OPeNDAP Reference Manual

Generated by Doxygen 1.2.14

Fri Jul 19 23:58:38 2002

Contents

Chapter 1

OPeNDAP Hierarchical Index

1.1 OPeNDAP Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 2

OPeNDAP Compound Index

2.1 OPeNDAP Compound List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 3

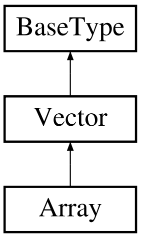
OPeNDAP Class Documentation

3.1 Array Class Reference

A multidimensional array of identical data types.

`#include <Array.h>`

Inheritance diagram for Array::



Public Methods

- Array (const string &n="", BaseType *v=0)

  Array constructor.

- Array (const Array &rhs)

  The Array copy constructor.

- virtual ~Array ()

The Array destructor.

- Array & **operator=** (const Array &rhs)
- virtual BaseType ∗ **ptr_duplicate** ()=0
- virtual bool read (const string &dataset)=0
- void update_length (int size)

    Changes the size property of the array.

- void append_dim (int size, string name="")

    Add a dimension of a given size.

- void add_constraint (Pix p, int start, int stride, int stop)

    Adds a dimension constraint to an Array.

- void reset_constraint ()

    Reset constraint to select entire array.

- void clear_constraint ()

    Clears the projection; add each projected dimension explicitly using `add_-constraint`*.*

- Pix first_dim ()

    Return first dimension of array.

- void next_dim (Pix &p)

    Return next array dimension.

- int dimension_size (Pix p, bool constrained=false)

    Returns the size of the dimension.

- int dimension_start (Pix p, bool constrained=false)

    Return the start index of a dimension.

- int dimension_stop (Pix p, bool constrained=false)

    Return the stop index of the constraint.

- int dimension_stride (Pix p, bool constrained=false)

    Returns the stride value of the constraint.

- string dimension_name (Pix p)

    Returns the name of the specified dimension.

- unsigned int dimensions (bool constrained=false)

Return the total number of dimensions in the array.

- virtual void print_decl (ostream &os, string space="", bool print_semi=true, bool constraint_info=false, bool constrained=false)

  Prints a DDS entry for the Array.

- virtual void print_val (ostream &os, string space="", bool print_decl_p=true)

  Print the value given the current constraint.

- virtual bool check_semantics (string &msg, bool all=false)

  Check semantic features of the Array.

Protected Methods

- void **_duplicate** (const Array &a)

3.1.1 Detailed Description

A multidimensional array of identical data types.

This class is used to hold arrays of other DODS data. The elements of the array can be simple or compound data types. There is no limit on the number of dimensions an array can have, or on the size of each dimension.

If desired, the user can give each dimension of an array a name. You can, for example, have a 360x180 array of temperatures, covering the whole globe with one-degree squares. In this case, you could name the first dimension “Longitude” and the second dimension “Latitude”. This can help prevent a great deal of confusion.

The Array is used as part of the Grid class, where the dimension names are crucial to its structure. The dimension names correspond to “Map” vectors, holding the actual values for that column of the array.

Each array dimension carries with it its own projection information. The projection inforamtion takes the form of three integers: the start, stop, and stride values. This is clearest with an example. Consider a one-dimensional array 10 elements long. If the start value of the dimension constraint is 3, then the constrained array appears to be seven elements long. If the stop value is changed to 7, then the array appears to be five elements long. If the stride is changed to two, the array will appear to be 3 elements long. Array constraints are written as: **[start:stride:stop]**.

```
A = [1 2 3 4 5 6 7 8 9 10]

A[3::] = [4 5 6 7 8 9 10]
```

```
A[3::7] = [4 5 6 7 8]

A[3:2:7] = [4 6 8]

A[0:3:9] = [1 4 7 10]
```

\note{DODS uses zero-based indexing.}

See also:
Grid , List

Definition at line 78 of file Array.h.

3.1.2 Constructor & Destructor Documentation

3.1.2.1 Array::Array (const string & *n* = "", BaseType ∗ *v* = 0)

Array constructor.

The Array constructor requires the name of the variable to be created, and the type of data the Array is to hold. The name may be omitted, which will create a nameless variable. The template pointer may also be omitted. Note that if the template pointer is omitted when the Array is created, it *must* be added (with `add_var()`) before `read()` or `deserialize()` is called.

Parameters:
- ***n*** A string containing the name of the variable to be created.
- ***v*** A pointer to a variable of the type to be included in the Array.

Definition at line 77 of file Array.cc.

3.1.3 Member Function Documentation

3.1.3.1 void Array::add_constraint (Pix *p*, int *start*, int *stride*, int *stop*)

Adds a dimension constraint to an Array.

Once a dimension has been created (see `append_dim()`), it can be "constrained". This will make the array appear to the rest of the world to be smaller than it is. This functions sets the projection for a dimension, and marks that dimension as part of the current projection.

A stride value <= 0 or > the array size is an error and causes `add_constraint` to return FALSE. Similarly, start or stop values greater than the dimension size will cause a FALSE return value.

Parameters:

p An index (of type Pix) pointing to the dimension in the list of dimensions.

start The start index of the constraint.

stride The stride value of the constraint.

stop The stop index of the constraint.

Returns:

void; in case of failure it throws an exception.

Definition at line 222 of file Array.cc.

References update_length.

3.1.3.2 void Array::append_dim (int *size*, string *name* = "")

Add a dimension of a given size.

Given a size and a name, this function adds a dimension to the array. For example, if the Array is already 10 elements long, calling `append_dim` with a size of 5 will transform the array into a 10x5 matrix. Calling it again with a size of 2 will create a 10x5x2 array, and so on. This sets Vector's length member as a side effect.

Parameters:

size The size of the desired new row.

name The name of the new dimension. This defaults to an empty string.

Definition at line 121 of file Array.cc.

References BaseType::name, and update_length.

3.1.3.3 bool Array::check_semantics (string & *msg*, bool *all* = false) `[virtual]`

Check semantic features of the Array.

This function checks semantic features of the Array. Currently, the only check specific to the Array is that there must be dimensions. The rest is inherited from `BaseType::check_semantics()`.

Returns:

A boolean value. FALSE means there was a problem.

Reimplemented from Vector.

Definition at line 607 of file Array.cc.

References BaseType::check_semantics.

3.1.3.4 void Array::clear_constraint ()

Clears the projection; add each projected dimension explicitly using `add_constraint`.

Tell the Array object to clear the constraint information about dimensions. Do this *once* before calling `add_constraint()` for each new constraint expression. Only the dimensions explicitly selected using `add_constraint()` will be sent.

Definition at line 177 of file Array.cc.

References Vector::set_length.

3.1.3.5 string Array::dimension_name (Pix *p*)

Returns the name of the specified dimension.

This function returns the name of the dimension indicated with *p*. Since this method is public, it is possible to call it before the Array object has been properly initialized. This will cause an exception. So don't do that.

Parameters:

p The Pix index of the dimension.

Returns:

A pointer to a string containing the dimension name.

Definition at line 456 of file Array.cc.

References dimension_name.

Referenced by dimension_name.

3.1.3.6 int Array::dimension_size (Pix *p*, bool *constrained* = false)

Returns the size of the dimension.

Return the size of the array dimension referred to by *p*. If the dimension is constrained (indicated with the *constrained* argument), the constrained size is returned.

Parameters:

p The Pix index of the dimension.

constrained If this parameter is TRUE, the function returns the constrained size of the array. If the dimension is not selected, the function returns zero. If it is FALSE, the function returns the dimension size whether or not the dimension is constrained. The default value is FALSE.

Returns:

An integer containing the size of the specified dimension.

Definition at line 318 of file Array.cc.

Referenced by Grid::check_semantics, print_val, and Grid::projection_yields_grid.

3.1.3.7 int Array::dimension_start (Pix *p*, bool *constrained* = false)

Return the start index of a dimension.

Use this function to return the start index of an array dimension. If the array is constrained (indicated with the *constrained* argument), the start index of the constrained array is returned (or zero if the dimension in question is not selected at all). See also `dimension_stop()` and `dimension_stride()`.

Parameters:

p The Pix index of the dimension.

constrained If this parameter is TRUE, the function returns the start index only if the dimension is constrained (subject to a start, stop, or stride constraint). If the dimension is not constrained, the function returns zero. If it is FALSE, the function returns the start index whether or not the dimension is constrained.

Returns:

The desired start index.

Definition at line 355 of file Array.cc.

References dimension_start.

Referenced by dimension_start, GSEClause::GSEClause, and Grid::projection_yields_grid.

3.1.3.8 int Array::dimension_stop (Pix *p*, bool *constrained* = false)

Return the stop index of the constraint.

Use this function to return the stop index of an array dimension. If the array is constrained (indicated with the *constrained* argument), the stop index of the constrained array is returned (or zero if the dimension in question is not selected at all). See also `dimension_start()` and `dimension_stride()`.

Parameters:

p The Pix index of the dimension.

constrained If this parameter is TRUE, the function returns the stop index only if the dimension is constrained (subject to a start, stop, or stride constraint). If the dimension is not constrained, the function returns zero. If it is FALSE, the function returns the stop index whether or not the dimension is constrained.

Returns:

The desired stop index.

Definition at line 391 of file Array.cc.

References dimension_stop.

Referenced by dimension_stop, GSEClause::GSEClause, and Grid::projection_yields_-grid.

3.1.3.9 int Array::dimension_stride (Pix *p*, bool *constrained* = false)

Returns the stride value of the constraint.

Use this function to return the stride value of an array dimension. If the array is constrained (indicated with the *constrained* argument), the stride value of the constrained array is returned (or zero if the dimension in question is not selected at all). See also `dimension_stop()` and `dimension_start()`.

Parameters:

p The Pix index of the dimension.

constrained If this parameter is TRUE, the function returns the stride value only if the dimension is constrained (subject to a start, stop, or stride constraint). If the dimension is not constrained, the function returns zero. If it is FALSE, the function returns the stride value whether or not the dimension is constrained.

Returns:

The stride value requested, or zero, if *constrained* is TRUE and the dimension is not selected.

Definition at line 428 of file Array.cc.

References dimension_stride.

Referenced by dimension_stride, and Grid::projection_yields_grid.

3.1.3.10 unsigned int Array::dimensions (bool *constrained* = false)

Return the total number of dimensions in the array.

Return the total number of dimensions contained in the array. When *constrained* is TRUE, return the number of dimensions given the most recently evaluated constraint expression.

Parameters:

constrained A boolean flag to indicate whether the array is constrained or not. By default, constrained is FALSE.

Definition at line 288 of file Array.cc.

References first_dim, and next_dim.

Referenced by Grid::check_semantics, and print_val.

3.1.3.11 Pix Array::first_dim ()

Return first dimension of array.

Returns a pointer to the first dimension of the array. Use `next_dim()` to return successive dimensions.

Returns:

A **Pix** object indicating the first array dimension.

Definition at line 257 of file Array.cc.

Referenced by Grid::check_semantics, dimensions, GSEClause::GSEClause, print_val, and Grid::projection_yields_grid.

3.1.3.12 void Array::next_dim (Pix & *p*)

Return next array dimension.

Given a dimension index, increments it to point to the next dimension. Use `first_dim()` to return the first dimensions.

Parameters:

p A **Pix** object indicating the array dimension immediately before the desired one.

Returns:

A **Pix** object indicating the array dimension immediately after the one specified by *p*.

Definition at line 272 of file Array.cc.

Referenced by Grid::check_semantics, dimensions, print_val, and Grid::projection_yields_grid.

3.1.3.13 void Array::print_decl (ostream & *os*, string *space* = " ", bool *print_semi* = true, bool *constraint_info* = false, bool *constrained* = false) [virtual]

Prints a DDS entry for the Array.

Prints a declaration for the Array. This is what appears in a DDS. If the Array is constrained, the declaration will allocate only enough space for the constrained values.

Parameters:

os An output stream to prin on.

space A string containing spaces to precede the declaration.

print_semi A boolean indicating whether to print a semi-colon after the declaration. (TRUE means "print a semi-colon.")

constraint_info A boolean value. See `BaseType::print_decl()`.

constrained This argument should be TRUE if the Array is constrained, and FALSE otherwise.

Reimplemented from Vector.

Definition at line 490 of file Array.cc.

References BaseType::print_decl, print_decl, BaseType::print_val, BaseType::send_p, and Vector::var.

Referenced by print_decl, and print_val.

3.1.3.14 void Array::print_val (ostream & *os*, string *space* = "", bool *print_decl_p* = true) [virtual]

Print the value given the current constraint.

Prints the value of the entire (constrained) array.

Parameters:

os The output stream to print on.

space The space to use in printing.

print_decl_p A boolean value indicating whether you want the Array declaration to precede the Array value.

Reimplemented from Vector.

Definition at line 570 of file Array.cc.

References dimension_size, dimensions, first_dim, next_dim, and print_decl.

3.1.3.15 virtual bool Array::read (const string & *dataset*) `[pure virtual]`

This function should read local data and fill in the data buffer. When reading the data, the read function should use the constraint and selection information available for each dimension of the array to decide how much of the array to read. Only the values to be transmitted with `serialize()` must be read.

The implementation of this function is part of creating a new DODS server, and is left for the user. For other details, refer to the description of the `read()` function in the BaseType class.

See also:
BaseType::read

Implements Vector.

3.1.3.16 void Array::reset_constraint ()

Reset constraint to select entire array.

Resets the dimension constraint information so that the entire array is selected.

Definition at line 148 of file Array.cc.

References Vector::set_length, and update_length.

3.1.3.17 void Array::update_length (int *size*)

Changes the size property of the array.

Deprecated:
Changes the size property of the array. If the array exists, it is augmented by a factor of `size`. This does not change the actual size of the array.

Definition at line 51 of file Array.cc.

References Vector::length, and Vector::set_length.

Referenced by add_constraint, append_dim, and reset_constraint.

The documentation for this class was generated from the following files:

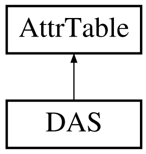
- Array.h
- Array.cc

3.2 AttrTable Class Reference

Contains the attributes for a dataset.

```
#include <AttrTable.h>
```

Inheritance diagram for AttrTable::



Public Methods

- unsigned int get_size () const

  Get the number of entries in this attribute table.

- string get_name ()

  Get the name of this attribute table.

- AttrTable * find_container (const string &target)

  Find an attribute with a given name.

- Pix find (const string &target, AttrTable **at)

  Find an attribute or container with a given name.

- void set_name (const string &n)

  Set the name of this attribute table.

- unsigned int append_attr (const string &name, const string &type, const string &value) throw (Error)

  Add an attribute to the table.

- unsigned int **append_attr** (const char *name, const char *type, const char *value) throw (Error)
- AttrTable * append_container (const string &name) throw (Error)

  Add a container to the attribute table.

- AttrTable * append_container (AttrTable *at, const string &name) throw (Error)

  Add a container to the attribute table.

- void add_container_alias (const string &name, AttrTable *src) throw (Error)

  Add an alias to a container held by this attribute table.

- void add_value_alias (AttrTable *das, const string &name, const string &source) throw (Error)

  Add an alias to an attribute held by this attribute table.

- bool attr_alias (const string &alias, AttrTable *at, const string &name)

  Adds an alias to the set of attributes. @name attr_alias().

- bool attr_alias (const string &alias, const string &name)
- void del_attr (const string &name, int i=-1)

  Deletes an attribute.

- void print (ostream &os, string pad="", bool dereference=false)

  Prints an attribute table.

- AttrTable * append_container (const string &name) throw (Error)
- AttrTable * append_container (AttrTable *at, const string &name) throw (Error)
- void add_container_alias (const string &name, AttrTable *src) throw (Error)
- void add_value_alias (AttrTable *das, const string &name, const string &source) throw (Error)
- void del_attr (const string &name, int i=-1)
- void print (ostream &os, string pad="", bool dereference=false)

Instance management functions

- **AttrTable** ()
- **AttrTable** (const AttrTable &rhs)
- virtual ~**AttrTable** ()
- AttrTable & **operator=** (const AttrTable &rhs)

Accessors using an attribute name

Each of the following accessors get information using the name of an attribute. They perform a simple search for the name in this attribute table only; sub-tables are not searched and the dot notation is not recognized.

- AttrTable * get_attr_table (const string &name)

  Get an attribute container.

- AttrTable ∗ **get_attr_table** (const char ∗name)
- string get_type (const string &name)

  Get the type name of an attribute within this attribute table.

- string **get_type** (const char ∗name)
- AttrType get_attr_type (const string &name)

  Get the type of an attribute.

- AttrType **get_attr_type** (const char ∗name)
- unsigned int get_attr_num (const string &name)

  Get the number of attributes in this container.

- unsigned int **get_attr_num** (const char ∗name)
- string get_attr (const string &name, unsigned int i=0)

  Get the value of an attribute.

- string **get_attr** (const char ∗name, unsigned int i=0)
- vector< string > ∗ get_attr_vector (const string &name)

  Get a vector-valued attribute.

- vector< string > ∗ **get_attr_vector** (const char ∗name)

Accessors using a Pix

The following accessors get information using a Pix pseudo-index pointer into the AttrTable structure.

- Pix first_attr ()

  Get a reference to the first entry in this attribute table.

- void next_attr (Pix &p)

  Increment a Pix pointer into the attribute table.

- string get_name (Pix p)

  Returns the name of the attribute table.

- bool is_container (Pix p)

  Returns true if the attribute is a container.

- AttrTable ∗ get_attr_table (Pix p)

  Return an attribute container.

- string get_type (Pix p)

  Get the type name of an attribute.

- AttrType get_attr_type (Pix p)

  Get the type of an attribute.

- unsigned int get_attr_num (Pix p)
  Get the number of attributes in this container.

- string get_attr (Pix p, unsigned int i=0)
  Return the value of an attribute.

- vector< string > * get_attr_vector (Pix p)
  Return a vector-valued attribute.

Instance management

- **AttrTable** ()
- **AttrTable** (const AttrTable &rhs)
- virtual ~**AttrTable** ()
- AttrTable & **operator=** (const AttrTable &rhs)

get_attr_table

- unsigned int get_size () const
- string get_name ()
- AttrTable * find_container (const string &target)
- Pix find (const string &target, AttrTable **at)
- AttrTable * get_attr_table (const string &name)
  Get an attribute container.

- AttrTable * **get_attr_table** (const char *name)

get_type

- string get_type (const string &name)
- string **get_type** (const char *name)

get_attr_type

- AttrType get_attr_type (const string &name)
- AttrType **get_attr_type** (const char *name)

get_attr_num

- unsigned int get_attr_num (const string &name)
- unsigned int **get_attr_num** (const char *name)

get_attr

- string get_attr (const string &name, unsigned int i=0)
- string **get_attr** (const char *name, unsigned int i=0)

get_attr_vector

- vector< string > * get_attr_vector (const string &name)
- vector< string > * **get_attr_vector** (const char *name)

get information using a Pix

- Accessors that use names Pix first_attr ()
- void next_attr (Pix &p)
- string get_name (Pix p)
- bool is_container (Pix p)
- AttrTable * get_attr_table (Pix p)
- string get_type (Pix p)
- AttrType get_attr_type (Pix p)
- unsigned int get_attr_num (Pix p)
- string get_attr (Pix p, unsigned int i=0)
- vector< string > * get_attr_vector (Pix p)

append_attr()

Adds an attribute to the table. If the given name already refers to an attribute, and the attribute has a value, the given value is appended to the attribute vector. Calling this function repeatedly is the way to append to an attribute vector.

The function returns an error condition if the attribute is a container, or if the type of the input value does not match the existing attribute's type. Use `append_container()` *to add container attributes.*

This method performs a simple search for `name` *in this attribute table only; subtables are not searched and the dot notation is not recognized.*

Returns:

Returns the length of the added attribute value.

Parameters:

name *The name of the attribute to add or modify.*

type *The type of the attribute to add or modify.*

value *The value to add to the attribute table.*

- Accessors that use a pix Accessors void set_name (const string &n)
- unsigned int **append_attr** (const string &name, const string &type, const string &value) throw (Error)
- unsigned int **append_attr** (const char *name, const char *type, const char *value) throw (Error)

attr_alias()

Adds an alias to the set of attributes. Once an alias is inserted into an attribute table, reading the attributes for alias *will return those stored for* name.

Two forms for this function exist: one searches for name *in the AttrTable referenced by* at *while the other uses* `this`. *You can use* `DAS::get_attr_table()` *to get the attribute table for an arbitrary name.*

See also:

get_attr_table

Deprecated:

- bool attr_alias (const string &alias, AttrTable *at, const string &name)
- bool attr_alias (const string &alias, const string &name)

Protected Methods

- void clone (const AttrTable &at)
- void simple_print (ostream &os, string pad, Pix p, bool dereference)
- void clone (const AttrTable &at)
- void simple_print (ostream &os, string pad, Pix p, bool dereference)

3.2.1 Detailed Description

Contains the attributes for a dataset.

An AttrTable ("Attribute Table") stores a set of names and, for each name, either a type and a value, or another attribute table. The attribute value can be a vector containing many values of the same type. The attributes can have any of the types listed in the `AttrType` list. However, all attribute types are stored as string data, except for the container type, which is stored as a pointer to another attribute table.

Each element in the attribute table can itself be an attribute table. The table can also contain "alias" attributes whose value is given by the value of another attribute to which it is linked.

The attribute tables have a standard printed representation. There is a member function `print()` for writing this form. Use the `DAS::parse()` function to read the printed form.

An attribute table might look something like this:

```
string long_name "Weekly Means of Sea Surface Temperature";
actual_range {
    Float64 min -1.8;
    Float64 max 35.09;
}
string units "degC";
conversion_data {
    Float64 add_offset 0.;
    Float64 scale_factor 0.0099999998;
}
Int32 missing_value 32767;
```

Here, `long_name`, `units`, and `missing_value` are simple attributes, and `actual_range` and `conversion_data` are container attributes containing other attribute tables.

See also:
DAS , AttrType

Definition at line 108 of file AttrTable.h.

3.2.2 Member Function Documentation

3.2.2.1 void AttrTable::add_container_alias (const string & *name*, AttrTable * *src*) throw (Error)

Add an alias to a container held by this attribute table.

Parameters:
name The name of the alias. May *not* use dot notation.
src The existing attribute container to alias.

Exceptions:
Error if an attribute, container or alias called `name` already exists in this attribute table.

3.2.2.2 void AttrTable::add_container_alias (const string & *name*, AttrTable * *src*) throw (Error)

Add an alias to a container held by this attribute table.

Parameters:
name The name of the alias. May *not* use dot notation.
src The existing attribute container to alias.

Exceptions:
Error if an attribute, container or alias called `name` already exists in this attribute table.

Definition at line 604 of file AttrTable.cc.

3.2.2.3 void AttrTable::add_value_alias (AttrTable * *das*, const string & *name*, const string & *source*) throw (Error)

Add an alias to an attribute held by this attribute table.

Parameters:

name The name of the alias. May *not* use dot notation.

source The name of the attribute to alias. May use dot notation.

Exceptions:

Error if the attribute table already contains an attribute, container or alias called `name` or if an attribute called `source` does not exist.

3.2.2.4 void AttrTable::add_value_alias (AttrTable * *das*, const string & *name*, const string & *source*) throw (Error)

Add an alias to an attribute held by this attribute table.

Parameters:

das

name The name of the alias. May *not* use dot notation.

source The name of the attribute to alias. May use dot notation.

Exceptions:

Error if the attribute table already contains an attribute, container or alias called `name` or if an attribute called `source` does not exist.

Definition at line 635 of file AttrTable.cc.

References attr_map, get_attr_table, and is_container.

Referenced by attr_alias.

3.2.2.5 unsigned int AttrTable::append_attr (const string & *name*, const string & *type*, const string & *attr*) throw (Error)

Add an attribute to the table.

If the given name already refers to an attribute, and the attribute has a value, the given value is appended to the attribute vector. Calling this function repeatedly is the way to append to an attribute vector.

The function returns an error condition if the attribute is a container, or if the type of the input value does not match the existing attribute's type. Use `append_container()` to add container attributes.

This method performs a simple search for `name` in this attribute table only; sub-tables are not searched and the dot notation is not recognized.

Returns:

Returns the length of the added attribute value.

Parameters:

name The name of the attribute to add or modify.

type The type of the attribute to add or modify.

attr The value to add to the attribute table.

Definition at line 511 of file AttrTable.cc.

3.2.2.6 AttrTable* AttrTable::append_container (AttrTable * *at*, const string & *name*) throw (Error)

Append a new attribute container to this attribute table. The new container is `at` and its name is set to `name`. If this attribute table already contains an attribute container called `name` an exception is thrown.

Returns:

A pointer to the new AttrTable object.

3.2.2.7 AttrTable* AttrTable::append_container (const string & *name*) throw (Error)

Create and append an attribute container to this AttrTable. If this attribute table already contains an attribute container called `name` an exception is thrown.

Returns:

A pointer to the new AttrTable object.

3.2.2.8 AttrTable * AttrTable::append_container (AttrTable * *at*, const string & *name*) throw (Error)

Add a container to the attribute table.

Append a new attribute container to this attribute table. The new container is `at` and its name is set to `name`. If this attribute table already contains an attribute container called `name` an exception is thrown.

Returns:

A pointer to the new AttrTable object.

Definition at line 576 of file AttrTable.cc.

3.2.2.9 AttrTable * AttrTable::append_container (const string & *name*) throw (Error)

Add a container to the attribute table.

Create and append an attribute container to this AttrTable. If this attribute table already contains an attribute container called `name` an exception is thrown.

Returns:

A pointer to the new AttrTable object.

Definition at line 561 of file AttrTable.cc.

Referenced by DAS::add_table, and DAS::DAS.

3.2.2.10 bool AttrTable::attr_alias (const string & *alias*, const string & *name*)

Deprecated:

Parameters:

alias The alias to insert into the attribute table.

name The name of the already-existing attribute to which the alias will refer.

3.2.2.11 bool AttrTable::attr_alias (const string & *alias*, AttrTable * *at*, const string & *name*)

Parameters:

alias The alias to insert into the attribute table.

name The name of the already-existing attribute to which the alias will refer.

Deprecated:

Parameters:

at An attribute table in which to insert the alias.

3.2.2.12 bool AttrTable::attr_alias (const string & *alias*, const string & *name*)

Deprecated:

Parameters:

alias The alias to insert into the attribute table.

name The name of the already-existing attribute to which the alias will refer.

Definition at line 711 of file AttrTable.cc.

References attr_alias.

3.2.2.13 bool AttrTable::attr_alias (const string & *alias*, AttrTable ∗ *at*, const string & *name*)

Adds an alias to the set of attributes. @name attr_alias().

Once an alias is inserted into an attribute table, reading the attributes for *alias* will return those stored for *name*.

Two forms for this function exist: one searches for *name* in the AttrTable referenced by *at* while the other uses `this`. You can use `DAS::get_attr_table()` to get the attribute table for an arbitrary name.

See also:

get_attr_table

Deprecated:

Parameters:

alias The alias to insert into the attribute table.

name The name of the already-existing attribute to which the alias will refer.

at An attribute table in which to insert the alias.

Definition at line 700 of file AttrTable.cc.

References add_value_alias.

Referenced by attr_alias.

3.2.2.14 void AttrTable::clone (const AttrTable & *at*) `[protected]`

Clone the given attribute table in `this`.

3.2.2.15 void AttrTable::clone (const AttrTable & *at*) `[protected]`

Clone the given attribute table in `this`.

Definition at line 178 of file AttrTable.cc.

References attr_map.

3.2.2.16 void AttrTable::del_attr (const string & *name*, int *i* = -1)

Delete the attribute named `name`. If `i` is given, and the attribute has a vector value, delete the `i`th element of the vector.

You can use this function to delete container attributes, although the `i` parameter has no meaning for that operation.

Parameters:

name The name of the attribute to delete. This can be an attribute of any type, including containers. However, this method looks only in this attribute table and does not recognize the dot notation.

i If the named attribute is a vector, and `i` is non-negative, the i-th entry in the vector is deleted, and the array is repacked. If `i` equals -1 (the default), the entire attribute is deleted.

3.2.2.17 void AttrTable::del_attr (const string & *name*, int *i* = -1)

Deletes an attribute.

Delete the attribute named `name`. If `i` is given, and the attribute has a vector value, delete the `i`th element of the vector.

You can use this function to delete container attributes, although the `i` parameter has no meaning for that operation.

Parameters:

name The name of the attribute to delete. This can be an attribute of any type, including containers. However, this method looks only in this attribute table and does not recognize the dot notation.

i If the named attribute is a vector, and `i` is non-negative, the i-th entry in the vector is deleted, and the array is repacked. If `i` equals -1 (the default), the entire attribute is deleted.

Definition at line 739 of file AttrTable.cc.

3.2.2.18 Pix AttrTable::find (const string & *target*, AttrTable ∗∗ *at*)

Look for an attribute or an attribute container. If used to search for an attribute container, this method returns the container's *parent* using the value-result parameter `at` and a reference to the container using the Pix return value. If used to search for an attribute, the attribute's container is returned using `at`; the attribute itself can be accessed using the Pix return value.

Parameters:

target The name (using dot notation) of the attribute or container to find.

at A value-result used to return the attribute container in which `target` was found. Null if `target` was not found.

Returns:

A Pix which can be used to access `target` from within `at`. Null if the attribute or container does not exist.

3.2.2.19 Pix AttrTable::find (const string & *target*, AttrTable ∗∗ *at*)

Find an attribute or container with a given name.

Look for an attribute or an attribute container. If used to search for an attribute container, this method returns the container's *parent* using the value-result parameter `at` and a reference to the container using the Pix return value. If used to search for an attribute, the attribute's container is returned using `at`; the attribute itself can be accessed using the Pix return value.

Parameters:

target The name (using dot notation) of the attribute or container to find.

at A value-result used to return the attribute container in which `target` was found. Null if `target` was not found.

Returns:

A Pix which can be used to access `target` from within `at`. Null if the attribute or container does not exist.

Definition at line 56 of file AttrTable.cc.

References find_container.

Referenced by find_container.

3.2.2.20 AttrTable∗ AttrTable::find_container (const string & *target*)

Look in this attribute table for an attribute container named `target`. The search starts at this attribute table; `target` should use the dot notation to name containers held within children of this attribute table.

To search the entire DAS object, make sure to invoke this method from that object.

Parameters:

target The attribute container to find.

Returns:

A pointer to the attribute table or null if the container cannot be found.

