

Hi,

this is more and more fun! Please find attached a base object definition which contains a few useful features that you can nicely inherit in anything else:

The object contains a NAME and a UVALUE field which can be set with the Init method (where NAME must be a string and UVALUE can be anything). It contains a method for displaying an error message either as a dialog box or on the log screen (controlled by the no_dialog keyword), and -- and this I am most proud of -- it contains a method for duplicating itself.

And this works for any object that inherits from it. Thanks to James Tappin who gave me the idea to create a structure with the object fields (stru = { object }), and to JD Smith who had the Struct_Assign idea.

What the Copy() method does is to create a copy of self as simply as

```
;; Initialize an empty object of the same type as self
clone = Obj_New( Obj_Class(self) )
```

```
;; Do a structure assignment to copy all fields verbatim
Struct_Assign, self, clone, verbose=verbose
```

But that leaves you with the problem that pointers and object references in the new object still point to the data and objects from the old object. So I apply James' trick with a little variant:

```
ok = Execute('ostru = {'+Obj_Class(self)+'}')
```

in order to get a structure equivalent of the current object. Then I loop through the structure tags and whenever there is a pointer, I call a method CopyPointers which does just what it says (took me a while to avoid the traps of pointer arrays and pointers pointing to object references, but it is still remarkably short, I think). Object references are handled by calling the Copy method of the referenced object. You can control some of this with keywords (no_pointers, no_objects, free_pointers, or free_objects).

The code is heavily documented, so there is hope even if you don't understand what I was talking about here ;-)

In order to use the base object in a meaningful way, you should inherit from it. To show you how this is done, I also attach a class named MGS_BaseVariable which allows consistent treatment of (scientific) data. Just look for all occurrences of MGS_BaseObject in the source of mgs_basevariable__define.pro, and you see what is needed. Here is an example how to use this all (you can find

this at the end of mgs_basevariable__define.pro):

PRO example

```
lon = obj_new('MGS_BaseVariable', findgen(128), name='LON' )
lat = obj_new('MGS_BaseVariable', findgen(64), name='LAT' )

var1 = obj_new('MGS_BaseVariable', dist(128,64), lon, lat, $
              name='Ozone', long_name='Ozone concentration',
units='ppb')

var2 = var1->Copy()
var2->SetProperty, name='CO', long_name='Carbon monoxide
concentration'

;; retrieve the name property from both variables to check they are
different:
var1->getproperty,name=name1
var2->getproperty,name=name2
print,'Name 1 = ',name1,' Name2 = ',name2

;; invert the dimensions of the first variable
lon->setproperty,data=reverse(findgen(128))
lat->setproperty,data=reverse(findgen(64))

;; print the first element of lon and lat for both variables:
var1->getproperty,dim1=d1,dim2=d2
d1->getData,lon1
d2->getData,lat1
var2->getproperty,dim1=d1,dim2=d2
d1->getData,lon2
d2->getData,lat2
print,'First elements of : '
print,'lon1 = ',lon1[0],' lon2 = ',lon2[0]
print,'lat1 = ',lat1[0],' lat2 = ',lat2[0]

;; clean up
Obj_Destroy, d1 ; copy of lon in var2
Obj_Destroy, d2 ; copy of lat in var2
Obj_Destroy, lon
Obj_Destroy, lat

Obj_Destroy, var1
Obj_Destroy, var2
```

END

Cheers,
Martin

```
--
[[
[[ Dr. Martin Schultz  Max-Planck-Institut fuer Meteorologie  [[
[[      Bundesstr. 55, 20146 Hamburg      [[
[[      phone: +49 40 41173-308      [[
[[      fax:  +49 40 41173-298      [[
[[ martin.schultz@dkrz.de      [[
[[
]]
]]
;+
; NAME:
;   mgs_baseobject (object)
;
; PURPOSE:
;   This object provides basic functionality in terms of error
;   handling and allows the user to store arbitrary information in
;   the UValue field. It provides the standard Init, Cleanup,
;   GetProperty, and SetProperty methods which will likely have to
;   be extended for derived objects. The following additional
;   methods are provided:
;   Undefine : render a variable argument as undefined
;   ErrorMessage : display an error message as dialog or on the log
;                  screen
;   Copy      : create a clone of the object and all its contents
;   Validate  : a dummy method for data consistency checks
;
; AUTHOR:
;
;   Dr. Martin Schultz
;   Max-Planck-Institut fuer Meteorologie
;   Bundesstr. 55, D-20146 Hamburg
;   email: martin.schultz@dkrz.de
;
; CATEGORY:
;   General object programming
;
; CALLING SEQUENCE:
;   This object will generally not be instanced in itself. To inherit
;   from it, add { ..., INHERITS MGS_BaseObject } to your
;   object__define procedure, call self->MGS_BaseObject::Init at the
;   beginning of your new object's Init method, and
;   self->MGS_BaseObject::Cleanup at the end of your new object's Cleanup method.
;
; EXAMPLE:
;   See the MGS_BaseVariable and MGS_Variable objects.
;
; MODIFICATION HISTORY:
```

```

; Martin G. Schultz, 25 May 2000: VERSION 1.00
; mgs, 07 Jun 2000 : - now allows for multiple line error messages
; mgs, 09 Jun 2000 : - added Copy and CopyPointers methods
;
;
;
;#####
;
; LICENSE
;
; This software is OSI Certified Open Source Software.
; OSI Certified is a certification mark of the Open Source Initiative.
;
; Copyright ©½ 2000 Martin Schultz
;
; This software is provided "as-is", without any express or
; implied warranty. In no event will the authors be held liable
; for any damages arising from the use of this software.
;
; Permission is granted to anyone to use this software for any
; purpose, including commercial applications, and to alter it and
; redistribute it freely, subject to the following restrictions:
;
; 1. The origin of this software must not be misrepresented; you must
;    not claim you wrote the original software. If you use this software
;    in a product, an acknowledgment in the product documentation
;    would be appreciated, but is not required.
;
; 2. Altered source versions must be plainly marked as such, and must
;    not be misrepresented as being the original software.
;
; 3. This notice may not be removed or altered from any source distribution.
;
; For more information on Open Source Software, visit the Open Source
; web site: http://www.opensource.org.
;
;#####

; =====
=====
; Base object methods:
;   Undefine    : render an argument variable undefined
;   ErrorMessage: display an error message as dialog or screen message
;   Validate     : make sure, object data is consistent

