
Subject: Re: IDL_Object example // help with
Posted by chris_torrence@NOSPAM on Mon, 03 Mar 2014 15:28:14 GMT
[View Forum Message](#) <> [Reply to Message](#)

On Sunday, March 2, 2014 1:11:53 PM UTC-7, Matthew Argall wrote:

> I solved the problem by turning the subscript ranges into a 1D array of indices. Here are the final results, if anyone is interested.

```
>
>
>
> https://dl.dropboxusercontent.com/u/42944960/mrreformindices .pro
>
> https://dl.dropboxusercontent.com/u/42944960/mrarray__define .pro
>
>
>
>
>
>
>
>
> IDL> a = obj_new('mrarray', findgen(2,4,4,2))
>
> IDL> print, reform(a[0, [0,2], 1:3, 1])
>
>    40.0000    44.0000
>
>    48.0000    52.0000
>
>    56.0000    60.0000
>
> IDL> a[0, [0,2], 1:3, 1] = findgen(6)
>
> IDL> print, reform(a[0, [0,2], 1:3, 1])
>
>    0.00000    3.00000
>
>    1.00000    4.00000
>
>    2.00000    5.00000
```

Hi Matthew,

Yes, that is indeed the best way to do that. Once you've converted your input subscripts into a 1D array, then you can just do the indexing.

If it helps, I'll append the code for IDL's Collection class to this post. It's actually a "helper" method that gets called from the List::_overloadBracketsLeftSide method.

Cheers,
Chris

```
;-----  
; Internal method to change the contents of pointer p  
;  
; p - pointer to an array  
; varname - name of the original list or hash variable  
; value - the new value for part of the array  
; isRange, i0...i7 - array indexing variables  
;  
pro Collection::_overloadBracketsLeftSideArray, p, varname, value, $  
  isRange, i0, i1, i2, i3, i4, i5, i6, i7  
  
  compile_opt idl2, hidden  
  
  ; Subtract one since we know we are indexing into the list.  
  nr = N_ELEMENTS(isRange) - 1  
  
  dim = SIZE(*p, /DIMENSIONS)  
  
  ; If our element is a List or Hash, call the bracket code on it directly.  
  ; Otherwise we would need to handle all of the different dimensions  
  ; using a massive nested "if" down below, since our naive method  
  ; of using an index array won't work for Lists or Hash.  
  if (ISA(*p, 'IDL_Object')) then begin  
    case (nr) of  
      1: (*p)._overloadBracketsLeftSide, *p, value, isRange[1:], i1  
      2: (*p)._overloadBracketsLeftSide, *p, value, isRange[1:], i1, i2  
      3: (*p)._overloadBracketsLeftSide, *p, value, isRange[1:], i1, i2, i3  
      4: (*p)._overloadBracketsLeftSide, *p, value, isRange[1:], i1, i2, i3, i4  
      5: (*p)._overloadBracketsLeftSide, *p, value, isRange[1:], i1, i2, i3, i4, i5  
      6: (*p)._overloadBracketsLeftSide, *p, value, isRange[1:], i1, i2, i3, i4, i5, i6  
      7: (*p)._overloadBracketsLeftSide, *p, value, isRange[1:], i1, i2, i3, i4, i5, i6, i7  
    endcase  
    return  
  endif  
  
  catch, iErr  
  if (iErr ne 0) then begin  
    catch, /CANCEL  
    msg = !error_state.msg  
    if (STRPOS(msg, 'subscript') ge 0) then begin  
      msg = 'Attempt to subscript ' + OBJ_CLASS(self) + $  
        ' element within ' + $
```

```

    varname + ' is out of range.'
endif
MESSAGE, msg, /NONAME
endif

```

```

; Highly optimized code for three dimensions or lower.
; Handle all combinations of subscript ranges or indices.

```

```

if (nr le 3) then begin
    if ISA(i3) then begin

        if (isRange[3]) then begin
            if (isRange[2]) then begin
                if (isRange[1]) then begin
                    (*p)[i1[0]:i1[1]:i1[2],i2[0]:i2[1]:i2[2],i3[0]:i3[1]:i3[2]] = value
                endif else begin
                    (*p)[i1,i2[0]:i2[1]:i2[2],i3[0]:i3[1]:i3[2]] = value
                endelse
            endif else begin
                if (isRange[1]) then begin
                    (*p)[i1[0]:i1[1]:i1[2],i2,i3[0]:i3[1]:i3[2]] = value
                endif else begin
                    (*p)[i1,i2,i3[0]:i3[1]:i3[2]] = value
                endelse
            endelse
        endif else begin
            if (isRange[2]) then begin
                if (isRange[1]) then begin
                    (*p)[i1[0]:i1[1]:i1[2],i2[0]:i2[1]:i2[2],i3] = value
                endif else begin
                    (*p)[i1,i2[0]:i2[1]:i2[2],i3] = value
                endelse
            endif else begin
                if (isRange[1]) then begin
                    (*p)[i1[0]:i1[1]:i1[2],i2,i3] = value
                endif else begin
                    (*p)[i1,i2,i3] = value
                endelse
            endelse
        endelse
    endif else if ISA(i2) then begin

        if (isRange[2]) then begin
            if (isRange[1]) then begin
                (*p)[i1[0]:i1[1]:i1[2],i2[0]:i2[1]:i2[2]] = value
            endif else begin
                (*p)[i1,i2[0]:i2[1]:i2[2]] = value
            endelse
        endif
    endif
endif