3.2.2.21 AttrTable ∗ AttrTable::find_container (const string & *target*)

Find an attribute with a given name.

Look in this attribute table for an attribute container named `target`. The search starts at this attribute table; `target` should use the dot notation to name containers held within children of this attribute table.

To search the entire DAS object, make sure to invoke this method from that object.

Parameters:

target The attribute container to find.

Returns:

A pointer to the attribute table or null if the container cannot be found.

Definition at line 99 of file AttrTable.cc.

References find.

Referenced by find, and get_attr_table.

3.2.2.22 Accessors that use names Pix AttrTable::first_attr ()

Get a reference to the first entry in this attribute table.

Returns:
Pix; returns null if there's nothing in the list.

3.2.2.23 Pix AttrTable::first_attr ()

Get a reference to the first entry in this attribute table.

Returns:
Pix; returns null if there's nothing in the list.

Definition at line 254 of file AttrTable.cc.

Referenced by DAS::first_var.

3.2.2.24 string AttrTable::get_attr (Pix *p*, unsigned int *i* = 0)

Returns the value of an attribute. If the attribute has a vector value, you can indicate which is the desired value with the index argument, `i`. If the argument is omitted, the first value is returned. If the attribute has only a single value, the index argument is ignored. If `i` is greater than the number of elements in the attribute, an error is produced.

All values in an attribute table are stored as string data. They may be converted to a more appropriate internal format by the calling program.

Parameters:
p

i The attribute value index, zero-based.

Returns:
If the indicated attribute is a container, this function returns the string "None". If using a name to refer to the attribute and the named attribute does not exist, return the empty string.

3.2.2.25 string AttrTable::get_attr (const string & *name*, unsigned int *i* = 0)

Get the value of an attribute. If the attribute has a vector value, you can indicate which is the desired value with the index argument, `i`. If the argument is omitted, the first value is returned. If the attribute has only a single value, the index argument is ignored. If `i` is greater than the number of elements in the attribute, an error is produced.

All values in an attribute table are stored as string data. They may be converted to a more appropriate internal format by the calling program.

3.2.2.26 string AttrTable::get_attr (Pix *p*, unsigned int *i* = 0)

Return the value of an attribute.

Returns the value of an attribute. If the attribute has a vector value, you can indicate which is the desired value with the index argument, `i`. If the argument is omitted, the first value is returned. If the attribute has only a single value, the index argument is ignored. If `i` is greater than the number of elements in the attribute, an error is produced.

All values in an attribute table are stored as string data. They may be converted to a more appropriate internal format by the calling program.

Parameters:

p

i The attribute value index, zero-based.

Returns:

If the indicated attribute is a container, this function returns the string "None". If using a name to refer to the attribute and the named attribute does not exist, return the empty string.

Definition at line 415 of file AttrTable.cc.

3.2.2.27 string AttrTable::get_attr (const string & *name*, unsigned int *i* = 0)

Get the value of an attribute.

Get the value of an attribute. If the attribute has a vector value, you can indicate which is the desired value with the index argument, `i`. If the argument is omitted, the first value is returned. If the attribute has only a single value, the index argument is ignored. If `i` is greater than the number of elements in the attribute, an error is produced.

All values in an attribute table are stored as string data. They may be converted to a more appropriate internal format by the calling program.

Definition at line 435 of file AttrTable.cc.

3.2.2.28 unsigned int AttrTable::get_attr_num (Pix *p*)

If the indicated attribute is a container attribute, this function returns the number of attributes in *its* attribute table. If the indicated attribute is not a container, the method returns the number of values for the attribute (1 for a scalar attribute, N for a vector attribute value).

Parameters:

p

Returns:

The number of elements in the attribute.

3.2.2.29 unsigned int AttrTable::get_attr_num (const string & *name*)

If the indicated attribute is a container attribute, this function returns the number of attributes in *its* attribute table. If the indicated attribute is not a container, the method returns the number of values for the attribute (1 for a scalar attribute, N for a vector attribute value).

3.2.2.30 unsigned int AttrTable::get_attr_num (Pix *p*)

Get the number of attributes in this container.

If the indicated attribute is a container attribute, this function returns the number of attributes in *its* attribute table. If the indicated attribute is not a container, the method returns the number of values for the attribute (1 for a scalar attribute, N for a vector attribute value).

Parameters:

p

Returns:

The number of elements in the attribute.

Definition at line 369 of file AttrTable.cc.

3.2.2.31 unsigned int AttrTable::get_attr_num (const string & *name*)

Get the number of attributes in this container.

If the indicated attribute is a container attribute, this function returns the number of attributes in *its* attribute table. If the indicated attribute is not a container, the method returns the number of values for the attribute (1 for a scalar attribute, N for a vector attribute value).

Definition at line 385 of file AttrTable.cc.

3.2.2.32 AttrTable* AttrTable::get_attr_table (Pix *p*)

Get the attribute container referenced by `p`. If no such container exists, then return null.

Parameters:
p Reference to a table contained by this object.

Returns:
The child attribute table.

3.2.2.33 AttrTable * AttrTable::get_attr_table (Pix *p*)

Return an attribute container.

Get the attribute container referenced by `p`. If no such container exists, then return null.

Parameters:
p Reference to a table contained by this object.

Returns:
The child attribute table.

Definition at line 291 of file AttrTable.cc.

References get_attr_table.

3.2.2.34 AttrType AttrTable::get_attr_type (Pix *p*)

Get the type of an attribute.

Parameters:
p

Returns:
The datatype of this attribute in an instance of AttrType.

3.2.2.35 AttrType AttrTable::get_attr_type (const string & *name*)

Get the type of an attribute using AttrType.

3.2.2.36 AttrType AttrTable::get_attr_type (Pix *p*)

Get the type of an attribute.

Parameters:

p

Returns:

The datatype of this attribute in an instance of AttrType.

Definition at line 338 of file AttrTable.cc.

3.2.2.37 AttrType AttrTable::get_attr_type (const string & *name*)

Get the type of an attribute.

Returns:

The `AttrType` value describing the attribute.

Definition at line 347 of file AttrTable.cc.

References get_attr_type.

Referenced by get_attr_type.

3.2.2.38 vector<string>* AttrTable::get_attr_vector (Pix *p*)

Returns a pointer to the vector of values associated with the attribute referenced by Pix `p` or named `name`.

Note that all values in an attribute table are stored as string data. They may be converted to a more appropriate internal format by the calling program.

Parameters:

p

Returns:

If the indicated attribute is a container, this function returns the null pointer. Otherwise returns a pointer to the the attribute vector value.

3.2.2.39 vector<string>* AttrTable::get_attr_vector (const string & *name*)

Get a pointer to the vector of values associated with the attribute referenced by Pix `p` or named `name`.

Note that all values in an attribute table are stored as string data. They may be converted to a more appropriate internal format by the calling program.

Returns:
If the indicated attribute is a container, this function returns the null pointer. Otherwise returns a pointer to the the attribute vector value.

3.2.2.40 vector< string > * AttrTable::get_attr_vector (Pix *p*)

Return a vector-valued attribute.

Returns a pointer to the vector of values associated with the attribute referenced by Pix `p` or named `name`.

Note that all values in an attribute table are stored as string data. They may be converted to a more appropriate internal format by the calling program.

Parameters:
p

Returns:
If the indicated attribute is a container, this function returns the null pointer. Otherwise returns a pointer to the the attribute vector value.

Definition at line 460 of file AttrTable.cc.

3.2.2.41 vector< string > * AttrTable::get_attr_vector (const string & *name*)

Get a vector-valued attribute.

Get a pointer to the vector of values associated with the attribute referenced by Pix `p` or named `name`.

Note that all values in an attribute table are stored as string data. They may be converted to a more appropriate internal format by the calling program.

Returns:
If the indicated attribute is a container, this function returns the null pointer. Otherwise returns a pointer to the the attribute vector value.

Definition at line 479 of file AttrTable.cc.

References get_attr_vector.

Referenced by get_attr_vector.

3.2.2.42 string AttrTable::get_name (Pix *p*)

Returns the name of the attribute table.

Reimplemented in DAS.

3.2.2.43 string AttrTable::get_name ()

Get the name of this attribute table.

Returns:
A string containing the name.

3.2.2.44 string AttrTable::get_name ()

Get the name of this attribute table.

Returns:
A string containing the name.

Definition at line 238 of file AttrTable.cc.

Referenced by DAS::get_name, print, and simple_print.

3.2.2.45 unsigned int AttrTable::get_size () const

Get the number of entries in this attribute table. Attributes that are containers count one attribute, as do attributes with both scalar and vector values.

Returns:
The number of entries.

3.2.2.46 unsigned int AttrTable::get_size () const

Get the number of entries in this attribute table.

Attributes that are containers count one attribute, as do attributes with both scalar and vector values.

Returns:

The number of entries.

Definition at line 230 of file AttrTable.cc.

3.2.2.47 string AttrTable::get_type (Pix *p*)

Get the type name of an attribute.

Parameters:

p

Returns:

A string with the name of this attribute datatype.

3.2.2.48 string AttrTable::get_type (const string & *name*)

Get the type name of an attribute within this attribute table.

3.2.2.49 string AttrTable::get_type (Pix *p*)

Get the type name of an attribute.

Parameters:

p

Returns:

A string with the name of this attribute datatype.

Definition at line 314 of file AttrTable.cc.

3.2.2.50 bool AttrTable::is_container (Pix *p*)

Returns true if the attribute is a container.

3.2.2.51 void AttrTable::next_attr (Pix & *p*)

Advance to the next element of this attribute table. Set to null when there are no more elements in the list.

Parameters:

p Pix to advance.

3.2.2.52 void AttrTable::next_attr (Pix & *p*)

Increment a Pix pointer into the attribute table.

Advance to the next element of this attribute table. Set to null when there are no more elements in the list.

Parameters:

p Pix to advance.

Definition at line 264 of file AttrTable.cc.

Referenced by DAS::next_var.

3.2.2.53 void AttrTable::print (ostream & *os*, string *pad* = "", bool *dereference* = false)

Prints an ASCII representation of the attribute table to the indicated output stream. The `pad` argument is prefixed to each line of the output to provide control of indentation.

Parameters:

os Print to the given output stream.

pad Indent elements of a table using this string of spaces. By default this is a string of four spaces

dereference If true, follow aliases. Default is false.

3.2.2.54 void AttrTable::print (ostream & *os*, string *pad* = " ", bool *dereference* = false)

Prints an attribute table.

Prints an ASCII representation of the attribute table to the indicated output stream. The `pad` argument is prefixed to each line of the output to provide control of indentation.

Parameters:

os Print to the given output stream.

pad Indent elements of a table using this string of spaces. By default this is a string of four spaces

dereference If true, follow aliases. Default is false.

Definition at line 801 of file AttrTable.cc.

References get_name, and simple_print.

Referenced by DAS::print.

3.2.2.55 Accessors that use a pix Accessors void AttrTable::set_name (const string & *n*)

Set the name of this attribute table.

Parameters:

n The new name of the attribute table.

3.2.2.56 void AttrTable::set_name (const string & *n*)

Set the name of this attribute table.

Parameters:

n The new name of the attribute table.

Definition at line 246 of file AttrTable.cc.

3.2.2.57 void AttrTable::simple_print (ostream & *os*, string *pad*, Pix *p*, bool *dereference*) `[protected]`

A simple printer that does nothing fancy with aliases.

3.2.2.58 void AttrTable::simple_print (ostream & *os*, string *pad*, Pix *p*, bool *dereference*) `[protected]`

A simple printer that does nothing fancy with aliases.

Definition at line 765 of file AttrTable.cc.

References get_name, and get_type.

Referenced by print.

The documentation for this class was generated from the following files:

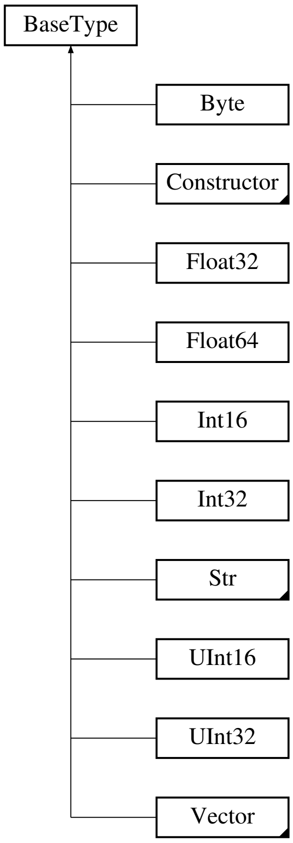
- AttrTable.h
- AttrTableOld.h
- AttrTable.cc
- AttrTableOld.cc

3.3 BaseType Class Reference

The basic data type for the DODS DAP types.

#include <BaseType.h>

Inheritance diagram for BaseType::



Public Methods

- BaseType (const string &n="", const Type &t=dods_null_c, xdrproc_t xdr=NULL)

    The BaseType constructor.

- BaseType (const BaseType &copy_from)

  The BaseType copy constructor.

- virtual ~**BaseType** ()
- virtual string toString ()
- BaseType & **operator=** (const BaseType &rhs)
- virtual BaseType ∗ **ptr_duplicate** ()=0
- string name () const

  Returns the name of the class instance.

- virtual void set_name (const string &n)

  Sets the name of the class instance.

- Type type () const

  Returns the type of the class instance.

- void set_type (const Type &t)

  Sets the type of the class instance.

- string type_name () const

  Returns the type of the class instance as a string.

- bool is_simple_type ()

  Returns true if the instance is a simple type variable.

- bool is_vector_type ()

  Returns true if the instance is a vector type variable.

- bool is_constructor_type ()

  Returns true if the instance is a constructor type variable.

- virtual int element_count (bool leaves=false)

  Count the members of constructor types.

- bool synthesized_p ()
- void set_synthesized_p (bool state)
- bool read_p ()

  Returns the value of the read_p# flag.

- virtual void set_read_p (bool state)

  Sets the value of the read_p# flag.

- bool send_p ()

  Returns the value of the send_p# flag.

- virtual void set_send_p (bool state)
- xdrproc_t xdr_coder ()

  Returns a function used to encode elements of an array.

- virtual void set_parent (BaseType ∗parent) throw (InternalErr)
- virtual BaseType ∗ get_parent ()
- virtual BaseType ∗ var (const string &name="", bool exact_match=true, btp_stack ∗s=0)

  Returns a pointer to a member of a constructor class.

- virtual BaseType ∗ var (const string &name, btp_stack &s)

  Returns a pointer to a member of a constructor class.

- virtual void add_var (BaseType ∗bt, Part part=nil)

  Adds the input data to the class instance.

- virtual unsigned int width ()=0

  Returns the size of the class instance data.

- virtual bool read (const string &dataset)=0

  Reads data into a local buffer.

- virtual unsigned int buf2val (void ∗∗val)=0

  Reads the class data.

- virtual unsigned int val2buf (void ∗val, bool reuse=false)=0

  Loads class data.

- virtual bool serialize (const string &dataset, DDS &dds, XDR ∗sink, bool ce_eval=true)=0

  Move data to the net.

- virtual bool deserialize (XDR ∗source, DDS ∗dds, bool reuse=false)=0

  Receive data from the net.

- bool **expunge** ()
- virtual void print_decl (ostream &os, string space="", bool print_semi=true, bool constraint_info=false, bool constrained=false)

  Print an ASCII representation of the variable structure.

- virtual void print_val (ostream &os, string space="", bool print_decl_p=true)=0

  Prints the value of the variable.

- virtual bool check_semantics (string &msg, bool all=false)

  Compare an object's current state with the sematics of its type.

- virtual bool ops (BaseType *b, int op, const string &dataset)

  The class relational operators.

Protected Methods

- void **_duplicate** (const BaseType &bt)

3.3.1 Detailed Description

The basic data type for the DODS DAP types.

This defines the basic data type features for the DODS data access protocol (DAP) data types. All the DAP type classes (Float64, Array, etc.) subclass it. This class is an abstract one; no variables will ever be stored as BaseType instances, only as instances of its child classes.

These classes and their methods give a user the capacity to set up sophisticated data types. They do *not* provide sophisticated ways to access and use this data. On the server side, in many cases, the class instances will have no data in them at all until the `serialize` function is called to send data to the client. On the client side, most DODS application programs will unpack the data promptly into whatever local data structure the programmer deems the most useful.

In order to use these classes on the server side of a DODS client/server connection, you must write a `read` method for each of the data types you expect to encounter in the application. This function, whose purpose is to read data from a local source into the class instance data buffer, is called in `serialize`, when the data is about to be sent to the client. The `read` function may be called earlier, in the case of data subset requests (constraint expressions) whose evaluation requires it. (For example, the constraint expression "`a,b&b>c`" requires that `c` be read even though it will not be sent.)

For some data types, the `read` function must be aware of the constraints to be returned. These cautions are outlined where they occur.

Definition at line 156 of file BaseType.h.

3.3.2 Constructor & Destructor Documentation

3.3.2.1 BaseType::BaseType (const string & *n* = "", const Type & *t* = dods_null_c, xdrproc_t *xdr* = NULL)

The BaseType constructor.

The BaseType constructor needs a name, a type, and the name of an XDR filter. The BaseType class exists to provide data to type classes that inherit from it. The constructors of those classes call the BaseType constructor; it is never called directly.

Note that the constructor (as well as the copy constructor via _duplicate) open/initialize the (XDRS ∗)s XDRIN and XDROUT to reference sdtin and stdout. This means that writing to std{in,out} must work correctly, and probably means that is must be OK to mix calls to cout/cin with calls that write to std{out,in} (it is for g++ with libg++ at version 2.6 or greater).

Parameters:

n A string containing the name of the new variable.

t The type of the variable.

xdr A pointer to an XDR filter to use to transmit the data in this variable to a client DODS process.

See also:

Type

Definition at line 71 of file BaseType.cc.

3.3.3 Member Function Documentation

3.3.3.1 void BaseType::add_var (BaseType ∗ *bt*, Part *part* = nil) `[virtual]`

Adds the input data to the class instance.

Adds a variable to an instance of a constructor class, such as Array, Structure and so on. This function is only used by those classes. The BaseType implementation simply prints an error message.

NB: When adding a variable to a constructor or an array (this is only important for a constructor), if that variable is itself a constructor you *must* add its children *before* you add call this method to add the variable to its parent. This method copies the variable allocating a new object. One way around this is to add the constructor, then get a BaseType pointer to it using `var()`.

Parameters:

bt The data to be added to the constructor type. The caller of this method *must* free memory it allocates for `v`. This method will make a deep copy of the object pointed to by `v`.

part The part of the constructor data to be modified.

See also:

Part

Reimplemented in Grid.

Definition at line 511 of file BaseType.cc.

3.3.3.2 virtual unsigned int BaseType::buf2val (void ** *val*) `[pure virtual]`

Reads the class data.

Reads the class data into the memory referenced by *val*. The caller must allocate enough storage to *val* to hold the class data. If *val* is NULL, however, memory will be allocated by this function with `new()`. Even if the memory is allocated this way, the caller is responsible for deallocating that memory. Array and List values for simple types are stored as C would store an array.

Parameters:

val A pointer to a pointer to the memory into which the class data will be copied. If the value pointed to is NULL, memory will be allocated to hold the data, and the pointer value modified accordingly. The calling program is responsible for deallocating the memory indicated by this pointer.

Returns:

The size (in bytes) of the information copied to *val*.

Implemented in Byte.

3.3.3.3 bool BaseType::check_semantics (string & *msg*, bool *all* = false) `[virtual]`

Compare an object's current state with the sematics of its type.

This function checks the class instance for internal consistency. This is important to check for complex constructor classes. For BaseType, an object is semantically correct if it has both a non-null name and type.

For example, an Int32 instance would return FALSE if it had no name or no type defined. A Grid instance might return FALSE for more complex reasons, such as having Map arrays of the wrong size or shape.

This function is used by the DDS class, and will rarely, if ever, be explicitly called by a DODS application program. A variable must pass this test before it is sent, but there may be many other stages in a retrieve operation where it would fail.

Returns:

Returns FALSE when the current state violates some aspect of the type semantics, TRUE otherwise.

Parameters:

msg A returned string, containing a message indicating the source of any problem.

all For complex constructor types (Grid, Sequence, Structure), this flag indicates whether to check the sematics of the member variables, too.

See also:

DDS::check_semantics

Reimplemented in Array.

Definition at line 649 of file BaseType.cc.

Referenced by Vector::check_semantics, Structure::check_semantics, Sequence::check_semantics, Grid::check_semantics, and Array::check_semantics.

3.3.3.4 virtual bool BaseType::deserialize (XDR ∗ *source*, DDS ∗ *dds*, bool *reuse* = false) `[pure virtual]`

Receive data from the net.

Receives data from the network connection identified by the `source` parameter. The data is put into the class data buffer according to the input `dds`.

This function is only used on the client side of the DODS client/server connection.

Parameters:

source A valid XDR pointer to the process connection to the net. This is generally created with a call to `new_xdrstdio()`.

dds The Data Descriptor Structure object corresponding to this dataset. See *The DODS User Manual* for information about this structure. This would have been received from the server in an earlier transmission.

reuse A boolean value, indicating whether the class internal data storage can be reused or not. If this argument is TRUE, the class buffer is assumed to be large enough to hold the incoming data, and it is *not* reallocated. If FALSE, new storage is allocated. If the internal buffer has not been allocated at all, this argument has no effect.

Returns:

Always returns TRUE.

Exceptions:

Error when a problem reading from the XDR stream is found.

See also:

DDS

Implemented in Byte.

Referenced by Structure::deserialize, Grid::deserialize, and Structure::print_all_vals.

3.3.3.5 int BaseType::element_count (bool *leaves* = false) `[virtual]`

Count the members of constructor types.

Return a count of the total number of variables in this variable. This is used to count the number of variables held by a constructor variable - for simple type and vector variables it always returns 1.

For compound data types, there are two ways to count members. You can count the members, or you can count the simple members and add that to the count of the compound members. For example, if a Structure contains an Int32 and another Structure that itself contains two Int32 members, the element count of the top-level structure could be two (one Int32 and one Structure) or three (one Int32 by itself and two Int32's in the subsidiary Structure). Use the *leaves* parameter to control which kind of counting you desire.

Returns:

Returns 1 for simple types. For compound members, the count depends on the *leaves* argument.

Parameters:

leaves This parameter is only relevant if the object contains other compound data types. If FALSE, the function counts only the data variables mentioned in the object's declaration. If TRUE, it counts the simple members, and adds that to the sum of the counts for the compound members. This parameter has no effect for simple type variables.

Reimplemented in Grid.

Definition at line 308 of file BaseType.cc.

Referenced by Vector::element_count, Structure::element_count, Sequence::element_count, and Grid::element_count.

3.3.3.6 BaseType * BaseType::get_parent () `[virtual]`

Return a pointer to the Constructor or Vector which holds (contains) this variable. If this variable is at the top level, this method returns null.

NB: The modifier for this property is protected. It should only be modified by Constructor and Vector.

Returns:

A BaseType pointer to the variable's parent.

Definition at line 421 of file BaseType.cc.

3.3.3.7 bool BaseType::ops (BaseType ∗ *b*, int *op*, const string & *dataset*) [virtual]

The class relational operators.

This function contains the relational operators used by the constraint expression evaluator in the DDS class. Each class that wants to be able to evaluate relational expressions must overload this function. The implementation in BaseType returns false and prints an error message.

The *op* argument refers to a table generated by bison from the constraint expression parser. Use statements like the following to correctly interpret its value:

```
switch (op) {
case EQUAL:
return i1 == i2;
case NOT_EQUAL:
return i1 != i2;
case GREATER:
return i1 > i2;
case GREATER_EQL:
return i1 >= i2;
case LESS:
return i1 < i2;
case LESS_EQL:
return i1 <= i2;
case REGEXP:
cerr << "Regular expressions not valid for integer values" << endl;
return false;
default:
cerr << "Unknown operator" << endl;
return false;
}
```

This function is used by the constraint expression evaluator.

Parameters:

b The value with which the instance value is to be compared.

op An integer index indicating which relational operator is implied. Choose one from the following: `EQUAL`, `NOT_EQUAL`, `GREATER`, `GREATER_EQL`, `LESS`, `LESS_EQL`, and `REGEXP`.

dataset The name of the dataset from which the instance's data has come (or is to come).

Returns:

The boolean value of the comparison.

Reimplemented in Byte.

Definition at line 710 of file BaseType.cc.

Referenced by List::member, and Clause::value.

3.3.3.8 void BaseType::print_decl (ostream & *os*, string *space* = " ", bool *print_semi* = true, bool *constraint_info* = false, bool *constrained* = false) [virtual]

Print an ASCII representation of the variable structure.

Write the variable's declaration in a C-style syntax. This function is used to create textual representation of the Data Descriptor Structure (DDS). See *The DODS User Manual* for information about this structure.

A simple array declaration might look like this:

```
Float64 lat[lat = 180];
```

While a more complex declaration (for a Grid, in this case), would look like this:

```
Grid {
ARRAY:
Int32 sst[time = 404][lat = 180][lon = 360];
MAPS:
Float64 time[time = 404];
Float64 lat[lat = 180];
Float64 lon[lon = 360];
} sst;
```

Parameters:

os The output stream on which to print the declaration.

space Each line of the declaration will begin with the characters in this string. Usually used for leading spaces.

print_semi A boolean value indicating whether to print a semicolon at the end of the declaration.

constraint_info A boolean value indicating whether constraint information is to be printed with the declaration. If the value of this parameter is TRUE, `print_decl()` prints the value of the variable's `send_p()` flag after the declaration.

constrained If this boolean value is TRUE, the variable's declaration is only printed if is the `send_p()` flag is TRUE. If a constraint expression is in place, and this variable is not requested, the `send_p()` flag is FALSE.

See also:
DDS , DDS::CE

Reimplemented in Array.

Definition at line 588 of file BaseType.cc.

References send_p, and type_name.

Referenced by Vector::print_decl, Grid::print_decl, Array::print_decl, UInt32::print_val, UInt16::print_val, Str::print_val, Int32::print_val, Int16::print_val, Float64::print_val, Float32::print_val, and Byte::print_val.

3.3.3.9 virtual void BaseType::print_val (ostream & *os*, string *space* = "", bool *print_decl_p* = true) `[pure virtual]`

Prints the value of the variable.

Prints the value of the variable, with its declaration. This function is primarily intended for debugging DODS applications. However, it can be overloaded and used to do some useful things. Take a look at the asciival and writeval clients, both of which overload this to output the values of variables in different ways.

NB: This function uses C++'s iostream to handle creating the print representations of simple type variables. For floating point numbers this is set to six digits of precision by default. If you want more precision (IEEE 64-bit floats have 15 digits of precision, 32-bit floats have 8), use the setprecision() I/O manipulator.

Parameters:

os The output stream on which to print the value.

space This value is passed to the `print_decl()` function, and controls the leading spaces of the output.

print_decl_p A boolean value controlling whether the variable declaration is printed as well as the value.

Implemented in Array.

Referenced by Structure::print_all_vals, Array::print_decl, Sequence::print_one_row, Vector::print_val, and Grid::print_val.

3.3.3.10 virtual bool BaseType::read (const string & *dataset*) `[pure virtual]`

Reads data into a local buffer.

Put the data into a local buffer so that it may be sent to a client. This operation involves reading data from whatever source (often a local disk), and filling out the fields in the data type class. This is the heart of the DODS DAP Class operation. Much of the work of implementing a new DODS server API consists in creating the `read()` functions to read various data types.

Note that this function is only for DODS servers. It has no use on the client side of a DODS client/server connection. The DODS client and server communicate their data with `serialize()` and `deserialize()`.

This method is not implemented for the BaseType class, nor for its children. However, it should be implemented for the specialized children of those classes. For example, it is not implemented for the Float64 class, but does exist for the NCFloat64 class, specialized to read data from local netCDF files.

This method should be implemented to throw Error when it encounters an unrecoverable error.

For an example of use, see the netCDF library classes. The netCDF library is part of the DODS source distribution, and can be found under `/export/home/tomfool/DODS/src/nc-dods`.

In some sub-classes, such as Array or Grid, the `read()` function must explicitly take into account constraint information stored with the class data. For example, the Grid::read method will be called when only one component of the Grid is to be sent. Your implementation of Grid::read should check send_p() for each member of the Grid before reading that member to avoid reading data into memory that won't be sent (and thus is not needed in memory).

Returns:

The function returns a boolean value, with TRUE indicating that read() should be called again because there's more data to read, and FALSE indicating there's no more data to read. Note that this behavior is necessary to properly handle variables that contain Sequences. WRONG! (9/5/2001 jhrg) Sequences now are read in one shot. The return value of this method should always be false.

Parameters:

dataset A string naming the dataset from which the data is to be read. The meaning of this string will vary among data APIs.

See also:

BaseType

Implemented in Array.

Referenced by rvalue::bvalue, UInt32::ops, UInt16::ops, Str::ops, Int32::ops, Int16::ops, Float64::ops, Float32::ops, and Byte::ops.

3.3.3.11 bool BaseType::read_p ()

Returns the value of the read_p# flag.

Returns the value of the `read_p` flag. This flag is TRUE when the class instance contains a valid value, and FALSE before a valid value has been read.

Definition at line 342 of file BaseType.cc.

Referenced by UInt32::ops, UInt16::ops, Str::ops, Int32::ops, Int16::ops, Float64::ops, Float32::ops, Byte::ops, Sequence::read_row, Vector::serialize, UInt32::serialize, UInt16::serialize, Structure::serialize, Str::serialize, Int32::serialize, Int16::serialize, Grid::serialize, Float64::serialize, Float32::serialize, and Byte::serialize.

3.3.3.12 bool BaseType::send_p ()

Returns the value of the send_p# flag.

Returns the value of the `send_p` flag. This flag is TRUE if this variable is to be sent to the client. This is determined by evaluating the constraint expression. The `_send_p` flag is set to TRUE for all variables in the constraint expression's "projection" clause.

Definition at line 376 of file BaseType.cc.

Referenced by Grid::components, Vector::print_decl, Structure::print_decl, Sequence::print_decl, Grid::print_decl, print_decl, Array::print_decl, Grid::print_val, Grid::projection_yields_grid, DDS::send, Structure::serialize, and Grid::serialize.

3.3.3.13 virtual bool BaseType::serialize (const string & *dataset*, DDS & *dds*, XDR * *sink*, bool *ce_eval* = true) `[pure virtual]`

Move data to the net.

Sends the data from the indicated (local) dataset through the connection identified by the *sink* parameter. If the data is not already incorporated into the DDS object, read the data from the dataset.

