Subject: Re: getting structure elements by tag name string
Posted by JD Smith on Wed, 08 Jan 2003 16:09:25 GMT
View Forum Message <> Reply to Message

On Wed, 08 Jan 2003 06:40:55 -0700, David Fanning wrote:

> s@visita2.die.upm.es (s@visita2.die.upm.es) writes:
>
>> I try to use a structure as a bad substitute for a hash map (hashs or
>> associative lists or dictionaries or anything similar don't exist in
>> IDL, do they?)
>
> Humm. I guess I'm used to thinking of IDL as nothing *more* than one
> giant hash map. Are you familiar with the WHERE function, or with array
> operations, in general?
>

Actually, hashes are something altogether different from other data structures, and are sorely missed from IDL. Their defining feature? They provide near constant-time lookup of array elements in lists of any size. Contrast this to the linear-time lookup WHERE provides, and you'll see why hashes are cool.

The basic idea of a hash is that you perform some magic (called the "hash function") on your hash key (e.g. "a" in this case). The hash function produces an index value (like 42), which is used to index a large array where the matching value is stored. Assuming your hash function is perfect, nothing besides "a" would hash to the same index value, and you could extract your value in one steop. Of course this isn't really true in practice, so typically you have a very small linked list in each "bucket" of the hash array, and perform a secondary linear search through it to find the correct element and return the desired value. The art of hashes is to find a decent, cheap-to-compute hash function that does a good job "spreading" data around the hash array, to minimize empty (and hence wasted) elements, and keep the secondary search small (or non-existent).

Unfortunately, it's quite difficult to build good hash functions and matching hash table sizes which strike a decent balance between wasted memory, lookup speed, and flexibility for growing/shrinking the hash array. There's no reason a hash couldn't be implemented using regular IDL arrays and pointers, along with a suitable hash function, but I suspect it would be an underperformer, compared to more finely tuned offerings of other languages. Might be a good exercise to try though.

>> Suppose I have a structure like this: s = {a:7,b:9}
>>
>> and I have a variable that contains a tag name: tag = 'a'

```
>>
>>  Now I want to put the value of the structures element with name tag
>>  into a new variable called value:
>>  value = s.tag
>>
>>  This doesn't work of course, because tag is undefined for this
>>  structure. But how can I get this the elements value?
>>
>>  I found a somewhat crude way:
>>  res = execute('value =' + 's.' + tag)
>>
>>  That works, but it's really ugly. Is there a better way to do this?
>
>  I should think so. Hash tables generally have ways to add, delete, and
>  find items. That almost sounds like a linked list to me.
>
>     http://www.dfanning.com/programs/linked_list__define.pro
>
```

Similar functionality, completely different implementation. Linked lists are linear just like WHERE, and, further, require pointer dereference at every step! Their main (and not insignificant) advantage: you can add or remove elements with only a constant small penalty, and you preserve ordering (which hashes expressly do *not*).

```
>>  And, is there a way to delete an element from a structure?
>
>  Do you mean delete a "field" from a structure? If you have anonymous
>  structures (see your "s" structure, above), then you could just
>  re-define your structure with that field:
>
>     s = (a:s.a}
```

Structs are poor replacements for hashes. Sometimes I use a pair of vectors: a key vector, and a value vector. A linear search through the key vector using WHERE is the poor-man's lookup, but of course all your values have to be of the same type, unless you use a pointer array for the value vector. E.g., for your case:

```
s_key=['a','b'] & s_val=[7,9]
tag='a'
wh=where(s_key eq tag)
val=s_val[wh[0]]
```

You can also be cheeky and "store" your key vector as tag names in a structure:

```
s={a:7,b:9}
```

```
s_key=tag_names(s)
tag='a'
wh=where(s_key eq strupcase(tag))
val=s.(wh[0])
```

with the advantage that you can have different types of values without resorting to pointers, and the disadvantages that you can't easily remove or append new tag/value pairs, and that tags are case insensitive ("A" and "a" are the same). Creative use of CREATE_STRUCT will allow you to concatenate, or even remove key/tag pairs from the middle of structures, but this usually requires copying the full structure, which is very inefficient for large amounts of data. The same caveat of course holds true for adding/deleing elements when using a pair of arrays to masquerade as a hash.

One other poor man's pseudo-hash technique: if you just need to know whether a key exists at all in your "hash" and don't need the value, you can speed it up slightly with something like:

```
if NOT array_equal(s_key ne tag,1b) then print,'Got '+tag
```

which stops as soon as it finds a key which is "not not equal" (i.e. "equal") to tag, rather than continuing on through the array. Ideally, you could actually return the index where this happened and use it to replace WHERE, but ARRAY_EQUAL doesn't allow this at present.

Good luck,

JD