```

```

; -----
; Undefine:
; This method applies a trick published by Andrew Coole, Adelaide, AUS
; to render an undefined variable from within an IDL subroutine.

pro MGS_BaseObject::Undefine, arg

    if (n_elements(arg) eq 0) then return ; already undefined
    tempvar = SIZE(TEMPORARY(arg))

end

; -----
; ErrorMessage:
; This method displays an error (or warning) message. Default is to
; display a Dialog_Message, unless the object's no_dialog flag is set
; or the no_dialog keyword is provided at the call of this method. IDL
; versions prior to 5.3 need also ensure that the current device
; supports widgets.
; This method is particularly useful in Catch error handlers. See
; other methods of this object for examples.
; Note that this object's Init method cannot call ErrorMessage,
; however if you establish an error handler after the
; self->MGS_BaseObject::Init call, you should be able to use this method.

pro MGS_BaseObject::ErrorMessage, $
    thisMessage, $ ; The text of the error message
    no_dialog=no_dialog, $ ; Don't display a dialog
    No_ErrorState=No_ErrorState, $ ; Don't display the error status
    Warning=Warning ; Display a warning instead of an error

;; establish own error handler
;; internalMessage = 'Strange internal error.'
CATCH, theError
IF theError NE 0 THEN BEGIN
    CATCH, /Cancel
    Message, internalMessage, /Continue
    RETURN
ENDIF

internalMessage = 'Strange internal error.'

;; determine whether dialog shall be used:
;; - default is object's no_dialog value
;; - this gets overwritten with local no_dialog keyword
;; - finally a test is made whether current device supports dialogs

```

```

;; at all (only necessary for IDL < 5.3)

Use_Dialog = ( self.no_dialog EQ 0 )
IF N_Elements(No_Dialog) GT 0 THEN Use_Dialog = not Keyword_Set(no_dialog)
IF FLOAT(!Version.Release) LT 5.3 AND (!D.FLAGS AND 65536L) EQ 0 THEN $
    Use_Dialog = 0 ; cannot display dialog on current device

;; Get calling routine's name
Help, Calls=CallStack
Caller = ( Str_Sep( StrCompress(CallStack[1]), " " ) )[0]
Caller = StrUpCase(Caller)
IF Caller EQ '$MAIN$' THEN Caller = " ELSE Caller = Caller + ': '

;; Compose message string
IF N_Elements(thisMessage) EQ 0 THEN thisMessage = "
IF StrTrim(thisMessage[0], 2) NE " THEN $
    theMessage = thisMessage $
ELSE $
    theMessage = 'Unspecified Error.'

;; Display error message as dialog
IF Use_Dialog THEN BEGIN
    IF N_Elements(theMessage) GT 1 THEN BEGIN
        IF NOT Keyword_Set(No_ErrorState) THEN $
            theMessage = [ Caller+theMessage[0], $
                theMessage[1:*], $
                !Error_State.Msg ]
    ENDIF ELSE BEGIN
        IF NOT Keyword_Set(No_ErrorState) THEN $
            theMessage = [ Caller+theMessage, $
                !Error_State.Msg ]
    ENDELSE
    IF Keyword_Set(Warning) THEN $
        ok = Dialog_Message(theMessage) $
    ELSE $
        ok = Dialog_Message(theMessage, /Error)
ENDIF ELSE BEGIN
    ;; Display error message as text
    Message,theMessage[0], /Info
    IF N_Elements(theMessage) GT 1 THEN BEGIN
        FOR i=1L, N_Elements(theMessage)-1 DO $
            Message,theMessage[i], /Info, /NoName
    ENDIF
    IF NOT Keyword_Set(No_ErrorState) THEN $
        Print, !Error_State.Msg
ENDELSE

;; Clear error status

```

Message, /Reset

end

```
; -----  
; CopyPointers:  
; This method returns a physical copy of the pointer or pointer  
; array given as argument. If a pointer contains other pointers, the  
; method is called recursively.  
; If the pointer or pointer array contains object references, an  
; attempt is made to call that object's Copy method so that the object  
; is copied physically rather than only by reference. If an object has  
; no Copy method (or if you want to preserve the link), set the  
; no_objects keyword.
```

FUNCTION MGS_BaseObject::CopyPointers, p, no_objects=no_objects

```
;; Set dummy default  
result = Ptr_New()  
  
;; Find out if p is a pointer array  
N = N_Elements(p)  
IF N GT 1 THEN result = PtrArr(N)  
  
;; Go through elements of p and duplicate them  
FOR i=0L, N-1 DO BEGIN  
  IF Ptr_Valid( p[i] ) THEN BEGIN  
    ;; Check if pointer contains another pointer  
    type = Size( *p[i], /TName )  
    IF type EQ 'POINTER' THEN $          ; recursive pointer copy  
      result[i] = Ptr_New( self->CopyPointers( *p[i] ) ) $  
    ELSE IF type EQ 'OBJREF' AND not Keyword_Set(no_objects) THEN BEGIN  
      result[i] = Ptr_New( (*p[i])->Copy() ) ; physical copy of object  
    ENDIF ELSE $  
      result[i] = Ptr_New( *p[i] )      ; copy of object reference  
    ENDIF  
  ENDFOR  
  
  RETURN, result  
END
```

```
; -----  
; Copy:  
; This method makes a copy of the object itself (cloning). The result  
; is an empty (invalid) object reference if something goes  
; wrong. Pointers are copied explicitly unless the no_pointers
```

```
; keyword is set. Objects are copied via recursive calls to their Copy
; method (which means they should have one). If you don't want to copy
; the objects but only the object references, use the no_objects
; keyword. With the free_pointers and free_objects keywords you can
; instead delete the pointer and object references from the clone.
;
```

```
FUNCTION MGS_BaseObject::Copy, no_pointers=no_pointers, $
    no_objects=no_objects, $
    free_pointers=free_pointers, $
    free_objects=free_objects, $
    verbose=verbose
```

```
:: set default result
dummy = Obj_New()
```

```
Catch, theError
    Catch, /Cancel
IF theError NE 0 THEN BEGIN
    Catch, /Cancel
    self->ErrorMessage, 'Error building object clone!'
    IF Obj_Valid(clone) THEN Obj_Destroy, clone
    RETURN, dummy
ENDIF
```

```
:: Initialize an empty object of the same type as self
clone = Obj_New( Obj_Class(self) )
```

```
:: Do a structure assignment to copy all fields verbatim
Struct_Assign, self, clone, verbose=verbose
```

```
:: Now apply a trick to get a structure with the current
;; object's tag names and types
;; (courtesy James Tappin)
ok = Execute('ostru = {'+Obj_Class(self)+'}')
IF not ok THEN RETURN, dummy
```

```
:: Go through tags and check which ones are pointers or objects:
;; these must be copied explicitly
IF Keyword_Set(verbose) THEN help,ostru,/stru
```

```
:: Pointers
FOR i=0L, N_Tags(ostru)-1 DO BEGIN
    tname = Size(ostru.(i), /TName)
    ;; Copy pointer contents or free them
    IF tname EQ 'POINTER' THEN BEGIN
        IF Keyword_Set(free_pointers) THEN BEGIN
            clone.(i) = Ptr_New()
```

```

ENDIF ELSE IF not Keyword_Set(no_pointers) THEN BEGIN
  IF Keyword_Set(verbose) THEN $
    print,'Copying pointer ',(tag_names(ostru))[i],'...'
    clone.(i) = self->CopyPointers( self.(i), no_objects=no_objects )
  ENDIF
ENDIF
;;
ENDFOR

```

```

;; Objects
FOR i=0L, N_Tags(ostru)-1 DO BEGIN
  tname = Size(ostru.(i), /TName)
  ;; Copy pointer contents or free them
  IF tname EQ 'OBJREF' THEN BEGIN
    IF Keyword_Set(free_objects) THEN BEGIN
      clone.(i) = Obj_New()
    ENDIF ELSE IF not Keyword_Set(no_objects) THEN BEGIN
      IF Keyword_Set(verbose) THEN $
        print,'Copying object ',(tag_names(ostru))[i],'...'
      IF Obj_Valid( clone.(i) ) THEN $
        clone.(i) = self.(i)->Copy()
      ENDIF
    ENDIF
  ENDIF
  ;;
ENDFOR

```

```

IF Keyword_Set(verbose) THEN print,'Done.'

