leaks like this.

2. For debugging purposes, if you happen to "lose" some data you had pointed to, you can recover it, i.e. reconnect a pointer to it. In the above example, you would say:

```
IDL> b=ptr_valid(/CAST,1)
IDL> help
% At  $MAIN$
A               POINTER   = <PtrHeapVar2>
B               POINTER   = <PtrHeapVar1>
```

so now PtrHeapVar1 has a pointer to it after all. The memory leak has been averted, and more importantly, you now have access to that possibly important data. Use this method only in emergencies, and for debugging code which develops leaks (see 1.).

3. When desperate to stem the flood of leaking memory, you can use heap_gc, which looks for all heap variables without one or more pointers pointing to

them, and free's them. This is not recommended for anything other than debugging purposes, except as a quick fix when something has to be done yesterday.

It's not terribly hard to write leak free programs. Just keep these few things in mind.

Good Luck,

JD

--
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 |*|

Subject: Re: Object Data and pointer assignments
Posted by davidf on Thu, 09 Mar 2000 08:00:00 GMT
View Forum Message <> Reply to Message

Ben Tupper (tupper@seadas.bigelow.org) writes:

> I am in the middle of wrtting my first object from scratch. Scratch is
> a good word since I'm doing a lot of that on my head. I'm hoping to get
> some advice on organization of data. I need 4 pieces of data (one 2d
> arrays and two structures that vary in size according to the size of the
> arrays) plus six keywords that I need to get/set. Currently, I have
> defined each of the 3 bits of data as null pointers in the BLAH__DEFINE
> procedure.
>
> In the BLAH::INIT function, the user passes one of the two arrays as an
> argument. At that point I reassign one of the pointers to...
>
>   Self.InArray = Ptr_New(InArray).
>
> I think I understand why I can reassign the structure field when going
> from a null pointer to a filled pointer. On second thought, I don't
> understand it but I can accept that it works. It's the next step I need
> help on.

The reason you need to use an actual pointer (Ptr_New) here,
is that you *don't* have a pointer from the BLAH__DEFINE
module. What you have done in that module is said that the
definition of the InArray field *will be* a pointer. In other
words, the BLAH__DEFINE module only *defines* the object and

its fields, it doesn't assign anything to the self object. This
is what must be done by the INIT method.

> I would like to change the contents of this field later to some other
> value (a differently sized array.) Here's where the ice under me gets
> very very thin and my eyes get misty. In the BLAH::SETPROPERTY method,
> I don't know if I should free this pointer before reassigning (and does
> that leave the structure field undefined?), or if I should simply
> overwrite it as I did in the INIT function. If I reassign the filed
> to a new pointer, what happens to the previously occupied heap space?
> Have I sprung a leak?

To reassign the pointer to something else (after it has been
defined by the INIT method), you simple de-reference the pointer:

```
*self.InArray = newStruct
```

You don't leak any memory because IDL is managing this
whole process for you. (Remember, these pointers are
not real pointers in the C sense. They are really
glorified variables in the IDL sense.) This is the
bestest feature of IDL pointers. :-)

If you overwrite the pointer like this:

```
self.InArray = Ptr_New(newStruct)
```

you *will* leak memory because now you destroyed the
only reference to that pointer area of memory. You could
do this:

```
Ptr_Free, self.InArray
self.InArray = Ptr_New(newStruct)
```

But what is the point, if IDL can do it all for you?