This function is only used on the server side of the client/server connection, and is generally only called from the DDS::send() function. It has no BaseType implementation; each datatype child class supplies its own implementation.

Parameters:

dataset The (local) name of dataset to be read.

dds The Data Descriptor Structure object corresponding to this dataset. See *The DODS User Manual* for information about this structure.

sink A valid XDR pointer generally created with a call to `new_xdrstdio()`. This typically routes data to a TCP/IP socket.

ce_eval A boolean value indicating whether to evaluate the DODS constraint expression that may accompany this dataset. The constraint expression is stored in *dds*.

Returns:

This method always returns true. Older versions used the return value to signal success or failure.

Exceptions:

InternalErr .

Error .

See also:

DDS

Implemented in Byte.

Referenced by DDS::send, Structure::serialize, and Grid::serialize.

3.3.3.14 void BaseType::set_parent (BaseType ∗ *parent*) throw (InternalErr) [virtual]

Set the `parent` property for this variable. Only instances of Constructor or Vector should call this method.

Parameters:

parent Pointer to the Constructor of Vector parent variable.

Exceptions:

InternalErr thrown if called with anything other than a Constructor or Vector.

Definition at line 402 of file BaseType.cc.

Referenced by Vector::add_var, Structure::add_var, Sequence::add_var, Grid::add_var, and Vector::Vector.

3.3.3.15 void BaseType::set_read_p (bool *state*) [virtual]

Sets the value of the read_p# flag.

Sets the value of the `read_p` flag. This flag is TRUE when the class instance contains a valid value, and FALSE before a valid value has been read. This is meant to be called from the `read()` function. Data is ready to be sent when *both* the `_send_p` and `_read_p` flags are set to TRUE.

Parameters:

state The logical state to set the `read_p` flag.

Reimplemented in Grid.

Definition at line 356 of file BaseType.cc.

References name.

Referenced by Vector::set_read_p, Structure::set_read_p, Sequence::set_read_p, Grid::set_read_p, and Clause::value.

3.3.3.16 void BaseType::set_send_p (bool *state*) `[virtual]`

Sets the value of the `send_p` flag. This function is meant to be called from `serialize()`. Data is ready to be sent when *both* the `_send_p` and `_read_p` flags are set to TRUE.

Parameters:

state The logical state to set the `send_p` flag.

Reimplemented in Grid.

Definition at line 389 of file BaseType.cc.

Referenced by DDS::mark, Vector::set_send_p, Structure::set_send_p, Sequence::set_send_p, and Grid::set_send_p.

3.3.3.17 void BaseType::set_synthesized_p (bool *state*)

Set the synthesized flag. Before setting this flag be sure to set the `read_p()` state. Once this flag is set you cannot alter the state of the `read_p` flag!

See also:

synthesized_p()

Definition at line 328 of file BaseType.cc.

3.3.3.18 bool BaseType::synthesized_p ()

Returns true if the variable is a synthesized variable. A synthesized variable is one that is added to the dataset by the server (usually with a 'projection function'.

Definition at line 317 of file BaseType.cc.

3.3.3.19 string BaseType::toString () `[virtual]`

Write out the object's internal fields in a string. To be used for debugging when regular inspection w/ddd or gdb isn't enough.

Returns:
A string which shows the object's internal stuff.

Reimplemented in Sequence.

Definition at line 109 of file BaseType.cc.

Referenced by Sequence::add_var, GSEClause::GSEClause, and Sequence::toString.

3.3.3.20 virtual unsigned int BaseType::val2buf (void * *val*, bool *reuse* = false) `[pure virtual]`

Loads class data.

Store the value pointed to by *val* in the object's internal buffer. This function does not perform any checks, so users must be sure that the thing pointed to can actually be stored in the object's buffer. For example, an array cannot easily be fit into the data buffer for an Int32 object.

Only simple objects (Int, Float, Byte, and so on) and arrays and lists of these simple objects may be stored using this function. To put data into more complex constructor functions, use the functions provided by that class. For example, use the `var()` and `add_var()` members of the Grid class to manipulate data in that class.

Parameters:

val A pointer to the data to be inserted into the class data buffer.

reuse A boolean value, indicating whether the class internal data storage can be reused or not. If this argument is TRUE, the class buffer is assumed to be large enough to hold the incoming data, and it is *not* reallocated. If FALSE, new storage is allocated. If the internal buffer has not been allocated at all, this argument has no effect. This is currently used only in the Vector class.

Returns:
The size (in bytes) of the information copied from *val*.

See also:
Grid , Vector::val2buf

Implemented in Byte.

Referenced by Vector::var.

3.3.3.21 BaseType * BaseType::var (const string & *name*, btp_stack & *s*) `[virtual]`

Returns a pointer to a member of a constructor class.

This version of var(...) searches for *name* and returns a pointer to the BaseType object if found. It uses the same search algorithm as above when *exact_match* is false. In addition to returning a pointer to the variable, it pushes onto *s* a BaseType pointer to each constructor type that ultimately contains *name*.

\note{The BaseType implementation always returns null. }

Deprecated:

Parameters:

name Find the variable whose name is *name*.

s Record the path to *name*.

Returns:

A pointer to the named variable.

Reimplemented in Grid.

Definition at line 483 of file BaseType.cc.

3.3.3.22 BaseType ∗ BaseType::var (const string & *name* = "", bool *exact_match* = true, btp_stack ∗ *s* = 0) `[virtual]`

Returns a pointer to a member of a constructor class.

Returns a pointer to the contained variable in a composite class. The composite classes are those made up of aggregated simple data types. Array, Grid, and Structure are composite types, while Int and Float are simple types. This function is only used by composite classes. The BaseType implementation always returns null.

Several of the subclasses overload this function with alternate access methods that make sense for that particular data type. For example, the Array class defines a `*var(int i)` method that returns the ith entry in the Array data, and the Structure provides a `*var(Pix p)` function using a pseudo-index to access the different members of the structure.

Parameters:

name The name of the class member.

exact_match True if only interested in variables whose full names match `name` exactly. If false, returns the first variable whose name matches `name`. For example, if `name` is x and point.x is a variable, then var("x", false) would return a `BaseType` pointer to point.x. If `exact_match` was `true` then `name` would need to be "point.x" for `var` to return that pointer. This feature simplifies constraint expressions for datasets which have complex, nested, constructor variables.

Returns:

A pointer to the member named in the *name* argument. If no name is given, the function returns the first (only) variable. For example, an Array has only one variable, while a Structure can have many.

Reimplemented in Grid.

Definition at line 463 of file BaseType.cc.

Referenced by Vector::var.

3.3.3.23 virtual unsigned int BaseType::width () `[pure virtual]`

Returns the size of the class instance data.

Return the number of bytes that are required to hold the instance's value. In the case of simple types such as Int32, this is the size of one Int32 (four bytes). For a String or Url type, `width()` returns the number of bytes needed for a `String *` variable, not the bytes needed for all the characters, since that value cannot be determined from type information alone. For Structure, and other constructor types size() returns the number of bytes needed to store pointers to the C++ objects.

Implemented in Byte.

Referenced by Vector::buf2val, Vector::deserialize, Vector::serialize, Vector::val2buf, Vector::var, Vector::width, Structure::width, Sequence::width, and Grid::width.

3.3.3.24 xdrproc_t BaseType::xdr_coder ()

Returns a function used to encode elements of an array.

The `xdr_coder` function (also "filter primitive") is used to encode and decode each element in a multiple element data structure. These functions are used to convert data to and from its local representation to the XDR representation, which is used to transmit and receive the data. See `man xdr` for more information about the available XDR filter primitives.

Note that this class data is only used for multiple element data types. The simple data types (Int, Float, and so on), are translated directly.

Returns:

A C function used to encode data in the XDR format.

Definition at line 536 of file BaseType.cc.

Referenced by Vector::deserialize, and Vector::serialize.

The documentation for this class was generated from the following files:

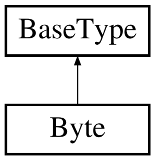
- BaseType.h
- BaseType.cc

3.4 Byte Class Reference

Holds a single byte.

`#include <Byte.h>`

Inheritance diagram for Byte::



Public Methods

- Byte (const string &n="")

  The Byte constructor.

- virtual ~**Byte** ()
- **Byte** (const Byte &copy_from)
- Byte & **operator=** (const Byte &rhs)
- virtual BaseType * **ptr_duplicate** ()=0
- virtual unsigned int width ()

  Returns the size of the class instance data.

- virtual bool serialize (const string &dataset, DDS &dds, XDR *sink, bool ce_eval=true)
- virtual bool deserialize (XDR *source, DDS *dds, bool reuse=false)

  Deserialize the char on stdin and put the result in `_BUF`.

- virtual bool read (const string &dataset)=0

  Reads data into a local buffer.

- virtual unsigned int val2buf (void *val, bool reuse=false)
- virtual unsigned int buf2val (void **val)

  Reads the class data.

- virtual void print_val (ostream &os, string space="", bool print_decl_p=true)

  Prints the value of the variable.

- virtual bool ops (BaseType *b, int op, const string &dataset)

  The class relational operators.

Protected Attributes

- dods_byte **_buf**

Friends

- class Int16
- class UInt16
- class Int32
- class UInt32
- class Float32
- class Float64

3.4.1 Detailed Description

Holds a single byte.

This class is used to hold eight bits of information. No sign information is implied in its value.

See also:
BaseType

Definition at line 46 of file Byte.h.

3.4.2 Constructor & Destructor Documentation

3.4.2.1 Byte::Byte (const string & *n* = "")

The Byte constructor.

The Byte constructor requires only the name of the variable to be created. The name may be omitted, which will create a nameless variable. This may be adequate for some applications.

Note:

Even though Byte is a cardinal type, xdr_char is *not* used to transport Byte arrays over the network. Instead, Byte is a special case handled in Array.

Parameters:

n A string containing the name of the variable to be created.

Definition at line 54 of file Byte.cc.

3.4.3 Member Function Documentation

3.4.3.1 unsigned int Byte::buf2val (void ** *val*) `[virtual]`

Reads the class data.

Reads the class data into the memory referenced by *val*. The caller must allocate enough storage to *val* to hold the class data. If *val* is NULL, however, memory will be allocated by this function with `new()`. Even if the memory is allocated this way, the caller is responsible for deallocating that memory. Array and List values for simple types are stored as C would store an array.

Parameters:

val A pointer to a pointer to the memory into which the class data will be copied. If the value pointed to is NULL, memory will be allocated to hold the data, and the pointer value modified accordingly. The calling program is responsible for deallocating the memory indicated by this pointer.

Returns:

The size (in bytes) of the information copied to *val*.

Implements BaseType.

Definition at line 148 of file Byte.cc.

References width.

3.4.3.2 bool Byte::ops (BaseType * *b*, int *op*, const string & *dataset*) `[virtual]`

The class relational operators.

This function contains the relational operators used by the constraint expression evaluator in the DDS class. Each class that wants to be able to evaluate relational expressions must overload this function. The implementation in BaseType returns false and prints an error message.

The *op* argument refers to a table generated by bison from the constraint expression parser. Use statements like the following to correctly interpret its value:

```
switch (op) {
case EQUAL:
return i1 == i2;
case NOT_EQUAL:
return i1 != i2;
case GREATER:
return i1 > i2;
case GREATER_EQL:
return i1 >= i2;
case LESS:
return i1 < i2;
case LESS_EQL:
return i1 <= i2;
case REGEXP:
cerr << "Regular expressions not valid for integer values" << endl;
return false;
default:
cerr << "Unknown operator" << endl;
return false;
}
```

This function is used by the constraint expression evaluator.

Parameters:

b The value with which the instance value is to be compared.

op An integer index indicating which relational operator is implied. Choose one from the following: `EQUAL`, `NOT_EQUAL`, `GREATER`, `GREATER_EQL`, `LESS`, `LESS_EQL`, and `REGEXP`.

dataset The name of the dataset from which the instance's data has come (or is to come).

Returns:

The boolean value of the comparison.

Reimplemented from BaseType.

Definition at line 177 of file Byte.cc.

References BaseType::read, read, BaseType::read_p, and BaseType::type.

3.4.3.3 void Byte::print_val (ostream & *os*, string *space* = "", bool *print_decl_p* = true) `[virtual]`

Prints the value of the variable.

Prints the value of the variable, with its declaration. This function is primarily intended for debugging DODS applications. However, it can be overloaded and used to do some useful things. Take a look at the asciival and writeval clients, both of which overload this to output the values of variables in different ways.

NB: This function uses C++'s iostream to handle creating the print representations of simple type variables. For floating point numbers this is set to six digits of precision by default. If you want more precision (IEEE 64-bit floats have 15 digits of precision, 32-bit floats have 8), use the setprecision() I/O manipulator.

Parameters:

os The output stream on which to print the value.

space This value is passed to the `print_decl()` function, and controls the leading spaces of the output.

print_decl_p A boolean value controlling whether the variable declaration is printed as well as the value.

Implements BaseType.

Definition at line 166 of file Byte.cc.

References BaseType::print_decl.

3.4.3.4 virtual bool Byte::read (const string & *dataset*) `[pure virtual]`

Reads data into a local buffer.

Put the data into a local buffer so that it may be sent to a client. This operation involves reading data from whatever source (often a local disk), and filling out the fields in the data type class. This is the heart of the DODS DAP Class operation. Much of the work of implementing a new DODS server API consists in creating the `read()` functions to read various data types.

Note that this function is only for DODS servers. It has no use on the client side of a DODS client/server connection. The DODS client and server communicate their data with `serialize()` and `deserialize()`.

This method is not implemented for the BaseType class, nor for its children. However, it should be implemented for the specialized children of those classes. For example, it is not implemented for the Float64 class, but does exist for the NCFloat64 class, specialized to read data from local netCDF files.

This method should be implemented to throw Error when it encounters an unrecoverable error.

For an example of use, see the netCDF library classes. The netCDF library is part of the DODS source distribution, and can be found under `/export/home/tomfool/DODS/src/nc-dods`.

In some sub-classes, such as Array or Grid, the `read()` function must explicitly take into account constraint information stored with the class data. For example, the Grid::read method will be called when only one component of the Grid is to be sent. Your implementation of Grid::read should check send_p() for each member of the Grid before reading that member to avoid reading data into memory that won't be sent (and thus is not needed in memory).

Returns:

The function returns a boolean value, with TRUE indicating that read() should be called again because there's more data to read, and FALSE indicating there's no more data to read. Note that this behavior is necessary to properly handle variables that contain Sequences. WRONG! (9/5/2001 jhrg) Sequences now are read in one shot. The return value of this method should always be false.

Parameters:

dataset A string naming the dataset from which the data is to be read. The meaning of this string will vary among data APIs.

See also:

BaseType

Implements BaseType.

Referenced by ops, and serialize.

3.4.3.5 bool Byte::serialize (const string & *dataset*, DDS & *dds*, XDR ∗ *sink*, bool *ce_eval* = true) `[virtual]`

Serialize the contents of member _BUF (the object's internal buffer, used to hold data) and write the result to stdout. If FLUSH is true, write the contents of the output buffer to the kernel. FLUSH is false by default. If CE_EVAL is true, evaluate the current constraint expression; only send data if the CE evaluates to true.

Note:

See the comment in BaseType about why we don't use XDR_CODER here.

Returns:

False if a failure to read, send or flush is detected, true otherwise.

Implements BaseType.

Definition at line 96 of file Byte.cc.

References DDS::eval_selection, read, and BaseType::read_p.

3.4.3.6 unsigned int Byte::val2buf (void * *val*, bool *reuse* = false) [virtual]

Store the value referenced by *val* in the object's internal buffer. *reuse* has no effect because this class does not dynamically allocate storage for the internal buffer.

Returns:

The size (in bytes) of the value's representation.

Implements BaseType.

Definition at line 133 of file Byte.cc.

References width.

3.4.3.7 unsigned int Byte::width () [virtual]

Returns the size of the class instance data.

Return the number of bytes that are required to hold the instance's value. In the case of simple types such as Int32, this is the size of one Int32 (four bytes). For a String or Url type, `width()` returns the number of bytes needed for a `String *` variable, not the bytes needed for all the characters, since that value cannot be determined from type information alone. For Structure, and other constructor types size() returns the number of bytes needed to store pointers to the C++ objects.

Implements BaseType.

Definition at line 77 of file Byte.cc.

Referenced by buf2val, and val2buf.

3.4.4 Friends And Related Function Documentation

3.4.4.1 friend class Int16 [friend]

This class allows Int16, ..., Float64 access to `_buf` to simplify and speed up the relational operators.

NB: According to Stroustrup it does not matter where (public, private or protected) friend classes are declared.

Definition at line 52 of file Byte.h.

The documentation for this class was generated from the following files:

- Byte.h

- Byte.cc

3.5 Clause Struct Reference

Holds a fragment of a constraint expression.

`#include <Clause.h>`

Public Methods

- **Clause** (const int oper, rvalue ∗a1, rvalue_list ∗rv)
- **Clause** (bool_func func, rvalue_list ∗rv)
- **Clause** (btp_func func, rvalue_list ∗rv)
- **Clause** ()
- virtual ~**Clause** ()
- bool OK ()

  Checks the "representation invariant" of a clause.
- bool boolean_clause ()

  Return true if the clause returns a boolean value.
- bool value_clause ()

  Return true if the clause returns a value in a BaseType pointer.
- bool value (const string &dataset, DDS &dds)

  Evaluate a clause which returns a boolean value.
- bool value (const string &dataset, DDS &dds, BaseType ∗∗value)

  Evaluate a clause that returns a value via a BaseType pointer.

3.5.1 Detailed Description

Holds a fragment of a constraint expression.

The selection part of a a DODS constraint expression may contain one or more clauses, separated by ampersands (&). This is modeled in the DDS class structure as a singly-linked list of Clause objects. In addition, a constraint expression may be a single function call, also represented in the DDS using an instance of Clause.

Each clause object can contain a representation of one of three possible forms:

1. A relational clause, where an operator tests the relation between two operands. This kind of clause evaluates to a boolean value. For example: `a > b`.
2. A boolean function, where some function operates on arguments in the clause to return a boolean value. For example, consider a scalar A and a list L. The clause `find(A,L)` might return TRUE if A is a member of L (if the `find()` function is defined).
3. A clause that returns a pointer to a DODS BaseType value. This is a clause that evaluates to some data value (be it scalar or vector). For example, `sig0()` might be included in the constraint expression parser to calculate density from pressure, temperature, and salinity. In this case, `sig0(p,t,s)` would be a clause that evaluates to a data value.

This might be a bit confusing; in the first, and by far more common, form of constraint expressions (CEs) only the first two types of clauses may appear. In the second form of the CE only the last type of clause may occur. The Clause class, however, can store them all.

The Clause object holds the constraint expression *after* it has been parsed. The parser renders the relational operator into an integer, and the functions into pointers.

See also:
DDS::parse_constraint

Definition at line 67 of file Clause.h.

The documentation for this struct was generated from the following files:

- Clause.h
- Clause.cc

3.6 Cmp Class Template Reference

`#include <Operators.h>`

Static Public Methods

- bool **eq** (T1 v1, T2 v2)
- bool **ne** (T1 v1, T2 v2)
- bool **gr** (T1 v1, T2 v2)
- bool **ge** (T1 v1, T2 v2)
- bool **lt** (T1 v1, T2 v2)
- bool **le** (T1 v1, T2 v2)
- bool **re** (T1 v1, T2 v2)

3.6.1 Detailed Description

template<class T1, class T2> class Cmp< T1, T2 >

Compare two numerical types, both of which are either signed or unsigned. This class is one implementation of the comparison policy used by rops.

See also:
rops , USCmp @SUCmp

Definition at line 37 of file Operators.h.

The documentation for this class was generated from the following file:

- Operators.h

3.7 Connect Class Reference

Holds information about the link from a DODS client to a dataset.

```
#include <Connect.h>
```

Public Methods

- Connect (string name, bool www_verbose_errors=false, bool accept_-deflate=true, string uname="", string password="")

  Create an instance of Connect.

- Connect (const Connect &copy_from)
- virtual ~**Connect** ()
- Connect & **operator=** (const Connect &rhs)
- bool get_www_errors_to_stderr ()

  Gets the state of the `www_errors_to_stderr` *property.*

- void set_www_errors_to_stderr (bool state)

  Sets the `www_errors_to_stderr` *property.*

- string get_accept_types ()

  Retrieve a list of types a DODS client can understand.

- void set_accept_types (const string &types)

  Set the types a client can accept.

- string get_cache_control ()

  Get the cache control directive.

- void set_cache_control (const string &caching)

  Set the cache control header value.

- bool fetch_url (string &url, bool async=false) throw (Error)

  Dereference a URL.

- FILE * output ()

  Access the information contained in this Connect instance.

- void close_output ()

  Close the object's output stream if it is not NULL or STDOUT.

- XDR ∗ source ()

  Access the XDR input stream (source) for this connection.

- bool is_local ()

  Does this object refer to a local file?

- string URL (bool CE=true)

  Get the object's URL.

- string CE ()

  Get the Connect's constraint expression.

- ObjectType type ()

  What type is the most recent object sent from the server?

- EncodingType encoding ()

  What type of encoding was used on the data in the stream?

- string server_version ()
- DAS & das ()

  Return a reference to the Connect's DAS object.

- DDS & dds ()

  Return a reference to the Connect's DDS object.

- Error & error ()

  Get a reference to the last Error object.

- void ∗ gui ()

  Returns a pointer to a Gui object.

- bool request_das (bool gui=false, const string &ext="das") throw (Error, InternalErr)

  Get the DAS from a server.

- bool request_dds (bool gui=false, const string &ext="dds") throw (Error, InternalErr)

  Get the DDS from a server.

- DDS * request_data (string expr, bool gui=false, bool async=false, const string &ext="dods") throw (Error, InternalErr)

  Calls `fetch_url()` *to retrieve data from the server.*

- DDS * read_data (FILE *data_source, bool gui=false, bool async=false) throw (Error, InternalErr)

  Read cached data from a disk file.

- void set_credentials (string u, string p)

  Set the credentials for responding to challenges while dereferencing URLs.

- void disable_cache ()

Friends

- BOOL **dods_username_password** (HTRequest *request, HTAlertOpcode, int, const char *, void *, HTAlertPar *reply)
- BOOL **dods_progress** (HTRequest *request, HTAlertOpcode op, int, const char *, void *input, HTAlertPar *)
- int **timeout_handler** (HTRequest *request)
- int **description_handler** (HTRequest *request, HTResponse *response, const char *token, const char *val)
- int **encoding_handler** (HTRequest *request, HTResponse *response, const char *token, const char *val)
- int **server_handler** (HTRequest *request, HTResponse *response, const char *token, const char *val)
- int **header_handler** (HTRequest *request, HTResponse *response, const char *token, const char *val)
- void **process_www_errors** (HTList *listerr, HTRequest *request) throw (Error)

3.7.1 Detailed Description

Holds information about the link from a DODS client to a dataset.

Connect objects are used as containers for information pertaining to the connection a user program makes to a dataset. The dataset may be either local (for example, a file on the user's own computer) or a remote dataset. In the latter case a DODS URL will be used to reference the dataset, instead of a filename.

Connect contains methods which can be used to read the DODS DAS and DDS objects from the remote dataset as well as reading data. The class understands in a rudimentary way how DODS constraint expressions are formed and how to manage them.

Connect also provides additional services such as automatic decompression of compressed data, transmission progress reports and error processing. Refer to the Gui and Error classes for more information about these features. See the DODSFilter class for information on servers that compress data.

Note:

This file may be built two different ways. If the compile-time symbol `GUI` is defined, a version of Connect is built which will use a simple graphical window to display transmission status information. The graphical display is useful for interactive clients, but won't work for non-interactive programs. For those clients, build the DAP library with `GUI` *undefined*. Without `GUI` defined, errors will result in Connect throwing an Error object. Clients can catch these objects and try to correct the error.

See also:

DDS , DAS , DODSFilter , Error , Gui

Author:

jhrg

Definition at line 158 of file Connect.h.

3.7.2 Constructor & Destructor Documentation

3.7.2.1 Connect::Connect (string *name*, bool *www_verbose_errors* = false, bool *accept_deflate* = true, string *uname* = "", string *password* = "")

Create an instance of Connect.

The Connect constructor requires a `name`, which is the URL to which the connection is to be made. You can specify that you want to see the `verbose` form of any WWW library errors. This can be useful for debugging. The default is to suppress these errors. Callers can use the `accept_deflate` parameter to request that servers are told the client (caller of Connect ctor) *can* process return documents that are compressed with gzip.

Parameters:

name The URL for the virtual connection.

www_verbose_errors False: show only WWW Fatal errors, True: show WWW informational messages, too. This affects message display but not exceptions. If Connect is compiled to throw exceptions for certain WWW errors, it will do so regardless of the value of this parameter.

accept_deflate Provides compile-time control for on-the-fly compression. If True clients will ask servers to compress responses.

uname If given along woth password, supply this as the Username in all HTTP requests.

password Use this as the password with `uname` above.

Definition at line 927 of file Connect.cc.

3.7.2.2 Connect::Connect (const Connect & *copy_from*)

The Connect copy constructor.

Definition at line 1000 of file Connect.cc.

3.7.3 Member Function Documentation

3.7.3.1 string Connect::CE ()

Get the Connect's constraint expression.

Return the constraint expression (CE) part of the Connect URL. Note that this CE is supplied as part of the URL passed to the Connect's constructor. It is not the CE passed to the `request_data()` function.

Returns:

A string containing the constraint expression (if any) submitted to the Connect object's constructor.

Definition at line 1782 of file Connect.cc.

3.7.3.2 void Connect::close_output ()

Close the object's output stream if it is not NULL or STDOUT.

Close the output stream of the Connect object. This closes the FILE pointer returned by `output()`. In addition, it also deletes the internal XDR stream object, although users should not have to know about that.

Definition at line 1300 of file Connect.cc.

3.7.3.3 DAS & Connect::das ()

Return a reference to the Connect's DAS object.

All DODS datasets define a Data Attribute Structure (DAS), to hold a variety of information about the variables in a dataset. This function returns the DAS for the dataset indicated by this Connect object.

Returns:
A reference to the DAS object.

See also:
DAS

Definition at line 1803 of file Connect.cc.

3.7.3.4 DDS & Connect::dds ()

Return a reference to the Connect's DDS object.

All DODS datasets define a Data Descriptor Structure (DDS), to hold the data type of each of the variables in a dataset. This function returns the DDS for the dataset indicated by this Connect object.

Returns:
A reference to the DDS object.

See also:
DDS

Definition at line 1820 of file Connect.cc.

3.7.3.5 void Connect::disable_cache ()

Disable any further use of the client-side cache. In a future version of this software, this should be handled so that the www library is not initialized with the cache running by default.

Definition at line 1862 of file Connect.cc.

3.7.3.6 EncodingType Connect::encoding ()

What type of encoding was used on the data in the stream?

During the parse of the message headers returned from the dereferenced URL, the encoding type is set. Use this function to read that type information. This will be valid *before* the return object is completely parsed so it can be used to decide which decoder to call (if any) to read the data remaining in the input stream.

The encoding types are currently limited to x-plain (no special decoding required) and x-gzip (compressed using GNU's gzip).

Returns:
The type of the compression.

See also:
EncodingType

Definition at line 1347 of file Connect.cc.

3.7.3.7 Error & Connect::error ()

Get a reference to the last Error object.

The DODS server uses Error objects to signal error conditions to the client. If an error condition has occurred while fetching a URL, the Connect object will contain an Error object with information about that error. The Error object may also contain a program to run to remedy the error. This function returns the latest Error object received by the Connect object.

Returns:
The last Error object sent from the server. If no error has been sent from the server, returns a reference to an empty error object.

See also:
Error

Definition at line 1841 of file Connect.cc.

3.7.3.8 bool Connect::fetch_url (string & *url*, bool *async* = false) throw (Error)

Dereference a URL.

Fetch the contents of the indicated URL and put its contents into an output file. A pointer to this file can be retrieved with the `output()` function. *A program that uses that FILE pointer must be sure not to close it with* `fclose()`. Instead, use `Connect::close_output()`. If *async* is TRUE, then the operation is asynchronous, and the function returns before the data transfer is complete. This method is here so that Connect can be used to read from any URL, not just URLs which return

DAS, DDS or DataDDS objects. Because of this, `fetch_url` neither automatically appends a suffix nor does it route the response through any of the parsers which decode responses from a DODS server.

This method also scans the URL for a username/passwd. If present, extracts them, places them in the Connect object, and rebuilds the URL without them.

This method is here so that Connect can be used to read from any URL, not just URLs which return DAS, DDS or DataDDS objects. Because of this, `fetch_url` neither automatically appends a suffix nor does it route the response through any of the parsers which decode responses from a DODS server. In addition, `fetch_url` also does *not* escape any characters in the URL. That is the responsibility of the caller. This includes characters that break URLs and those that break DODS CEs.

Note:

Note that the asynchronous transfer feature of DODS is not currently enabled. All invocations of this function will be synchronous, no matter what the value of the *async* parameter.

Returns:

Always returns true.

Exceptions:

Error indicates some problem reading from the web server (not the DODS server, other methods report those errors).

Parameters:

url A string containing the URL to be dereferenced. The data referred to by this URL will wind up available through a file pointer retrieved from the `output()` function.

async Not currently implemented.

See also:

Connect::output

Definition at line 1211 of file Connect.cc.

3.7.3.9 string Connect::get_accept_types ()

Retrieve a list of types a DODS client can understand.

Gets the current string of 'accepted types.' This string lists all of the DODS datatypes that the client can grok.

See also:

set_accepted_types

Returns:
A string listing the types this client declares to servers it can understand.

Definition at line 1129 of file Connect.cc.

3.7.3.10 string Connect::get_cache_control ()

Get the cache control directive.

Get the string which describes the default cache control value. This is sent with all outgoing messages.

Note:
N.b. The libwww 5.2.9 cache does not honor this.

Returns:
The cache control header value.

Definition at line 1156 of file Connect.cc.

3.7.3.11 bool Connect::get_www_errors_to_stderr ()

Gets the state of the `www_errors_to_stderr` property.

If TRUE this means that http errors will be printed to stderr in addition to being reported in the Error object. If FALSE only the Error object will be used.

Returns:
TRUE if WWW errors should got to stderr, FALSE if only the Error object should be used.

Definition at line 1090 of file Connect.cc.

3.7.3.12 void * Connect::gui ()

Returns a pointer to a Gui object.

