Subject: Re: Pointers in IDL
Posted by JD Smith on Wed, 14 Apr 2004 01:08:01 GMT
View Forum Message <> Reply to Message

On Tue, 13 Apr 2004 11:29:45 -0400, Benjamin Hornberger wrote:

> Hi all,
>
> I still don't understand all aspects of pointers in IDL. 2 Questions:
>
> 1. What are null pointers for? I read that they can't be dereferenced. What
> is their purpose then? The Gumley book writes (pg. 61): "Null pointers are
> used when a pointer must be created, but the variable ... does not yet
> exist." What would I do then when the variable does exist later and I want
> the pointer to point to it? Wouldn't I use ptr_new(/allocate_heap) in the
> first place, i.e. not create a null pointer but a pointer to an undefined
> variable? Can anyone give an example when I would use ptr_new()?
>
> 2. If I point a pointer to a variable (e.g. *ptr=indgen(100)) and later
> point it to a smaller variable (*ptr=indgen(50)), do I have a memory leak?
> I.e., do I have to free it before I re-reference it?
>
> I want to write a GUI which can open files which contain arrays of varying
> size. Is it ok to define a pointer in the GUI to hold these arrays
> (ptr=ptr_new(/allocate_heap)), and then whenever I open a new file, just
> dereference to the new array (*ptr=array)? Or do I have to free the pointer
> when I close one file and open another one?

Here is the absolute best way to think of pointers in IDL (and it has the advantage that it's actually the real way they are handled internally, I believe). A pointer is *nothing* more than a specially accessed, but otherwise regular-old variable. Anything you can do with a variable, you can do with a de-referenced pointer. Actions like:

```
IDL> *ptr=indgen(100)
IDL> *ptr=indgen(5)
```

are just as allowed as if you had used a regular variable:

```
IDL> var=indgen(100)
IDL> var=indgen(5)
```

Another particular application of the "a deferenced-pointer is really just a variable" rule which many may not know.... you can pass them by reference! Suppose you have a function set_to_pi which sets its argument to PI:

```
pro set_to_pi,arg
  arg=!PI
end
```

You won't be surprised when:

```
IDL> set_to_pi,a
IDL> print,a
      3.14159
```

but would you believe:

```
IDL> b=ptr_new(/ALLOCATE_HEAP)
IDL> set_to_pi,*b
IDL> print,*b
      3.14159
```

That's right, *b is just a variable, and set_to_pi doesn't know the difference: it's passed in by reference and dutifully set to PI. When b is a null pointer, it is *not* a variable, it's just a loose end waiting to be tied to one.

In IDL, pointers always point to a special stash of regular-old variables called "heap variables". There's nothing out of the ordinary about them, except they can only be accessed through pointers (or objects, but that's a side issue). In all other ways, they are just normal IDL variables. Heap variables even get funny names internally:

```
IDL> print,b
<PtrHeapVar2>
```

Here we see b points to "ptrheapvar2", i.e. the second variable on the "pointer heap". We can even have a look at that variable directly:

```
IDL> help,/heap
Heap Variables:
   # Pointer: 2
   # Object : 0

<PtrHeapVar1>   INT      =       1
<PtrHeapVar2>   FLOAT    =      3.14159
```

In some languages you can have pointers pointing to normal variables, but not in IDL: a variable is either of the normal variety (like 'a' above), or of the heap variety (like 'ptrheapvar2' above). The only distinction, again, is how you access them. The first kind spring

into existence as IDL runs, the latter have to be specifically requested with PTR_NEW(). Once you have this mental model in mind (and specifically forget any baggage you may bring from an understanding of pointers in C), it will all seem much clearer.

JD