```

```

endif else begin
  if (isRange[1]) then begin
    (*p)[i1[0]:i1[1]:i1[2],i2] = value
  endif else begin
    (*p)[i1,i2] = value
  endelse
endelse

```

```

endif else begin
  ; just i1
  if (isRange[1]) then begin
    (*p)[i1[0]:i1[1]:i1[2]] = value
  endif else begin
    (*p)[i1] = value
  endelse
endelse

```

```

return
endif

```

;*** Brute force code for 4D or higher arrays.

```

; Compute the number of subelements for each indexing dimension.
nd = LON64ARR(nr)

```

; Adjust for negative indices or "*" indices.

```

if (nr ge 7 && isRange[7]) then begin
  if (i7[0] lt 0) then i7[0] += dim[6]
  if (i7[1] lt 0) then i7[1] += dim[6]
endif
if (nr ge 6 && isRange[6]) then begin
  if (i6[0] lt 0) then i6[0] += dim[5]
  if (i6[1] lt 0) then i6[1] += dim[5]
endif
if (nr ge 5 && isRange[5]) then begin
  if (i5[0] lt 0) then i5[0] += dim[4]
  if (i5[1] lt 0) then i5[1] += dim[4]
endif
if (nr ge 4 && isRange[4]) then begin
  if (i4[0] lt 0) then i4[0] += dim[3]
  if (i4[1] lt 0) then i4[1] += dim[3]
endif
if (nr ge 3 && isRange[3]) then begin
  if (i3[0] lt 0) then i3[0] += dim[2]
  if (i3[1] lt 0) then i3[1] += dim[2]
endif
if (nr ge 2 && isRange[2]) then begin

```

```

    if (i2[0] lt 0) then i2[0] += dim[1]
    if (i2[1] lt 0) then i2[1] += dim[1]
endif
if (nr ge 1 && isRange[1]) then begin
    if (i1[0] lt 0) then i1[0] += dim[0]
    if (i1[1] lt 0) then i1[1] += dim[0]
endif

switch (nr) of
7: nd[6] = isRange[7] ? (i7[1] - i7[0])/i7[2] + 1 : N_ELEMENTS(i7)
6: nd[5] = isRange[6] ? (i6[1] - i6[0])/i6[2] + 1 : N_ELEMENTS(i6)
5: nd[4] = isRange[5] ? (i5[1] - i5[0])/i5[2] + 1 : N_ELEMENTS(i5)
4: nd[3] = isRange[4] ? (i4[1] - i4[0])/i4[2] + 1 : N_ELEMENTS(i4)
3: nd[2] = isRange[3] ? (i3[1] - i3[0])/i3[2] + 1 : N_ELEMENTS(i3)
2: nd[1] = isRange[2] ? (i2[1] - i2[0])/i2[2] + 1 : N_ELEMENTS(i2)
1: nd[0] = isRange[1] ? (i1[1] - i1[0])/i1[2] + 1 : N_ELEMENTS(i1)
endswitch

; Compute the cumulative and total number of elements.
ncum = PRODUCT(nd, /INTEGER, /CUMULATIVE)
n = PRODUCT(nd, /INTEGER)
indices = n eq 1 ? 0LL : LON64ARR(n)

; Product of all "previous" dimensions.
pdim = PRODUCT(dim, /CUMULATIVE, /INTEGER)

; Construct the indices for the first dimension.
ix = (isRange[1] ? i1[0] + i1[2]*L64INDGEN(nd[0]) : i1)
if ((nd[0] ne n) && ISA(ix,/ARRAY)) then ix = REBIN(ix, nd[0], n/nd[0])
indices += ix

; Construct the indices for the higher dimensions.
for i=1,nr-1 do begin
    ; Get the raw indices for this particular dimension.
    case (i) of
    1: ix = (isRange[2] ? i2[0] + i2[2]*L64INDGEN(nd[i]) : i2)
    2: ix = (isRange[3] ? i3[0] + i3[2]*L64INDGEN(nd[i]) : i3)
    3: ix = (isRange[4] ? i4[0] + i4[2]*L64INDGEN(nd[i]) : i4)
    4: ix = (isRange[5] ? i5[0] + i5[2]*L64INDGEN(nd[i]) : i5)
    5: ix = (isRange[6] ? i6[0] + i6[2]*L64INDGEN(nd[i]) : i6)
    6: ix = (isRange[7] ? i7[0] + i7[2]*L64INDGEN(nd[i]) : i7)
    endcase
    ; Multiply indices for this dimension by all the lower
    ; dimensions since this dimension spans them.
    ix *= pdim[i-1]
    ix = REFORM(ix, 1, nd[i], /OVERWRITE)
    ; Use rebin to replicate the indices - make them "repeat" for all of

```

```
; the lower dimensions, and "duplicate" them for the higher dims.  
if nd[i] ne n then ix = REBIN(ix, ncum[i-1], nd[i], n/ncum[i])  
indices += ix  
endfor  
  
(*p)[indices] = value  
end
```