```

```

RETURN, clone

```

```

END

```

```

; -----
; Validate:
; This method is just a dummy for derived objects

```

```

pro MGS_BaseObject::Validate

```

```

; help,self ; #####

```

```

end

```

```

; =====
; =====
; Standard object methods:
;   GetProperty : retrieve object values
;   SetProperty : set object values
;   Cleanup    : free object pointers and destroy

```

```

;   Init       : initialize object

; -----
; SetProperty:
; This method extracts specific object values and returns them to the
; user. Derived objects should overwrite and extend this method to
; return the extra information stored in them.

pro MGS_BaseObject::GetProperty, $
    name=name,      $ ; The variable name
    uvalue=uvalue,  $ ; A user defined value
    _Ref_Extra=extra

    Catch, theError
    IF theError NE 0 THEN BEGIN
        Catch, /Cancel
        self->ErrorMessage, !Error_State.Msg
        RETURN
    ENDIF

    name = self.name

; Handle pointer values
    IF Arg_Present(uvalue) THEN BEGIN
help,self.uvalue
        IF Ptr_Valid(self.uvalue) THEN $
            uvalue = *self.uvalue $
        ELSE $
            self->undefine, uvalue
        ENDIF
    END

end

; -----
; SetProperty:
; This method sets specific object values. Derived objects should
; overwrite and extend this method to allow storing additional information.

pro MGS_BaseObject::SetProperty, $
    name=name, $      ; The variable name
    no_copy=no_copy,  $ ; No_copy flag for ptr_new (data and uvalue)
    no_dialog=no_dialog, $ ; Flag to indicate if interactive dialogs are used
    uvalue=uvalue,    $ ; A user defined value
    _Ref_Extra=extra

    Catch, theError

```

```

IF theError NE 0 THEN BEGIN
    Catch, /Cancel
    self->ErrorMessage, !Error_State.Msg
    RETURN
ENDIF

```

```

;; Test arguments
IF N_Elements(name) GT 0 THEN self.name = name
IF N_Elements(no_dialog) GT 0 THEN self.no_dialog = Keyword_Set(no_dialog)

```

```

IF N_Elements(uvalue) GT 0 THEN BEGIN
    IF Ptr_Valid(self.uvalue) THEN Ptr_Free, self.uvalue
    self.uvalue = Ptr_New(uvalue)
ENDIF

```

```

; make sure everything is ok
self->Validate

```

```

end

```

```

; -----
; Cleanup:
; This method frees all data stored in the object. Derived objects
; should call self->MGS_BaseObject::Cleanup at the end of their own Cleanup
; method.

```

```

pro MGS_BaseObject::Cleanup

```

```

    IF Ptr_Valid(self.uvalue) THEN Ptr_Free, self.uvalue

```

```

end

```

```

; -----
; Init:
; This method initializes the object values. Derived objects should
; call self->MGS_BaseObject::Init at the beginning of their own Init method
; as in:
;   IF not self->MGS_BaseObject::Init(name=name,...) THEN RETURN
; Note that you can use _REF_EXTRA keywords to handle the keywords for
; the MGS_BaseObject transparently.

```

```

function MGS_BaseObject::Init, $
    name=name, $      ; The object name
    uvalue=uvalue,    $ ; A user defined value
    no_copy=no_copy,  $ ; No_Copy flag for ptr_new
    no_dialog=no_dialog, $ ; Flag indicating use of interactive dialogs
    _Ref_Extra=extra   ; For future additions

```

```

Catch, theError
IF theError NE 0 THEN BEGIN
  Catch, /Cancel
  IF Keyword_Set(no_dialog) THEN BEGIN
    theMessage = 'Error initializing object: '+ !Error_State.Msg
    Message, theMessage, /Continue
  ENDIF ELSE BEGIN
    theMessage = [ Routine_Name(), !Error_State.Msg ]
    ok = Dialog_Message(theMessage)
  ENDELSE
  RETURN, 0
ENDIF

;; Test arguments
IF N_Elements(name) EQ 0 THEN name = "

no_copy = Keyword_Set(no_copy)
no_dialog = Keyword_Set(no_dialog)

;; populate object
self.name = name
self.no_dialog = no_dialog

IF N_Elements(uvalue) GT 0 THEN self.uvalue = Ptr_New(uvalue, no_copy=no_copy) $
ELSE self.uvalue = Ptr_New()

; make sure everything is ok
self->MGS_BaseObject::Validate

; reset error state variable
Message, /Reset

return, 1
end

; -----
; This is the object definition. Derived objects should create a new
; structure and append these fields via the INHERITS MGS_BaseObject
; syntax.

pro MGS_BaseObject__Define

  objectClass = { mgs_baseobject,    $ ; The object class
    name      : ",    $ ; The object name
    no_dialog : 0,    $ ; Flag indicating use of interactive dialogs
    uvalue    : ptr_new() $ ; A user defined value
  }

