Subject: Is there a less clunky syntax for retrieving data from an array of hashes?
Posted by Matt Francis on Thu, 29 Aug 2013 02:44:13 GMT
View Forum Message <> Reply to Message

I'm (finally) playing around with the 'new' IDL LIST() and HASH() features. I must say I'm fairly un-impressed so far. The syntax is very clunky in the same way the custom object syntax has always been clunky. They look and feel like what they are, bolted on bits to the language with enough effort put in to say 'we have this feature' without actually implementing them in a usable way.

Anyhoo, enough belly aching. Is there a more natural way to do this:

```
>  h = hash()
>  h['a']=1
>  h['b']=2
>  harr = replicate(h,10)
```

Now I want to pull out a value:

```
>  harr[0]['a']
```

Causes an error

```
>  (harr[0])['a']
```

works, but makes for some horribly difficult to read code, particularly if you need to further nest the data structures with more hashes in hashes or arrays. Is there any way to improve this (possibly via compiler flags?).

Note that the same is true if you have a hash of hashes, you still need the fully enclosing brackets, Yuck!

Subject: Re: Is there a less clunky syntax for retrieving data from an array of hashes?
Posted by chris_torrence@NOSPAM on Thu, 29 Aug 2013 06:14:18 GMT
View Forum Message <> Reply to Message

Hi Bogdanovist,

I'm going to respond to your post in reverse order. Both the List and Hash were really designed to be "scalar" objects. It is just a happy coincident that you can create arrays of them, but I would strongly recommend that you do not do that. Instead, you would be better off just putting your data into the List or Hash. Since you can put data of any type, you could just create a List of Hashes to do what you want.

Then, if you do that, you can use the new array syntax to access array elements (or hash/list elements) within a list or hash. For example:

```
IDL> a = List(Hash('key', findgen(10)))
IDL> help, a[0, 'key', 5:8]
<Expression>    FLOAT     = Array[4]
IDL> a[0, 'key', 5:8] = 6
IDL> print, a[0, 'key', 5:8]
      6.00000      6.00000      6.00000      6.00000
```

Hopefully this will address your needs for the List/Hash.

Regarding your first paragraph, it would be great to get specific information. Besides the above questions about array indexing, what else do you think is missing or bad about the List & Hash implementation? Specific use cases are very helpful!

By the way, we don't add anything just to check off on a marketing list. Everything we add has at least one or more customers who have requested it, or it is something that we personally would like to have in the language or need for our other products. We don't have time to add fluff features.

All that being said, there are certain limitations that we have with our existing code, the IDL parser, and the IDL interpreter. In many cases (like array indexing), we cannot change the existing implementation without serious risk of breaking backwards compatibility. So even though some features may appear to look "bolted on," there is always careful design and architecture that is done behind the scenes. Finally, we rely on you, the users, to give us feedback about bugs or missing features, so that we can address them in the next release. So, let us know!

Cheers,
Chris
IDL Project Lead
ExelisVIS

Subject: Re: Is there a less clunky syntax for retrieving data from an array of hashes?
Posted by David Fanning on Thu, 29 Aug 2013 12:41:28 GMT
View Forum Message <> Reply to Message

kagoldberg@lbl.gov writes:

>
> In my opinion, that syntax clunkiness is a small price to pay for what you get by using hashes and lists. I program a lot with objects, and with objects, you have to declare and manage all of the fields explicitly. Every time I add a field, I have to put it in the __define, then in the ::Cleanup, then in the ::GetProperty, then in the ::SetProperty, then in ::Init, then write code to handle it properly. We all probably do these tedious mechanics. And unless you use
pointers (talk about clunky), those object fields have a fixed type in advance.
>
> Now with hashes, you can do just about anything with any type of data, changing it with the

freedom of a pointer, but without the cleanup hassles. Need a new hash field? Go ahead and just start using it on the fly. Does a certain field already exist? Just ask .hasKey(). Lists can .Count() .Add and .Remove in ways that are cleaner than the IDL_Container()'s way of doing things. And, you can very easily pass them around without copying memory, so every routine can share and modify the same hash flexibly, and changes appear in all places at once, if you catch my drift.
>
> Once you start using lists and hashes, you start programming in 'foreach' instead of 'for', and you save the preliminary steps of counting your elements before defining the loop range.
>
> And don't forget, hashes do have an elegant way to reach inside, as follows.
> h = hash()
> h['a'] = [10,20,30]
> print, h['a', 1]
> h['a'] = findgen(5,10)
> print, h['a', 2, 3]
>
> Make sure to condition your data so you don't ask for index values out of range, and these tricks work very nicely.
>
> Thank you, IDL, for hashes and lists.

Thanks for this post.

