Subject: Ptr_Free that recurses (including pointers within structures)
Posted by philaldis on Thu, 04 Mar 1999 08:00:00 GMT
View Forum Message <> Reply to Message

While no-one replied to a little P.S. I put on a former post saying that I would have this program in a few days, I'll put it here anyway,
since it's the first program I have written at home, in my own time (oh dear) so I'm allowed to put it out. I realise that people don't
normally put the programs on the newsgroup but as it is short and I haven't really set up my web page yet, I thought that people wouldn't
mind.

It's only a short little routine but it's pretty powerful. It will free up any memory referenced by an original pointer(s). I challenge (how
dangerous) anyone to find a 'heap hierachy' that it can't free up. It will free up structures within structures, object refs etc. I just
thought that it might be quite useful to some people just to be able to recurse through the heap, although it certainly isn't designed to
stop careful heap memory clearing up. I've used it in a circular linked list so that when the user replaces a node or whatever, it will free
up all the memory that was referenced by the pointer in the linked list.

The program also requires a slightly modified coyotelist, which just has an extra remove method. The programs are below so scroll down
and cut and paste into files : Ptr_Free_Recursively.pro
  coyotelist__define.pro

Cheers,
Phil Aldis

P.S. If you do want to reply then could you also cc to my e-mail address : philaldis@geocities.com as my news feed at home is not very good.

Program Number One: Ptr_Free_Recursively.pro

```
;+
; NAME:
;  PTR_FREE_RECURSIVELY
;
; PURPOSE:
;  This routine when passed a pointer will free up any memory associated
;  with that pointer. This includes structures, which could contain
;  pointers, or other structures which could in turn contain more
;  pointers. It also destroys objects it comes across. It can also
;  cope with null pointers that are found on its travels.
;
; AUTHOR:
```

```
;
;  Philip Aldis
;  14 Milton Road,
;  Bentley Heath,
;  B93 8AA.
;  Phone: 0044 1564 773437
;  E-Mail:philaldis@geocities.com
;
; CATEGORY:
;  Heap Memory
;
; CALLING SEQUENCE:
;  Ptr_Free_Recursively, AValidPointer
;
; INPUTS:
;  InitialPointer: The pointer whose 'heap memory hierachy' will be freed.
;
; KEYWORD PARAMETERS:
;  None.
;
; OUTPUTS:
;  None.
;
; COMMON BLOCKS:
;  None, of course.
;
; SIDE EFFECTS:
;  Well as far as I can see, there are none. But please try and find them
;  so I can improve the program.
;
; RESTRICTIONS:
;  The program will destroy any objects it meets but if the cleanup
;  methods for those objects are not tight then there will be leakage.
;  There are no restrictions on the number of levels of referencing -
;  the routine is totally general.
;  Also you must have my modified version of coyotelist (DWF).
;
; EXAMPLE:
; First you have to have a valid pointer and while obviously your hierachy
; doesn't have to be as complictated as this, the example shows just what
; it can do.
;
; ThisPointer = Ptr_New( [Ptr_New(), $
;       Ptr_New({h:'hello', point0:ptr_new(ptr_new(ptr_new($
;         {string:'hello', point1:ptr_new(),point2:ptr_new('hello')}))), $
;           point3:ptr_new(), b:'bye'}), $
;       Ptr_New('Hello'), $
;       Ptr_New({Object:Obj_New('Coyotelist'), Point4:Ptr_New(), $
```

```
;           Point5:[ptr_new('hello'),ptr_new('bye')]})
$
;        ])
;
; Ptr_Free_Recursively, ThisPointer
;       .............and all your memory will be gone.
;
; MODIFICATION HISTORY:
;  Written by: Phil Aldis
; March 3rd, 1999: Fixed the last few bugs and attempted to make the comments
;   a bit clearer.   PJA
;-

PRO Ptr_Free_Recursively, InitialPointer
 ;This Program takes a pointer and goes through all the
 ;memory freeing it up.

struct = {  ROUTENODE,  $
          Pointer:Ptr_New(),  $ ;The pointer that is currently being followed
                ;for this particular route node.
          StructurePointer:Ptr_New(), $ ;In the case of when a structure is found it
                ;points to the structure.
