Subject: Re: compile a routine wich inlude a commun
Posted by JD Smith on Tue, 24 Jan 2006 20:55:35 GMT
View Forum Message <> Reply to Message

On Mon, 23 Jan 2006 21:03:27 -0700, David Fanning wrote:

> Paul Van Delst writes:
>
>> Well, then they're *not* pointers. They're, umm, "copy/move enablers".
>> :o)
>
> I have a feeling JD is going to come to our rescue and sort the whole
> thing out for us. At least, I *hope* so. :-)

IDL pointers are really properly called "references", and are similar to references in many other languages, like Perl. In some ways, they are more limited, since they can only reference a special pool of variables which are otherwise inaccessible: the "heap variables".

In many languages, there are two ways to make a reference: a reference can be made of a pre-existing normal variable, or a new "anonymous" reference can be made, essentially referring to a freshly made heap variable. IDL only supports on the latter method -- anonymous references -- not the former method. There is no "address-of" operator, so there is no way to take an address of an existing variable, ala:

```
IDL> x=2
IDL> new_ptr=&x ;; No such thing
```

It just doesn't exist. Pointer heap variables and normal variables will forever live their separate lives. You *can* cheaply re-assign the actual memory pointed to by a regular variable to a pointer heap variable, and visa versa, using the method David already outlined:

```
IDL> x=indgen(1000000L)
IDL> p=ptr_new(x,/NO_COPY) ;x now undefined
IDL> x=temporary(*p) ;*p now undefined
```

But at no time can you have more than one reference to a given normal IDL variable (e.g. two ways to modify it). You can, of course, have multiple references to each pointer heap variable (or none at all, which is a great way to leak memory). This is covered in the pointer tutorial.

One very basic fact about IDL pointers is that they are very heavy, both in terms of compute time, and memory. Try this:

```
IDL> m=memory(/current)
IDL> a=ptrarr(10000L,/ALLOCATE_HEAP) & for i=0,10000L-1 do *a[i]=1L
IDL> print,(memory(/current)-m)/1024
        382
IDL> a=0
IDL> m=memory(/current)
IDL> a=replicate(1L,10000L)
IDL> print,(memory(/current)-m)/1024
         39
```

Using a pointer to store 10000 long integers (1L) takes approximately 10x as much memory as storing those 10000 integers directly in an array. This is because even *empty* IDL variables take up around 35 bytes of memory. All this extra memory goes to all the additional information associated with each and every IDL variable (mostly things like variable type, array lengths, etc. -- see idl_export.h if you are curious).

Here's how to see that:

```
IDL> a=0
IDL> m=memory(/current)
IDL> a=ptrarr(10000L,/ALLOCATE_HEAP)
IDL> print,float(memory(/current)-m)/10000L
      36.2015
```

So, "pointer" is really a bad name; "anonymous reference" would be a better (if longer) name. The problem with "pointer" is it is comes with no small amount of baggage from association with pointers in languages like C. Unlike C pointers, which are very lightweight, these "pointers" can't really be used for large, nested data structures, without a fairly large memory and speed penalty (not to mention the penalty that looping over many pointers entails). For this reason, the best use of pointers in IDL is to provide persistent, flexible storage for more typical IDL variable types, like large arrays or structures, which can be accessed and operated on without this penalty. Also unlike C pointers, IDL pointers can't be used to access or write memory outside of the IDL address space, so in that sense are much safer than C pointers.

JD