The DODS client can display graphic information to a user with the DODS Graphical User Interface (GUI). Typically used for a progress indicator, the GUI is simply a shell capable of interpreting arbitrary graphical commands (with tcl/tk). The Gui object is created anew when the Connect object is first created. This function returns a pointer to the Gui object, so you can modify the GUI as desired.

This member will be removed since its presence makes it hard to build Gui and non-gui versions of the DAP. The Gui object is accessed is in Connect and Error, but in the later case an instance of Gui is always passed to the instance of Error. Thus, even though it is a dubious design, we can use the private member `_gui` and pass the pointer to outside classes. Eventually, Connect must be redesigned.

Returns:
a pointer to the Gui object associated with this connection.

Deprecated:

See also:
Gui

Definition at line 1719 of file Connect.cc.

3.7.3.13 bool Connect::is_local ()

Does this object refer to a local file?

The Connect class can be used for "connections" to local files. This means that local files can continue to be accessed with a DODS-compliant API.

Returns:
Return TRUE if the Connect object refers to a local file, otherwise returns FALSE.

Definition at line 1737 of file Connect.cc.

3.7.3.14 FILE * Connect::output ()

Access the information contained in this Connect instance.

Returns a file pointer which can be used to read the data fetched from a URL.

Note that occasionally this may be directed to `stdout`. If this is the case, users should avoid closing it.

See also:
Connect::fetch_url

Returns:
A `(FILE *)` indicating a file containing the data received from a dereferenced URL.

Definition at line 1267 of file Connect.cc.

3.7.3.15 DDS ∗ Connect::read_data (FILE ∗ *data_source*, bool *gui_p* = false, bool *async* = false) throw (Error, InternalErr)

Read cached data from a disk file.

Returns:

A reference to the DataDDS object which contains the variables (BaseType pointers) generated from the DDS sent with the data.

Parameters:

data_source A file pointer to the cache file.

gui_p If this is TRUE, use the DODS client GUI. See the Gui class for a description of this feature.

async If this is TRUE, this function reads data asynchronously, returning before the read completes. Synchronous reads are the default, and the only possible action as of DODS version 2.15.

See also:

DataDDS , Gui

Definition at line 1671 of file Connect.cc.

3.7.3.16 bool Connect::request_das (bool *gui_p* = false, const string & *ext* = "das") throw (Error, InternalErr)

Get the DAS from a server.

Reads the DAS corresponding to the dataset in the Connect object's URL. Although DODS does not support usig CEs with DAS requests, if present in the Connect object's instance, they will be escaped and passed as the query string of the request.

Returns:

TRUE if the DAS was successfully received. FALSE otherwise.

Parameters:

gui_p If TRUE, use the client GUI. Most DAS's are too small to make this worthwhile.

ext The extension to append to the URL to retrieve the dataset DAS. This parameter is included for compatibility with future versions of the DODS software. It currently defaults to the only possible working value, "das".

Definition at line 1378 of file Connect.cc.

3.7.3.17 DDS * Connect::request_data (string *expr*, bool *gui_p* = false, bool *async* = false, const string & *ext* = "dods") throw (Error, InternalErr)

Calls `fetch_url()` to retrieve data from the server.

Reads data from the Connect object's server. This method sets up the BaseType variables in a DDS, and sends a request using `fetch_url()`. Upon return, it caches the data on a disk, then unpacks it into the DDS storage. Unlike `fetch_url`, this method escapes the CE part of the URL.

Returns:

A reference to the DataDDS object which contains the variables (BaseType pointers) generated from the DDS sent with the data. These variables are guaranteed to be large enough to hold the data, even if the constraint expression changed the type of the variable from that which appeared in the original DDS received from the dataset when this connection was made.

Parameters:

expr A string containign a constraint expression. The function adds the clauses of this constraint expression to the Connect object's original CE. If the constraint expression contains one or more Sequences, these must be the *last* objects specified in the projection clause. If you request N variables and M of them are Sequences, all the M sequences must follow the N-M other variables.

gui_p If this is TRUE, use the DODS client GUI. See the Gui class for a description of this feature.

async If this is TRUE, this function reads data asynchronously, returning before the read completes. Synchronous reads are the default, and the only possible action as of DODS version 2.15.

ext The extension to append to the URL to retrieve the dataset data. This parameter is included for compatibility with future versions of the DODS software. It currently defaults to the only possible working value, "dods".

See also:

DataDDS , Gui

Definition at line 1621 of file Connect.cc.

3.7.3.18 bool Connect::request_dds (bool *gui_p* = false, const string & *ext* = "dds") throw (Error, InternalErr)

Get the DDS from a server.

Reads the DDS corresponding to the dataset in the Connect object's URL. Although CEs are rarely used with this method, if present it will be escaped.

Returns:
TRUE if the DDS was successfully received. FALSE otherwise.

Parameters:
gui_p If TRUE, use the client GUI. Most DDS's are too small to make this worthwhile.
ext The extension to append to the URL to retrieve the dataset DDS. This parameter is included for compatibility with future versions of the DODS software. It currently defaults to the only possible working value, "dds".

Definition at line 1459 of file Connect.cc.

3.7.3.19 string Connect::server_version ()

Returns a string containing the version of DODS used by the server.

Definition at line 1355 of file Connect.cc.

3.7.3.20 void Connect::set_accept_types (const string & *types*)

Set the types a client can accept.

Sets the list of accepted types. This string is meant to list all of the DODS datatypes that the client can grok and is sent to the server using the XDODS-Accept-Types MIME header. The server will try to only send back to the client datatypes that are listed. If the value of this header is 'All', then the server assumes that the client can process all of the DODS datatypes. If only one or two types are *not* understood, then they can be listed, each one prefixed by '!'. Thus, if a client does not understand 'Sequences', it could set types to '!Sequences' as opposed to listing all of the DODS datatypes. Multiple values are separated by commas (,).

Not all servers will honor this and some requests may not be possible to express with a very limited set of datatypes.

Note:
By default (if this function is never called) the value 'All' is used.

Parameters:
types The string listing datatypes understood by this client.

Definition at line 1116 of file Connect.cc.

3.7.3.21 void Connect::set_cache_control (const string & *caching*)

Set the cache control header value.

http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html

See also:
get_cache_control , HTTP/1.1 Specification (RFC 2616), sec. 14.9.

Parameters:
caching A string containing the desired caching directive. Should be `no-cache` to disable caching.

Definition at line 1143 of file Connect.cc.

3.7.3.22 void Connect::set_credentials (string *u*, string *p*)

Set the credentials for responding to challenges while dereferencing URLs.

Parameters:
u The username.

p The password.

See also:
extract_auth_info()

Definition at line 1852 of file Connect.cc.

3.7.3.23 void Connect::set_www_errors_to_stderr (bool *state*)

Sets the `www_errors_to_stderr` property.

This controls where http errors get printed.

See also:
get_www_errors_to_stderr

Parameters:
state The desired state of the property. TRUE means that http errors get printed on stderr as well as in the Error object. FALSE prints them only in the Error object.

Definition at line 1075 of file Connect.cc.

3.7.3.24 XDR ∗ Connect::source ()

Access the XDR input stream (source) for this connection.

The data retrieved from a remote DODS server will be in XDR format. Use this function to initialize an XDR decoder for that data and to return an XDR pointer to the data.

Returns:
Returns a XDR pointer tied to the current output stream.

See also:
Connect::output

Definition at line 1284 of file Connect.cc.

3.7.3.25 ObjectType Connect::type ()

What type is the most recent object sent from the server?

During the parse of the message headers returned from the dereferenced URL, the object type is set. Use this function to read that type information. This will be valid *before* the return object is completely parsed so it can be used to decide which parser to call to read the data remaining in the input stream.

The object types are Data, DAS, DDS, Error, and undefined.

Returns:
The type of the object.

See also:
ObjectType

Definition at line 1327 of file Connect.cc.

3.7.3.26 string Connect::URL (bool *ce* = true)

Get the object's URL.

Return the Connect object's URL in a string. The URL was set by the class constructor, and may not be reset. If you want to open another URL, you must create another Connect object. There is a Connections class created to handle the management of multiple Connect objects.

See also:
Connections

Returns:

A string containing the URL of the data to which the Connect object refers. If the object refers to local data, the function returns the null string.

Parameters:

ce If TRUE, the returned URL will include any constraint expression enclosed with the Connect object's URL (including the `?`). If FALSE, any constraint expression will be removed from the URL. The default is TRUE.

Definition at line 1759 of file Connect.cc.

The documentation for this class was generated from the following files:

- Connect.h
- Connect.cc

3.8 Connections Class Template Reference

Holds an ensemble of Connect objects.

`#include <Connections.h>`

Public Methods

- Connections (int max_connections=MAX_CONNECTIONS)
- Connections (const Connections< T > &cs)
- virtual ~**Connections** ()
- Connections & **operator=** (const Connections< T > &rhs)
- T & operator[ ] (int i)

  Reference a Connect object in a Connection collection. Reference a single Connect object from a set.

- int add_connect (T c)

  Add a Connect object to the array.

- void del_connect (int i)

  Delete a Connect object from the array.

3.8.1 Detailed Description

template<class T> class Connections< T >

Holds an ensemble of Connect objects.

This object holds a collection of Connect objects. It contains methods to add and delete Connect objects from the collection, and defines the index operator ([]) to retrieve individual objects.

See also:
Connect

Definition at line 48 of file Connections.h.

3.8.2 Constructor & Destructor Documentation

3.8.2.1 template<class T> Connections< T >::Connections (int *i* = MAX_CONNECTIONS)

The constructor for the Connections object.

Parameters:

i The maximum number of open Connect objects this Connections object can hold. This number should not be bigger than the `MAX_CONNECTIONS` parameter in the Connections header file.

Definition at line 52 of file Connections.cc.

3.8.2.2 template<class T> Connections< T >::Connections (const Connections< T > & *cs*)

The specialized copy constructor.

Definition at line 64 of file Connections.cc.

References _max_con.

3.8.3 Member Function Documentation

3.8.3.1 template<class T> int Connections< T >::add_connect (T *c*)

Add a Connect object to the array.

Returns:

The index of the newly added Connect object or -1 if the allocated space for the array is full.

Parameters:

c The Connect object to add to the array.

Definition at line 134 of file Connections.cc.

3.8.3.2 template<class T> void Connections< T >::del_connect (int *i*)

Delete a Connect object from the array.

Remove a connection from the array of managed conections. The caller is responsible for deleting the Connect $*$ BEFORE calling this member function.

Returns:
void

Parameters:
i The integer index of the object to remove from the array.

Definition at line 163 of file Connections.cc.

3.8.3.3]

template<class T> T & Connections< T >::operator[ ] (int *i*)

Reference a Connect object in a Connection collection. Reference a single Connect object from a set.

This operator is how you retrieve Connect objects from the Connection collection.

Returns:
This function returns a reference to an element of the Connections array. It returns a reference so that this operator will work on both the left and right side of the assignment operator.

Definition at line 117 of file Connections.cc.

The documentation for this class was generated from the following files:

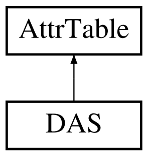
- Connections.h
- Connections.cc

3.9 DAS Class Reference

Hold attribute data for a DODS dataset.

```
#include <DAS.h>
```

Inheritance diagram for DAS::



Public Methods

- DAS (AttrTable *dflt=(AttrTable *) NULL, unsigned int sz=0)
- DAS (AttrTable *attr_table, string name)
- virtual ~**DAS** ()
- Pix first_var ()

  Returns a pointer to the first attribute table.

- void next_var (Pix &p)

  Increments an attribute table pointer to indicate the next table in the series.

- string get_name (Pix p)

  Returns the name of the indicated attribute table.

- void print (ostream &os=cout, bool dereference=false)

  Creates an ASCII representation of a DAS on the given output stream.

- AttrTable * get_table (Pix p)

  Returns the indicated attribute table.

- AttrTable * get_table (const string &name)

  Returns the attribute table with the given name string.

- AttrTable * get_table (const char *name)

  Returns the attribute table with the given name.

- AttrTable ∗ add_table (const string &name, AttrTable ∗at)
  Adds an attribute table to the DAS.
- AttrTable ∗ add_table (const char ∗name, AttrTable ∗at)
  Adds an attribute table to the DAS.

parse()

Reads a DAS in from an external source.

- void parse (string fname)
  Reads a DAS from the named file.
- void parse (int fd)
  Read attributes from a file descriptor.
- void parse (FILE ∗in=stdin)
  Reads a DAS from an open file descriptor.

Protected Methods

- AttrTable ∗ **das_find** (string name)

3.9.1 Detailed Description

Hold attribute data for a DODS dataset.

The Data Attribute Structure is a set of name-value pairs used to describe the data in a particular dataset. The name-value pairs are called the "attributes". The values may be of any of the DODS simple data types (Byte, Int16, UInt16, Int32, UInt32, Float32, Float64, String and URL), and may be scalar or vector. Note that all values are actually stored as String data, making the easy to read/check using a web browser. \label{api:das}

A value may also consist of a set of other name-value pairs. This makes it possible to nest collections of attributes, giving rise to a hierarchy of attributes. DODS uses this structure to provide information about variables in a dataset. For example, consider the dataset used in the DDS example earlier.

In the following example of a DAS, several of the attribute collections have names corresponding to the names of variables in the DDS example. The attributes in that collection are said to belong to that variable. For example, the `lat` variable has an attribute "units" of "degrees_north".

```
Attributes {
    GLOBAL {
        String title "Reynolds Optimum Interpolation (OI) SST";
    }
    lat {
        String units "degrees_north";
        String long_name "Latitude";
        Float64 actual_range 89.5, -89.5;
    }
    lon {
        String units "degrees_east";
        String long_name "Longitude";
        Float64 actual_range 0.5, 359.5;
    }
    time {
        String units "days since 1-1-1 00:00:00";
        String long_name "Time";
        Float64 actual_range 726468., 729289.;
        String delta_t "0000-00-07 00:00:00";
    }
    sst {
        String long_name "Weekly Means of Sea Surface Temperature";
        Float64 actual_range -1.8, 35.09;
        String units "degC";
        Float64 add_offset 0.;
        Float64 scale_factor 0.0099999998;
        Int32 missing_value 32767;
    }
}
```

Attributes may have arbitrary names, although in most datasets it is important to choose these names so a reader will know what they describe. In the above example, the "GLOBAL" attribute provides information about the entire dataset.

Data attribute information is an important part of the the data provided to a DODS client by a server, and the DAS is how this data is packaged for sending (and how it is received).

The DAS class is simply a sequence of attribute tables and names. It may be thought of as the top level of the attribute hierarchy.

See also:
DDS , AttrTable

Definition at line 105 of file DAS.h.

3.9.2 Constructor & Destructor Documentation

3.9.2.1 DAS::DAS (AttrTable * *dflt* = (AttrTable *)NULL, unsigned int *sz* = 0)

Create a DAS from a single attribute table.

Note:
In an older version of this class, `dflt` and `sz` initialized a hash table. That is no longer used and these params should no longer matter. Note that this constructor is effectively the empty constructor. 11/23/98 jhrg

Parameters:
dflt A pointer to a valid attribute table.

sz The number of entries in the table. This is unused. It was part of the ctor when DAS used the old GNU VHMap class. I switched from that to a SLList of struct toplevel_entry objects because the VHMap class had bugs I didn't want to fix. 11/23/98 jhrg

Definition at line 64 of file DAS.cc.

3.9.2.2 DAS::DAS (AttrTable * *attr*, string *name*)

Create a DAS object with one attribute table. Use `append_attr()` to add additional attributes.

See also:
append_attr()

Parameters:
attr The initial AttrTable.

name The name of the DAS parent structure.

Definition at line 76 of file DAS.cc.

References AttrTable::append_container.

3.9.3 Member Function Documentation

3.9.3.1 AttrTable * DAS::get_table (const char * *name*)

Returns the attribute table with the given name.

This function is necessary because (char *) arguments will be converted to Pixs (and not strings). Without this member function get_table(name) needs a cast, and it seems tough to believe folks will always remember that.

Definition at line 141 of file DAS.cc.

References get_table.

3.9.3.2 void DAS::parse (FILE * *in* = stdin)

Reads a DAS from an open file descriptor.

Read attributes from in (which defaults to stdin). If dasrestart() fails, return false, otherwise return the status of dasparse().

Definition at line 232 of file DAS.cc.

3.9.3.3 void DAS::parse (int *fd*)

Read attributes from a file descriptor.

If the file descriptor cannot be fdopen'd, return false, otherwise return the status of the mfunc parse.

Note:

Added call to dup() within fdopen so that once the FILE * is closed the decriptor fd will not also be closed (instead the duplicate descriptor will be closed). Thus further information can be read from the descriptor fd.

Definition at line 205 of file DAS.cc.

References parse.

3.9.3.4 void DAS::parse (string *fname*)

Reads a DAS from the named file.

Read attributes from a file. Returns false if unable to open the file, otherwise returns the result of the mfunc parse.

Definition at line 181 of file DAS.cc.

Referenced by parse, and DODSFilter::read_ancillary_das.

3.9.3.5 void DAS::print (ostream & *os* = cout, bool *dereference* = false)

Creates an ASCII representation of a DAS on the given output stream.

Write attributes from tables to 'out' (which defaults to stdout). Return true.

When an identifier contains a character that contains characters that cannot be present in a URL (e.g., a space) AttrTable::print replaces those characters with WWW escape codes. 7/13/2001 jhrg

Definition at line 267 of file DAS.cc.

References AttrTable::print.

Referenced by DODSFilter::send_das.

The documentation for this class was generated from the following files:

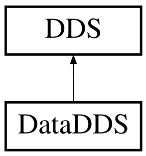
- DAS.h
- DAS.cc

3.10 DataDDS Class Reference

Holds a DODS DDS.

```
#include <DataDDS.h>
```

Inheritance diagram for DataDDS::



3.10.1 Detailed Description

Holds a DODS DDS.

This class adds some useful state information to the DDS structure. It is for use on the client side of the DODS connection.

See also:
Connect

Definition at line 39 of file DataDDS.h.

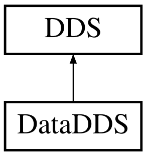
The documentation for this class was generated from the following files:

- DataDDS.h
- DataDDS.cc

3.11 DDS Class Reference

```
#include <DDS.h>
```

Inheritance diagram for DDS::



Public Methods

- DDS (const string &n="")
- DDS (const DDS &dds)
- virtual ~**DDS** ()
- DDS & **operator=** (const DDS &rhs)
- void add_var (BaseType *bt)

  Adds a variable to the DDS.

- void del_var (const string &n)

  Removes a variable from the DDS.

- BaseType * **var** (const string &n)
- BaseType * **var** (const char *n)
- BaseType * var (const string &n, btp_stack &s)
- BaseType * var (const string &n, btp_stack *s=0)

  Find the variable with the given name.

- BaseType * var (const char *n, btp_stack *s=0)

  Returns a pointer to the named variable.

- BaseType * var (Pix p)
- Pix first_var ()

  Returns the first variable in the DDS.

- void next_var (Pix &p)

  Increments the DDS variable counter to point at the next variable.

- int num_var ()

  Returns the number of variables in the DDS.

- void append_constant (BaseType *btp)
- bool send (const string &dataset, const string &constraint, FILE *out, bool compressed=true, const string &cgi_ver="", time_t lmt=0)
- void **mark_all** (bool state)
- bool mark (const string &name, bool state)

  Mark the `send_p` *flag of the named variable to* state.

- bool check_semantics (bool all=false)

  Check the semantics of each of the variables represented in the DDS.

Dataset Name Accessors

Get and set the dataset's name. This is the name of the dataset itself, and is not to be confused with the name of the file or disk on which it is stored.

- string get_dataset_name ()
- void set_dataset_name (const string &n)

File Name Accessor

Get and set the dataset's filename. This is the physical location on a disk where the dataset exists. The dataset name is simply a title.

See also:
Dataset Name Accessors

- string filename ()
- void filename (const string &fn)

External Function Accessors

Each DDS carries with it a list of external functions it can use to evaluate a constraint expression. If a constraint contains any of these functions, the entries in the list allow the parser to evaluate it. The functions are of two types: those that return boolean values, and those that return real (also called BaseType) values.

These methods are used to manipulate this list of known external functions.

- void add_function (const string &name, bool_func f)

  Add a boolean function to the list.

- void add_function (const string &name, btp_func f)

  Add a BaseType function to the list.

- void add_function (const string &name, proj_func f)
  Add a projection function to the list.
- bool find_function (const string &name, bool_func ∗f) const
  Find a Boolean function with a given name in the function list.
- bool find_function (const string &name, btp_func ∗f) const
  Find a BaseType function with a given name in the function list.
- bool find_function (const string &name, proj_func ∗f) const
  Find a projection function with a given name in the function list.

Constraint Expression

These member functions are used to access and manipulate the constraint expression that may be part of a DDS. Most of them are only used by the constraint expression parser.

Refer to The DODS User Manual *for a complete description of constraint expressions.*

See also:
Clause

- Pix first_clause ()
- void next_clause (Pix &p)
- Clause & clause (Pix p)
- bool clause_value (Pix p, const string &dataset)
- void append_clause (int op, rvalue ∗arg1, rvalue_list ∗arg2)
  Add a clause to a constraint expression.
- void append_clause (bool_func func, rvalue_list ∗args)
  Add a clause to a constraint expression.
- void append_clause (btp_func func, rvalue_list ∗args)
  Add a clause to a constraint expression.
- bool functional_expression ()
  Does the current constraint expression return a BaseType pointer?
- bool boolean_expression ()
  Does the current constraint expression return a boolean value?
- BaseType ∗ eval_function (const string &dataset)
  Evaluate a function-valued constraint expression.

- bool eval_selection (const string &dataset)

  Evaluate a boolean-valued constraint expression.

parse_constraint

Parse a constraint expression.

- void parse_constraint (const string &constraint, ostream &os=cout, bool server=true)

  Parse the constraint expression given the current DDS.

- void parse_constraint (const string &constraint, FILE *out, bool server=true)

  Parse the constraint expression given the current DDS.

Read Methods

Read a DDS from a file. This method calls a generated parser, `ddsparse()`, *to interpret an ASCII representation of a DDS, and regenerate that DDS in memory.*

See also:
Print Methods

- void parse (string fname)

  Parse a DDS from a file with the given name.

- void parse (int fd)

  Parse a DDS from a file indicated by the input file descriptor.

- void parse (FILE *in=stdin)

  Parse a DDS from a file indicated by the input file descriptor.

Print Methods

These methods create an ASCII representation of the DDS. This is the form in which the DDS is transmitted to the client process. A DDS can be output entire, or subject to the constraint of a constraint expression. In most cases, a constrained DDS will be smaller than the original, since the purpose of the expression is to discard some of the data in the dataset.

See also:
Read Methods

- void print (ostream &os=cout)

  Print the entire DDS on the specified output stream.

- void print (FILE ∗out)
  Print the entire DDS to the specified file.
- void print_constrained (ostream &os=cout)
  Print the constrained DDS to the specified output stream.
- void print_constrained (FILE ∗out)
  Print a constrained DDS to the specified file.

Protected Methods

- void **duplicate** (const DDS &dds)
- BaseType ∗ **leaf_match** (const string &name, btp_stack ∗s=0)
- BaseType ∗ **exact_match** (const string &name, btp_stack ∗s=0)

3.11.1 Detailed Description

The DODS Data Descriptor Object (DDS) is a data structure used by the DODS software to describe datasets and subsets of those datasets. The DDS may be thought of as the declarations for the data structures that will hold data requested by some DODS client. Part of the job of a DODS server is to build a suitable DDS for a specific dataset and to send it to the client. Depending on the data access API in use, this may involve reading part of the dataset and inferring the DDS. Other APIs may require the server simply to read some ancillary data file with the DDS in it. \label{api:dds}

On the server side, in addition to the data declarations, the DDS holds the clauses of any constraint expression that may have accompanied the data request from the DODS client. The DDS object includes methods for modifying the DDS according to the given constraint expression. It also has methods for directly modifying a DDS, and for transmitting it from a server to a client.

For the client, the DDS object includes methods for reading the persistent form of the object sent from a server. This includes parsing the ASCII representation of the object and, possibly, reading data received from a server into a data object.

Note that the class DDS is used to instantiate both DDS and DataDDS objects. A DDS that is empty (contains no actual data) is used by servers to send structural information to the client. The same DDS can becomes a DataDDS when data values are bound to the variables it defines.

For a complete description of the DDS layout and protocol, please refer to *The DODS User Guide*.

The DDS has an ASCII representation, which is what is transmitted from a DODS server to a client. Here is the DDS representation of an entire dataset containing a time series of worldwide grids of sea surface temperatures:

```
Dataset {
    Float64 lat[lat = 180];
    Float64 lon[lon = 360];
    Float64 time[time = 404];
    Grid {
     ARRAY:
        Int32 sst[time = 404][lat = 180][lon = 360];
     MAPS:
        Float64 time[time = 404];
        Float64 lat[lat = 180];
        Float64 lon[lon = 360];
    } sst;
} weekly;
```

If the data request to this dataset includes a constraint expression, the corresponding DDS might be different. For example, if the request was only for northern hemisphere data at a specific time, the above DDS might be modified to appear like this:

```
Dataset {
    Grid {
     ARRAY:
        Int32 sst[time = 1][lat = 90][lon = 360];
     MAPS:
        Float64 time[time = 1];
        Float64 lat[lat = 90];
        Float64 lon[lon = 360];
    } sst;
} weekly;
```

Since the constraint has narrowed the area of interest, the range of latitude values has been halved, and there is only one time value in the returned array. Note that the simple arrays (`lat`, `lon`, and `time`) described in the dataset are also part of the `sst` Grid object. They can be requested by themselves or as part of that larger object.

See the *The DODS User Guide*, or the documentation of the BaseType class for descriptions of the DODS data types.

See also:
BaseType , DAS

Definition at line 133 of file DDS.h.

3.11.2 Constructor & Destructor Documentation

3.11.2.1 DDS::DDS (const string & *n* = ””)

Creates a DDS with the given string for its name.

Definition at line 86 of file DDS.cc.

References add_function.

3.11.2.2 DDS::DDS (const DDS & *rhs*)

The DDS copy constructor.

Definition at line 96 of file DDS.cc.

3.11.3 Member Function Documentation

3.11.3.1 void DDS::append_clause (btp_func *func*, rvalue_list ∗ *args*)

Add a clause to a constraint expression.

This function adds a real-valued (BaseType) function clause to the constraint expression.

Parameters:

func A pointer to a BaseType function from the list of supported functions.

args A list of arguments to that function.

Definition at line 441 of file DDS.cc.

References clause.

3.11.3.2 void DDS::append_clause (bool_func *func*, rvalue_list ∗ *args*)

Add a clause to a constraint expression.

This function adds a boolean function clause to the constraint expression.

Parameters:

func A pointer to a boolean function from the list of supported functions.

args A list of arguments to that function.

Definition at line 424 of file DDS.cc.

References clause.

3.11.3.3 void DDS::append_clause (int *op*, rvalue ∗ *arg1*, rvalue_list ∗ *arg2*)

Add a clause to a constraint expression.

This function adds an operator clause to the constraint expression.

Parameters:

op An integer indicating the operator in use. These values are generated by {\tt bison}.

arg1 A pointer to the argument on the left side of the operator.

arg2 A pointer to a list of the arguments on the right side of the operator.

Definition at line 407 of file DDS.cc.

References clause.

3.11.3.4 void DDS::append_constant (BaseType ∗ *btp*)

The DDS maintains a list of BaseType pointers for all the constants that the constraint expression parser generates. These objects are deleted by the DDS destructor. Note that there are no list accessors; these constants are never accessed from the list. The list is simply a convenient way to make sure the constants are disposed of properly.

the constraint expression parser.

Definition at line 456 of file DDS.cc.

3.11.3.5 bool DDS::check_semantics (bool *all* = false)

Check the semantics of each of the variables represented in the DDS.

Check the semantics of the DDS describing a complete dataset. If ALL is true, check not only the semantics of THIS->TABLE, but also recurrsively all ctor types in the THIS->TABLE. By default, ALL is false since parsing a DDS input file runns semantic checks on all variables (but not the dataset itself.

Returns:

TRUE if the conventions for the DDS are not violated, FALSE otherwise.

Parameters:

all If true, recursively check the individual members of compound variables.

See also:

BaseType::check_semantics

Definition at line 818 of file DDS.cc.

3.11.3.6 Clause & DDS::clause (Pix *p*)

Returns a clause of a parsed constraint expression.

Definition at line 369 of file DDS.cc.

Referenced by append_clause, boolean_expression, eval_function, eval_selection, and functional_expression.

3.11.3.7 bool DDS::clause_value (Pix *p*, const string & *dataset*)

Returns the value of the indicated clause of a constraint expression.

Definition at line 384 of file DDS.cc.

3.11.3.8 void DDS::filename (const string & *fn*)

Set the dataset's filename.

Definition at line 174 of file DDS.cc.

3.11.3.9 string DDS::filename ()

Gets the dataset file name.

Definition at line 167 of file DDS.cc.

Referenced by List::member.

3.11.3.10 Pix DDS::first_clause ()

Returns a pointer to the first clause in a parsed constraint expression.

Definition at line 343 of file DDS.cc.

Referenced by boolean_expression, eval_function, eval_selection, and functional_expression.

3.11.3.11 string DDS::get_dataset_name ()

Returns the dataset's name.

Definition at line 143 of file DDS.cc.

3.11.3.12 bool DDS::mark (const string & *n*, bool *state*)

Mark the `send_p` flag of the named variable to *state*.

Mark the named variable by setting its SEND_P flag to STATE (true indicates that it is to be sent). Names must be fully qualified.

Note:
For aggregate types this sets each part to STATE when STATE is True. Thus, if State is True and N is 'exp1.test', then both 'exp1' and 'test' have their SEND_P flag set to True. If STATE is False, then the SEND_P flag of the 'test' is set to False, but 'exp1' is left unchanged. This means that a single variable can be removed from the current projection without removing all the other children of its parent. See the mfunc set_send_p().

Returns:
True if the named variable was found, false otherwise.

Definition at line 1024 of file DDS.cc.

References BaseType::name, BaseType::set_send_p, and var.

3.11.3.13 void DDS::next_clause (Pix & *p*)

Increments a pointer to indicate the next clause in a parsed constraint expression.

Definition at line 355 of file DDS.cc.

Referenced by boolean_expression, and eval_selection.

