Subject: Re: On Pointers and Culture
Posted by JD Smith on Fri, 04 Mar 2005 20:01:09 GMT
View Forum Message <> Reply to Message

On Fri, 04 Mar 2005 07:05:14 -0700, David Fanning wrote:

> Antonio Santiago writes:
>
>> If you are new on IDL pointers remember are not the same as C pointers.
>
> Another way they are different, and this surprises even people who have
> used them for awhile, is that they are much more forgiving of rough
> handling. More like IDL variables, really, than pointers. So you can do
> something like this:
>
>   ptr = Ptr_New([1,4,6]) ; Pointer to array *ptr = 8.4 ; Pointer now
>   points to scalar Print, *ptr
>   8.4
>
> And there is no leaking of memory because IDL manages all this for you, as
> it does with variables. Naturally, this is dangerous, etc., but it is so,
> so nice. :-)

Not only are pointer variables "more like IDL variables", they *are*
IDL variables. Full, regular old variables, which, rather than having
a name (like myVar) and local scope inside your routine, have a global
scope and a heap variable number. E.g., try:

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

<PtrHeapVar1>   INT       =       1
```

Note that there are two parts to a pointer: the pointer variable (here
`p'), and the "heap" variable it refers to (here `PtrHeapVar1'). The
latter cannot be accessed, except through the former, using the
infamous dereference ("*") operator. Actually more than one pointer
variable can refer to the same heap variable (or none can! --- see
below).

Think of a pointer variable as holding nothing besides than a little
slip of paper which says "Go look at pointer heap variable #1". That
PtrHeapVar1 is a real variable, again, it just has global scope and an
unusual way to access it. I can do anything to it I could do to a
real variable:

```
IDL> *p=findgen(10,10)
IDL> help,/heap
Heap Variables:
   # Pointer: 1
   # Object : 0

<PtrHeapVar2>   FLOAT     = Array[10, 10]
```

I just changed the contents of that heap variable. And no, this is no more dangerous than doing this:

```
IDL> a=1
IDL> a=findgen(10,10)
```

Here's another important aspect of the "pointer heap variables are just regular variables" rule: I can pass memory *without copying* between "real" and "heap" variables, using TEMPORARY. Let's try it:

```
IDL> *p=lindgen(250000000L)
IDL> a=temporary(*p)
IDL> help,/heap
Heap Variables:
   # Pointer: 1
   # Object : 0

<PtrHeapVar1>   UNDEFINED = <Undefined>
IDL> help,/memory
heap memory used: 1000969834, max: 1000969930, gets:     7404, frees:     7060
IDL> *p=temporary(a)
IDL> help,/memory
heap memory used: 1000969871, max: 1000969890, gets:     7407, frees:     7061
```

Notice this proceeds quite rapidly, and without any additional memory used: IDL just transferred PtrHeapVar1's data content in memory to a, and then back. What about poor PtHeapVar1 in the meantime? It's still there, it's just undefined, just like a regular variable would be (because again, it *is* a regular variable). What if I just try to copy the data directly:

```
IDL> a=*p
<__long__ delay while copying that 1GB of data into the swap>
IDL> help,/memory
heap memory used: 2000969853, max: 2000969964, gets:     7411, frees:     7063
```

Ouch! So remember, pointers are lightweight (remember the slip of paper?), but their contents aren't necessarily (in fact usually not).

What about passing variables by reference into routines, can we do that? Try compiling this little procedure:

```
pro set_to_twelve,a
  a=12
end
```

And then:

```
IDL> help,/heap
Heap Variables:
    # Pointer: 1
    # Object : 0

<PtrHeapVar1>   LONG      = Array[250000000]
IDL> set_to_twelve,*p
IDL> help,/heap
Heap Variables:
    # Pointer: 1
    # Object : 0

<PtrHeapVar1>   INT       =       12
```

Yes! We see that *p (aka PtrHeapVar1) is just as good as a "regular" variable, like "a" (again, because it *is* a regular variable -- OK I'll quit beating the deceased equine). But wait, it gets even better! You know how you can't pass, say, a structure member by reference?

```
IDL> st={val:1}
IDL> set_to_twelve,st.val
IDL> print,st
{       1}
```

Hmmm, that's not what we wanted. A standard trap most IDL users learn to tiptoe around early on. But, let's say st.val was a pointer.
Watch carefully, nothing up my sleeves:

```
IDL> *p=0
IDL> st={val:p}
IDL> print,*st.val
       0
IDL> set_to_twelve,*st.val
IDL> print,*st.val
      12
```

Oh yea, passing a structure member by reference (sort of)! Glorious.

What if you lose that pointer variable `p'? Just because PtrHeapVar1 is on a global heap, doesn't mean the regular pointer variable `p' (remember, the slip of paper) is available everywhere. It has a normal scope just like all non-heap variables, and can disappear if you're careless:

```
IDL> help,p
P               POINTER   = <PtrHeapVar1>
IDL> delvar,p  ;; whoops, lost it!
IDL> help,/heap
Heap Variables:
   # Pointer: 1
   # Object : 0

<PtrHeapVar1>   INT      =      12
```

There's our heap variable sitting there with nobody pointing to him. What ever to do? If you're desperate, you can:

```
IDL> new_p=ptr_valid(1,/CAST)
IDL> print,*new_p
     12
```

We've recovered our pointer heap variable, saving him from certain death. What form of death would that be? Well, if again we lose track of that heap variable, and use HEAP_GC, like the grim reaper it harvests poor PtrHeapVar1 with a quick stroke of the scythe:

```
IDL> delvar,new_p
IDL> heap_gc,/verbose
<PtrHeapVar1>   INT      =      12
IDL> help,/heap
Heap Variables:
   # Pointer: 0
   # Object : 0
```

Alas, poor PtrHeapVar1, I knew him well.

Forget what you know about pointers. IDL pointers are not like pointers in C or most other languages you've used. They are a bit akin to "references" in some languages. You've seen many of their advantages, but what about disadvantages? Well, although lightweight in IDL terms, when compared to, e.g., C pointers, they are fairly heavy, since they contain all of the baggage, and ability to take on any form, of regular IDL variables (which they actually are!... oh sorry).

What this means is that the familiar pointer-based data structures you know and love, like trees and multiply linked lists, can of course be implemented using IDL pointers, but they are typically much slower than you might like, especially if they require lots and lots of pointers to define their structure (as trees do, for instance). In C, a linked list can compare favorably to an array in terms of linear searches, etc. (and is much faster for arbitrary insertion). In IDL, you'll probably find WHERE on an array to be many hundreds of times faster than searching a linked-list using IDL pointers. So what does that mean? If you have a favorite data structure that would revolutionize your programming, beg RSI to implement it as a built-in (my top choice would be a real hash object).

There's lots more to know (like how to use precedence to interact with deeply nested pointer data), but that's the basic story.

JD