          TagNumber:0,    $ ;In the case of when a structure is found it is the tag number
               ;currently being processed.
          Element:0L  $  ;The element of this route node that's being looked at
              ;currently being
         }
FOR i=0, N_Elements(InitialPointer)-1 DO BEGIN

   Route = Obj_New('coyotelist')
   CurrentElement = 0   ;This is the current element being looked at. This only comes into
        ;with structures.
   CurrentPointer = InitialPointer[i] ;This is the current pointer
   CurrentRouteNumber = -1  ;The index in the coyotelist that is our current number
   FinishedYet = 0  ;A flag to determine if we've reached the end
   MoveBackOne = 0  ; A flag that is used when we have to move back a node

   WHILE FinishedYet NE 1 DO BEGIN
  ;Keep looping until eventually it's determined that the end is here
      NoNewRoute = 0

      IF Ptr_Valid(CurrentPointer) THEN BEGIN
         IF Size((*CurrentPointer)[CurrentElement], /type) EQ 10 THEN BEGIN
    ;If the current pointer points to another pointer then
    ;put in a route node, move the currentroutenumber on one
    ;and set the current pointer to the pointer which we have
    ;just found
            Route->Add_Node, {ROUTENODE, CurrentPointer, Ptr_New() , 0, 0}
```

```
            CurrentRouteNumber = CurrentRouteNumber+1
            CurrentElement = 0
            CurrentPointer = (*CurrentPointer)[CurrentElement]

         ENDIF ELSE IF Size((*CurrentPointer)[CurrentElement], /type) EQ 8 THEN BEGIN
      ;If the current pointer points to a structure then add a new
      ;route node. Set the structure pointer to point to the structure
      ;that is found. Set the current pointer to point to the first
      ;tag. This will create a copy of the first tag and put it on
      ;the heap. Then set the routenode.pointer to point to the current
      ;pointer. In this way we can fool the program into thinking
      ;that the routenode.pointer points to a pointer which in turn will
      ;point to copies of all of the tags. (Stay with me on this one - it
      ;was the best way to do it)
            ThisStructure = {ROUTENODE, Ptr_New(), CurrentPointer, 0, 0}
            CurrentPointer = Ptr_New((*CurrentPointer).(0))
            ThisStructure.Pointer = Ptr_New(CurrentPointer)
            Route->Add_Node, ThisStructure
            CurrentRouteNumber = CurrentRouteNumber+1
            CurrentElement = 0

         ENDIF ELSE  NoNewRoute = 1

      ENDIF ELSE NoNewRoute = 1

      IF NoNewRoute EQ 1 THEN BEGIN
   ;If no new route was found then there's probably some heap memory that
   ;needs freeing
         IF Ptr_Valid(CurrentPointer) THEN BEGIN
            IF Obj_Valid(*CurrentPointer) THEN Obj_Destroy, *CurrentPointer
            Ptr_Free, CurrentPointer
         ENDIF

         ThisNode = Route->Get_Item(CurrentRouteNumber)

   ;If we've reached the last element in this particular pointer array. Having
   ;a single pointer i sthen simply a special case of the general case.
         IF (*ThisNode).Element EQ N_Elements(*((*ThisNode).Pointer))-1 THEN BEGIN

    ;Are we delaing with structure?
            IF Ptr_Valid((*ThisNode).StructurePointer) THEN BEGIN
              ;Have we reached the last tag?
                IF (*ThisNode).TagNumber EQ N_Tags(*((*ThisNode).StructurePointer))-1 THEN
BEGIN
                   ;If the currenttroutenumber = 0 and we're at the last
                   ;tag, we've reached the end.
                  IF CurrentRouteNumber EQ 0 THEN BEGIN
      FinishedYet = 1
```

```
                ENDIF ELSE BEGIN
                  ;If not then set the flag to indicate that we simply need
                  ;to move back one route step.
                    MoveBackOne = 1
                ENDELSE
              ENDIF ELSE BEGIN
                ;We're simply moving on the tag. So free up the pointer, which
                ;pointed to the temppointer, (which has already been freed just after
                ;IF nonewroute EQ 1. Then increase the tag number field. Remember
                ;we have the actual pointer to this routenode so we are changing it.