3.11.3.14 void DDS::parse (FILE ∗ *in* = stdin)

Parse a DDS from a file indicated by the input file descriptor.

Read structure from IN (which defaults to stdin). If ddsrestart() fails, return false, otherwise return the status of ddsparse().

Definition at line 663 of file DDS.cc.

3.11.3.15 void DDS::parse_constraint (const string & *constraint*, FILE ∗ *out*, bool *server* = true)

Parse the constraint expression given the current DDS.

Evaluate the constraint expression; return the value of the expression. As a side effect, mark the DDS so that BaseType's mfuncs can be used to correctly read the variable's value and send it to the client.

Returns:
void

Parameters:
constraint A string containing the constraint expression.
out A FILE pointer to which error objects should be wrtten.
server If true, send errors back to client instead of displaying errors on the default output stream.

Definition at line 881 of file DDS.cc.

References parse_constraint.

3.11.3.16 void DDS::parse_constraint (const string & *constraint*, ostream & *os* = cout, bool *server* = true)

Parse the constraint expression given the current DDS.

Evaluate the constraint expression; return the value of the expression. As a side effect, mark the DDS so that BaseType's mfuncs can be used to correctly read the variable's value and send it to the client.

Returns:
void

Parameters:
constraint A string containing the constraint expression.
os The output stream on which to write error objects and messages.
server If true, send errors back to client instead of displaying errors on the default output stream.

Definition at line 853 of file DDS.cc.

Referenced by parse_constraint, send, and DODSFilter::send_dds.

3.11.3.17 void DDS::print (ostream & *os* = cout)

Print the entire DDS on the specified output stream.

Write structure from tables to OUT (which defaults to stdout).

Returns:
TRUE

Definition at line 692 of file DDS.cc.

Referenced by print, and DODSFilter::send_dds.

3.11.3.18 void DDS::print_constrained (FILE * *out*)

Print a constrained DDS to the specified file.

Print those parts (variables) of the DDS structure to OS that are marked to be sent after evaluating the constraint expression.

Note:
This function only works for scalars at the top level.

Returns:
true.

Definition at line 752 of file DDS.cc.

References print_constrained.

3.11.3.19 void DDS::print_constrained (ostream & *os* = cout)

Print the constrained DDS to the specified output stream.

Print those parts (variables) of the DDS structure to OS that are marked to be sent after evaluating the constraint expression.

Note:
This function only works for scalars at the top level.

Returns:
true.

Definition at line 729 of file DDS.cc.

Referenced by print_constrained, send, and DODSFilter::send_dds.

3.11.3.20 bool DDS::send (const string & *dataset*, const string & *constraint*, FILE * *out*, bool *compressed* = true, const string & *cgi_ver* = "", time_t *lmt* = 0)

This function sends the variables described in the constrained DDS to the output described by the FILE pointer. This function calls `parse_constraint()`, `BaseType::read()`, and `BaseType::serialize()`.

Returns:
True if successful, false otherwise.

Parameters:

dataset The name of the dataset to send.

constraint A string containing the entire constraint expression received in the original data request.

out A pointer to the output buffer for the data.

compressed If true, send compressed data.

cgi_ver version information from the caller that identifies the server sending data.

lmt A `time_t` value to use in making a "Last Modifed" header for the sent data.

See also:

parse_constraint , BaseType::read , BaseType::serialize

Definition at line 950 of file DDS.cc.

References eval_function, first_var, functional_expression, next_var, parse_constraint, print_constrained, BaseType::send_p, BaseType::serialize, and var.

Referenced by DODSFilter::send_data.

3.11.3.21 void DDS::set_dataset_name (const string & *n*)

Sets the dataset name.

Definition at line 150 of file DDS.cc.

3.11.3.22 BaseType ∗ DDS::var (Pix *p*)

Returns a pointer to the indicated variable.

Definition at line 325 of file DDS.cc.

3.11.3.23 BaseType ∗ DDS::var (const char ∗ *n*, btp_stack ∗ *s* = 0)

Returns a pointer to the named variable.

This is necessary because (char ∗) can be cast to Pix (because Pix is really (void ∗)). This must take precedence over the creation of a temporary object (the string).

Returns:

A pointer to the variable or null if not found.

Definition at line 302 of file DDS.cc.

References var.

3.11.3.24 BaseType ∗ DDS::var (const string & *n*, btp_stack ∗ *s* = 0)

Find the variable with the given name.

Returns a pointer to the named variable. If the name contains one or more field separators then the function looks for a variable whose name matches exactly. If the name contains no field separators then the funciton looks first in the top level and then in all subsequent levels and returns the first occurrence found. In general, this function searches constructor types in the order in which they appear in the DDS, but there is no requirement that it do so.

\note{If a dataset contains two constructor types which have field names that are the same (say point.x and pair.x) you should always use fully qualified names to get each of those variables.}

Returns:
A BaseType pointer to the variable or null if not found.

Definition at line 235 of file DDS.cc.

3.11.3.25 BaseType ∗ DDS::var (const string & *n*, btp_stack & *s*)

Search for for variable *n* as above but record all compound type variables which ultimately contain *n* on *s*. This stack can then be used to mark the contained compound-type variables as part of the current projection.

Returns:
A BaseType pointer to the variable *n* or 0 if *n* could not be found.

Definition at line 214 of file DDS.cc.

Referenced by mark, send, and var.

The documentation for this class was generated from the following files:

- DDS.h
- DDS.cc

3.12 DODSFilter Class Reference

Common functions for DODS server filter programs.

`#include <DODSFilter.h>`

Public Methods

- DODSFilter (int argc, char *argv[ ])

  DODSFilter constructor.

- virtual ~**DODSFilter** ()
- bool OK ()

  Check whether the DODSFilter was initialized with valid arguments.

- bool version ()

  Should the filter send version information to the client program?

- virtual bool is_conditional ()

  Is this request conditional?

- string get_cgi_version ()
- void set_cgi_version (string version)
- string get_ce ()

  Get the constraint expression.

- virtual void **set_ce** (string _ce)
- string get_dataset_name ()

  Get the dataset name.

- virtual void **set_dataset_name** (const string _dataset)
- virtual string get_dataset_version ()

  Get the version information for the dataset.

- virtual time_t get_dataset_last_modified_time ()
- virtual time_t get_das_last_modified_time (const string &anc_location="")
- virtual time_t get_dds_last_modified_time (const string &anc_location="")
- virtual time_t get_data_last_modified_time (const string &anc_location="")
- virtual time_t get_request_if_modified_since ()

- virtual string get_cache_dir ()

  Get the cache directory.

- string get_accept_types ()
- virtual void read_ancillary_das (DAS &das, string anc_location="")

  Test if ancillary data must be read.

- virtual void read_ancillary_dds (DDS &dds, string anc_location="")

  Test if ancillary data must be read.

- void print_usage ()

  Print usage information for a filter program.

- void send_version_info ()

  Send version information back to the client program.

- virtual void **send_das** (DAS &das, const string &anc_location="")
- virtual void send_das (ostream &os, DAS &das, const string &anc_location="")

  Transmit a DAS.

- virtual void **send_dds** (DDS &dds, bool constrained=false, const string &anc_location="")
- virtual void send_dds (ostream &os, DDS &dds, bool constrained=false, const string &anc_location="")

  Transmit a DDS.

- virtual void send_data (DDS &dds, FILE ∗data_stream, const string &anc_location="")

  Transmit data.

3.12.1 Detailed Description

Common functions for DODS server filter programs.

When a DODS server receives a request from a DODS client, the server CGI script dispatches the request to one of several "filter" programs. Each filter is responsible for returning a different aspect of the dataset information: one is for data, one is for the dataset DDS, one is for the dataset DAS, and a fourth is for a usage message describing the server itself. Some installations may have additional optional filters.

The filter program receives a data request from the dispatch script. It receives its operating parameters from the command line, like any UNIX command, and it returns its output to standard output, which the httpd server packages up into a reply to the client.

This class contains some common functions for the filter programs used to make up the DODS data servers. The filter programs do not *have* to be called by a CGI program, but that is the normal mechanism by which they are invoked.

Author:
jhrg 8/26/97

Definition at line 55 of file DODSFilter.h.

3.12.2 Constructor & Destructor Documentation

3.12.2.1 DODSFilter::DODSFilter (int *argc*, char * *argv*[ ])

DODSFilter constructor.

Create an instance of DODSFilter using the command line arguments passed by the CGI (or other) program. The default constructor is private; this and the copy constructor (which is just the default copy constructor) are the only way to create an instance of DODSFilter.

These are the valid options:

filename The name of the file on which the filter is to operate. Usually this would be the file whose data has been requested.

`-c` Send compressed data. Data are compressed using the deflate program. The W3C's libwww will recognize this and automatically decompress these data.

`-e` ***expression*** This option specifies a non-blank constraint expression used to subsample a dataset.

`-v` ***cgi-version*** Set the CGI/Server version to `cgi-version`. This is a way for the caller to set version information passed back to the client either as the response to a version request of in the response headers.

`-V` Specifies that this request is just for version information. Servers can check to see if this was given using the `version` mfunc. Note that version information is sent from within DODSFilter so that sophisticated servers can support versioning data sources inaddition to the server software.

`-d` ***ancdir*** Specifies that ancillary data be sought in the *ancdir* directory.

`-f` ***ancfile*** Specifies that ancillary data may be found in a file called *ancfile*.

`-r` ***cache directory*** Specify a directory to use if/when files are to be cached. Not all handlers support caching and each uses its own rules tailored to a specific file or data type.

`-t` ***list of types*** Specifies a list of types accepted by the client. This information is passed to a server by a client using the XDODS-Accept-Types header. The comma separated list contains each type the client can understand *or*, each type the client does *not* understand. In the latter case the type names are prefixed by a {\tt !}. If the list contains only the keyword 'All', then the client is declaring that it can understand all DODS types.

`-l` ***time*** Indicates that the request is a conditional request; send a complete response if and only if the data has changed since *time*. If it has not changed since *time*, then send a 304 (Not Modified) response. The *time* parameter is the `Last-Modified` time from an If-Modified-Since condition GET request. It is given in seconds since the start of the Unix epoch (Midnight, 1 Jan 1970).

Definition at line 109 of file DODSFilter.cc.

3.12.3 Member Function Documentation

3.12.3.1 string DODSFilter::get_accept_types ()

Get the list of accepted datatypes sent by the client. If no list was sent, return the string 'All'.

NB: The funny spelling 'accept types' instead of 'accepted types' mirrors the name of the HTTP request header field name which in turn mirrors the common practice of using 'accept' over 'accepted'.

See also:
DODSFilter

Returns:
A string containing a list of the accepted types.

Definition at line 409 of file DODSFilter.cc.

3.12.3.2 string DODSFilter::get_cache_dir () `[virtual]`

Get the cache directory.

The `cache_dir` is used to hold the cached .dds and .das files. By default, this returns an empty string (store cache files in current directory.

Returns:
A string object that contains the cache file directory.

Definition at line 394 of file DODSFilter.cc.

3.12.3.3 string DODSFilter::get_ce ()

Get the constraint expression.

Return the entire constraint expression in a string. This includes both the projection and selection clauses, but not the question mark.

Returns:
A string object that contains the constraint expression.

Definition at line 244 of file DODSFilter.cc.

3.12.3.4 string DODSFilter::get_cgi_version ()

Return the version information passed to the instance when it was created. This string is passed to the DODSFilter ctor using the -v option.

Returns:
The version string supplied at initialization.

Definition at line 232 of file DODSFilter.cc.

3.12.3.5 time_t DODSFilter::get_das_last_modified_time (const string & *anc_location* = "") `[virtual]`

Get the last modified time for the dataset's DAS. This time, given in seconds since the epoch (1 Jan 1970 00:00:00 GMT), is the greater of the datasets's and any ancillary DAS' last modified time.

Parameters:
anc_location A directory to search for ancillary files (in addition to the CWD).

Returns:
Time of last modification of the DAS.

See also:
get_dataset_last_modified_time() , get_dds_last_modified_time()

Definition at line 324 of file DODSFilter.cc.

References get_dataset_last_modified_time.

Referenced by send_das.

3.12.3.6 time_t DODSFilter::get_data_last_modified_time (const string & *anc_location* = "") [virtual]

Get the last modified time to be used for a particular data request. This method should look at both the contraint expression and any ancillary files for this dataset. The implementation provided here returns the latest time returned by the `get_dataset`...(), `get_das`...() and `get_dds`...() methods and does not currently check the CE.

Parameters:

anc_location A directory to search for ancillary files (in addition to the CWD).

Returns:

Time of last modification of the data.

See also:

get_dataset_last_modified_time() , get_das_last_modified_time() , get_dds_last_modified_time()

Definition at line 360 of file DODSFilter.cc.

References get_dataset_last_modified_time.

Referenced by send_data.

3.12.3.7 time_t DODSFilter::get_dataset_last_modified_time () [virtual]

Get the dataset's last modified time. This returns the time at which the dataset was last modified as defined by UNIX's notion of modification. This does not take into account the modification of an ancillary DAS or DDS. Time is given in seconds since the epoch (1 Jan 1970 00:00:00 GMT).

This method perform a simple check on the file named by the dataset given when the DODSFilter instance was created. If the dataset is not a filter, this method returns the current time. Servers which provide access to non-file-based data should subclass DODSFilter and supply a more suitable version of this method.

From the stat(2) man page: "Traditionally, `st_mtime` is changed by mknod(2), utime(2), and write(2). The `st_mtime` is not changed for changes in owner, group, hard link count, or mode."

Returns:

Time of the last modification in seconds since the epoch.

See also:

get_das_last_modified_time() , get_dds_last_modified_time()

Definition at line 309 of file DODSFilter.cc.

Referenced by get_das_last_modified_time, get_data_last_modified_time, and get_dds_-last_modified_time.

3.12.3.8 string DODSFilter::get_dataset_name ()

Get the dataset name.

The “dataset name” is the filename or other string that the filter program will use to access the data. In some cases this will indicate a disk file containing the data. In others, it may represent a database query or some other exotic data access method.

Returns:
A string object that contains the name of the dataset.

Definition at line 264 of file DODSFilter.cc.

3.12.3.9 string DODSFilter::get_dataset_version () `[virtual]`

Get the version information for the dataset.

To read version information that is specific to a certain dataset, override this method with an implementation that does what you want. By default, this returns an empty string.

Returns:
A string object that contains the dataset version information.

Definition at line 283 of file DODSFilter.cc.

Referenced by send_version_info.

3.12.3.10 time_t DODSFilter::get_dds_last_modified_time (const string & *anc_location* = "") `[virtual]`

Get the last modified time for the dataset's DDS. This time, given in seconds since the epoch (1 Jan 1970 00:00:00 GMT), is the greater of the datasets's and any ancillary DDS' last modified time.

Returns:
Time of last modification of the DDS.

See also:
get_dataset_last_modified_time() , get_dds_last_modified_time()

Definition at line 339 of file DODSFilter.cc.

References get_dataset_last_modified_time.

Referenced by send_dds.

3.12.3.11 time_t DODSFilter::get_request_if_modified_since () [virtual]

Get the value of a conditional request's If-Modified-Since header. This value is used to determine if the request should get a full response or a Not Modified (304) response. The time is given in seconds since the Unix epoch (midnight, 1 Jan 1970). If no time was given with the request, this methods returns -1.

Returns:
If-Modified-Since time from a condition GET request.

Definition at line 382 of file DODSFilter.cc.

Referenced by send_das, send_data, and send_dds.

3.12.3.12 bool DODSFilter::is_conditional () [virtual]

Is this request conditional?

Returns:
True if the request is conditional.

See also:
get_request_if_modified_since().

Definition at line 202 of file DODSFilter.cc.

Referenced by send_das, send_data, and send_dds.

3.12.3.13 bool DODSFilter::OK ()

Check whether the DODSFilter was initialized with valid arguments.

Use this function to test whether the options passed via argc and argv are valid.

Returns:
True if the class state is OK, false otherwise.

Definition at line 173 of file DODSFilter.cc.

3.12.3.14 void DODSFilter::print_usage ()

Print usage information for a filter program.

This message is printed when the filter program is incorrectly invoked by the dispatch CGI. This is an error in the server installation or the CGI implementation, so the error message is written to stderr instead of stdout. A server's stderr messages show up in the httpd log file. In addition, an error object is sent back to the client program telling them that the server is broken.

Definition at line 469 of file DODSFilter.cc.

3.12.3.15 void DODSFilter::read_ancillary_das (DAS & *das*, string *anc_location* = "") `[virtual]`

Test if ancillary data must be read.

Read the ancillary DAS information and merge it into the input DAS object.

Parameters:

das A DAS object that will be augmented with the ancillary data attributes.

anc_location The directory in which the external DAS file resides.

Returns:

void

See also:

DAS

Definition at line 424 of file DODSFilter.cc.

References DAS::parse.

3.12.3.16 void DODSFilter::read_ancillary_dds (DDS & *dds*, string *anc_location* = "") `[virtual]`

Test if ancillary data must be read.

Read the ancillary DDS information and merge it into the input DDS object.

Parameters:

dds A DDS object that will be augmented with the ancillary data properties.

anc_location The directory in which the external DAS file resides.

Returns:

void

See also:
DDS

Definition at line 445 of file DODSFilter.cc.

References DDS::parse.

3.12.3.17 void DODSFilter::send_das (ostream & *os*, DAS & *das*, const string & *anc_location* = "") [virtual]

Transmit a DAS.

This function formats and prints an ASCII representation of a DAS on stdout. This has the effect of sending the DAS object back to the client program.

Parameters:
- ***os*** The output stream to which the DAS is to be sent.
- ***das*** The DAS object to be sent.
- ***anc_location*** The directory in which the external DAS file resides.

Returns:
void

See also:
DAS

Definition at line 517 of file DODSFilter.cc.

References get_das_last_modified_time, get_request_if_modified_since, is_conditional, and DAS::print.

3.12.3.18 void DODSFilter::send_data (DDS & *dds*, FILE * *data_stream*, const string & *anc_location* = "") [virtual]

Transmit data.

Send the data in the DDS object back to the client program. The data is encoded in XDR format, and enclosed in a MIME document which is all sent to stdout. This has the effect of sending it back to the client.

Parameters:
- ***dds*** A DDS object containing the data to be sent.
- ***data_stream*** A pointer to the XDR sink into which the data is to be put for encoding and transmission.
- ***anc_location*** A directory to search for ancillary files (in addition to the CWD). This is used in a call to get_data_last_modified_time().

Returns:
void

See also:
DDS

Definition at line 593 of file DODSFilter.cc.

References get_data_last_modified_time, get_request_if_modified_since, is_conditional, and DDS::send.

3.12.3.19 void DODSFilter::send_dds (ostream & *os*, DDS & *dds*, bool *constrained* = false, const string & *anc_location* = "") `[virtual]`

Transmit a DDS.

This function formats and prints an ASCII representation of a DDS on stdout. When called by a CGI program, this has the effect of sending a DDS object back to the client program. Either an entire DDS or a constrained DDS may be sent.

Parameters:
os The output stream to which the DAS is to be sent.

dds The DDS to send back to a client.

constrained If this argument is true, evaluate the current constraint expression and send the 'constrained DDS' back to the client.

anc_location The directory in which the external DAS file resides.

Returns:
void

See also:
DDS

Definition at line 551 of file DODSFilter.cc.

References get_dds_last_modified_time, get_request_if_modified_since, is_conditional, DDS::parse_constraint, DDS::print, and DDS::print_constrained.

3.12.3.20 void DODSFilter::send_version_info ()

Send version information back to the client program.

This function formats and sends to stdout version information from the httpd server, the server dispatch scripts, the DODS core software, and (optionally) the dataset.

Definition at line 487 of file DODSFilter.cc.

References get_dataset_version.

3.12.3.21 void DODSFilter::set_cgi_version (string *version*)

Set the CGI/Server version number. Servers use this when answering requests for version information. The vesion 'number' should include both the name of the server (e.g., `ff_dods`) as well as the version number. Since this information is typically divined by configure, it's up to the executable to poke the correct value in using this method.

Note that the -v switch that this class understands is deprecated since it is usually called by Perl code. It makes more sense to have the actual C++ software set the version string.

Parameters:

version A version string for this server.

Definition at line 221 of file DODSFilter.cc.

References version.

3.12.3.22 bool DODSFilter::version ()

Should the filter send version information to the client program?

Use this function to check whether the client requested version information. In addition to returning version information about the DODS software, the server can also provide version information about the dataset itself.

Returns:

TRUE if the -v option was given indicating that the filter should send version information back to the client, FALSE otherwise.

See also:

DODSFilter::send_version_info

Definition at line 192 of file DODSFilter.cc.

Referenced by set_cgi_version.

The documentation for this class was generated from the following files:

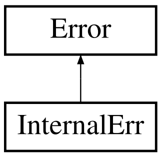
- DODSFilter.h
- DODSFilter.cc

3.13 Error Class Reference

A class for error processing.

```
#include <Error.h>
```

Inheritance diagram for Error::



Public Methods

- Error (const Error &copy_from)

  Error object copy constructor.

- virtual ~**Error** ()
- Error & operator= (const Error &rhs)
- bool OK () const

  Is the Error object valid?

- bool parse (FILE ∗fp)

  Parse an Error object.

- void print (ostream &os=cout) const

  Print the Error object on the given output stream.

- ErrorCode get_error_code () const

  Get the error code.

- string get_error_message () const

  Get the error message.

- ProgramType get_program_type () const

  get the program type.

- const char ∗ get_program () const

Get the error handling program.

- void set_error_code (ErrorCode ec=undefined_error)

  Set the error code.

- void set_error_message (string msg="")

  Set the error message.

- void set_program_type (ProgramType pt=undefined_prog_type)

  Set the program type.

- void set_program (char ∗program)

  Set the error handling program.

- void display_message (void ∗gui=0) const

  Display the error message in a dialog box or on stderr.

- string correct_error (void ∗gui) const

  Run the error correction program or print the error message.

- Error (ErrorCode ec, string msg)

  Create an Error object with an error code and a message.

- Error (string msg)

  Create an Error object with only a message.

- Error (ErrorCode ec, string msg, ProgramType pt, char ∗pgm)

  Create a descriptive Error object.

- Error ()

  Create a default Error object.

Deprecated methods

These methods should not be used because combining the accessors and mutators makes using const objects almost impossible for clients of this class.

- string error_message (string msg="")

  Get or set the error code.

- ProgramType program_type (ProgramType pt=undefined_prog_type)

  Get or set the program type.

- char * program (char *program=0)
  Get or set the error correction program.
- ErrorCode error_code (ErrorCode ec=undefined_error)
  Get or set the error code.

Protected Attributes

- ErrorCode **_error_code**
- string **_error_message**
- ProgramType **_program_type**
- char * **_program**

3.13.1 Detailed Description

A class for error processing.

The Error class is used to transport error information from the server to the client within DODS. It can also be used on the client side only. Errors consist of an error code, string and optionally a function/program. The code can be used to quickly distinguish between certain common errors while the string is used to convey information about the error to the user. The error code should never be displayed to the user. The program or function can be used for error correction controlled by the user.

This class is used on both clients and servers. The `print()` and `parse()` methods are used to send the object back and forth.

Author:
jhrg

Definition at line 106 of file Error.h.

3.13.2 Constructor & Destructor Documentation

3.13.2.1 Error::Error (ErrorCode *ec*, string *msg*)

Create an Error object with an error code and a message.

It is not possible to create an Error object with only an error code; you must supply at a minimum a code and a message. In this case the correction program and program type will be null. In addition, if a program type is given a program *must* also be given.

Other class constructors should be the only callers of this object's default constructor.

Definition at line 48 of file Error.cc.

3.13.2.2 Error::Error (string *msg*)

Create an Error object with only a message.

I've modified Error so that only a message string needs to be given. In this case the constructor will use the message code `unknown_error`. It is a bit misleading, but those codes are turning out to be more of a problem than anything else since they don't seem very useful but are a pain to supply. (10/11/2000 jhrg)

Definition at line 65 of file Error.cc.

3.13.3 Member Function Documentation

3.13.3.1 string Error::correct_error (void ∗ *pgui*) const

Run the error correction program or print the error message.

This function runs the error correction program, if possible, and returns a string that can be used as the 'corrected' value. If there is no error correction program or it is not possible to run the program, the function simply displays the error message. If the error correction program cannot be run, the function returns the null string.

If program() is true, then source the code it returns in the Gui object and return the value of Gui::command(). Note that this means that the code in the program must run itself (i.e., in addition to any procedure definitions, etc. it must also contain the necessary instructions to popup an initial window).

Returns:
A corrected string or "".

Parameters:
pgui A pointer to a Gui class object handling a GUI process on the client. f this function is called with a null Gui pointer then the message text is displayed on stderr.

See also:
Gui , display_message

Definition at line 459 of file Error.cc.

References display_message, and OK.

3.13.3.2 void Error::display_message (void * *pgui* = 0) const

Display the error message in a dialog box or on stderr.

Either display the error message in a dialog box and offer the user a single 'OK' button or print the message to standard error. If `gui` is not given, then use stderr. In addition, the class Gui provides other means for the user to control how messages are displayed and those may be used to select either graphical or text devices.

Note that the void `* gui` gets cast to a pointer to Gui when Error.cc is compiled with the preprocessor symbol `GUI` defined. When that symbol is not defined, the method ignores the param `gui`. I've hidden the type (GUI *) because when `GUI` is not defined the dap++ library is built without the Gui class.

Parameters:

pgui A pointer to a valid Gui class instance. This would be attached to a GUI process running on a client machine, and that process will display the message. If the pointer is not provided, the message will be displayed on the client's stderr.

See also:

Gui , correct_error

Definition at line 344 of file Error.cc.

References OK, Gui::show_gui, and Gui::simple_error.

Referenced by correct_error.

3.13.3.3 ErrorCode Error::error_code (ErrorCode *ec* = undefined_error)

Get or set the error code.

With no argument, returns the Error object's error code. With an argument, sets the error code to that value.

Deprecated:

Returns:

The Error object's error code.

Parameters:

ec The error code. If this is not included, the undefined error code will be stored.

Definition at line 245 of file Error.cc.

References OK.

3.13.3.4 string Error::error_message (string *msg* = "")

Get or set the error code.

With no argument, return a copy of the objet's error message string. With an argument, set the object's error message to that string.

Deprecated:

Parameters:

msg The error message string. If this is omitted, the function simply returns a copy of the current message string.

Returns:

A copy of the Error object's message string.

Definition at line 291 of file Error.cc.

References OK.

3.13.3.5 bool Error::OK () const

Is the Error object valid?

Use this function to determine whether an Error object is valid. To be a valid, an Error object must either be: 1) empty, 2) contain a message and a program or 3) contain only a message.

NB: This mfunc does not test for malformed messages or programs - ones where the code is defined but not the 'data'.

Returns:

TRUE if the object is valid, FALSE otherwise.

Definition at line 134 of file Error.cc.

References program.

Referenced by correct_error, display_message, error_code, error_message, get_error_code, get_error_message, get_program_type, InternalErr::OK, operator=, parse, print, program_type, set_error_code, and set_error_message.

3.13.3.6 Error & Error::operator= (const Error & *rhs*)

The assignment operator copies the error correction.

Definition at line 100 of file Error.cc.

References _error_code, _error_message, _program, _program_type, and OK.

3.13.3.7 bool Error::parse (FILE * *fp*)

Parse an Error object.

Given an input stream (FILE *) `fp`, parse an Error object from stream. Values for fields of the Error object are parsed and THIS is set accordingly. This is how a DODS client might receive an error object from a server.

Parameters:

fp A valid file pointer to an input stream.

Returns:

TRUE if no error was detected, FALSE otherwise.

Definition at line 166 of file Error.cc.

References OK.

3.13.3.8 void Error::print (ostream & *os* = cout) const

Print the Error object on the given output stream.

Creates a printable representation of the Error object. It is suitable for framing, and also for printing and sending over a network.

The printed representation produced by this function can be parsed by the parse() memeber function. Thus parse and print form a symetrical pair that can be used to send and receive an Error object over the network in a MIME document.

Parameters:

os A pointer to the output stream on which the Error object is to be rendered.

Definition at line 210 of file Error.cc.

References OK.

3.13.3.9 char * Error::program (char * *pgm* = 0)

Get or set the error correction program.

With no argument, return the error correction program. With an argument, set the error correction program to a copy of that value.

Note that this is not a pointer to a function, but a character string containing the entire tcl, Java, or other program.

Deprecated:

Returns:
the error correction program.

Definition at line 411 of file Error.cc.

Referenced by OK.

3.13.3.10 ProgramType Error::program_type (ProgramType *pt* = undefined_prog_type)

Get or set the program type.

With no argument, return the program type of the error object. With an argument, set the error object's program type field.

Deprecated:

Returns:
The program type of the object.

See also:
ProgramType

Definition at line 369 of file Error.cc.

References OK.

3.13.3.11 void Error::set_error_code (ErrorCode *ec* = undefined_error)

Set the error code.

If the input error code is an empty string, the error code is set to "undefined error."

Definition at line 273 of file Error.cc.

References OK.

3.13.3.12 void Error::set_program_type (ProgramType *pt* = undefined_prog_type)

Set the program type.

The default program type t is undefined_prog_type.

Definition at line 395 of file Error.cc.

The documentation for this class was generated from the following files:

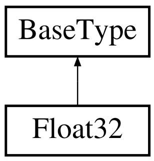
- Error.h
- Error.cc

3.14 Float32 Class Reference

Holds a 32-bit floating point value.

```
#include <Float32.h>
```

Inheritance diagram for Float32::



Public Methods

- Float32 (const string &n="")
- **Float32** (const Float32 &copy_from)
- Float32 & **operator=** (const Float32 &rhs)
- virtual ~**Float32** ()
- virtual BaseType * **ptr_duplicate** ()=0
- virtual unsigned int width ()

  Returns the size of the class instance data.

- virtual bool serialize (const string &dataset, DDS &dds, XDR *sink, bool ce_eval=true)

  Move data to the net.

- virtual bool deserialize (XDR *source, DDS *dds, bool reuse=false)

  Receive data from the net.

- virtual bool read (const string &dataset)=0

  Reads data into a local buffer.

- virtual unsigned int val2buf (void *val, bool reuse=false)

  Loads class data.

- virtual unsigned int buf2val (void **val)

  Reads the class data.

- virtual void print_val (ostream &os, string space="", bool print_decl_p=true)
  Prints the value of the variable.

