Subject: Re: LIST performance
Posted by penteado on Sun, 07 Nov 2010 01:31:34 GMT
View Forum Message <> Reply to Message

On Nov 6, 10:03 pm, JD Smith <jdtsmith.nos...@yahoo.com> wrote:
> I'm still trying to decide how useful LIST and HASH are.

I am surprised they are that slow. But that was not an issue in most (maybe all) of the many uses I had of them in the past few months. I had been wanting lists and hashes for a long time, and was starting to move to Python for their lack in IDL.

It is too much work, or too awkward, to store and retrieve variable or heterogeneous things using only arrays, pointers and structures. Most of the time they are not so large that performance is an issue, much more important is that the code is short and clean. Which is why lists and hashes are present in every(?) decent modern language.

One simple example is how much nicer histogram's reverse indices become with lists (and !null):

http://www.ppenteado.net/idl/histogram_pp.html

A more complicated example is the use of lists and hashes to set and retrieve multiple properties in the class I made to provide a plot grid (like multiplot does for DG):

http://www.ppenteado.net/idl/pp_multiplot__define.html

Properties can be retrieved or set as lists, with one element for each column/line/plot. That way, for instance, xtickvalues can be a list with arrays of different lengths for the columns. And hashes can store properties by their names. That is just for the API. Internally, the code would be a horrible mess without lists and hashes to store things that are heterogeneous and variable (in number of elements and shape/type of the elements).

The current lists and hashes can (and should) mature in future versions, becoming more efficient, perhaps providing some specialized subclasses (like homogeneous lists). This can happen by reimplementation or inheritance and overloading, in either IDL code or DLMs.

For instance, a derived list could be made to implement the expand-concatenate algorithm, providing the same interface, in an easy way to implement, as only a few methods would have to be written. All the classes I had made for this kind of functionality before IDL 8 were incomplete, as it was too much work to implement every relevant

method, with all the needed error-checking, documentation and testing.
Now, inherited classes can be made, a lot less work.

Other algorithms can even make use of the built-in lists: when the buffer gets full, instead of concatenating arrays, a new buffer array is made, and added to a list where each element is a buffer array. At the end, these arrays get put together into a flat array.

Java is an example where a bunch of different implementations are provided in the standard library, which through inheritance keep the same API for each kind of collection. For instance, it has 5 types of lists, and 9 types of maps (hashes, in IDL's nomenclature). Given that IDL's lists and hashes are regular classes, inheritance and overloading allow to easily make a similar variety of containers.