                ;Then set the currentpointer to the next tag, and the thisnode.pointer
                ;the current pointer.
                  Ptr_Free, (*ThisNode).Pointer
                  (*ThisNode).TagNumber = (*ThisNode).TagNumber + 1
                  CurrentPointer =  Ptr_New((*((*ThisNode).StructurePointer)).((*ThisNode).TagNu
mber))
                  (*ThisNode).Pointer = Ptr_New(CurrentPointer)
                  CurrentElement = 0
             ENDELSE

    ;If we're not dealing with a structure
          ENDIF ELSE IF CurrentRouteNumber EQ 0 THEN BEGIN
            ;If the currenttroutenumber = 0 and we're at the last element and
            ;we're not dealing with a structure - we've reached the end.
    FinishedYet = 1
          ENDIF ELSE BEGIN
            ;If not then set the flag to indicate that ew simply need
                ;to move back one route step.
              MoveBackOne = 1
   ENDELSE
        ENDIF ELSE BEGIN
          ;We've not reached the end.

          IF Ptr_Valid((*ThisNode).StructurePointer) THEN BEGIN
            ;The structure must contain an array. We don't
            ;fiddle with the pointer, we simply increment
            ;the current element.
           CurrentElement = CurrentElement+1
           (*ThisNode).Element = (*ThisNode).Element+1
          ENDIF ELSE BEGIN
            ;There is no structure so we have to move onto the next
            ;element in this pointer array.
           (*(ThisNode)).Element = (*(ThisNode)).Element+1
           CurrentPointer = (*((*(ThisNode)).Pointer))[(*(ThisNode)).Element]
           CurrentElement = 0
          ENDELSE
        ENDELSE
```

```
IF FinishedYet EQ 1 THEN BEGIN
  ;If we've finished then free up the routenode.pointer and the
  ;structurepointer, which may well be invalid.
 Ptr_Free, (*ThisNode).Pointer
           Ptr_Free, (*ThisNode).StructurePointer
ENDIF ELSE IF MoveBackOne EQ 1 THEN BEGIN
  ;We have to now move back a node. First free up the pointer and the
  ;structure pointer, which could well be invalid.
 Ptr_Free, (*ThisNode).pointer
 Ptr_Free, (*ThisNode).StructurePointer
  ;Then decrease the currentroutenumber and get the nextitem (well
  ;really it's now the current item)
 CurrentRouteNumber = CurrentRouteNumber-1
 NextNode = Route->Get_Item(CurrentRouteNumber)


 IF Ptr_Valid((*NextNode).StructurePointer) THEN BEGIN
   ;If we're dealing with a structure
   ;If this current tag wasn't the last then ...
    IF (*NextNode).TagNumber NE N_Tags(*((*NextNode).StructurePointer))-1 $
       THEN BEGIN
        ;Find out where they left off and increase the tag number,
        ;as well as setting the current pointer appropriately.
       (*NextNode).TagNumber = (*NextNode).TagNumber + 1
       CurrentPointer = Ptr_New((*((*NextNode).StructurePointer)).$
                        ((*NextNode).TagNumber))
    ENDIF ELSE BEGIN
      ;Okay, if the current was the last one then set the current pointer
      ;to be null. This means that no new routenode is added, the relevant
      ;pointers are cleared up and the process move on.
       CurrentPointer = Ptr_New()
    ENDELSE
  ;Free the pointer to the temp pointer, and then set the nextnode.pointer
  ;to point to the current pointer, as wel as setting the element to 0
    Ptr_Free, (*NextNode).Pointer
    (*NextNode).Pointer = Ptr_New(CurrentPointer)
    (*NextNode).Element = 0
 ENDIF ELSE BEGIN
   ;If we've not reached the end then increase the element number.
   ;Then set the current pointer appropriately. If the end has been
   ;reached then pointers will be invalid and it will all be trapped
   ;by the n_elements test on the next loop.
    IF (*NextNode).Element NE N_Elements(*((*NextNode).Pointer))-1 $
          THEN (*NextNode).Element = (*NextNode).Element + 1
    CurrentPointer = (*((*NextNode).Pointer))[(*NextNode).Element]

 ENDELSE
  ;Remove the previous node
```

```
    Route->Remove_Node, CurrentRouteNumber+1
    CurrentElement = 0
   ENDIF
       ENDIF
        ;Reset flag
     MoveBackOne = 0
 ENDWHILE
    Obj_Destroy, Route
ENDFOR
END
```

Next Program coyotelist__define.pro.....

