Subject: LIST extensions
Posted by Paul Van Delst[1] on Tue, 14 Dec 2010 21:51:11 GMT
View Forum Message <> Reply to Message

Hello,

Sorry to keep flogging this particular topic, but I'm currently up to my eyes in adapting some code from using pointers
to using lists (partly for curiosity, partly to avoid nested pointer dereferencing), and I've found the very simple
methods below to be useful in dealing with the "is it empty?", "are there nulls in the list?" etc type of questions I've
been asking here.

Comments?

cheers,

paulv

```
;+
; NAME:
;       List::Empty
;
; PURPOSE:
;       The List::Empty function method return TRUE if a list contains
;       no elements, FALSE otherwise.
;
; CALLING SEQUENCE:
;       result = Obj->[List::]Empty()
;-
FUNCTION List::Empty
  RETURN, N_ELEMENTS(self) EQ 0
END


;+
; NAME:
;       List::Length
;
; PURPOSE:
;       The List::Length function method returns the number of
;       elements in a list.
;
; CALLING SEQUENCE:
;       result = Obj->[List::]Length()
;-
```

```
FUNCTION List::Length
  RETURN, N_ELEMENTS(self)
END


;+
; NAME:
;       List::n_Items
;
; PURPOSE:
;       The List::n_Items function method returns the number of
;       non-null elements in a list.
;
; CALLING SEQUENCE:
;       result = Obj->[List::]n_Items()
;-
FUNCTION List::n_Items
  idx = WHERE(self NE !NULL, count)
  RETURN, count
END


;+
; NAME:
;       List::Compact
;
; PURPOSE:
;       The List::Compact function method returns a copy of a list
;       with all null elements removed.
;
; CALLING SEQUENCE:
;       result = Obj->[List::]Compact()
;-
FUNCTION List::Compact
  idx = WHERE(self NE !NULL, count)
  RETURN, ( count GT 0 ) ? self[idx] : LIST()
END
```

Subject: Re: LIST extensions
Posted by Jeremy Bailin on Thu, 23 Dec 2010 14:00:10 GMT

View Forum Message <> Reply to Message

On Dec 22, 6:47 pm, Paulo Penteado <pp.pente...@gmail.com> wrote:
> On Dec 14, 8:31 pm, Paulo Penteado <pp.pente...@gmail.com> wrote:
>
>> This is very pertinent. Those are important needed features / bug
>> fixes that were missed in the way lists and hashes were implemented.

>> They are good ideas to add to the class I started implementing, which
>> will also include deep copy, conversion to/from pointer arrays, and
>> fix those issues with non existing elements.
>
> Other changes I am considering to put in my derived classes:
>
> 1) Make lists do nothing (as hashes already do) if !null is used as
> index on _overloadBracketsLeftSide.
>
> 2) Make lists and hashes return !null when !null is used as an index
> (now they throw an error).
>
> 3) Make lists and hashes accept !null on the _overloadPlus method and
> do nothing, instead of throwing an error.
>
> (3) is to work in conjunction with (2), so that lists/hashes can be
> added to indexed lists/hashes, without having to verify if the index
> is not !null.
>
> Any thoughts?

I think (3) is a good idea on its own, regardless of (2). !null seems
to me a natural convention for an empty list/hash.

(but this is coming from someone who hasn't yet used IDL8...)

-Jeremy.

Subject: Re: LIST extensions
Posted by penteado on Sun, 02 Jan 2011 07:14:06 GMT
View Forum Message <> Reply to Message

On Jan 2, 4:37 am, Paulo Penteado <pp.pente...@gmail.com> wrote:
> I have really been finding inconvenient the lack of these, and noticed
> another shortcoming: _overloadPlus should add to a list something that
> is not a list. So that
>
> l1=list()
> l2=list(1,2,3)
> w=where(l2.toarray() eq 2)
> l1+=l2[w]
>
> Does not throw an error. As it is now, it takes a lot of work to
> select elements from a list with where(): not only it is necessary to
> test for no results (because !null is not accepted as index for
> lists), but it is also necessary to test for a single match, as a list
> indexed by a scalar (or 1-element array) returns the list element,

> which cannot be concatenated to a list (unless the element happens to
> be a list, which would not throw an error, but would concatenate in
> the wrong way).
>
> An alternative is not change _overloadPlus, but change
> _overloadBracketsRightSide to return a 1-element list when given a 1-
> element array as index. It should still return the element when
> indexed by a scalar.
>
> And doing these things also makes me think that, for syntatic sugar,
> there should be a list::where() method that would simply call where()
> on the list's toarray() result. Or where() should automatically call
> toarray() if given a list.

Also, the remove method should not remove elements when the index it is given is !null. For the same reason, as, in the example above,

l2.remove,where(l2.toarray() lt 0,/null)

Will remove the last element of l2, instead of removing nothing.

But this has an implementation difficulty: how can a routine distinguish between being given !null for an argument, and not being given that argument? I remember this being asked, but do not remember if there was an answer. The only way I can think of now is to try to concatenate something to it, and catch the error that gets thrown in case the variable is not !null. As in:

```
function pp_isnull,a
compile_opt idl2,logical_predicate
catch,err
if err then begin
  catch,/cancel
  ret=0
endif else begin
  b=[a,a]
  ret=1
endelse
return,ret
end
```