Subject: Re: saving variables between calls to a procedure?
Posted by JD Smith on Thu, 01 Aug 2002 17:59:14 GMT
View Forum Message <> Reply to Message

On Thu, 01 Aug 2002 07:45:52 -0700, Paul van Delst wrote:

```
> Mark Hadfield wrote:
>>
>> "David Fanning" <david@dfanning.com> wrote in message
>> news:MPG.17b21214ff7d9b35989944@news.frii.com...
>>
>>> Paul van Delst (paul.vandelst@noaa.gov) writes:
>>>
>>>> > pro define,ptr
>>>> >  *ptr=[*ptr,10]
>>>> > end
>>>>
>>>> Hmm. That seems like an extremely dangerous thing to do - couldn't
>>>> you clobber something by concatenating like that? If IDL is smart
>>>> enough to recognise that the next bit of memory may be used by
>>>> something else it then seems that you would end up with a
>>>> non-contiguous data structure (in the figurative).
>>>
>>> This doesn't seem dangerous to me (perhaps because I use the
>>> construct all the time). It seems like one of those wonderful things
>>> IDL occasionally does that makes you think to yourself "Now, by God,
>>> that's how software *ought* to work!"
>>>
>>> In any case, it works, over and over and over. And it never occurred
>>> to me that non-contiguous data storage could be involved, even
>>> remotely.
>>
>> Extending an array a with the a = [a,b] syntax doesn't create a
>> non-contiguous data structure.
>
> I'm not worried about extending arrays like a=[a,b], but pointers ala
> *ptr=[*ptr,10]. If ptr is pointing to some data structure (scalar or
> array), it seems feasible to me that the next position in memory could
> be occupied by something else. If one then does a *ptr=[*ptr,10], what
> happens to what was in that memory? Is it "protected" somehow so that
> the value "10" is then placed in the next "free" spot in memory giving a
> non-contiguous array [not efficient, but safe], or are the contents
> clobbered and replaced by the value "10" [very very bad], or are the
> contents moved to some other memory location so that the original
> position can be used to store the value "10" [relatively efficient(?)
> and safe].
```

The best way to think of pointers in IDL is to forget what you know from other languages like C, and regard them as special unnamed, but otherwise perfectly normal IDL variables which live in a global heap available everywhere, and thus aren't created/destroyed when you enter or exit routines.

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

<PtrHeapVar1>   STRING    = 'test'
IDL> resurrected=ptr_valid(1,/cast)
IDL> print,*resurrected
test
```

Here we see we can recall the pointer heap variable off the heap. And just as for normal IDL variables, you can arbitrarily reassign them, extend them, change the data type of, etc.:

```
IDL> b=ptr_new('x')
IDL> *b=10.1
IDL> *b=[*b,10000]
IDL> print,*b
      10.1000      10000.0
```

This is exactly analogous to:

```
IDL> b='x'
IDL> b=10.1
IDL> b=[b,10000]
IDL> print,b
      10.1000      10000.0
```

The only difference between "b" and "*b" is the former has a name, and a scope (in the $MAIN$ level here), whereas the latter is global, and only accessible through a pointer. In both cases we were able to change from a string to a scalar floating value to a two index array.

Of course, this ease of type/size/layout changing hides lots of behind-the-scenes memory-juggling, some of which may be rather inefficient, so it pays to have a little more familiarity with the methods these operations use internally.

JD