```
;+
; NAME:
;   COYOTELIST__DEFINE
;
; PURPOSE:
;   The purpose of this program is to implement a list that
;   is linked in both the forward and backward directions. There
;   is no restriction as to what can be stored in a linked list
;   node. The linked list is implemented as an object.
;-

PRO COYOTELIST::DELETE_ALL_NODES

; This method deletes all of the nodes.

WHILE Ptr_Valid(self.head) DO BEGIN
    currentNode = *self.head
    Ptr_Free, currentNode.previous
    Ptr_Free, currentNode.item
    self.head = currentNode.next
ENDWHILE

    ; Update the count.
```

```
self.count = 0

END

 ;------------------------------------------------------------ -------------

PRO COYOTELIST::ADD_NODE, item

; This method adds an item to the tail of the list.

   ; Be sure you have an item to add.

IF N_Elements(item) EQ 0 THEN BEGIN
   ok = Dialog_Message('Must pass an ITEM to add to the list.')
   RETURN
ENDIF

IF self.count EQ 0 THEN BEGIN

      ; Create a new node.
   currentNode = Ptr_New({ COYOTELIST_NODE })

      ; Add the item to the node.
   (*currentNode).item = Ptr_New(item)

      ; The head and tail point to current node.
   self.head = currentNode
   self.tail = currentNode

      ; Update the node count.
   self.count = self.count + 1

ENDIF ELSE BEGIN

      ; Create a new node.
   currentNode = Ptr_New({ COYOTELIST_NODE })

      ; Set the next field of the previous node.
   (*self.tail).next = currentNode

      ; Add the item to the current node.
   (*currentNode).item = Ptr_New(item)

      ; Set the previous field to point to previous node.
   (*currentNode).previous = self.tail

      ; Update the tail field to point to current node.
   self.tail = currentNode
```

```
        ; Update the node count.
    self.count = self.count + 1
ENDELSE
END

 ;------------------------------------------------------------ -------------

PRO COYOTELIST::REMOVE_NODE, INDEX
 ;This removes the selected node or if it is an object reference, it searches
 ;through for that object and then deletes it. The object reference is only
 ;valid if the list is being used to store object references.

IF Obj_Valid(Index) THEN BEGIN
 ;They have given an object reference, so lets go find it

  ;loop through the data items until we find the right one

 flag = 0
 counter = -1
  ;start at head
 CurrentNode = self.head

 WHILE flag NE 1 AND counter NE self.count-1 DO BEGIN

  counter=counter+1

  IF Obj_Valid((*CurrentNode).item) THEN BEGIN

   IF Index EQ (*CurrentNode).item THEN flag = 1

  ENDIF

 ENDWHILE

IF flag EQ 1 THEN Index = counter ELSE rtb=rbtnb

ENDIF

 ;now we've got the number of the item they want to delete so lets get to that node

currentnode=self.head
FOR i=0, index-1 DO currentNode = (*currentNode).next

 ;now free up the data

Ptr_Free, (*currentNode).item
```

```
 ;now join up the previous pointers

IF index EQ 0 THEN BEGIN
 ;if they are deleting the first item and
 ;there is not only one item in the list

 IF self.count NE 1 THEN BEGIN
  (*((*currentNode).next)).previous = Ptr_New()
  self.head = (*(currentNode)).next

 ENDIF

ENDIF ELSE IF index EQ self.count-1 THEN BEGIN
 ;if they are deleting the last item

 (*((*currentNode).previous)).next = Ptr_New()
 self.tail = (*(currentNode)).previous

ENDIF ELSE BEGIN

 (*((*currentNode).previous)).next = (*currentNode).next
 (*((*currentNode).next)).previous = (*currentNode).previous

ENDELSE

Ptr_Free, currentNode

self.count=self.count-1

END

 ;------------------------------------------------------------ -------------

FUNCTION COYOTELIST::GET_ITEM, index

; This method returns a pointer to the information
; stored in the list. Ask for the item by number or
; order in the list (list numbers start at 0).