```

end

```
;+
; NAME:
;   mgs_basevariable (object)
;
; PURPOSE:
;   This object stores (scientific) data and the information
;   relevant for plotting and (simple) arithmetic analysis.
;   It provides methods for extracting data and computing
;   basic statistics.
;   It inherits from MGS_BaseObject which provides basic
;   functionality for displaying error messages and storing a user
;   value.
;   The BaseVariable object does not contain information or methods
;   for treatment of missing values or limited data range. These
;   are added in the definition of MGS_Variable.
;   Data of REAL, DOUBLE, LONG, INT, or BYTE type is fully supported,
;   for other types BaseVariable serves as a container object only
;   (i.e. you cannot compute a mean, extract a range etc.).
;
; AUTHOR:
;
;   Dr. Martin Schultz
;   Max-Planck-Institut fuer Meteorologie
;   Bundesstr. 55, D-20146 Hamburg
;   email: martin.schultz@dkrz.de
;
; CATEGORY:
;
;
; CALLING SEQUENCE:
;
;
;
; EXAMPLE:
;
;
;
; MODIFICATION HISTORY:
;   Martin G. Schultz, 30 May 2000: VERSION 1.00
;
;
; -
;
; #####
```

```

; LICENSE
;
; This software is OSI Certified Open Source Software.
; OSI Certified is a certification mark of the Open Source Initiative.
;
; Copyright © 2000 Martin Schultz
;
; This software is provided "as-is", without any express or
; implied warranty. In no event will the authors be held liable
; for any damages arising from the use of this software.
;
; Permission is granted to anyone to use this software for any
; purpose, including commercial applications, and to alter it and
; redistribute it freely, subject to the following restrictions:
;
; 1. The origin of this software must not be misrepresented; you must
;    not claim you wrote the original software. If you use this software
;    in a product, an acknowledgment in the product documentation
;    would be appreciated, but is not required.
;
; 2. Altered source versions must be plainly marked as such, and must
;    not be misrepresented as being the original software.
;
; 3. This notice may not be removed or altered from any source distribution.
;
; For more information on Open Source Software, visit the Open Source
; web site: http://www.opensource.org.
;
; #####
;
; -----
;
; =====
; =====
; Utility methods:
;   ConvertType : Convert a data argument to a specific type
;   ExtractHyperslice : Return data array with one dimension fixed
;   Mean       : Compute the mean of the data stored in the object
;
; -----
; ConvertType:
;   This method accepts a data argument and converts it to a specified
;   data type. The data type can be given either as string with the
;   tname keyword or by setting one of the logical flags double, float,
;   long, integer, or byte, or by specifying an (integer) type value
;   (see IDL online help on data type values). If the self keyword is
;   set, the object's internal data array is converted. In this case, no

```

```

; data argument must be present.
; The truncate, round, and clip keywords determine, how data are
; converted from a "higher" to a "lower" variable type. The truncate
; keyword is purely for readability, truncation is IDL's default for
; type conversion. Alternatively, you can round float data to the
; nearest (long) integer. Be aware that you can end up with unreasonable
; values in both cases, unless the clip keyword is set (example:
; "print, fix(32768L)" yields -32768). Clip will ensure that the data
; does not extend beyond the valid data range given by the new data type.
; The minimum and maximum possible (valid) data values for the new
; type can be returned via the min_valid and max_valid keywords. In
; order to avoid machine specific differences, the limit for double
; precision values is set to 1.D300, and the limit for float is set to
; 1.E31.

```

```

pro MGS_BaseVariable::ConvertType, $
    data,          $ ; The data to be converted
    self=selfdata, $ ; Convert object's own data
    tname=tname,   $ ; A string containing the new data type name
    double=double, $ ; Convert data to double
    float=float,   $ ; Convert data to float
    long=long,     $ ; Convert data to long
    integer=integer, $ ; Convert data to integer
    byte=byte,     $ ; Convert data to byte
    type=type,     $ ; Convert to type number (see online help)
    truncate=truncate, $ ; Truncate the data if necessary (default)
    round=round,   $ ; Round the data if converting to integer type
    clip=clip,     $ ; Clip data to valid range before converting
    min_valid=min_valid, $ ; Return the minimum possible value for the new type
    max_valid=max_valid ; Return the maximum possible value for the new type

```

```

Catch, theError
IF theError NE 0 THEN BEGIN
    Catch, /Cancel
    self->ErrorMessage, !Error_State.Msg
    RETURN
ENDIF

```

```

;; Determine new data type
IF N_Elements(tname) GT 0 THEN BEGIN
    newtype = StrUpCase(tname)
ENDIF ELSE BEGIN
    IF Keyword_Set(double) THEN newtype = 'DOUBLE'
    IF Keyword_Set(float) THEN newtype = 'FLOAT'
    IF Keyword_Set(long) THEN newtype = 'LONG'
    IF Keyword_Set(integer) THEN newtype = 'INTEGER'

```

```

IF Keyword_Set(byte) THEN newtype = 'BYTE'
IF N_Elements(type) GT 0 THEN BEGIN
    typedef = [ 'BYTE', 'INT', 'LONG', 'FLOAT', 'DOUBLE' ]
    newtype = typedef[ ( type[0]-1 ) > 0 < 4 ]
ENDIF
ENDELSE

;; Unsupported type, return
IF N_Elements(newtype) EQ 0 THEN RETURN

;; Set minimum and maximum valid values
;; Need to apply trick for most negative integers
CASE newtype OF
    'DOUBLE' : BEGIN
        min_valid = -1.0D300
        max_valid = +1.0D300
        typeid = 5
    END
    'FLOAT' : BEGIN
        min_valid = -1.031
        max_valid = +1.031
        typeid = 4
    END
    'LONG' : BEGIN
        min_valid = long(-2LL^31) ; -2147483648L
        max_valid = +2147483647L
        typeid = 3
    END
    'INT' : BEGIN
        min_valid = fix(-2L^15) ; -32768
        max_valid = +32767
        typeid = 2
    END
    'BYTE' : BEGIN
        min_valid = 0B
        max_valid = 255B
        typeid = 1
    END
    ELSE : RETURN ; unsupported type set via tname keyword
ENDCASE

;; If no data is passed, look for the self keyword. If set,
;; retrieve object's internal data
IF N_Elements(data) EQ 0 THEN BEGIN
    IF Keyword_Set(selfdata) THEN self->GetData, data
    IF N_Elements(data) EQ 0 THEN RETURN
ENDIF

```

```

;; Determine current data type
datatype = Size(data, /TYPE)

;; If new data type is "larger" than or equal to the original type,
;; the type conversion is trivial.
IF ( datatype LE typeid ) THEN BEGIN
    CASE newtype OF
        'DOUBLE' : data = Double(temporary(data))
        'FLOAT'  : data = Float(temporary(data))
        'LONG'   : data = Long(temporary(data))
        'INT'    : data = Fix(temporary(data))
        'BYTE'   : data = Byte(temporary(data))
    ENDCASE
    RETURN
ENDIF

;; If clipping requested, get out-of-bounds indices before
;; converting.
IF Keyword_Set(clip) THEN BEGIN
    low = Where( data LT min_valid, lcount )
    high = Where( data GT max_valid, hcount )
ENDIF ELSE BEGIN
    lcount = 0L
    hcount = 0L
ENDELSE

;; Round or truncate data (or convert double to float)
IF Keyword_Set(round) THEN BEGIN
    CASE newtype OF
        'DOUBLE' : data = Double(temporary(data)) ; shouldn't occur
        'FLOAT'  : data = Float(temporary(data))
        'LONG'   : data = Round(temporary(data))
        'INT'    : data = Fix(Round(temporary(data)))
        'BYTE'   : data = Byte(Round(temporary(data)))
    ENDCASE
ENDIF ELSE BEGIN
    CASE newtype OF
        'DOUBLE' : data = Double(temporary(data)) ; shouldn't occur
        'FLOAT'  : data = Float(temporary(data))
        'LONG'   : data = Long(temporary(data))
        'INT'    : data = Fix(temporary(data))
        'BYTE'   : data = Byte(temporary(data))
    ENDCASE
ENDELSE

;; Apply clipping where necessary
IF lcount GT 0L THEN data[low] = min_valid
IF hcount GT 0L THEN data[high] = max_valid

```

```

;; If self keyword is set, store data back in object
IF Keyword_Set(selfdata) THEN BEGIN
    Ptr_Free, self.data
    self.data = Ptr_New(data)
    self.tname = Size(data, /Tname)
ENDIF

```

end

```

; -----
; ExtractHyperslice:
; This method returns an N-1 dimensional subarray of the data
; argument which is extracted by fixing the given dimension to the
; given index value. If the self keyword is set, the hyperslice is
; taken from the object's data.
; If data (or the objects data field) contains no data, an undefined
; variable is returned.