- virtual bool ops (BaseType *b, int op, const string &dataset)
  The class relational operators.

Protected Attributes

- dods_float32 **_buf**

Friends

- class Byte
- class Int16
- class UInt16
- class Int32
- class UInt32
- class Float64

3.14.1 Detailed Description

Holds a 32-bit floating point value.

See also:
BaseType

Definition at line 46 of file Float32.h.

3.14.2 Constructor & Destructor Documentation

3.14.2.1 Float32::Float32 (const string & *n* = "")

The Float32 constructor requires only the name of the variable to be created. The name may be omitted, which will create a nameless variable. This may be adequate for some applications.

Parameters:
n A string containing the name of the variable to be created.

Definition at line 47 of file Float32.cc.

3.14.3 Member Function Documentation

3.14.3.1 unsigned int Float32::buf2val (void ** *val*) [virtual]

Reads the class data.

Reads the class data into the memory referenced by *val*. The caller must allocate enough storage to *val* to hold the class data. If *val* is NULL, however, memory will be allocated by this function with `new()`. Even if the memory is allocated this way, the caller is responsible for deallocating that memory. Array and List values for simple types are stored as C would store an array.

Parameters:

val A pointer to a pointer to the memory into which the class data will be copied. If the value pointed to is NULL, memory will be allocated to hold the data, and the pointer value modified accordingly. The calling program is responsible for deallocating the memory indicated by this pointer.

Returns:

The size (in bytes) of the information copied to *val*.

Implements BaseType.

Definition at line 123 of file Float32.cc.

References width.

3.14.3.2 bool Float32::deserialize (XDR * *source*, DDS * *dds*, bool *reuse* = false) [virtual]

Receive data from the net.

Receives data from the network connection identified by the `source` parameter. The data is put into the class data buffer according to the input `dds`.

This function is only used on the client side of the DODS client/server connection.

Parameters:

source A valid XDR pointer to the process connection to the net. This is generally created with a call to `new_xdrstdio()`.

dds The Data Descriptor Structure object corresponding to this dataset. See *The DODS User Manual* for information about this structure. This would have been received from the server in an earlier transmission.

reuse A boolean value, indicating whether the class internal data storage can be reused or not. If this argument is TRUE, the class buffer is assumed to be large enough to hold the incoming data, and it is *not* reallocated. If FALSE, new storage is allocated. If the internal buffer has not been allocated at all, this argument has no effect.

Returns:
Always returns TRUE.

Exceptions:
Error when a problem reading from the XDR stream is found.

See also:
DDS

Implements BaseType.

Definition at line 96 of file Float32.cc.

3.14.3.3 bool Float32::ops (BaseType * *b*, int *op*, const string & *dataset*) [virtual]

The class relational operators.

This function contains the relational operators used by the constraint expression evaluator in the DDS class. Each class that wants to be able to evaluate relational expressions must overload this function. The implementation in BaseType returns false and prints an error message.

The *op* argument refers to a table generated by bison from the constraint expression parser. Use statements like the following to correctly interpret its value:

```
switch (op) {
case EQUAL:
return i1 == i2;
case NOT_EQUAL:
return i1 != i2;
case GREATER:
return i1 > i2;
case GREATER_EQL:
return i1 >= i2;
case LESS:
return i1 < i2;
case LESS_EQL:
return i1 <= i2;
case REGEXP:
cerr << "Regular expressions not valid for integer values" << endl;
return false;
default:
```

```
cerr << "Unknown operator" << endl;
return false;
}
```

This function is used by the constraint expression evaluator.

Parameters:

b The value with which the instance value is to be compared.

op An integer index indicating which relational operator is implied. Choose one from the following: `EQUAL`, `NOT_EQUAL`, `GREATER`, `GREATER_EQL`, `LESS`, `LESS_EQL`, and `REGEXP`.

dataset The name of the dataset from which the instance's data has come (or is to come).

Returns:

The boolean value of the comparison.

Reimplemented from BaseType.

Definition at line 152 of file Float32.cc.

References BaseType::read, read, BaseType::read_p, and BaseType::type.

3.14.3.4 void Float32::print_val (ostream & *os*, string *space* = "", bool *print_decl_p* = true) `[virtual]`

Prints the value of the variable.

Prints the value of the variable, with its declaration. This function is primarily intended for debugging DODS applications. However, it can be overloaded and used to do some useful things. Take a look at the asciival and writeval clients, both of which overload this to output the values of variables in different ways.

NB: This function uses C++'s iostream to handle creating the print representations of simple type variables. For floating point numbers this is set to six digits of precision by default. If you want more precision (IEEE 64-bit floats have 15 digits of precision, 32-bit floats have 8), use the setprecision() I/O manipulator.

Parameters:

os The output stream on which to print the value.

space This value is passed to the `print_decl()` function, and controls the leading spaces of the output.

print_decl_p A boolean value controlling whether the variable declaration is printed as well as the value.

Implements BaseType.

Definition at line 139 of file Float32.cc.

References BaseType::print_decl.

3.14.3.5 virtual bool Float32::read (const string & *dataset*) `[pure virtual]`

Reads data into a local buffer.

Put the data into a local buffer so that it may be sent to a client. This operation involves reading data from whatever source (often a local disk), and filling out the fields in the data type class. This is the heart of the DODS DAP Class operation. Much of the work of implementing a new DODS server API consists in creating the `read()` functions to read various data types.

Note that this function is only for DODS servers. It has no use on the client side of a DODS client/server connection. The DODS client and server communicate their data with `serialize()` and `deserialize()`.

This method is not implemented for the BaseType class, nor for its children. However, it should be implemented for the specialized children of those classes. For example, it is not implemented for the Float64 class, but does exist for the NCFloat64 class, specialized to read data from local netCDF files.

This method should be implemented to throw Error when it encounters an unrecoverable error.

For an example of use, see the netCDF library classes. The netCDF library is part of the DODS source distribution, and can be found under `/export/home/tomfool/DODS/src/nc-dods`.

In some sub-classes, such as Array or Grid, the `read()` function must explicitly take into account constraint information stored with the class data. For example, the Grid::read method will be called when only one component of the Grid is to be sent. Your implementation of Grid::read should check send_p() for each member of the Grid before reading that member to avoid reading data into memory that won't be sent (and thus is not needed in memory).

Returns:

The function returns a boolean value, with TRUE indicating that read() should be called again because there's more data to read, and FALSE indicating there's no more data to read. Note that this behavior is necessary to properly handle variables that contain Sequences. WRONG! (9/5/2001 jhrg) Sequences now are read in one shot. The return value of this method should always be false.

Parameters:

dataset A string naming the dataset from which the data is to be read. The meaning of this string will vary among data APIs.

See also:
BaseType

Implements BaseType.

Referenced by ops, and serialize.

3.14.3.6 bool Float32::serialize (const string & *dataset*, DDS & *dds*, XDR * *sink*, bool *ce_eval* = true) [virtual]

Move data to the net.

Sends the data from the indicated (local) dataset through the connection identified by the *sink* parameter. If the data is not already incorporated into the DDS object, read the data from the dataset.

This function is only used on the server side of the client/server connection, and is generally only called from the DDS::send() function. It has no BaseType implementation; each datatype child class supplies its own implementation.

Parameters:

dataset The (local) name of dataset to be read.

dds The Data Descriptor Structure object corresponding to this dataset. See *The DODS User Manual* for information about this structure.

sink A valid XDR pointer generally created with a call to `new_xdrstdio()`. This typically routes data to a TCP/IP socket.

ce_eval A boolean value indicating whether to evaluate the DODS constraint expression that may accompany this dataset. The constraint expression is stored in *dds*.

Returns:
This method always returns true. Older versions used the return value to signal success or failure.

Exceptions:

InternalErr .

Error .

See also:
DDS

Implements BaseType.

Definition at line 77 of file Float32.cc.

References DDS::eval_selection, read, and BaseType::read_p.

3.14.3.7 unsigned int Float32::val2buf (void * *val*, bool *reuse* = false) [virtual]

Loads class data.

Store the value pointed to by *val* in the object's internal buffer. This function does not perform any checks, so users must be sure that the thing pointed to can actually be stored in the object's buffer. For example, an array cannot easily be fit into the data buffer for an Int32 object.

Only simple objects (Int, Float, Byte, and so on) and arrays and lists of these simple objects may be stored using this function. To put data into more complex constructor functions, use the functions provided by that class. For example, use the `var()` and `add_var()` members of the Grid class to manipulate data in that class.

Parameters:

val A pointer to the data to be inserted into the class data buffer.

reuse A boolean value, indicating whether the class internal data storage can be reused or not. If this argument is TRUE, the class buffer is assumed to be large enough to hold the incoming data, and it is *not* reallocated. If FALSE, new storage is allocated. If the internal buffer has not been allocated at all, this argument has no effect. This is currently used only in the Vector class.

Returns:

The size (in bytes) of the information copied from *val*.

See also:

Grid , Vector::val2buf

Implements BaseType.

Definition at line 107 of file Float32.cc.

References width.

3.14.3.8 unsigned int Float32::width () [virtual]

Returns the size of the class instance data.

Return the number of bytes that are required to hold the instance's value. In the case of simple types such as Int32, this is the size of one Int32 (four bytes). For a String or Url type, `width()` returns the number of bytes needed for a `String *` variable, not the bytes needed for all the characters, since that value cannot be determined from type information alone. For Structure, and other constructor types size() returns the number of bytes needed to store pointers to the C++ objects.

Implements BaseType.

Definition at line 71 of file Float32.cc.

Referenced by buf2val, and val2buf.

3.14.4 Friends And Related Function Documentation

3.14.4.1 friend class Byte `[friend]`

This class allows Byte, ..., Float64 access to `_buf` to simplify and speed up the relational operators.

NB: According to Stroustrup it does not matter where (public, private or protected) friend classes are declared.

Definition at line 53 of file Float32.h.

The documentation for this class was generated from the following files:

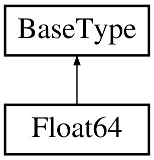
- Float32.h
- Float32.cc

3.15 Float64 Class Reference

Holds a 64-bit (double precision) floating point value.

`#include <Float64.h>`

Inheritance diagram for Float64::



Public Methods

- Float64 (const string &n="")
- virtual ~**Float64** ()
- **Float64** (const Float64 &copy_from)
- Float64 & **operator=** (const Float64 &rhs)
- virtual BaseType * **ptr_duplicate** ()=0
- virtual unsigned int width ()

    Returns the size of the class instance data.

- virtual bool serialize (const string &dataset, DDS &dds, XDR *sink, bool ce_-eval=true)

    Move data to the net.

- virtual bool deserialize (XDR *source, DDS *dds, bool reuse=false)

    Receive data from the net.

- virtual bool read (const string &dataset)=0

    Reads data into a local buffer.

- virtual unsigned int val2buf (void *val, bool reuse=false)

    Loads class data.

- virtual unsigned int buf2val (void **val)

    Reads the class data.

- virtual void print_val (ostream &os, string space="", bool print_decl_p=true)

  Prints the value of the variable.

- virtual bool ops (BaseType *b, int op, const string &dataset)

  The class relational operators.

Protected Attributes

- dods_float64 **_buf**

Friends

- class Byte
- class Int16
- class UInt16
- class Int32
- class UInt32
- class Float32

3.15.1 Detailed Description

Holds a 64-bit (double precision) floating point value.

See also:
  BaseType

Definition at line 46 of file Float64.h.

3.15.2 Constructor & Destructor Documentation

3.15.2.1 Float64::Float64 (const string & *n* = "")

The Float64 constructor requires only the name of the variable to be created. The name may be omitted, which will create a nameless variable. This may be adequate for some applications.

Parameters:
  n A string containing the name of the variable to be created.

Definition at line 49 of file Float64.cc.

3.15.3 Member Function Documentation

3.15.3.1 unsigned int Float64::buf2val (void ** *val*) [virtual]

Reads the class data.

Reads the class data into the memory referenced by *val*. The caller must allocate enough storage to *val* to hold the class data. If *val* is NULL, however, memory will be allocated by this function with `new()`. Even if the memory is allocated this way, the caller is responsible for deallocating that memory. Array and List values for simple types are stored as C would store an array.

Parameters:

val A pointer to a pointer to the memory into which the class data will be copied. If the value pointed to is NULL, memory will be allocated to hold the data, and the pointer value modified accordingly. The calling program is responsible for deallocating the memory indicated by this pointer.

Returns:

The size (in bytes) of the information copied to *val*.

Implements BaseType.

Definition at line 125 of file Float64.cc.

References width.

3.15.3.2 bool Float64::deserialize (XDR * *source*, DDS * *dds*, bool *reuse* = false) [virtual]

Receive data from the net.

Receives data from the network connection identified by the `source` parameter. The data is put into the class data buffer according to the input `dds`.

This function is only used on the client side of the DODS client/server connection.

Parameters:

source A valid XDR pointer to the process connection to the net. This is generally created with a call to `new_xdrstdio()`.

dds The Data Descriptor Structure object corresponding to this dataset. See *The DODS User Manual* for information about this structure. This would have been received from the server in an earlier transmission.

reuse A boolean value, indicating whether the class internal data storage can be reused or not. If this argument is TRUE, the class buffer is assumed to be large enough to hold the incoming data, and it is *not* reallocated. If FALSE, new storage is allocated. If the internal buffer has not been allocated at all, this argument has no effect.

Returns:
Always returns TRUE.

Exceptions:
Error when a problem reading from the XDR stream is found.

See also:
DDS

Implements BaseType.

Definition at line 98 of file Float64.cc.

3.15.3.3 bool Float64::ops (BaseType ∗ *b*, int *op*, const string & *dataset*) [virtual]

The class relational operators.

This function contains the relational operators used by the constraint expression evaluator in the DDS class. Each class that wants to be able to evaluate relational expressions must overload this function. The implementation in BaseType returns false and prints an error message.

The *op* argument refers to a table generated by bison from the constraint expression parser. Use statements like the following to correctly interpret its value:

```
switch (op) {
case EQUAL:
return i1 == i2;
case NOT_EQUAL:
return i1 != i2;
case GREATER:
return i1 > i2;
case GREATER_EQL:
return i1 >= i2;
case LESS:
return i1 < i2;
case LESS_EQL:
return i1 <= i2;
case REGEXP:
cerr << "Regular expressions not valid for integer values" << endl;
return false;
default:
```

```
cerr << "Unknown operator" << endl;
return false;
}
```

This function is used by the constraint expression evaluator.

Parameters:

b The value with which the instance value is to be compared.

op An integer index indicating which relational operator is implied. Choose one from the following: `EQUAL`, `NOT_EQUAL`, `GREATER`, `GREATER_EQL`, `LESS`, `LESS_EQL`, and `REGEXP`.

dataset The name of the dataset from which the instance's data has come (or is to come).

Returns:

The boolean value of the comparison.

Reimplemented from BaseType.

Definition at line 154 of file Float64.cc.

References BaseType::read, read, BaseType::read_p, and BaseType::type.

3.15.3.4 void Float64::print_val (ostream & *os*, string *space* = "", bool *print_decl_p* = true) `[virtual]`

Prints the value of the variable.

Prints the value of the variable, with its declaration. This function is primarily intended for debugging DODS applications. However, it can be overloaded and used to do some useful things. Take a look at the asciival and writeval clients, both of which overload this to output the values of variables in different ways.

NB: This function uses C++'s iostream to handle creating the print representations of simple type variables. For floating point numbers this is set to six digits of precision by default. If you want more precision (IEEE 64-bit floats have 15 digits of precision, 32-bit floats have 8), use the setprecision() I/O manipulator.

Parameters:

os The output stream on which to print the value.

space This value is passed to the `print_decl()` function, and controls the leading spaces of the output.

print_decl_p A boolean value controlling whether the variable declaration is printed as well as the value.

Implements BaseType.

Definition at line 141 of file Float64.cc.

References BaseType::print_decl.

3.15.3.5 virtual bool Float64::read (const string & *dataset*) [pure virtual]

Reads data into a local buffer.

Put the data into a local buffer so that it may be sent to a client. This operation involves reading data from whatever source (often a local disk), and filling out the fields in the data type class. This is the heart of the DODS DAP Class operation. Much of the work of implementing a new DODS server API consists in creating the `read()` functions to read various data types.

Note that this function is only for DODS servers. It has no use on the client side of a DODS client/server connection. The DODS client and server communicate their data with `serialize()` and `deserialize()`.

This method is not implemented for the BaseType class, nor for its children. However, it should be implemented for the specialized children of those classes. For example, it is not implemented for the Float64 class, but does exist for the NCFloat64 class, specialized to read data from local netCDF files.

This method should be implemented to throw Error when it encounters an unrecoverable error.

For an example of use, see the netCDF library classes. The netCDF library is part of the DODS source distribution, and can be found under `/export/home/tomfool/DODS/src/nc-dods`.

In some sub-classes, such as Array or Grid, the `read()` function must explicitly take into account constraint information stored with the class data. For example, the Grid::read method will be called when only one component of the Grid is to be sent. Your implementation of Grid::read should check send_p() for each member of the Grid before reading that member to avoid reading data into memory that won't be sent (and thus is not needed in memory).

Returns:

The function returns a boolean value, with TRUE indicating that read() should be called again because there's more data to read, and FALSE indicating there's no more data to read. Note that this behavior is necessary to properly handle variables that contain Sequences. WRONG! (9/5/2001 jhrg) Sequences now are read in one shot. The return value of this method should always be false.

Parameters:

dataset A string naming the dataset from which the data is to be read. The meaning of this string will vary among data APIs.

See also:
BaseType

Implements BaseType.

Referenced by ops, and serialize.

3.15.3.6 bool Float64::serialize (const string & *dataset*, DDS & *dds*, XDR $*$ *sink*, bool *ce_eval* = true) `[virtual]`

Move data to the net.

Sends the data from the indicated (local) dataset through the connection identified by the *sink* parameter. If the data is not already incorporated into the DDS object, read the data from the dataset.

This function is only used on the server side of the client/server connection, and is generally only called from the DDS::send() function. It has no BaseType implementation; each datatype child class supplies its own implementation.

Parameters:

dataset The (local) name of dataset to be read.

dds The Data Descriptor Structure object corresponding to this dataset. See *The DODS User Manual* for information about this structure.

sink A valid XDR pointer generally created with a call to `new_xdrstdio()`. This typically routes data to a TCP/IP socket.

ce_eval A boolean value indicating whether to evaluate the DODS constraint expression that may accompany this dataset. The constraint expression is stored in *dds*.

Returns:
This method always returns true. Older versions used the return value to signal success or failure.

Exceptions:

InternalErr .

Error .

See also:
DDS

Implements BaseType.

Definition at line 79 of file Float64.cc.

References DDS::eval_selection, read, and BaseType::read_p.

3.15.3.7 unsigned int Float64::val2buf (void * *val*, bool *reuse* = false) [virtual]

Loads class data.

Store the value pointed to by *val* in the object's internal buffer. This function does not perform any checks, so users must be sure that the thing pointed to can actually be stored in the object's buffer. For example, an array cannot easily be fit into the data buffer for an Int32 object.

Only simple objects (Int, Float, Byte, and so on) and arrays and lists of these simple objects may be stored using this function. To put data into more complex constructor functions, use the functions provided by that class. For example, use the `var()` and `add_var()` members of the Grid class to manipulate data in that class.

Parameters:

val A pointer to the data to be inserted into the class data buffer.

reuse A boolean value, indicating whether the class internal data storage can be reused or not. If this argument is TRUE, the class buffer is assumed to be large enough to hold the incoming data, and it is *not* reallocated. If FALSE, new storage is allocated. If the internal buffer has not been allocated at all, this argument has no effect. This is currently used only in the Vector class.

Returns:

The size (in bytes) of the information copied from *val*.

See also:

Grid , Vector::val2buf

Implements BaseType.

Definition at line 109 of file Float64.cc.

References width.

3.15.3.8 unsigned int Float64::width () [virtual]

Returns the size of the class instance data.

Return the number of bytes that are required to hold the instance's value. In the case of simple types such as Int32, this is the size of one Int32 (four bytes). For a String or Url type, `width()` returns the number of bytes needed for a `String *` variable, not the bytes needed for all the characters, since that value cannot be determined from type information alone. For Structure, and other constructor types size() returns the number of bytes needed to store pointers to the C++ objects.

Implements BaseType.

Definition at line 73 of file Float64.cc.

Referenced by buf2val, and val2buf.

3.15.4 Friends And Related Function Documentation

3.15.4.1 friend class Byte `[friend]`

This class allows Byte, ..., Float32 access to `_buf` to simplify and speed up the relational operators.

NB: According to Stroustrup it does not matter where (public, private or protected) friend classes are declared.

Definition at line 52 of file Float64.h.

The documentation for this class was generated from the following files:

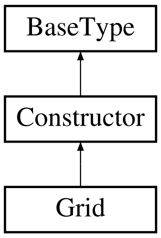
- Float64.h
- Float64.cc

3.16 Grid Class Reference

Holds the Grid data type.

```
#include <Grid.h>
```

Inheritance diagram for Grid::



Public Methods

- Grid (const string &n="")

  The Grid constructor.

- Grid (const Grid &rhs)

  The Grid copy constructor.

- virtual ∼**Grid** ()
- Grid & **operator=** (const Grid &rhs)
- virtual BaseType ∗ **ptr_duplicate** ()=0
- virtual int element_count (bool leaves=false)

  Count the members of constructor types.

- virtual void set_send_p (bool state)
- virtual void set_read_p (bool state)

  Sets the value of the read_p# flag.

- virtual BaseType ∗ var (const string &name, bool exact_match=true, btp_stack ∗s=0)
- virtual BaseType ∗ var (const string &name, btp_stack &s)

  Returns a pointer to a member of a constructor class.

- virtual void add_var (BaseType ∗bt, Part part)

Adds the input data to the class instance.

- BaseType ∗ array_var ()

  Returns the Grid Array.

- Pix first_map_var ()

  Returns the index of the first Map vector.

- void next_map_var (Pix &p)

  Increments the Map vector index.

- BaseType ∗ map_var (Pix p)

  Given an index, returns the corresponding Map vector.

- virtual unsigned int width ()

  Returns the size of the class instance data.

- virtual int components (bool constrained=false)

  Returns the number of components in the Grid object.

- virtual bool projection_yields_grid ()
- virtual bool serialize (const string &dataset, DDS &dds, XDR ∗sink, bool ce_eval=true)

  Move data to the net.

- virtual bool deserialize (XDR ∗source, DDS ∗dds, bool reuse=false)

  Receive data from the net.

- virtual bool read (const string &dataset)=0

  Reads data into a local buffer.

- virtual unsigned int val2buf (void ∗buf, bool reuse=false)

  Returns the size of the Grid type.

- virtual unsigned int buf2val (void ∗∗val)
- virtual void print_decl (ostream &os, string space="", bool print_semi=true, bool constraint_info=false, bool constrained=false)

  Prints the Grid declaration only if a valid Grid.

- virtual void print_val (ostream &os, string space="", bool print_decl_p=true)

  Prints the value of the variable.

- virtual bool check_semantics (string &msg, bool all=false)

  Return true if this Grid is well formed.

3.16.1 Detailed Description

Holds the Grid data type.

The Grid data type is a collection of an Array and a set of "Map" vectors. The Map vectors are one-dimensional arrays corresponding to each dimension of the central Array. Using this scheme, a Grid can represent, in a rectilinear array, data which is not in reality rectilinear. An example will help make it clear.

Assume that the following array contains measurements of some real quantity, conducted at nine different points in space:

```
A = [ 1  2  3  4 ]
    [ 2  4  6  8 ]
    [ 3  6  9  12]
```

To locate this Array in the real world, we could note the location of one corner of the grid, and the grid spacing. This would allow us to calculate the location of any of the other points of the Array.

This approach will not work, however, unless the grid spacing is precisely regular. If the distance between Row 1 and Row 2 is not the same as the distance between Row 2 and Row 3, the scheme will break down. The solution is to equip the Array with two Map vectors that define the location of each row or column of the array:

```
     A = [ 1  2  3  4 ] Row = [ 0 ]
         [ 2  4  6  8 ]       [ 3 ]
         [ 3  6  9  12]       [ 8 ]

Column = [ 0  2  8  27]
```

The real location of the point in the first row and column of the array is now exactly fixed at (0,0), and the point in the last row and last column is at (8,27).

The Grid data type has two parts: an Array, and a singly-linked list of Map vectors to describe the Array. The access functions for this class include a function to return the Array (`array_var()`), and a set of functions for cycling through the list of Map vectors.

See also:
Array

Definition at line 89 of file Grid.h.

3.16.2 Constructor & Destructor Documentation

3.16.2.1 Grid::Grid (const string & *n* = "")

The Grid constructor.

The Grid constructor requires only the name of the variable to be created. The name may be omitted, which will create a nameless variable. This may be adequate for some applications.

Parameters:

n A string containing the name of the variable to be created.

Definition at line 60 of file Grid.cc.

3.16.3 Member Function Documentation

3.16.3.1 void Grid::add_var (BaseType * *bt*, Part *part*) `[virtual]`

Adds the input data to the class instance.

Adds a variable to an instance of a constructor class, such as Array, Structure and so on. This function is only used by those classes. The BaseType implementation simply prints an error message.

NB: When adding a variable to a constructor or an array (this is only important for a constructor), if that variable is itself a constructor you *must* add its children *before* you add call this method to add the variable to its parent. This method copies the variable allocating a new object. One way around this is to add the constructor, then get a BaseType pointer to it using `var()`.

Parameters:

bt The data to be added to the constructor type. The caller of this method *must* free memory it allocates for `v`. This method will make a deep copy of the object pointed to by `v`.

part The part of the constructor data to be modified.

See also:

Part

Reimplemented from BaseType.

Definition at line 245 of file Grid.cc.

References BaseType::ptr_duplicate, and BaseType::set_parent.

3.16.3.2 unsigned int Grid::buf2val (void ** *val*) [virtual]

Returns the size of the Grid type. Use the `buf2val()` functions of the member elements to read values from the Grid buffer.

Implements BaseType.

Definition at line 194 of file Grid.cc.

3.16.3.3 bool Grid::check_semantics (string & *msg*, bool *all* = false) [virtual]

Return true if this Grid is well formed.

The array dimensions and number of map vectors must match and both the array and maps must be of simple-type elements.

Reimplemented from BaseType.

Definition at line 515 of file Grid.cc.

References BaseType::check_semantics, Array::dimension_size, Array::dimensions, Array::first_dim, BaseType::is_simple_type, BaseType::name, Array::next_dim, BaseType::type, BaseType::type_name, and Vector::var.

3.16.3.4 int Grid::components (bool *constrained* = false) [virtual]

Returns the number of components in the Grid object.

Returns the number of components in the Grid object. This is equal to one plus the number of Map vectors. If there is a constraint expression in effect, the number of dimensions needed may be smaller than the actual number in the stored data. (Or the Array might not even be requested.) In this case, a user can request the smaller number with the *constrained* flag.

Returns:

The number of components in the Grid object.

Parameters:

constrained If TRUE, the function returns the number of components contained in the constrained Grid. Since a constraint expression might well eliminate one or more of the Grid dimensions, this number can be lower than the actual number of components. If this parameter is FALSE (the default), the actual number of components will be returned.

Definition at line 324 of file Grid.cc.

References BaseType::send_p.

Referenced by print_decl.

3.16.3.5 bool Grid::deserialize (XDR * *source*, DDS * *dds*, bool *reuse* = false) [virtual]

Receive data from the net.

Receives data from the network connection identified by the `source` parameter. The data is put into the class data buffer according to the input `dds`.

This function is only used on the client side of the DODS client/server connection.

Parameters:

source A valid XDR pointer to the process connection to the net. This is generally created with a call to `new_xdrstdio()`.

dds The Data Descriptor Structure object corresponding to this dataset. See *The DODS User Manual* for information about this structure. This would have been received from the server in an earlier transmission.

reuse A boolean value, indicating whether the class internal data storage can be reused or not. If this argument is TRUE, the class buffer is assumed to be large enough to hold the incoming data, and it is *not* reallocated. If FALSE, new storage is allocated. If the internal buffer has not been allocated at all, this argument has no effect.

Returns:

Always returns TRUE.

Exceptions:

Error when a problem reading from the XDR stream is found.

See also:

DDS

Implements BaseType.

Definition at line 167 of file Grid.cc.

References BaseType::deserialize.

3.16.3.6 int Grid::element_count (bool *leaves* = false) [virtual]

Count the members of constructor types.

Return a count of the total number of variables in this variable. This is used to count the number of variables held by a constructor variable - for simple type and vector variables it always returns 1.

For compound data types, there are two ways to count members. You can count the members, or you can count the simple members and add that to the count of the compound members. For example, if a Structure contains an Int32 and another Structure that itself contains two Int32 members, the element count of the top-level structure could be two (one Int32 and one Structure) or three (one Int32 by itself and two Int32's in the subsidiary Structure). Use the *leaves* parameter to control which kind of counting you desire.

Returns:

Returns 1 for simple types. For compound members, the count depends on the *leaves* argument.

Parameters:

leaves This parameter is only relevant if the object contains other compound data types. If FALSE, the function counts only the data variables mentioned in the object's declaration. If TRUE, it counts the simple members, and adds that to the sum of the counts for the compound members. This parameter has no effect for simple type variables.

Reimplemented from BaseType.

Definition at line 99 of file Grid.cc.

References array_var, BaseType::element_count, first_map_var, map_var, and next_map_var.

3.16.3.7 void Grid::print_decl (ostream & *os*, string *space* = " ", bool *print_semi* = true, bool *constraint_info* = false, bool *constrained* = false) [virtual]

Prints the Grid declaration only if a valid Grid.

This mfunc has been modified so that if the user projects a Grid such that the number of components in the Grid no longer match up properly (the Array dimension no longer matches the number of Map vectors) whatever is in the projection is sent not as a Grid but as a simple array or Structure of Arrays, whichever is appropriate.

See also:

Array , Structure

Reimplemented from BaseType.

Definition at line 413 of file Grid.cc.

References components, BaseType::name, BaseType::print_decl, projection_yields_grid, BaseType::send_p, and BaseType::type_name.

Referenced by print_val.

3.16.3.8 void Grid::print_val (ostream & *os*, string *space* = "", bool *print_decl_p* = true) [virtual]

Prints the value of the variable.

Prints the value of the variable, with its declaration. This function is primarily intended for debugging DODS applications. However, it can be overloaded and used to do some useful things. Take a look at the asciival and writeval clients, both of which overload this to output the values of variables in different ways.