I have to admit that lists and hashes tend to confuse me, and I have had a hard time figuring out what I would use them for. (Not to mention that using them would suddenly put the Coyote Library out of the reach of hundreds of users.) So, I read this article with a great deal of interest.

I think I understand most of your points, and I know as well as you the tedium of adding new fields to objects. But, as I was reading I got a queasy feeling in my stomach. Doesn't all this ad hoc adding of fields and changing things around on the fly result in programs that are nearly impossible to understand and/or maintain? How do you manage that aspect of using lists and hashes?

Cheers,

David

--
David Fanning, Ph.D.
Fanning Software Consulting, Inc.
Coyote's Guide to IDL Programming: http://www.idlcoyote.com/
Sepore ma de ni thue. ("Perhaps thou speakest truth.")

Subject: Re: Is there a less clunky syntax for retrieving data from an array of hashes?
Posted by kagoldberg on Thu, 29 Aug 2013 18:01:10 GMT
View Forum Message <> Reply to Message

David,
That's a good point. With hashes vs. objects, the lack of rigidity leads to a potential reduction in transparency. One can always look at an object's __define method to see what fields are there (and what other classes we inherit from.) I think it's more of a matter of transferring some of the programming burden elsewhere, but the benefits for pure data handling are there.

For graphics I use objects built from IDLgr... pieces in the conventional way, and the predictable structure of fields and methods makes a great framework. By comparison, hashes don't have classes or methods, so we're only focusing on the data--accessing and updating the data freely.

Here's an example of where I appreciate the flexibility of hashes. Say I have a folder containing 1000 images, and a file or files containing their metadata (100 pieces of metadata per image, where each field may be a scalar, text, or a small array.) In addition to the stored metadata, I want to add-on a few fields that are specific to the way I'm processing the stored data, to give me continuity from one session to the next.

For me, the old way of approaching this is with a structure array. Create a structure to contain the metadata from each individual image, and contain all of it those structures in an array, indexed 0,1,2,... One field in the structure is the image file name.

Now every time I access the metadata, I use where() to find the array index of the structure I want, then I take that index and get at the structure fields. If some images were missing some metadata that others have, I need to rely on flag or default values to realize that (adding steps to the processing). Adding fields to the structure for ad-hoc information requires me to add it to everyone's metadata structure (it's a class thing). OK, that system works, and it's not too bad. It worked for me for years.

Now consider my approach with hashes. YMMV.

1. Each image gets its own metadata hash, analogous to the structure, except that it's perfectly acceptable for some data to be missing, or added to one image's metadata and not all of them. The hash will not complain that it doesn't match the others. I can add fields related to the image processing by just declaring hash['newField'] = 'easy'. The hash fields can handle what would otherwise require pointers. Like this

```
hash['a'] = 15.
hash['a'] = [15., 16., 17.]
hash['a'] = !null   ; (Structures hate this one.)
```

2. Now we need to bundle up all of these individual hashes, like the structure array. One way is with a list(). Lists are nice because you don't have to know their count in advance. But with lists you have to somehow remember which list index corresponds to which image, or you'll be stuck searching every time. Here's where hashes shine. Imagine a hash of metadata hashes, using the filenames as the keys.

So with h1 as the metadata hash for 'image_001.tif' (containing whatever), then we assign

```
masterHash = hash()
masterHash['image_001.tif'] = h1
masterHash['image_001.tif'] = h2
...
```

Accessing metadata from an image now looks like this (notice, no searching)

```
xy = masterHash['image_001.tif', 'xy']
  which is a short way of writing
xy = (masterHash['image_001.tif'])['xy']
  (i.e. Give me the 'xy' field in the hash keyed as 'image_001.tif')
```

If you want to be careful, you can test for existence first. A full-paranoid example would be like this

```
file = 'image_001.tif'
xy = !null  ;--- test for this in the end.
if masterHash.haskey(file) $
  then if masterHash[file].hasKey('xy') $
    then xy = masterHash[file, 'xy']
```

In the above example, we ask if the masterHash knows about the image called 'image_001.tif'. Then once that's established, we ask if the metadata contains an 'xy' field.

With structures, it might go something like this, where we know for sure that xy is a field.

```
file = 'image_001.tif'
xy = !null
w = where(sArray.filename EQ file, count)
if count GT 0 $
  then xy = sArray[w[0]].xy
```

Adding new images is less painful in the hash, we just

```
masterHash[newFile] = hNew
```

In the structure case we might append a new structure element to the array.

```
sArray = [sArray, newStruct]
```

It's nice to be able to pass the hash around as an argument to functions. Since it's object-like, it is always a reference to the hash--no memory copying, no multiple-copies.

TL;DR Hashes are great for image metadata because they're flexible, you can key them by filename in a hash of hashes so there's no WHERE searching for your data, you can pass around the reference without copying memory, you can add new, custom fields or images on the fly. Give them a try.