```

```

function MGS_BaseVariable::ExtractHyperslice, $
    data,      $ ; The data array from which to extract a slice
    dimension, $ ; The dimension to keep fixed
    index,     $ ; The index value for the fixed dimension
    self=selfdata ; Flag to indicate use of object's own data

```

```

self->Undefine, result

```

```

Catch, theError
IF theError NE 0 THEN BEGIN
    Catch, /Cancel
    self->ErrorMessage, !Error_State.Msg
    RETURN, result
ENDIF

```

```

;; Check if all three arguments are present and valid
IF N_PARAMS() NE 3 THEN BEGIN
    self->ErrorMessage, 'Usage: obj->ExtractHyperslice, data, dimension, index'
    RETURN, result
ENDIF
IF N_Elements(dimension) NE 1 THEN BEGIN
    self->ErrorMessage, 'Dimension argument must be scalar!'
    RETURN, result
ENDIF
IF N_Elements(index) NE 1 THEN BEGIN
    self->ErrorMessage, 'Index argument must be scalar!'
    RETURN, result

```

ENDIF

;; If no data is passed, look for the self keyword. If set,
;; retrieve object's internal data

IF N_Elements(data) EQ 0 THEN BEGIN

IF Keyword_Set(selfdata) THEN self->GetData, thisdata

IF N_Elements(thisdata) EQ 0 THEN RETURN, result

ENDIF ELSE BEGIN

thisdata = data

ENDELSE

;; Get dimensions of data array

ndims = Size(thisdata, /N_Dimensions)

dims = Size(thisdata, /Dimensions)

;; Make sure dimension and index have correct type

thisdim = Fix(dimension > 1) ; automatically convert 0 to 1

thisindex = Long(index)

;; Make sure, dimension and index are valid

IF thisdim GT ndims THEN BEGIN

self->ErrorMessage, 'Invalid dimension ('+StrTrim(thisdim,2)+') !'

RETURN, result

ENDIF

IF thisindex LT 0L OR thisindex GE dims[thisdim-1] THEN BEGIN

self->ErrorMessage, 'Invalid index ('+StrTrim(thisindex,2)+') !'

RETURN, result

ENDIF

;; OK, go ahead with the extraction

CASE thisdim OF

1 : result = data[thisindex,*,*,*,*,*]

2 : result = data[* ,thisindex,*,*,*,*]

3 : result = data[* ,*,thisindex,*,*,*]

4 : result = data[* ,*,*,thisindex,*,*]

5 : result = data[* ,*,*,*,thisindex,*]

6 : result = data[* ,*,*,*,*,thisindex,*]

7 : result = data[* ,*,*,*,*,*,thisindex,*]

8 : result = data[* ,*,*,*,*,*,*,thisindex]

ENDCASE

RETURN, result

end

; -----

; Mean:

; This method computes the mean of the variable. The dimension

; argument allows to compute the mean for all elements of this dimension,

; i.e. in this case mean returns a vector rather than a scalar.

```
function MGS_BaseVariable::Mean, $
    dimension    ; [optional] Keep this dimension fixed

result = !Values.F_NaN

Catch, theError
IF theError NE 0 THEN BEGIN
    Catch, /Cancel
    self->ErrorMessage, !Error_State.Msg
    RETURN, result
ENDIF

;; If no data is loaded, return
self->GetData, data
IF N_Elements(data) EQ 0 THEN BEGIN
    ;; self->ErrorMessage, 'No data stored in variable '+self.name
    RETURN, result
ENDIF

;; Check data type
IF (self.tname NE 'DOUBLE' $
    AND self.tname NE 'FLOAT' $
    AND self.tname NE 'LONG' $
    AND self.tname NE 'INT' $
    AND self.tname NE 'BYTE') THEN BEGIN
    self->ErrorMessage, 'Data type '+self.tname+ $
        ' does not permit mathematical operations.'
    RETURN, result
ENDIF

;; If dimension argument is passed, evaluate mean across that
;; dimension
IF N_Elements(dimension) EQ 1 THEN BEGIN
    thisdim = Fix(dimension > 1) ; automatically convert 0 to 1

    ;; Make sure dimensions are in range
    ndims = Size(data, /N_Dimensions)
    dims = Size(data, /Dimensions)
    IF thisdim GT ndims THEN BEGIN
        self->ErrorMessage, 'Invalid dimension attribute!'
        RETURN, result
    ENDIF
    ;; Create result array of correct type (float or double) and
    ;; size
```

```

    result = fltarr(dims[thisdim-1])
    IF self.tname EQ 'DOUBLE' THEN result = Double(result)

    ;; Compute the respective mean values
    FOR i=0L, dims[thisdim-1]-1 DO $
        result[i] = Mean( self->ExtractHyperslice(data, thisdim, i) )

    ENDIF ELSE BEGIN
        result = Mean(*self.data)
    ENDELSE

    RETURN, result

end

; =====
; =====
; Methods for data manipulation:
;   DeleteData   : free data pointer and reset dimension values
;   ScaleData    : apply scaling factor and offset to data values

; -----
; DeleteData:
;   This method frees the data pointer and optionally resets the
;   dimension values

pro MGS_BaseVariable::DeleteData, $
    reset_dimensions=reset_dimensions

    Catch, theError
    IF theError NE 0 THEN BEGIN
        Catch, /Cancel
        self->ErrorMessage, !Error_State.Msg
        RETURN
    ENDIF

    IF Ptr_Valid(self.data) THEN Ptr_Free, self.data

    IF Keyword_Set(reset_dimensions) THEN BEGIN
        self.ndims = 0
        self.dims = 0L
    ENDIF

    ;; Don't touch dimvars for they may be hard to regenerate. Dimvars
    ;; can be deleted via the SetDimVar method.

end