NB: This function uses C++'s iostream to handle creating the print representations of simple type variables. For floating point numbers this is set to six digits of precision by default. If you want more precision (IEEE 64-bit floats have 15 digits of precision, 32-bit floats have 8), use the setprecision() I/O manipulator.

Parameters:

os The output stream on which to print the value.

space This value is passed to the `print_decl()` function, and controls the leading spaces of the output.

print_decl_p A boolean value controlling whether the variable declaration is printed as well as the value.

Implements BaseType.

Definition at line 480 of file Grid.cc.

References print_decl, BaseType::print_val, projection_yields_grid, and BaseType::send_p.

3.16.3.9 bool Grid::projection_yields_grid () [virtual]

Returns TRUE if the current projection will yield a Grid that will pass the `check_semantics()` function. A Grid that, when projected, will not pass the `check_semantics()` function must be sent as either a Structure of Arrays or a single Array depending on the projection.

The function first checks to see whether the Array is present. Then, for each dimension in the Array part, the function checks the corresponding Map vector to make sure it is present in the projected Grid. If for each projected dimension in the Array component, there is a matching Map vector, then the Grid is valid.

Returns:
TRUE if the projected grid is still a Grid. FALSE otherwise.

Definition at line 362 of file Grid.cc.

References Array::dimension_size, Array::dimension_start, Array::dimension_stop, Array::dimension_stride, Array::first_dim, first_map_var, map_var, Array::next_dim, next_map_var, and BaseType::send_p.

Referenced by print_decl, and print_val.

3.16.3.10 virtual bool Grid::read (const string & *dataset*) `[pure virtual]`

Reads data into a local buffer.

Put the data into a local buffer so that it may be sent to a client. This operation involves reading data from whatever source (often a local disk), and filling out the fields in the data type class. This is the heart of the DODS DAP Class operation. Much of the work of implementing a new DODS server API consists in creating the `read()` functions to read various data types.

Note that this function is only for DODS servers. It has no use on the client side of a DODS client/server connection. The DODS client and server communicate their data with `serialize()` and `deserialize()`.

This method is not implemented for the BaseType class, nor for its children. However, it should be implemented for the specialized children of those classes. For example, it is not implemented for the Float64 class, but does exist for the NCFloat64 class, specialized to read data from local netCDF files.

This method should be implemented to throw Error when it encounters an unrecoverable error.

For an example of use, see the netCDF library classes. The netCDF library is part of the DODS source distribution, and can be found under `/export/home/tomfool/DODS/src/nc-dods`.

In some sub-classes, such as Array or Grid, the `read()` function must explicitly take into account constraint information stored with the class data. For example, the Grid::read method will be called when only one component of the Grid is to be sent. Your implementation of Grid::read should check send_p() for each member of the Grid before reading that member to avoid reading data into memory that won't be sent (and thus is not needed in memory).

Returns:
The function returns a boolean value, with TRUE indicating that read() should be called again because there's more data to read, and FALSE indicating there's no more data to read. Note that this behavior is necessary to properly handle variables that contain Sequences. WRONG! (9/5/2001 jhrg) Sequences now are read in one shot. The return value of this method should always be false.

Parameters:

dataset A string naming the dataset from which the data is to be read. The meaning of this string will vary among data APIs.

See also:

BaseType

Implements BaseType.

Referenced by serialize.

3.16.3.11 bool Grid::serialize (const string & *dataset*, DDS & *dds*, XDR ∗ *sink*, bool *ce_eval* = true) `[virtual]`

Move data to the net.

Sends the data from the indicated (local) dataset through the connection identified by the *sink* parameter. If the data is not already incorporated into the DDS object, read the data from the dataset.

This function is only used on the server side of the client/server connection, and is generally only called from the DDS::send() function. It has no BaseType implementation; each datatype child class supplies its own implementation.

Parameters:

dataset The (local) name of dataset to be read.

dds The Data Descriptor Structure object corresponding to this dataset. See *The DODS User Manual* for information about this structure.

sink A valid XDR pointer generally created with a call to `new_xdrstdio()`. This typically routes data to a TCP/IP socket.

ce_eval A boolean value indicating whether to evaluate the DODS constraint expression that may accompany this dataset. The constraint expression is stored in *dds*.

Returns:

This method always returns true. Older versions used the return value to signal success or failure.

Exceptions:

InternalErr .

Error .

See also:

DDS

Implements BaseType.

Definition at line 147 of file Grid.cc.

References DDS::eval_selection, read, BaseType::read_p, BaseType::send_p, and BaseType::serialize.

3.16.3.12 void Grid::set_read_p (bool *state*) [virtual]

Sets the value of the read_p# flag.

Sets the value of the `read_p` flag. This flag is TRUE when the class instance contains a valid value, and FALSE before a valid value has been read. This is meant to be called from the `read()` function. Data is ready to be sent when *both* the `_send_p` and `_read_p` flags are set to TRUE.

Parameters:
state The logical state to set the `read_p` flag.

Reimplemented from BaseType.

Definition at line 125 of file Grid.cc.

References BaseType::set_read_p.

3.16.3.13 void Grid::set_send_p (bool *state*) [virtual]

Sets the value of the `send_p` flag. This function is meant to be called from `serialize()`. Data is ready to be sent when *both* the `_send_p` and `_read_p` flags are set to TRUE.

Parameters:
state The logical state to set the `send_p` flag.

Reimplemented from BaseType.

Definition at line 114 of file Grid.cc.

References BaseType::set_send_p.

3.16.3.14 unsigned int Grid::val2buf (void * *buf*, bool *reuse* = false) [virtual]

Returns the size of the Grid type.

Use the `val2buf()` functions of the member elements to insert values into the Grid buffer.

Returns:
The size (in bytes) of the value's representation.

Implements BaseType.

Definition at line 185 of file Grid.cc.

3.16.3.15 BaseType ∗ Grid::var (const string & *name*, btp_stack & *s*) [virtual]

Returns a pointer to a member of a constructor class.

This version of var(...) searches for *name* and returns a pointer to the BaseType object if found. It uses the same search algorithm as above when *exact_match* is false. In addition to returning a pointer to the variable, it pushes onto *s* a BaseType pointer to each constructor type that ultimately contains *name*.

\note{The BaseType implementation always returns null. }

Deprecated:

Parameters:
name Find the variable whose name is *name*.
s Record the path to *name*.

Returns:
A pointer to the named variable.

Reimplemented from BaseType.

Definition at line 200 of file Grid.cc.

References BaseType::name, and var.

3.16.3.16 BaseType ∗ Grid::var (const string & *name*, bool *exact_match* = true, btp_stack ∗ *s* = 0) [virtual]

Note the parameter *exact_match* is not used by this member function.

See also:
BaseType

Reimplemented from BaseType.

Definition at line 210 of file Grid.cc.

References BaseType::name.

Referenced by GSEClause::GSEClause, and var.

3.16.3.17 unsigned int Grid::width () `[virtual]`

Returns the size of the class instance data.

Return the number of bytes that are required to hold the instance's value. In the case of simple types such as Int32, this is the size of one Int32 (four bytes). For a String or Url type, `width()` returns the number of bytes needed for a `String *` variable, not the bytes needed for all the characters, since that value cannot be determined from type information alone. For Structure, and other constructor types size() returns the number of bytes needed to store pointers to the C++ objects.

Implements BaseType.

Definition at line 136 of file Grid.cc.

References BaseType::width.

The documentation for this class was generated from the following files:

- Grid.h
- Grid.cc

3.17 gse_arg Struct Reference

```
#include <parser.h>
```

Public Methods

- **gse_arg** ()
- **gse_arg** (Grid *g)
- virtual ~**gse_arg** ()
- void **set_gsec** (GSEClause *gsec)
- GSEClause * **get_gsec** ()
- void **set_grid** (Grid *g)
- Grid * **get_grid** ()
- void **set_status** (int stat)
- int **get_status** ()

Public Attributes

- GSEClause * **_gsec**
- Grid * **_grid**
- int **_status**

3.17.1 Detailed Description

Argument to the GSE parser. Assumes all errors will be signalled by throws; Error objects are not returned.

Definition at line 70 of file parser.h.

The documentation for this struct was generated from the following file:

- parser.h

3.18 GSEClause Class Reference

`#include <GSEClause.h>`

Public Methods

- bool OK () const

Constructors

- GSEClause (Grid *grid, const string &map, const double value, const relop op)
  Create an instance using discrete parameters.
- GSEClause (Grid *grid, const string &map, const double value1, const relop op1, const double value2, const relop op2)
  Create an instance using discrete parameters.
- GSEClause (Grid *grid, const string &expr)
  Create an instance using a grid and an expression.
- GSEClause (Grid *grid, char *expr)
  Create an instance using a grid and an expression.

Accessors

- Array * get_map () const
  Get a pointer to the map variable constrained by this clause.
- string get_map_name () const
  Get the name of the map variable constrained by this clause.
- int get_start () const
  Get the starting index of the clause's map variable as constrained by this clause.
- int get_stop () const
  Get the stopping index of the clause's map variable as constrained by this clause.
- string get_map_min_value () const
  Get the minimum map vector value.

- string get_map_max_value () const
  Get the maximum map vector value.

Mutators

- void set_map (Array *map)
  Set the pointer to the map vector contrained by this clause.
- void set_start (int start)
  Set the starting index.
- void set_stop (int stop)
  Set the stopping index.

3.18.1 Detailed Description

Holds the results of parsing one of the Grid Selection Expression clauses. The Grid selection function takes a set of clauses as arguments and must create one instance of this class for each of those clauses. The GridSelectionExpr class holds N instances of this class.

Author:
James Gallagher

See also:
GridSelectionExpr

Definition at line 51 of file GSEClause.h.

3.18.2 Member Function Documentation

3.18.2.1 Array * GSEClause::get_map () const

Get a pointer to the map variable constrained by this clause.

Returns:
The Array object.

Definition at line 273 of file GSEClause.cc.

3.18.2.2 string GSEClause::get_map_max_value () const

Get the maximum map vector value.

Useful in messages back to users.

Returns:
The maximum map vetor value.

Definition at line 346 of file GSEClause.cc.

3.18.2.3 string GSEClause::get_map_min_value () const

Get the minimum map vector value.

Useful in messages back to users.

Returns:
The minimum map vetor value.

Definition at line 336 of file GSEClause.cc.

3.18.2.4 string GSEClause::get_map_name () const

Get the name of the map variable constrained by this clause.

Returns:
The Array object's name.

Definition at line 291 of file GSEClause.cc.

References BaseType::name.

3.18.2.5 int GSEClause::get_start () const

Get the starting index of the clause's map variable as constrained by this clause.

Returns:
The start index.

Definition at line 300 of file GSEClause.cc.

3.18.2.6 int GSEClause::get_stop () const

Get the stopping index of the clause's map variable as constrained by this clause.

Returns:
The stop index.

Definition at line 317 of file GSEClause.cc.

3.18.2.7 bool GSEClause::OK () const

Class invariant.

Returns:
True if the object is valid, otherwise False.

Definition at line 260 of file GSEClause.cc.

3.18.2.8 void GSEClause::set_map (Array * *map*)

Set the pointer to the map vector contrained by this clause.

Note that this method also sets the name of the map vector.

Returns:
void

Definition at line 283 of file GSEClause.cc.

3.18.2.9 void GSEClause::set_start (int *start*)

Set the starting index.

Returns:
void

Definition at line 308 of file GSEClause.cc.

3.18.2.10 void GSEClause::set_stop (int *stop*)

Set the stopping index.

Returns:
void

Definition at line 326 of file GSEClause.cc.

The documentation for this class was generated from the following files:

- GSEClause.h
- GSEClause.cc

3.19 Gui Class Reference

Implements a graphic user interface for a DODS client.

`#include <Gui.h>`

Public Methods

- virtual ~**Gui** ()
- bool progress_visible (int state=-1)

  Set or test progress indicator visibility.

- bool show_gui (int state=-1)

  Should the GUI be used?

- bool command (string command)

  Send the GUI process a command.

- bool password (string &user, string &pass)

  Get username and password information.

- bool simple_error (string msg)
- bool percent_bar (double pro, int *usr_cancel)

  Moves the progress bar back and forth in its slot. If user presses "Cancel" button parameter `usr_cancel` *is true.*

Gui();

- Gui (const string cmd)

  The Gui class constructor.

- **Gui** ()

3.19.1 Detailed Description

Implements a graphic user interface for a DODS client.

This class manages a simple graphical user interface (GUI) for a DODS client. The GUI is implemented by an embedded Tcl/Tk interpreter and a short tk script (see dods_gui.tcl). The script contains a number of functions which are loaded into the embedded interpreter. The functions defined by the script can be run using the methods of Gui. Some of the functions can be accessed only through certain special methods (e.g., `simple_error`) while others can be accessed using their name via the command(...) method.

Reimplement as a singleton ***

The Gui class was/is intended to support the transfer progress indicator. It also supports display of various error messages and provides a way for users to supply username and password pairs for secure sites.

Do not confuse the DODS Client GUI, managed by this class, with a DODS GUI Client, such as the DODS Matlab GUI, used for the graphical display of DODS datasets. The two can work together, but otherwise have nothing to do with each other.

The GUI process uses the following environment variables:

`DODS_USE_GUI` If this variable exists and equals “no”, the progress indicator will not appear. If it does not exist, or if it has any other value, the progress indicator will be activated.

Note that for historical reasons there is no copy constructor or compare operators.

Definition at line 66 of file Gui.h.

3.19.2 Constructor & Destructor Documentation

3.19.2.1 Gui::Gui (const string *cmd*)

The Gui class constructor.

Create the instance of Gui and initialize the embedded interpreter. In previous versions of this class it was possible to change the way the interpreter was initialized. However, that is no longer an option.

If the constructor is called with a string argument, send that string to the interpreter as an initial command.

Parameters:

cmd Send the Tcl/Tk interpreter this command.

Definition at line 63 of file Gui.cc.

3.19.3 Member Function Documentation

3.19.3.1 bool Gui::command (string *command*)

Send the GUI process a command.

Submits a command to the GUI interpreter for execution.

It may be that users turn off the GUI (because they are sick of it). In that case, this command will simply return false. It does no harm to call the command when no GUI is running. Care should be taken to send only valid commands—invalid commands may put the GUI into an inoperable state.

If the GUI is not running, this function starts it.

Returns:
TRUE if the command succeeded, FALSE otherwise.

Parameters:
command A command to submit to the GUI process for execution.

Definition at line 165 of file Gui.cc.

References command, and show_gui.

Referenced by command.

3.19.3.2 bool Gui::password (string & *user*, string & *pass*)

Get username and password information.

Pops up a username/password dialog box. The values entered by the user are returned in the parameters `user` and `pass`.

Parameters:
user The user's name.
pass The user's password.

Returns:
true if the command succeeded, false otherwise.

Definition at line 218 of file Gui.cc.

References show_gui.

3.19.3.3 bool Gui::percent_bar (double *pro*, int ∗ *usr_cancel*)

Moves the progress bar back and forth in its slot. If user presses "Cancel" button parameter `usr_cancel` is true.

Parameters:

pro Percent of progress.

usr_cancel The cancel button.

Definition at line 255 of file Gui.cc.

References show_gui.

3.19.3.4 bool Gui::progress_visible (int *state* = -1)

Set or test progress indicator visibility.

This function can be used to query whether the progress indicator is visible or not. It may also be used to set whether it is to be seen or not.

Parameters:

state If *state* is omitted (or is -1, the default), the function returns the visibility state. If *state* is included, the function sets the visibility state to the given value.

Returns:

Returns TRUE if the GUI is visible, false otherwise.

Definition at line 142 of file Gui.cc.

3.19.3.5 bool Gui::show_gui (int *state* = -1)

Should the GUI be used?

This function returns the state of two internal flags. The first, under program control, indicates whether or not to use the GUI at all. This is a private value called `_show_gui`. The second flag is derived from the value of the environment variable `DODS_USE_GUI`, and is set when the class instance is created. It is also a private value (`_user_wants_gui`), and cannot be modified. The GUI is displayed only if both these flags are TRUE.

This function can be used to modify the value of the `_show_gui` flag.

Parameters:

state If *state* is omitted (or is -1, the default), the function returns the result of the operation `_show_gui` AND `_user_wants_gui`. If *state* is included, the function sets the value of the `_show_gui` flag.

Returns:
The value of `_show_gui` AND `_user_wants_gui`.

Definition at line 305 of file Gui.cc.

Referenced by command, Error::display_message, password, percent_bar, and simple_-error.

3.19.3.6 bool Gui::simple_error (string *error_msg*)

Print an error message and wait for the user to acknowledge it.

Parameters:
error_msg message to be printed.

Definition at line 188 of file Gui.cc.

References show_gui.

Referenced by Error::display_message.

The documentation for this class was generated from the following files:

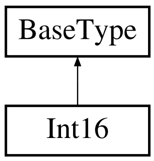
- Gui.h
- Gui.cc

3.20 Int16 Class Reference

Holds a 16-bit signed integer value.

```
#include <Int16.h>
```

Inheritance diagram for Int16::



Public Methods

- **Int16** (const string &n="")
- virtual ~**Int16** ()
- **Int16** (const Int16 &copy_from)
- Int16 & **operator=** (const Int16 &rhs)
- virtual BaseType * **ptr_duplicate** ()=0
- virtual unsigned int width ()

  Returns the size of the class instance data.

- virtual bool serialize (const string &dataset, DDS &dds, XDR *sink, bool ce_eval=true)

  Move data to the net.

- virtual bool deserialize (XDR *source, DDS *dds, bool reuse=false)

  Receive data from the net.

- virtual bool read (const string &dataset)=0

  Reads data into a local buffer.

- virtual unsigned int val2buf (void *val, bool reuse=false)

  Loads class data.

- virtual unsigned int buf2val (void **val)

  Reads the class data.

- virtual void print_val (ostream &os, string space="", bool print_decl_p=true)

  Prints the value of the variable.

- virtual bool ops (BaseType *b, int op, const string &dataset)

  The class relational operators.

Protected Attributes

- dods_int16 **_buf**

Friends

- class Byte
- class UInt16
- class Int32
- class UInt32
- class Float32
- class Float64

3.20.1 Detailed Description

Holds a 16-bit signed integer value.

Definition at line 43 of file Int16.h.

3.20.2 Member Function Documentation

3.20.2.1 unsigned int Int16::buf2val (void ** *val*) `[virtual]`

Reads the class data.

Reads the class data into the memory referenced by *val*. The caller must allocate enough storage to *val* to hold the class data. If *val* is NULL, however, memory will be allocated by this function with `new()`. Even if the memory is allocated this way, the caller is responsible for deallocating that memory. Array and List values for simple types are stored as C would store an array.

Parameters:

val A pointer to a pointer to the memory into which the class data will be copied. If the value pointed to is NULL, memory will be allocated to hold the data, and the pointer value modified accordingly. The calling program is responsible for deallocating the memory indicated by this pointer.

Returns:

The size (in bytes) of the information copied to *val*.

Implements BaseType.

Definition at line 115 of file Int16.cc.

References width.

3.20.2.2 bool Int16::deserialize (XDR * *source*, DDS * *dds*, bool *reuse* = false) `[virtual]`

Receive data from the net.

Receives data from the network connection identified by the `source` parameter. The data is put into the class data buffer according to the input `dds`.

This function is only used on the client side of the DODS client/server connection.

Parameters:

source A valid XDR pointer to the process connection to the net. This is generally created with a call to `new_xdrstdio()`.

dds The Data Descriptor Structure object corresponding to this dataset. See *The DODS User Manual* for information about this structure. This would have been received from the server in an earlier transmission.

reuse A boolean value, indicating whether the class internal data storage can be reused or not. If this argument is TRUE, the class buffer is assumed to be large enough to hold the incoming data, and it is *not* reallocated. If FALSE, new storage is allocated. If the internal buffer has not been allocated at all, this argument has no effect.

Returns:

Always returns TRUE.

Exceptions:

Error when a problem reading from the XDR stream is found.

See also:

DDS

Implements BaseType.

Definition at line 88 of file Int16.cc.

3.20.2.3 bool Int16::ops (BaseType * *b*, int *op*, const string & *dataset*) [virtual]

The class relational operators.

This function contains the relational operators used by the constraint expression evaluator in the DDS class. Each class that wants to be able to evaluate relational expressions must overload this function. The implementation in BaseType returns false and prints an error message.

The *op* argument refers to a table generated by bison from the constraint expression parser. Use statements like the following to correctly interpret its value:

```
switch (op) {
case EQUAL:
return i1 == i2;
case NOT_EQUAL:
return i1 != i2;
case GREATER:
return i1 > i2;
case GREATER_EQL:
return i1 >= i2;
case LESS:
return i1 < i2;
case LESS_EQL:
return i1 <= i2;
case REGEXP:
cerr << "Regular expressions not valid for integer values" << endl;
return false;
default:
cerr << "Unknown operator" << endl;
return false;
}
```

This function is used by the constraint expression evaluator.

Parameters:

b The value with which the instance value is to be compared.

op An integer index indicating which relational operator is implied. Choose one from the following: `EQUAL`, `NOT_EQUAL`, `GREATER`, `GREATER_EQL`, `LESS`, `LESS_EQL`, and `REGEXP`.

dataset The name of the dataset from which the instance's data has come (or is to come).

Returns:

The boolean value of the comparison.

Reimplemented from BaseType.

Definition at line 144 of file Int16.cc.

References BaseType::read, read, BaseType::read_p, and BaseType::type.

3.20.2.4 void Int16::print_val (ostream & *os*, string *space* = "", bool *print_decl_p* = true) [virtual]

Prints the value of the variable.

Prints the value of the variable, with its declaration. This function is primarily intended for debugging DODS applications. However, it can be overloaded and used to do some useful things. Take a look at the asciival and writeval clients, both of which overload this to output the values of variables in different ways.

NB: This function uses C++'s iostream to handle creating the print representations of simple type variables. For floating point numbers this is set to six digits of precision by default. If you want more precision (IEEE 64-bit floats have 15 digits of precision, 32-bit floats have 8), use the setprecision() I/O manipulator.

Parameters:

os The output stream on which to print the value.

space This value is passed to the `print_decl()` function, and controls the leading spaces of the output.

print_decl_p A boolean value controlling whether the variable declaration is printed as well as the value.

Implements BaseType.

Definition at line 133 of file Int16.cc.

References BaseType::print_decl.

3.20.2.5 virtual bool Int16::read (const string & *dataset*) [pure virtual]

Reads data into a local buffer.

Put the data into a local buffer so that it may be sent to a client. This operation involves reading data from whatever source (often a local disk), and filling out the fields in the data type class. This is the heart of the DODS DAP Class operation. Much of the work of implementing a new DODS server API consists in creating the `read()` functions to read various data types.

Note that this function is only for DODS servers. It has no use on the client side of a DODS client/server connection. The DODS client and server communicate their data with `serialize()` and `deserialize()`.

This method is not implemented for the BaseType class, nor for its children. However, it should be implemented for the specialized children of those classes. For example,

it is not implemented for the Float64 class, but does exist for the NCFloat64 class, specialized to read data from local netCDF files.

This method should be implemented to throw Error when it encounters an unrecoverable error.

For an example of use, see the netCDF library classes. The netCDF library is part of the DODS source distribution, and can be found under `/export/home/tomfool/DODS/src/nc-dods`.

In some sub-classes, such as Array or Grid, the `read()` function must explicitly take into account constraint information stored with the class data. For example, the Grid::read method will be called when only one component of the Grid is to be sent. Your implementation of Grid::read should check send_p() for each member of the Grid before reading that member to avoid reading data into memory that won't be sent (and thus is not needed in memory).

Returns:

The function returns a boolean value, with TRUE indicating that read() should be called again because there's more data to read, and FALSE indicating there's no more data to read. Note that this behavior is necessary to properly handle variables that contain Sequences. WRONG! (9/5/2001 jhrg) Sequences now are read in one shot. The return value of this method should always be false.

Parameters:

dataset A string naming the dataset from which the data is to be read. The meaning of this string will vary among data APIs.

See also:

BaseType

Implements BaseType.

Referenced by ops, and serialize.

3.20.2.6 bool Int16::serialize (const string & *dataset*, DDS & *dds*, XDR ∗ *sink*, bool *ce_eval* = true) `[virtual]`

Move data to the net.

Sends the data from the indicated (local) dataset through the connection identified by the *sink* parameter. If the data is not already incorporated into the DDS object, read the data from the dataset.

This function is only used on the server side of the client/server connection, and is generally only called from the DDS::send() function. It has no BaseType implementation; each datatype child class supplies its own implementation.

Parameters:

dataset The (local) name of dataset to be read.

dds The Data Descriptor Structure object corresponding to this dataset. See *The DODS User Manual* for information about this structure.

sink A valid XDR pointer generally created with a call to `new_xdrstdio()`. This typically routes data to a TCP/IP socket.

ce_eval A boolean value indicating whether to evaluate the DODS constraint expression that may accompany this dataset. The constraint expression is stored in *dds*.

Returns:

This method always returns true. Older versions used the return value to signal success or failure.

Exceptions:

InternalErr .

Error .

See also:

DDS

Implements BaseType.

Definition at line 69 of file Int16.cc.

References DDS::eval_selection, read, and BaseType::read_p.

3.20.2.7 unsigned int Int16::val2buf (void * *val*, bool *reuse* = false) `[virtual]`

Loads class data.

Store the value pointed to by *val* in the object's internal buffer. This function does not perform any checks, so users must be sure that the thing pointed to can actually be stored in the object's buffer. For example, an array cannot easily be fit into the data buffer for an Int32 object.

Only simple objects (Int, Float, Byte, and so on) and arrays and lists of these simple objects may be stored using this function. To put data into more complex constructor functions, use the functions provided by that class. For example, use the `var()` and `add_var()` members of the Grid class to manipulate data in that class.

Parameters:

val A pointer to the data to be inserted into the class data buffer.

reuse A boolean value, indicating whether the class internal data storage can be reused or not. If this argument is TRUE, the class buffer is assumed to be

large enough to hold the incoming data, and it is *not* reallocated. If FALSE, new storage is allocated. If the internal buffer has not been allocated at all, this argument has no effect. This is currently used only in the Vector class.

Returns:
The size (in bytes) of the information copied from *val*.

See also:
Grid , Vector::val2buf

Implements BaseType.

Definition at line 99 of file Int16.cc.

References width.

3.20.2.8 unsigned int Int16::width () `[virtual]`

Returns the size of the class instance data.

Return the number of bytes that are required to hold the instance's value. In the case of simple types such as Int32, this is the size of one Int32 (four bytes). For a String or Url type, `width()` returns the number of bytes needed for a `String *` variable, not the bytes needed for all the characters, since that value cannot be determined from type information alone. For Structure, and other constructor types size() returns the number of bytes needed to store pointers to the C++ objects.

Implements BaseType.

Definition at line 63 of file Int16.cc.

Referenced by buf2val, and val2buf.

3.20.3 Friends And Related Function Documentation

3.20.3.1 friend class Byte `[friend]`

This class allows Byte, ..., Float64 access to `_buf` to simplify and speed up the relational operators.

NB: According to Stroustrup it does not matter where (public, private or protected) friend classes are declared.

Definition at line 49 of file Int16.h.

The documentation for this class was generated from the following files:

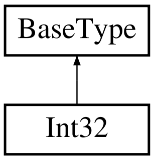
- Int16.h
- Int16.cc

3.21 Int32 Class Reference

Holds a 32-bit signed integer.

`#include <Int32.h>`

Inheritance diagram for Int32::



Public Methods

- Int32 (const string &n="")
- **Int32** (const Int32 &copy_from)
- Int32 & **operator=** (const Int32 &rhs)
- virtual ~**Int32** ()
- virtual BaseType * **ptr_duplicate** ()=0
- virtual unsigned int width ()

  Returns the size of the class instance data.

- virtual bool serialize (const string &dataset, DDS &dds, XDR *sink, bool ce_-eval=true)

  Move data to the net.

- virtual bool deserialize (XDR *source, DDS *dds, bool reuse=false)

  Receive data from the net.

- virtual bool read (const string &dataset)=0

  Reads data into a local buffer.

- virtual unsigned int val2buf (void *val, bool reuse=false)

  Loads class data.

- virtual unsigned int buf2val (void **val)

  Reads the class data.

- virtual void print_val (ostream &os, string space="", bool print_decl_p=true)

    Prints the value of the variable.

- virtual bool ops (BaseType ∗b, int op, const string &dataset)

    The class relational operators.

Protected Attributes

- dods_int32 **_buf**

Friends

- class Byte
- class Int16
- class UInt16
- class UInt32
- class Float32
- class Float64

3.21.1 Detailed Description

Holds a 32-bit signed integer.

See also:
BaseType

Definition at line 46 of file Int32.h.

3.21.2 Constructor & Destructor Documentation

3.21.2.1 Int32::Int32 (const string & *n* = "")

The Int32 constructor requires only the name of the variable to be created. The name may be omitted, which will create a nameless variable. This may be adequate for some applications.

Parameters:
n A string containing the name of the variable to be created.

Definition at line 47 of file Int32.cc.

3.21.3 Member Function Documentation

3.21.3.1 unsigned int Int32::buf2val (void ** *val*) [virtual]

Reads the class data.

Reads the class data into the memory referenced by *val*. The caller must allocate enough storage to *val* to hold the class data. If *val* is NULL, however, memory will be allocated by this function with `new()`. Even if the memory is allocated this way, the caller is responsible for deallocating that memory. Array and List values for simple types are stored as C would store an array.

Parameters:

val A pointer to a pointer to the memory into which the class data will be copied. If the value pointed to is NULL, memory will be allocated to hold the data, and the pointer value modified accordingly. The calling program is responsible for deallocating the memory indicated by this pointer.

Returns:

The size (in bytes) of the information copied to *val*.

Implements BaseType.

Definition at line 128 of file Int32.cc.

References width.

3.21.3.2 bool Int32::deserialize (XDR * *source*, DDS * *dds*, bool *reuse* = false) [virtual]

Receive data from the net.

Receives data from the network connection identified by the `source` parameter. The data is put into the class data buffer according to the input `dds`.

This function is only used on the client side of the DODS client/server connection.

Parameters:

source A valid XDR pointer to the process connection to the net. This is generally created with a call to `new_xdrstdio()`.

dds The Data Descriptor Structure object corresponding to this dataset. See *The DODS User Manual* for information about this structure. This would have been received from the server in an earlier transmission.

reuse A boolean value, indicating whether the class internal data storage can be reused or not. If this argument is TRUE, the class buffer is assumed to be large enough to hold the incoming data, and it is *not* reallocated. If FALSE, new storage is allocated. If the internal buffer has not been allocated at all, this argument has no effect.