   ; Gets last item by default.

IF N_Params() EQ 0 THEN index = self.count - 1

    ; Make sure there are items in the list.

IF self.count EQ 0 THEN BEGIN
   ok = Dialog_Message('Nothing is currently stored in the list.')
   RETURN, Ptr_New()
```

```
ENDIF

IF index GT (self.count - 1) OR index LT 0 THEN BEGIN
   ok = Dialog_Message('Sorry. Requested node is not in list.')
   RETURN, Ptr_New()
ENDIF

    ; Start at the head of the list.

currentNode = self.head

    ; Find the item asked for by traversing the list.

FOR j=0, index-1 DO currentNode = (*currentNode).next

    ; Return the pointer to the item.

RETURN, (*currentNode).item
END

 ;------------------------------------------------------------ -------------

FUNCTION COYOTELIST::GET_NODE, index

; This method returns a pointer to the asked-for node.
; Ask for the node by number or order in the list
; (node numbers start at 0).

   ; Gets last node by default.

IF N_Params() EQ 0 THEN index = self.count - 1

    ; Make sure there are items in the list.

IF self.count EQ 0 THEN BEGIN
   ok = Dialog_Message('Nothing is currently stored in the list.')
   RETURN, Ptr_New()
ENDIF

IF index GT (self.count - 1) OR index LT 0 THEN BEGIN
   ok = Dialog_Message('Sorry. Requested node is not in list.')
   RETURN, Ptr_New()
ENDIF

    ; Start at the head of the list.

currentNode = self.head
```

```
    ; Find the item asked for by traversing the list.

FOR j=0, index-1 DO currentNode = (*currentNode).next

    ; Return the pointer to the node.

RETURN, currentNode
END

 ;--------------------------------------------------------- -------------

FUNCTION COYOTELIST::GET_COUNT

; This method returns the number of items in the list.

RETURN, self.count
END
 ;--------------------------------------------------------- -------------

PRO COYOTELIST::HELP, Print=print

; This method performs a HELP command on the items
; in the linked list. If the PRINT keyword is set, the
; data items are printed instead.

   ; Are there nodes to work with?

IF NOT Ptr_Valid(self.head) THEN BEGIN
    ok = Widget_Message('No nodes in Linked List.')
    RETURN
ENDIF

    ; First node.

currentNode = *self.head
IF Keyword_Set(print) THEN Print, *currentNode.item ELSE $
    Help, *currentNode.item

    ; The rest of the nodes. End of list indicated by null pointer.

WHILE currentNode.next NE Ptr_New() DO BEGIN
    nextNode = *currentNode.next
    IF Keyword_Set(print) THEN Print, *nextNode.item ELSE $
        Help, *nextNode.item
    currentNode = nextNode
ENDWHILE

END
```

```
;------------------------------------------------------------ -------------

PRO COYOTELIST::CLEANUP

; This method deletes all of the nodes and cleans up
; the objects pointers.

self->Delete_All_Nodes
Ptr_Free, self.head
Ptr_Free, self.tail
END
;------------------------------------------------------------ -------------

FUNCTION COYOTELIST::INIT, item

; Initialize the linked list. Add an item if required.

IF N_Params() EQ 0 THEN RETURN, 1
self->Add_Node, item
RETURN, 1
END

;------------------------------------------------------------ -------------

PRO COYOTELIST__DEFINE

; The implementation of a COYOTELIST object.

  struct = { COYOTELIST, $        ; The COYOTELIST object.
           head:Ptr_New(), $     ; A pointer to the first node.
           tail:Ptr_New(), $     ; A pointer to the last node.
           count:0L $            ; The number of nodes in the list.
           }

  struct = { COYOTELIST_NODE, $    ; The COYOTELIST NODE structure.
           previous:Ptr_New(), $ ; A pointer to the previous node.
           item:Ptr_New(), $     ; A pointer to the data item.
           next:Ptr_New()  $     ; A pointer to the next node.
           }

END
;------------------------------------------------------------ -------------
```