Subject: Re: Pass by value and performance
Posted by JD Smith on Fri, 16 Dec 2005 18:40:18 GMT
View Forum Message <> Reply to Message

> a = 10
> b = PTR_NEW(40)
> c = PTR_NEW(BYTARR(100))
>
>
> Conventional Mem. | HEAP memory
> (Managed by IDL but | (Jungle where you are
> without a Garbage Collector???) | responsible to free)
> ---------- ----
> a = 10 |
> b ----------|--> 40
> c ----------|--> [0,1,2,.....,99]

All memory is managed by IDL, without garbage collection. For normal variable memory, IDL takes care of allocating it when a variable goes into scope (e.g. you enter a procedure and a assign a value to a variable), and de-allocating when it goes out of scope (e.g. exiting a procedure). Heap memory is managed in the same way, except it only gets allocated when you act on a heap variable (via a pointer or object), and only gets de-allocated when you explicity free it (or use one of the heavy-handed clean-up routines like HEAP_FREE).

> Then if I call a function with:
>
> call_to_procedure, *c
>
> In bad C style I think I am passing the content of 'c', that is a BTYARR
> of 100 (BAD ???)

In C, everything is always passed by value. In IDL, everything is always passed by reference (more on this below). Here, you are passing by reference the heap variable which the pointer variable `c' points to. The fact that it is a pointer heap variable, and not a normal variable, is irrelevant.

In C, pointers are often used to avoid the pass-by-value overhead, so that the only thing passed by value is the lightweight pointer, and the full data it points to can be accessed efficiently and without copying. It's still passing by value, but it's such a small value, that you don't care, and, since the pointer gives you the address of the data you are really interested in, you can edit it at will. (As an aside, this form of lightweight pass-by-value is very likely what IDL uses at its C core to implement its default pass by reference

behavior).

In IDL, pointers aren't normally used for this purpose, since everything is passed by reference by default. In IDL, pointers are used more for storing arbitrarily-sized data inside of structures, and objects, and keeping global persistent data around as you jump from procedure to procedure. IDL pointers shouldn't really even be called pointers; probably "references" is a better description of them. C pointers give you indirect hardware access to a block of memory. IDL pointers give you indirect access to a special pool of normal IDL variables called heap variables, special only in their lifetime and access semantics, but otherwise exactly the same as normal IDL variables.

> In IDL is passed a reference to the content, that is like if I writte:
>
> d = BYTARR(100)
> call_to_procedure, d
>
> Is this right??
>
> Thanks a lot.

Yep, again, IDL *always* passes by reference. True of pointer heap variables, and normal variables alike. As far as by-value vs. by-reference, normal vs. pointer heap variables makes no difference whatsoever. The way to think about IDL pass-by-value is as follows:

```
IDL> a=randomu(sd,100,100)
IDL> do_something, a[0:10,20:30]
```

When you pass a[0:10,20:30] as an argument, IDL creates a temporary array variable to hold the smaller subscripted array. It then passes this temporary array variable *by reference* into the procedure, just like normal. You can set this temporary array to another value inside the procedure, and it won't complain:

```
pro do_something, array
  array=12
end
```

However, as soon as your procedure completes, that temporary array variable is automatically destroyed, and you have not managed to set anything. So, it's not that IDL ever passes by value, just that it occasionally automatically creates and destroys temporary variables, which make it appear that arguments have been passed by value.

JD