```

```
; -----
; Scale:
; This method performs a linear operation data*factor+offset on the
; data stored in the object. The data type may change according to the
; IDL rules (e.g. scaling a LONG array with a FLOAT factor yields a
; FLOAT array). Setting the preserve_type flag will convert the final
; result back to the original type.
; Extra keywords are passed to the ConvertType method if the
; Preserve_Type keyword is set. Useful options include Truncate,
; Round, or Clip (see ConvertType method above).
```

```
pro MGS_BaseVariable::Scale, $
    factor=factor, $
    offset=offset, $
    preserve_type=preserve_type, $
    _Ref_Extra=extra
```

```
Catch, theError
IF theError NE 0 THEN BEGIN
    Catch, /Cancel
    self->ErrorMessage, !Error_State.Msg
    RETURN
ENDIF
```

```
;; Define defaults
CASE self.tname OF
    'DOUBLE' : BEGIN
        DefFactor = 1.0D
        DefOffset = 0.0D
    END
    'FLOAT' : BEGIN
        DefFactor = 1.0
        DefOffset = 0.0
    END
    'LONG' : BEGIN
        DefFactor = 1L
        DefOffset = 0L
    END
    'INT' : BEGIN
        DefFactor = 1
        DefOffset = 0
    END
    'BYTE' : BEGIN
        DefFactor = 1B
        DefOffset = 0B
    END
```

```
ELSE : RETURN    ; no scaling possible
ENDCASE
```

```
;; Keyword checking
IF N_Elements(factor) EQ 0 THEN factor = DefFactor
IF N_Elements(offset) EQ 0 THEN offset = DefOffset

;; Retrieve a local copy of all data and a vector of the valid data
;; indices
self->GetData, data, ok=ok
```

```
;; If no data is loaded, return
IF N_Elements(data) EQ 0 THEN RETURN
```

```
;; Convert data type if necessary
dtype = Size(data, /TYPE)
ftype = ( Size(factor, /TYPE) > Size(offset, /TYPE) > 1 ) < 5
```

```
IF ftype GT dtype THEN self->ConvertType, data, type=ftype
```

```
;; Scale valid data
;; *** Performance note: if it makes a difference, distinguish
;; between the case for N_Elements(ok) EQ N_Elements(data) and
;; otherwise, applying the scale factor and offset to the complete
;; array in the first case - or even directly to the pointer
;; content.
data[ok] = data[ok] * factor + offset
```

```
;; Adjust data type name or convert data back to old type if
;; preserve_type keyword is set
IF Keyword_Set(preserve_type) THEN BEGIN
    self->ConvertType, data, tname=self.tname, _Extra=extra
ENDIF ELSE BEGIN
    self.tname = Size(data, /TName)
ENDELSE
```

```
;; Store data back in object
(*self.data) = data
```

```
end
```

```
; =====
=====
; Methods for data retrieval and dimension handling:
;   GetData   : return data from the object, optionally filtered for
;               validity etc.
;   GetDimVar : return the values for a specific dimension
```

```

;   SetDimVar : set the values for a specific dimension

; -----
; GetData:
;   This method returns all data satisfying object specific validation
; and/or selection criteria. As default, all data is returned with no
; questions asked.
;   The finite_only keyword ensures that no invalid floating point data is
; returned. Unless the substitute keyword is set at the same time, the
; data are returned as a vector (resulting from a "Where" query).
;   The ok and bad keywords return the indices of all data which
; satisfy (don't satisfy) the "normal" filter criteria. I.e. even if
; the finite_only keyword is not set, ok will only contain the indices
; of finite data. This feature is used e.g. in the Scale method, where
; the scaling factor and offset are only applied to valid data.
;   Overwrite this method in derived objects to
; accomodate specific needs. As an example, take a look at
; the MGS_Variable object, where data are filtered according to the
; three attributes missing_value, min_valid, and max_valid.

pro MGS_BaseVariable::GetData, $
    data,          $ ; The valid data
    finite_only=finite_only, $ ; Return only valid numerical data
    substitute=substitute, $ ; A replacement value for invalid data
    ok=ok,          $ ; returns the index of valid data
    bad=bad,         $ ; returns the index of bad data
    _Ref_Extra=extra

Catch, theError
IF theError NE 0 THEN BEGIN
    Catch, /Cancel
    self->ErrorMessage, !Error_State.Msg
    RETURN
ENDIF

;; Undefine data as a fail-save return value
self->Undefine, data
self->Undefine, ok
self->Undefine, bad

;; If no data are stored, that's all
IF not Ptr_Valid( self.data ) THEN RETURN

;; Get local copy of data array
data = *self.data

;; Apply filter

```

```
;; Note: filter is applied directly if no substitute value is
;; given. The inverse filter operation is performed otherwise. The
;; dimensionality of data is only preserved if a substitute value
;; is given!
```

```
IF N_Elements(substitute) GT 0 THEN BEGIN
  IF Keyword_Set(finite_only) THEN BEGIN
    bad = Where( finite(data) EQ 0, count )
    IF count GT 0 THEN data[bad] = substitute[0]
  ENDIF
ENDIF ELSE BEGIN
  IF Keyword_Set(finite_only) THEN BEGIN
    ok = Where( finite(data), count )
    IF count GT 0 THEN data = data[ok]
  ENDIF
ENDELSE
```

```
;; Make sure ok and bad indices are returned if requested
IF Arg_Present(ok) AND N_Elements(ok) EQ 0 THEN $
  ok = Where( finite(data), count )
IF Arg_Present(bad) AND N_Elements(bad) EQ 0 THEN $
  bad = Where( finite(data) EQ 0, count )
```

```
end
```

```
; -----
; GetDimVar:
; This method retrieves the values of a specific dimension. A more
; detailed explanation is given for the corresponding SetDimVar method
; below.
```

```
pro MGS_BaseVariable::GetDimVar, $
    dimid,      $ ; The dimension number
    dimvals     ; The dimension values
```

```
Catch, theError
; Catch, /Cancel ; ### for DEBUGGING
IF theError NE 0 THEN BEGIN
  Catch, /Cancel
  self->ErrorMessage, !Error_State.Msg
  RETURN
ENDIF
```

```
thisDim = ( dimid > 1 ) < 8 ; limit range
```

```
;; If dimension values are stored get them, otherwise make sure to
;; return an undefined variable
```

```

IF Ptr_Valid( self.dimvars[thisDim-1] ) THEN BEGIN
    dimvals = *(self.dimvars[thisDim-1])
ENDIF ELSE BEGIN
    self->Undefine, dimvals
ENDELSE

```

end

```

; -----
; SetDimVar:
; This method stores a vector (or an array) with dimension values into
; the object. If the dimvals parameter is undefined, the values of the
; respective dimension are deleted from the object.
; This method allows access to individual dimension variables and is
; called from the Init and SetProperty methods. Note that SetDimVar
; accepts a numerical argument to indicate the dimension number, whereas
; the SetProperty method uses named keywords.
; Note: Dimension numbers < 1 or > 8 are silently clipped.
; Example:
; To store longitude values into the first dimension and latitude
; values into the second dimension, use:
; theVariable->SetDimVar, 1, lon
; theVariable->SetDimVar, 2, lat
; The same effect could be achieved via:
; theVariable->SetProperty, dim1=lon, dim2=lat

```

```

pro MGS_BaseVariable::SetDimVar, $
    dimid,          $ ; The dimension number
    dimvals          ; The dimension values

```

```

Catch, theError
IF theError NE 0 THEN BEGIN
    Catch, /Cancel
    self->ErrorMessage, !Error_State.Msg
    RETURN
ENDIF