Cheers,

David

--
David Fanning, Ph.D.
Fanning Software Consulting
Phone: 970-221-0438 E-Mail: davidf@dfanning.com
Coyote's Guide to IDL Programming: http://www.dfanning.com/
Toll-Free IDL Book Orders: 1-888-461-0155

Subject: Re: Object Data and pointer assignments
Posted by John-David T. Smith on Fri, 10 Mar 2000 08:00:00 GMT
View Forum Message <> Reply to Message

"J.D. Smith" wrote:
>
> David Fanning wrote:
>>
>> J.D. Smith (jdsmith@astro.cornell.edu) writes:
>>
>>> Just to be clear... you are free to free self.inarray, and point it somewhere
>>> else, at any time. This can be useful if you have a list which is either empty
>>> (NULL pointer a.k.a. a dangling reference), or not (pointer to a list of finite
>>> size). If the list changes size, and becomes empty again, you can simply free
>>> it, which indicates its emptiness. If it then grows again, simply use ptr_new()
>>> to get another heap variable for it. So, while it might be easiest in some
>>> cases only to call ptr_new() once, in other cases it is useful to let a single
>>> member variable like self.inarray point to different heap variables over its
>>> life.
>>
>> Lord knows I need more excitement in my life if I'm quibbling with
>> quibbles, but let me make one suggestion:
>>
>> If I want to point to an "empty" variable, I prefer to
>> use a pointer to an undefined variable. The advantage
>> to me is that this is a VALID pointer, in contrast
>> to the NULL pointer, which is an invalid pointer.
>>
>> Note:
>>
>> IDL> a = Ptr_New()
>> IDL> Print, Ptr_Valid(a)
>> 0
>> IDL> *a = 5
>> % Unable to dereference NULL pointer: A.
>>
>> IDL> b = Ptr_New(/Allocate_Heap)
>> IDL> Print, Ptr_Valid(b)
>> 1
>> IDL> *b = 5
>>
>> I like this because it fits into the programming style
>> I've developed. For example:
>>
>> IF N_Elements(color) EQ 0 THEN color = 5
>> IF N_Elements(*b) EQ 0 THEN *b = 5
>>
>> But again, you must *initialize* this pointer to an
>> undefined variable in the INIT method, NOT in the __DEFINE

>> module.
>>
>
> That's a nice idea. I hadn't thought of doing it that way. In my method, the
> validity of the pointer is what indicates an empty vs. non-empty list. In your
> method, whether the variable pointed to by the pointer is defined provides the
> same distinction. With your method, you save yourself tests like:
>
> if ptr_valid(ptr) n_elem=0 else n_elem=n_elements(ptr)

meant:
if ptr_valid(ptr) n_elem=0 else n_elem=n_elements(*ptr)

of course.

>
> (of which I have *many*) in favor of:
>
> n_elem=n_elements(*ptr)
>
> This is very clean. To pay for that, though, each time your list (or whatever)
> reaches 0 size, you must do a:
>
> ptr_free,ptr
> ptr=ptr_new(/ALLOC)
>
> the latter line not being required in my method (a consequence of the
> indistinguishability of null pointers and dangling pointers). I think this
> trade is well worth it, though, and I will consider using your method in the
> future.

One nice feature of my method is the ability to "zero" many lists or data constructions quite simply. E.g. suppose I had a pointer "l" to a list of pointers, each to a list, along with a few other lists. To zero out all of those lists, I can simply say:

ptr_free,*l,l1,s.l2,...

whereas in your method, I'd have to say:

```
ptr_free,*l
for i=0,n_elements(l)-1 do *l[i]=ptr_new(/ALLOC)
l1=ptr_new(/ALLOC)
s.l2=ptr_new(/ALLOC)
...
```

which could introduce more room for errors. I'll let you know how I fare with your technique.

JD

--
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 |*|

Subject: Re: Object Data and pointer assignments
Posted by Ben Tupper on Fri, 10 Mar 2000 08:00:00 GMT
View Forum Message <> Reply to Message

"J.D. Smith" wrote:

>
> Think of self as an invisible final argument, passed by reference to every
> single method of the class.

I can do that.

> So my advice: have fun with your self.
>

I can do that, too.

Thanks for all the help.

Ben

--
Ben Tupper

Bigelow Laboratory for Ocean Science
tupper@seadas.bigelow.org

pemaquidriver@tidewater.net