```

```

thisDim = ( dimid > 1 ) < 8 ; limit range

```

```

IF Ptr_Valid( (self.dimvars)[thisDim-1] ) THEN Ptr_Free, (self.dimvars)[thisDim-1]

```

```

IF N_Elements(dimvals) GT 0 THEN BEGIN
    self.dimvars[thisDim-1] = Ptr_New(dimvals)
ENDIF ELSE BEGIN
    self.dimvars[thisDim-1] = Ptr_New()
ENDELSE

```

end

```
; =====  
=====
```

```
; Standard object methods:  
;   GetProperty : retrieve object values  
;   SetProperty : set object values  
;   Cleanup    : free object pointers and destroy  
;   Init       : initialize object
```

```
; -----  
; GetProperty:  
; This method extracts specific object values and returns them to the  
; user. Derived objects should overwrite and extend this method to  
; return the extra information stored in them.
```

```
pro MGS_BaseVariable::GetProperty, $  
    long_name=long_name, $ ; A more complete description of the variable  
    units=units,        $ ; The physical unit of the variable  
    tname=tname,        $ ; Data type name  
    data=data,          $ ; The data values  
    ndims=ndims,        $ ; The number of variable dimensions  
    dims=dims,          $ ; The size of each dimension  
    dimvars=dimvars,    $ ; Links to the dimension variables  
    dim1=dim1,          $ ; The values of the first dimension  
    dim2=dim2,          $ ; The values of the second dimension  
    dim3=dim3,          $ ; The values of the third dimension  
    dim4=dim4,          $ ; ...  
    dim5=dim5,          $  
    dim6=dim6,          $  
    dim7=dim7,          $  
    dim8=dim8,          $  
    _Ref_Extra=extra  
        ; Inherited keywords:  
        ; name      : The variable name  
        ; uvalue    : a user-defined value
```

```
Catch, theError  
IF theError NE 0 THEN BEGIN  
    Catch, /Cancel  
    self->ErrorMessage, !Error_State.Msg  
    RETURN  
ENDIF
```

```
;; Call GetProperty method from parent object  
self->MGS_BaseObject::GetProperty, _Extra=extra
```

```

long_name = self.long_name
units = self.units
tname = self.tname
ndims = self.ndims
dims = self.dims[0:ndims-1]
dimvars = self.dimvars

```

```

;; Handle pointer values
IF Arg_Present(data) THEN BEGIN
  IF Ptr_Valid(self.data) THEN $
    data = *self.data $
  ELSE $
    self->undefine, data
  ENDIF

```

```

;; Handle requests for dimension values
IF Arg_Present(dim1) THEN self->GetDimVar, 1, dim1
IF Arg_Present(dim2) THEN self->GetDimVar, 2, dim2
IF Arg_Present(dim3) THEN self->GetDimVar, 3, dim3
IF Arg_Present(dim4) THEN self->GetDimVar, 4, dim4
IF Arg_Present(dim5) THEN self->GetDimVar, 5, dim5
IF Arg_Present(dim6) THEN self->GetDimVar, 6, dim6
IF Arg_Present(dim7) THEN self->GetDimVar, 7, dim7
IF Arg_Present(dim8) THEN self->GetDimVar, 8, dim8

```

end

```

; -----
; SetProperty:
; This method sets specific object values. Derived objects should
; overwrite and extend this method to allow storing additional information.

```

```

pro MGS_BaseVariable::SetProperty, $
  long_name=long_name, $ ; A more complete description of the variable
  units=units,          $ ; The physical unit of the variable
  data=data,            $ ; The data values
  no_copy=no_copy,      $ ; No_copy flag for ptr_new (data and uvalue)
  dims=dims,            $ ; The size of each dimension
  dim1=dim1,            $ ; The values of the first dimension
  dim2=dim2,            $ ; The values of the second dimension
  dim3=dim3,            $ ; The values of the third dimension
  dim4=dim4,            $ ; ...
  dim5=dim5,            $
  dim6=dim6,            $
  dim7=dim7,            $
  dim8=dim8,            $
  _Ref_Extra=extra

```

```

; Inherited keywords:
; name      : The variable name
; no_dialog : Don't display
;           : interactive dialogs
; uvalue    : a user-defined value

```

```

Catch, theError
IF theError NE 0 THEN BEGIN
  Catch, /Cancel
  self->ErrorMessage, !Error_State.Msg
  RETURN
ENDIF

no_copy = Keyword_Set(no_copy)

;; Call SetProperty method of BaseObject
self->MGS_BaseObject::SetProperty, no_copy=no_copy, _Extra=extra

;; Test arguments
IF N_Elements(long_name) GT 0 THEN self.long_name = long_name
IF N_Elements(units) GT 0 THEN self.units = units

;; If data given, determine data type and dimensions
IF N_Elements(data) GT 0 THEN BEGIN
  self.tname = Size(data, /TName)
  self.ndims = Size(data, /N_Dimensions) > 1
  self.dims = Size(data, /Dimensions)
  IF Ptr_Valid(self.data) THEN Ptr_Free, self.data
  self.data = Ptr_New( data, no_copy=no_copy )
ENDIF ELSE BEGIN
;; otherwise set default dimension values
;; but only if no data is loaded
  IF not Ptr_Valid(self.data) THEN BEGIN
    IF N_Elements(tname) GT 0 THEN self.tname = StrUpCase(tname)
    IF N_Elements(dims) GT 0 THEN BEGIN
      self.ndims = N_Elements(dims) < 8
      self.dims[0:self.ndims-1] = dims[0:self.ndims-1]
    ENDIF
  ENDIF
ENDIF
ENDELSE

;; analyze dimension variables
;; no consistency check is made whether dimensions of dimvars agree
;; with data dimensions. This is done to allow storage of
;; irregularly gridded data where X and Y may be 2D arrays.
IF N_Elements(dim1) GT 0 THEN self->SetDimVar, 1, dim1
IF N_Elements(dim2) GT 0 THEN self->SetDimVar, 2, dim2
IF N_Elements(dim3) GT 0 THEN self->SetDimVar, 3, dim3

```

```

IF N_Elements(dim4) GT 0 THEN self->SetDimVar, 4, dim4
IF N_Elements(dim5) GT 0 THEN self->SetDimVar, 5, dim5
IF N_Elements(dim6) GT 0 THEN self->SetDimVar, 6, dim6
IF N_Elements(dim7) GT 0 THEN self->SetDimVar, 7, dim7
IF N_Elements(dim8) GT 0 THEN self->SetDimVar, 8, dim8

;; make sure everything is ok
;; (only needed if object contains its own Validate method)
; self->Validate

end

; -----
; Cleanup:
; This method frees all data stored in the object.

pro MGS_BaseVariable::Cleanup

    IF Ptr_Valid(self.data) THEN Ptr_Free, self.data
    FOR i=0,7 DO $
        IF Ptr_Valid(self.dimvars[i]) THEN Ptr_Free, self.dimvars[i]

;; Call parent's cleanup method
self->MGS_BaseObject::Cleanup

end

; -----
; Init:
; This method initializes the object values.
; Note that dim1-dim8 can hold any data type. Thus you can either
; specify values directly, or you can pass object references to other
; variable objects which contain the dimensional information.
; Example:
; lon=obj_new('mgs_basevariable', findgen(360), name='LON',long_name='Longitude')
; lat=obj_new('mgs_basevariable', findgen(180)-90., name='LAT',long_name='Latitude')
; var=obj_new('mgs_basevariable', data, name='Data', dim1=lon, dim2=lat)

function MGS_BaseVariable::Init, $
    data,          $ ; The data values
    dim1,          $ ; Values for first dimension (X)
    dim2,          $ ; Values for second dimension (Y)
    dim3,          $ ; Values for third dimension (Z or T)
    dim4,          $ ; ...
    dim5,          $
    dim6,          $
    dim7,          $
    dim8,          $

```

```

long_name=long_name, $ ; A more complete description of the variable
units=units,          $ ; The physical unit of the variable
no_copy=no_copy,      $ ; No_copy flag for ptr_new
dims=dims,            $ ; The size of each dimension
_Ref_Extra=extra      ; For future additions
                        ; Inherited keywords:
                        ; name      : The variable name
                        ; no_dialog : Don't display
                        ;           interactive dialogs
                        ; uvalue   : a user-defined value

```

```
no_copy = Keyword_Set(no_copy)
```

```
:: Initialize parent object first
```

```
IF not self->MGS_BaseObject::Init(no_copy=no_copy, _Extra=extra) THEN RETURN, 0
```

```
Catch, theError
```

```
IF theError NE 0 THEN BEGIN
```

```
    Catch, /Cancel
```

```
    self->ErrorMessage, 'Error initializing object!'
```

```
    RETURN, 0
```

```
ENDIF
```

```
:: Test arguments
```

```
IF N_Elements(long_name) EQ 0 THEN long_name = "
```

```
IF N_Elements(units) EQ 0 THEN units = "
```

```
:: If data given, determine data type and dimensions
```

```
IF N_Elements(data) GT 0 THEN BEGIN
```

```
    tname = Size(data, /TName)
```

```
    ndims = Size(data, /N_Dimensions) > 1
```

```
    dims = Size(data, /Dimensions)
```

```
    self.data = Ptr_New( data, no_copy=no_copy )
```

```
ENDIF ELSE BEGIN
```

```
:: otherwise set default dimension values
```

```
    IF N_Elements(tname) EQ 0 THEN tname = 'NOTHING'
```

```
    IF N_Elements(dims) EQ 0 THEN dims = 0L
```

```
    ndims = N_Elements(dims) < 8
```

```
    self.data = Ptr_New()
```

```
ENDELSE
```

```
:: populate object (data was done above)
```

```
self.long_name = long_name
```

```
self.units = units
```

```
self.tname = StrUpCase(tname)
```

```
self.ndims = ndims
```

```
self.dims = lonarr(8)
```

```
self.dims[0:ndims-1] = dims
```

```

;; analyze dimension variables
;; no consistency check is made whether dimensions of dimvars agree
;; with data dimensions. This is done to allow storage of
;; irregularly gridded data where X and Y may be 2D arrays.
self.dimvars = PtrArr(8)
IF N_Elements(dim1) GT 0 THEN self.dimvars[0] = Ptr_New(dim1)
IF N_Elements(dim2) GT 0 THEN self.dimvars[1] = Ptr_New(dim2)
; IF N_Elements(dim1) GT 0 THEN self->SetDimVar, 1, dim1
; IF N_Elements(dim2) GT 0 THEN self->SetDimVar, 2, dim2
IF N_Elements(dim3) GT 0 THEN self->SetDimVar, 3, dim3
IF N_Elements(dim4) GT 0 THEN self->SetDimVar, 4, dim4
IF N_Elements(dim5) GT 0 THEN self->SetDimVar, 5, dim5
IF N_Elements(dim6) GT 0 THEN self->SetDimVar, 6, dim6
IF N_Elements(dim7) GT 0 THEN self->SetDimVar, 7, dim7
IF N_Elements(dim8) GT 0 THEN self->SetDimVar, 8, dim8

;; make sure everything is ok
;; (only needed if object contains its own Validate method)
;; self->MGS_BaseVariable::Validate

;; Reset error state variable
Message, /Reset

return, 1
end

```

```

; -----
; This is the object definition. Derived objects should create a new
; structure and append these fields via the INHERITS MGS_BaseObject
; syntax.

```

```

pro MGS_BaseVariable__Define

```

```

objectClass = { MGS_BaseVariable, $ ; The object class
    long_name    : ", $ ; A more complete description of the variable
    units        : ", $ ; The physical unit of the variable
    tname        : ", $ ; The data type
    data         : ptr_new(), $ ; The data values
    ndims        : 0, $ ; The number of data dimensions
    dims         : lonarr(8), $ ; The size of each dimension
    dimvars      : ptrarr(8), $ ; The dimension values
    INHERITS MGS_BaseObject $

}

```

end

```
; -----  
; Example:  
; This example demonstrates a few features of the BaseVariable  
; object, notably how to make a copy and how to retrieve data from it.
```

PRO example

```
lon = obj_new('MGS_BaseVariable', findgen(128), name='LON' )  
lat = obj_new('MGS_BaseVariable', findgen(64), name='LAT' )  
  
var1 = obj_new('MGS_BaseVariable', dist(128,64), lon, lat, $  
              name='Ozone', long_name='Ozone concentration', units='ppb')  
  
var2 = var1->Copy()  
var2->SetProperty, name='CO', long_name='Carbon monoxide concentration'  
  
;; retrieve the name property from both variables to check they are different:  
var1->getproperty,name=name1  
var2->getproperty,name=name2  
print,'Name 1 = ',name1,' Name2 = ',name2  
  
;; invert the dimensions of the first variable  
lon->setproperty,data=reverse(findgen(128))  
lat->setproperty,data=reverse(findgen(64))  
  
;; print the first element of lon and lat for both variables:  
var1->getproperty,dim1=d1,dim2=d2  
d1->getData,lon1  
d2->getData,lat1  
var2->getproperty,dim1=d1,dim2=d2  
d1->getData,lon2  
d2->getData,lat2  
print,'First elements of :'  
print,'lon1 = ',lon1[0],' lon2 = ',lon2[0]  
print,'lat1 = ',lat1[0],' lat2 = ',lat2[0]  
  
;; clean up  
Obj_Destroy, d1 ; copy of lon in var2  
Obj_Destroy, d2 ; copy of lat in var2  
Obj_Destroy, lon  
Obj_Destroy, lat
```

Obj_Destroy, var1
Obj_Destroy, var2

END

File Attachments

- 1) [mgs_baseobject__define.pro](#), downloaded 97 times
 - 2) [mgs_basevariable__define.pro](#), downloaded 118 times
-