Returns:
Always returns TRUE.

Exceptions:
Error when a problem reading from the XDR stream is found.

See also:
DDS

Implements BaseType.

Definition at line 101 of file Int32.cc.

3.21.3.3 bool Int32::ops (BaseType ∗ *b*, int *op*, const string & *dataset*) [virtual]

The class relational operators.

This function contains the relational operators used by the constraint expression evaluator in the DDS class. Each class that wants to be able to evaluate relational expressions must overload this function. The implementation in BaseType returns false and prints an error message.

The *op* argument refers to a table generated by bison from the constraint expression parser. Use statements like the following to correctly interpret its value:

```
switch (op) {
case EQUAL:
return i1 == i2;
case NOT_EQUAL:
return i1 != i2;
case GREATER:
return i1 > i2;
case GREATER_EQL:
return i1 >= i2;
case LESS:
return i1 < i2;
case LESS_EQL:
return i1 <= i2;
case REGEXP:
cerr << "Regular expressions not valid for integer values" << endl;
return false;
default:
```

```
cerr << "Unknown operator" << endl;
return false;
}
```

This function is used by the constraint expression evaluator.

Parameters:

b The value with which the instance value is to be compared.

op An integer index indicating which relational operator is implied. Choose one from the following: `EQUAL`, `NOT_EQUAL`, `GREATER`, `GREATER_EQL`, `LESS`, `LESS_EQL`, and `REGEXP`.

dataset The name of the dataset from which the instance's data has come (or is to come).

Returns:

The boolean value of the comparison.

Reimplemented from BaseType.

Definition at line 157 of file Int32.cc.

References BaseType::read, read, BaseType::read_p, and BaseType::type.

3.21.3.4 void Int32::print_val (ostream & *os*, string *space* = "", bool *print_decl_p* = true) `[virtual]`

Prints the value of the variable.

Prints the value of the variable, with its declaration. This function is primarily intended for debugging DODS applications. However, it can be overloaded and used to do some useful things. Take a look at the asciival and writeval clients, both of which overload this to output the values of variables in different ways.

NB: This function uses C++'s iostream to handle creating the print representations of simple type variables. For floating point numbers this is set to six digits of precision by default. If you want more precision (IEEE 64-bit floats have 15 digits of precision, 32-bit floats have 8), use the setprecision() I/O manipulator.

Parameters:

os The output stream on which to print the value.

space This value is passed to the `print_decl()` function, and controls the leading spaces of the output.

print_decl_p A boolean value controlling whether the variable declaration is printed as well as the value.

Implements BaseType.

Definition at line 146 of file Int32.cc.

References BaseType::print_decl.

3.21.3.5 virtual bool Int32::read (const string & *dataset*) `[pure virtual]`

Reads data into a local buffer.

Put the data into a local buffer so that it may be sent to a client. This operation involves reading data from whatever source (often a local disk), and filling out the fields in the data type class. This is the heart of the DODS DAP Class operation. Much of the work of implementing a new DODS server API consists in creating the `read()` functions to read various data types.

Note that this function is only for DODS servers. It has no use on the client side of a DODS client/server connection. The DODS client and server communicate their data with `serialize()` and `deserialize()`.

This method is not implemented for the BaseType class, nor for its children. However, it should be implemented for the specialized children of those classes. For example, it is not implemented for the Float64 class, but does exist for the NCFloat64 class, specialized to read data from local netCDF files.

This method should be implemented to throw Error when it encounters an unrecoverable error.

For an example of use, see the netCDF library classes. The netCDF library is part of the DODS source distribution, and can be found under `/export/home/tomfool/DODS/src/nc-dods`.

In some sub-classes, such as Array or Grid, the `read()` function must explicitly take into account constraint information stored with the class data. For example, the Grid::read method will be called when only one component of the Grid is to be sent. Your implementation of Grid::read should check send_p() for each member of the Grid before reading that member to avoid reading data into memory that won't be sent (and thus is not needed in memory).

Returns:

The function returns a boolean value, with TRUE indicating that read() should be called again because there's more data to read, and FALSE indicating there's no more data to read. Note that this behavior is necessary to properly handle variables that contain Sequences. WRONG! (9/5/2001 jhrg) Sequences now are read in one shot. The return value of this method should always be false.

Parameters:

dataset A string naming the dataset from which the data is to be read. The meaning of this string will vary among data APIs.

See also:
BaseType

Implements BaseType.

Referenced by ops, and serialize.

3.21.3.6 bool Int32::serialize (const string & *dataset*, DDS & *dds*, XDR $*$ *sink*, bool *ce_eval* = true) [virtual]

Move data to the net.

Sends the data from the indicated (local) dataset through the connection identified by the *sink* parameter. If the data is not already incorporated into the DDS object, read the data from the dataset.

This function is only used on the server side of the client/server connection, and is generally only called from the DDS::send() function. It has no BaseType implementation; each datatype child class supplies its own implementation.

Parameters:

dataset The (local) name of dataset to be read.

dds The Data Descriptor Structure object corresponding to this dataset. See *The DODS User Manual* for information about this structure.

sink A valid XDR pointer generally created with a call to `new_xdrstdio()`. This typically routes data to a TCP/IP socket.

ce_eval A boolean value indicating whether to evaluate the DODS constraint expression that may accompany this dataset. The constraint expression is stored in *dds*.

Returns:
This method always returns true. Older versions used the return value to signal success or failure.

Exceptions:

InternalErr .

Error .

See also:
DDS

Implements BaseType.

Definition at line 82 of file Int32.cc.

References DDS::eval_selection, read, and BaseType::read_p.

3.21.3.7 unsigned int Int32::val2buf (void * *val*, bool *reuse* = false) `[virtual]`

Loads class data.

Store the value pointed to by *val* in the object's internal buffer. This function does not perform any checks, so users must be sure that the thing pointed to can actually be stored in the object's buffer. For example, an array cannot easily be fit into the data buffer for an Int32 object.

Only simple objects (Int, Float, Byte, and so on) and arrays and lists of these simple objects may be stored using this function. To put data into more complex constructor functions, use the functions provided by that class. For example, use the `var()` and `add_var()` members of the Grid class to manipulate data in that class.

Parameters:

val A pointer to the data to be inserted into the class data buffer.

reuse A boolean value, indicating whether the class internal data storage can be reused or not. If this argument is TRUE, the class buffer is assumed to be large enough to hold the incoming data, and it is *not* reallocated. If FALSE, new storage is allocated. If the internal buffer has not been allocated at all, this argument has no effect. This is currently used only in the Vector class.

Returns:

The size (in bytes) of the information copied from *val*.

See also:

Grid , Vector::val2buf

Implements BaseType.

Definition at line 112 of file Int32.cc.

References width.

3.21.3.8 unsigned int Int32::width () `[virtual]`

Returns the size of the class instance data.

Return the number of bytes that are required to hold the instance's value. In the case of simple types such as Int32, this is the size of one Int32 (four bytes). For a String or Url type, `width()` returns the number of bytes needed for a `String *` variable, not the bytes needed for all the characters, since that value cannot be determined from type information alone. For Structure, and other constructor types size() returns the number of bytes needed to store pointers to the C++ objects.

Implements BaseType.

Definition at line 76 of file Int32.cc.

Referenced by buf2val, and val2buf.

3.21.4 Friends And Related Function Documentation

3.21.4.1 friend class Byte `[friend]`

This class allows Byte, ..., Float64 access to `_buf` to simplify and speed up the relational operators.

NB: According to Stroustrup it does not matter where (public, private or protected) friend classes are declared.

Definition at line 52 of file Int32.h.

The documentation for this class was generated from the following files:

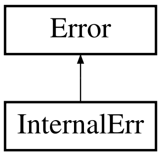
- Int32.h
- Int32.cc

3.22 InternalErr Class Reference

A class for software fault reporting.

```
#include <InternalErr.h>
```

Inheritance diagram for InternalErr::



Public Methods

- virtual ~**InternalErr** ()
- bool OK ()

  Is the InternalErr object valid?

- InternalErr (const string &msg)
- **InternalErr** (const string &file, const int &line, const string &msg)
- **InternalErr** (string msg, ProgramType pt, char *pgm)
- **InternalErr** ()
- **InternalErr** (const InternalErr &copy_from)

3.22.1 Detailed Description

A class for software fault reporting.

The InternalErr class is used to signal that somewhere inside DODS a software fault was found. This class wraps the message text in some boiler plate that asks the error be reported to us (DODS support).

NB: This class Adds some text to the message and might, in the future, hold information not also held in Error. However, all Error objects thrown on the server-side of DODS that cannot be resolved (and that's all of them for now, 5/3/99 jhrg) will be sent to the client-side *using Error objects*. That is, the information recorded in an InternalErr object will be sent by calling the `print(...)` mfunc of Error.

Author:
jhrg

Definition at line 44 of file InternalErr.h.

3.22.2 Constructor & Destructor Documentation

3.22.2.1 InternalErr::InternalErr (const string & *msg*)

These constructors always set the error code to `internal_error`.

Note that there is no way to specify an error correction program for an internal error.

Definition at line 35 of file InternalErr.cc.

3.22.3 Member Function Documentation

3.22.3.1 bool InternalErr::OK ()

Is the InternalErr object valid?

Returns:
TRUE if the object is valid, FALSE otherwise.

Definition at line 84 of file InternalErr.cc.

References Error::OK.

The documentation for this class was generated from the following files:

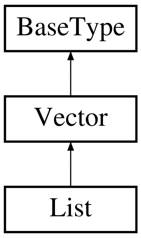
- InternalErr.h
- InternalErr.cc

3.23 List Class Reference

Holds a list of other objects.

`#include <List.h>`

Inheritance diagram for List::



Public Methods

- List (const string &n="", BaseType *v=0)
- List (const List &rhs)
- virtual ~**List** ()
- List & **operator=** (const List &rhs)
- virtual BaseType * **ptr_duplicate** ()=0
- virtual bool read (const string &dataset)=0

  Reads data into a local buffer.

- virtual bool member (BaseType *btp, DDS &dds)
- virtual bool null ()

Protected Methods

- void **_duplicate** (const List &l)

3.23.1 Detailed Description

Holds a list of other objects.

This class uses the Vector class to implement a simple list of DODS data types. A list is a simple sequence of data items, without the sophisticated subsetting and array indexing features of an Array.

DODS does not support Lists of Lists.

Definition at line 42 of file List.h.

3.23.2 Constructor & Destructor Documentation

3.23.2.1 List::List (const string & *n* = "", BaseType ∗ *v* = 0)

The List constructor requires the name of the variable to be created, and the type of data the list is to hold. The name may be omitted, which will create a nameless variable. The template pointer may not be omitted.

Parameters:

n A string containing the name of the variable to be created.

v A pointer to a variable of the type to be included in the list. This variable may not itself be a List.

Definition at line 53 of file List.cc.

3.23.2.2 List::List (const List & *rhs*)

The List copy constructor.

Definition at line 58 of file List.cc.

3.23.3 Member Function Documentation

3.23.3.1 bool List::member (BaseType ∗ *btp*, DDS & *dds*) `[virtual]`

Determine whether an object is a member of the list.

Returns:

TRUE if ∗*btp* is a member of the list.

Parameters:

btp A pointer to an object.

dds The fully-initialized data descriptor object corresponding to the dataset containing the list.

Definition at line 88 of file List.cc.

References DDS::filename, Vector::length, BaseType::ops, and Vector::var.

3.23.3.2 bool List::null () [virtual]

Returns TRUE if the list is empty.

Definition at line 103 of file List.cc.

References Vector::length.

3.23.3.3 virtual bool List::read (const string & *dataset*) [pure virtual]

Reads data into a local buffer.

Put the data into a local buffer so that it may be sent to a client. This operation involves reading data from whatever source (often a local disk), and filling out the fields in the data type class. This is the heart of the DODS DAP Class operation. Much of the work of implementing a new DODS server API consists in creating the `read()` functions to read various data types.

Note that this function is only for DODS servers. It has no use on the client side of a DODS client/server connection. The DODS client and server communicate their data with `serialize()` and `deserialize()`.

This method is not implemented for the BaseType class, nor for its children. However, it should be implemented for the specialized children of those classes. For example, it is not implemented for the Float64 class, but does exist for the NCFloat64 class, specialized to read data from local netCDF files.

This method should be implemented to throw Error when it encounters an unrecoverable error.

For an example of use, see the netCDF library classes. The netCDF library is part of the DODS source distribution, and can be found under `/export/home/tomfool/DODS/src/nc-dods`.

In some sub-classes, such as Array or Grid, the `read()` function must explicitly take into account constraint information stored with the class data. For example, the Grid::read method will be called when only one component of the Grid is to be sent. Your implementation of Grid::read should check send_p() for each member of the Grid before reading that member to avoid reading data into memory that won't be sent (and thus is not needed in memory).

Returns:

The function returns a boolean value, with TRUE indicating that read() should be called again because there's more data to read, and FALSE indicating there's no more data to read. Note that this behavior is necessary to properly handle variables that contain Sequences. WRONG! (9/5/2001 jhrg) Sequences now are read in one shot. The return value of this method should always be false.

Parameters:

dataset A string naming the dataset from which the data is to be read. The meaning of this string will vary among data APIs.

See also:

BaseType

Implements Vector.

The documentation for this class was generated from the following files:

- List.h
- List.cc

3.24 rvalue Class Reference

```
#include <RValue.h>
```

Public Methods

- **rvalue** (BaseType *bt)
- **rvalue** (btp_func f, rvalue_list *a)
- **rvalue** ()
- virtual ~**rvalue** ()
- string **value_name** ()
- BaseType * bvalue (const string &dataset, DDS &dds)

3.24.1 Detailed Description

Holds the rvalues for the parser, Clause objects and evaluator.

See also:
Clause.h , DDS.h , expr.y

Definition at line 19 of file RValue.h.

3.24.2 Member Function Documentation

3.24.2.1 BaseType * rvalue::bvalue (const string & *dataset*, DDS & *dds*)

Return the BaseType * that contains a value for a given rvalue. If the rvalue is a BaseType *, ensures that the read mfunc has been called. If the rvalue is a func_rvalue, evaluates the func_rvalue and returns the result.

NB: The functions referenced by func_rvalues must encapsulate their return values in BaseType *s.

Definition at line 65 of file RValue.cc.

References BaseType::read.

Referenced by Clause::value.

The documentation for this class was generated from the following files:

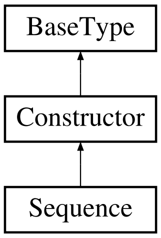
- RValue.h
- RValue.cc

3.25 Sequence Class Reference

Holds a sequence.

#include <Sequence.h>

Inheritance diagram for Sequence::



Public Methods

- Sequence (const string &n="")

  The Sequence constructor.

- Sequence (const Sequence &rhs)

  The Sequence copy constructor.

- virtual ~**Sequence** ()
- Sequence & operator= (const Sequence &rhs)
- virtual BaseType * **ptr_duplicate** ()=0
- virtual string toString ()
- virtual int element_count (bool leaves=false)

  Count the members of constructor types.

- virtual bool is_linear ()
- virtual void set_send_p (bool state)
- virtual void set_read_p (bool state)

  Sets the value of the read_p# flag.

- virtual unsigned int width ()

  Returns the size of the class instance data.

- virtual int length ()

- int **number_of_rows** ()
- virtual bool read_row (int row, const string &dataset, DDS &dds, bool ce_eval=true)
- int get_starting_row_number ()

  Get the starting row number.

- int get_row_stride ()

  Get the row stride.

- int get_ending_row_number ()

  Get the ending row number.

- void set_row_number_constraint (int start, int stop, int stride=1)
- virtual bool read (const string &dataset)=0
- virtual unsigned int val2buf (void ∗val, bool reuse=false)

  Loads class data.

- virtual unsigned int buf2val (void ∗∗val)

  Reads the class data.

- virtual BaseType ∗ var (const string &name, bool exact_match=true, btp_stack ∗s=0)

  Returns a pointer to a member of a constructor class.

- virtual BaseType ∗ var (const string &name, btp_stack &s)

  Returns a pointer to a member of a constructor class.

- BaseType ∗ var_value (size_t row, const string &name)

  Get the BaseType pointer to the named variable of a given row.

- BaseType ∗ var_value (size_t row, size_t i)

  Get the BaseType pointer to the i^{th} variable of row.

- BaseTypeRow ∗ row_value (size_t row)

  Get a whole row from the sequence.

- virtual void add_var (BaseType ∗, Part part=nil)

  Adds a variable to the Sequence.

- Pix first_var ()

  Returns an index to the first variable in a Sequence instance. This corresponds to the item in the first column of the table the Sequence represents. It is not the first row of the table.

- void next_var (Pix &p)

  Increments the Sequence instance. This returns a pointer to the next "column" in the Sequence, not the next row.

- BaseType ∗ var (Pix p)

  Returns a pointer to a Sequence member. This may be another Sequence.

- virtual void print_decl (ostream &os, string space="", bool print_semi=true, bool constraint_info=false, bool constrained=false)

  Print an ASCII representation of the variable structure.

- virtual void print_one_row (ostream &os, int row, string space, bool print_row_-num=false)
- virtual void print_val_by_rows (ostream &os, string space="", bool print_decl_-p=true, bool print_row_numners=true)

  Prints each row on its own line with a row number.

- virtual void print_val (ostream &os, string space="", bool print_decl_p=true)
- virtual void print_all_vals (ostream &os, XDR ∗src, DDS ∗dds, string space="", bool print_decl_p=true)

  Print the entire sequence.

- virtual bool check_semantics (string &msg, bool all=false)

  Compare an object's current state with the sematics of its type.

Public Attributes

- <<<<<<< Sequence.hvirtualboolserialize(conststring &dataset, DDS &dds, XDR ∗sink, boolce_eval=true);=======>>>>>> virtual boo deserialize)(XDR ∗source, DDS ∗dds, bool reuse=false)

  Deserialize (read from the network) the entire Sequence.

Protected Methods

- void **write_end_of_sequence** (XDR ∗sink)
- void **write_start_of_instance** (XDR ∗sink)
- unsigned char **read_marker** (XDR ∗source)
- bool **is_start_of_instance** (unsigned char marker)
- bool **is_end_of_sequence** (unsigned char marker)

3.25.1 Detailed Description

Holds a sequence.

This is the interface for the class Sequence. A sequence contains a single set of variables, all at the same lexical level just like a Structure. Like a Structure, a Sequence may contain other compound types, including other Sequences. Unlike a Structure, a Sequence defines a pattern that is repeated N times for a sequence of N elements. It is useful to think of a Sequence as representing a table of values (like a relational database), with each row of the table corresponding to a Sequence “instance.” (This usage can be confusing, since “instance” also refers to a particular item of class Sequence.) For example:

```
Sequence {
  String name;
  Int32 age;
} person;
```

This represents a Sequence of “person” records, each instance of which contains a name and an age:

```
Fred       34
Ralph      23
Andrea     29
...
```

A Sequence can be arbitrarily long, which is to say that its length is not part of its declaration. A Sequence can contain other Sequences:

```
Sequence {
  String name;
  Int32 age;
  Sequence {
    String friend;
  } friend_list;
} person;
```

This is still represented as a single table, but each row contains the elements of both the main Sequence and the nested one:

```
Fred       34      Norman
Fred       34      Andrea
Fred       34      Ralph
```

```
Fred        34     Lisa
Ralph       23     Norman
Ralph       23     Andrea
Ralph       23     Lisa
Ralph       23     Marth
Ralph       23     Throckmorton
Ralph       23     Helga
Ralph       23     Millicent
Andrea      29     Ralph
Andrea      29     Natasha
Andrea      29     Norman
...         ..     ...
```

Internally, the Sequence is represented by a vector of vectors. The members of the outer vector are the members of the Sequence. This includes the nested Sequences, as in the above example.

NB: Note that in the past this class had a different behavior. It held only one row at a time and the deserialize(...) method had to be called from within a loop. This is *no longer true*. Now the deserailize(...) method should be called once and will read the entire sequence's values from the server. All the values are now stored in an instance of Sequence, not just a single row's.

Because the length of a Sequence is indeterminate, there are changes to the behavior of the functions to read this class of data. The `read()` function for Sequence must be written so that successive calls return values for successive rows of the Sequence.

Similar to a C structure, you refer to members of Sequence elements with a "." notation. For example, if the Sequence has a member Sequence called "Tom" and Tom has a member Float32 called "shoe_size", you can refer to Tom's shoe size as "Tom.shoe_-size".

Definition at line 133 of file Sequence.h.

3.25.2 Constructor & Destructor Documentation

3.25.2.1 Sequence::Sequence (const string & *n* = "")

The Sequence constructor.

The Sequence constructor requires only the name of the variable to be created. The name may be omitted, which will create a nameless variable. This may be adequate for some applications.

Parameters:

n A string containing the name of the variable to be created.

Definition at line 133 of file Sequence.cc.

3.25.3 Member Function Documentation

3.25.3.1 void Sequence::add_var (BaseType * *bt*, Part *part* = nil) [virtual]

Adds a variable to the Sequence.

Remember that if you wish to add a member to a nested Sequence, you must use the `add_var()` of that Sequence. This means that variable names need not be unique among a set of nested Sequences.

Parameters:

bt A pointer to the DODS type variable to add to this Sequence.

part defaults to nil

Reimplemented from BaseType.

Definition at line 272 of file Sequence.cc.

References BaseType::ptr_duplicate, BaseType::set_parent, and BaseType::toString.

3.25.3.2 unsigned int Sequence::buf2val (void ** *val*) [virtual]

Reads the class data.

Reads the class data into the memory referenced by *val*. The caller must allocate enough storage to *val* to hold the class data. If *val* is NULL, however, memory will be allocated by this function with `new()`. Even if the memory is allocated this way, the caller is responsible for deallocating that memory. Array and List values for simple types are stored as C would store an array.

Parameters:

val A pointer to a pointer to the memory into which the class data will be copied. If the value pointed to is NULL, memory will be allocated to hold the data, and the pointer value modified accordingly. The calling program is responsible for deallocating the memory indicated by this pointer.

Returns:

The size (in bytes) of the information copied to *val*.

Implements BaseType.

Definition at line 777 of file Sequence.cc.

3.25.3.3 bool Sequence::check_semantics (string & *msg*, bool *all* = false) `[virtual]`

Compare an object's current state with the sematics of its type.

This function checks the class instance for internal consistency. This is important to check for complex constructor classes. For BaseType, an object is semantically correct if it has both a non-null name and type.

For example, an Int32 instance would return FALSE if it had no name or no type defined. A Grid instance might return FALSE for more complex reasons, such as having Map arrays of the wrong size or shape.

This function is used by the DDS class, and will rarely, if ever, be explicitly called by a DODS application program. A variable must pass this test before it is sent, but there may be many other stages in a retrieve operation where it would fail.

Returns:
Returns FALSE when the current state violates some aspect of the type semantics, TRUE otherwise.

Parameters:
- ***msg*** A returned string, containing a message indicating the source of any problem.
- ***all*** For complex constructor types (Grid, Sequence, Structure), this flag indicates whether to check the sematics of the member variables, too.

See also:
DDS::check_semantics

Reimplemented from BaseType.

Definition at line 929 of file Sequence.cc.

References BaseType::check_semantics, BaseType::name, and BaseType::type_name.

3.25.3.4 int Sequence::element_count (bool *leaves* = false) `[virtual]`

Count the members of constructor types.

Return a count of the total number of variables in this variable. This is used to count the number of variables held by a constructor variable - for simple type and vector variables it always returns 1.

For compound data types, there are two ways to count members. You can count the members, or you can count the simple members and add that to the count of the compound members. For example, if a Structure contains an Int32 and another Structure that itself contains two Int32 members, the element count of the top-level structure could be two (one Int32 and one Structure) or three (one Int32 by itself and two Int32's in the subsidiary Structure). Use the *leaves* parameter to control which kind of counting you desire.

Returns:

Returns 1 for simple types. For compound members, the count depends on the *leaves* argument.

Parameters:

leaves This parameter is only relevant if the object contains other compound data types. If FALSE, the function counts only the data variables mentioned in the object's declaration. If TRUE, it counts the simple members, and adds that to the sum of the counts for the compound members. This parameter has no effect for simple type variables.

Reimplemented from BaseType.

Definition at line 203 of file Sequence.cc.

References BaseType::element_count, first_var, next_var, and var.

Referenced by print_one_row.

3.25.3.5 int Sequence::get_ending_row_number ()

Get the ending row number.

Return the ending row number if the sequence was constrained using row numbers (instead of, or in addition to, a relational constraint). If a relational constraint was also given, the row number corresponds to the row number of the sequence *after* applying the relational constraint.

If the bracket notation was not used to constrain this sequence, this method returns -1.

Returns:

The ending row number.

Definition at line 726 of file Sequence.cc.

3.25.3.6 int Sequence::get_row_stride ()

Get the row stride.

Return the row stride number if the sequence was constrained using row numbers (instead of, or in addition to, a relational constraint). If a relational constraint was also given, the row stride is applied to the sequence *after* applying the relational constraint.

If the bracket notation was not used to constrain this sequence, this method returns -1.

Returns:
The row stride.

Definition at line 709 of file Sequence.cc.

3.25.3.7 int Sequence::get_starting_row_number ()

Get the starting row number.

Return the starting row number if the sequence was constrained using row numbers (instead of, or in addition to, a relational constraint). If a relational constraint was also given, the row number corresponds to the row number of the sequence *after* applying the relational constraint.

If the bracket notation was not used to constrain this sequence, this method returns -1.

Returns:
The starting row number.

Definition at line 693 of file Sequence.cc.

3.25.3.8 bool Sequence::is_linear () `[virtual]`

A linear sequence is any sequence that holds only simple types and/or a single linear sequence. So they can be nested sequences, but they cannot contain Arrays or Grids and they cannot contain more than one linear sequence at any given level. Structures are allowed since they can be flattened within the Sequence that holds them.

A better name for this would be is_flat or is_flatenable. 2/18/2002 jhrg

Returns:
True if the Sequence can be flattened, False otherwise.

Definition at line 216 of file Sequence.cc.

References first_var, BaseType::is_simple_type, next_var, BaseType::type, and var.

3.25.3.9 int Sequence::length () `[virtual]`

Returns the number of elements in a Sequence object. Note that this is *not* the number of items in a row, but the number of rows in the complete sequence object. To be meaningful, this must be computed after constraint expresseion (CE) evaluation. The purpose of this function is to facilitate translations between Sequence objects and Array objects, particularly when the Sequence is too large to be transferred from the server to the client in its entirety.

This function, to be useful, must be specialized for the API and data format in use.

Returns:
The base implentation returns -1, indicating that the length is not known. Subclasses specific to a particular API will have a more complete implementation.

Definition at line 478 of file Sequence.cc.

3.25.3.10 Sequence & Sequence::operator= (const Sequence & *rhs*)

Assigment. When overloading, make absolutely sure to call this version (use: dynamic_cast<Sequence &>(*this) = rhs;#).

Definition at line 174 of file Sequence.cc.

3.25.3.11 void Sequence::print_all_vals (ostream & *os*, XDR * *src*, DDS * *dds*, string *space* = "", bool *print_decl_p* = true) `[virtual]`

Print the entire sequence.

Prints a formatted version of an entire Sequence (all rows, all columns), including nested Sequences. This is meant to be used on the client side of a DODS connection, and the source of the Sequence data to be printed is specified with an XDR pointer.

This function is no longer needed, since print_val_by_rows.

Deprecated:

Parameters:

os The output stream on which to print the Sequence.

src The external source from which the data is to come. This is passed to `deserialize`.

dds The Data Descriptor Structure object corresponding to this dataset. See *The DODS User Manual* for information about this structure. This would have been received from the server in an earlier transmission.

space The leading spaces of the output.

print_decl_p If TRUE, prints the declaration of the Sequence as well as its data.

Definition at line 922 of file Sequence.cc.

References print_val.

3.25.3.12 void Sequence::print_decl (ostream & *os*, string *space* = " ", bool *print_semi* = true, bool *constraint_info* = false, bool *constrained* = false) [virtual]

Print an ASCII representation of the variable structure.

Write the variable's declaration in a C-style syntax. This function is used to create textual representation of the Data Descriptor Structure (DDS). See *The DODS User Manual* for information about this structure.

A simple array declaration might look like this:

```
Float64 lat[lat = 180];
```

While a more complex declaration (for a Grid, in this case), would look like this:

```
Grid {
ARRAY:
Int32 sst[time = 404][lat = 180][lon = 360];
MAPS:
Float64 time[time = 404];
Float64 lat[lat = 180];
Float64 lon[lon = 360];
} sst;
```

Parameters:

os The output stream on which to print the declaration.

space Each line of the declaration will begin with the characters in this string. Usually used for leading spaces.

print_semi A boolean value indicating whether to print a semicolon at the end of the declaration.

constraint_info A boolean value indicating whether constraint information is to be printed with the declaration. If the value of this parameter is TRUE, `print_decl()` prints the value of the variable's `send_p()` flag after the declaration.

constrained If this boolean value is TRUE, the variable's declaration is only printed if is the `send_p()` flag is TRUE. If a constraint expression is in place, and this variable is not requested, the `send_p()` flag is FALSE.

See also:

DDS , DDS::CE

Reimplemented from BaseType.

Definition at line 783 of file Sequence.cc.

References BaseType::name, BaseType::send_p, and BaseType::type_name.

Referenced by print_val_by_rows.

3.25.3.13 void Sequence::print_one_row (ostream & *os*, int *row*, string *space*, bool *print_row_num* = false) [virtual]

Print the i^{th} row of the sequence. This hopes in writing code that spits out sequences for testing. Sequences are now read all at once, this method does not change that.

Definition at line 810 of file Sequence.cc.

References element_count, BaseType::print_val, BaseType::type, and var_value.

Referenced by print_val_by_rows.

3.25.3.14 void Sequence::print_val (ostream & *os*, string *space* = "", bool *print_decl_p* = true) [virtual]

Print the values held by the sequence. This is used mostly for debugging.

Parameters:

os Where should the text go?

space Leading spaces, used by formatting code.

print_decl_p If True, print the variable's declaration.

Implements BaseType.

Definition at line 890 of file Sequence.cc.

References print_val_by_rows.

Referenced by print_all_vals.

3.25.3.15 void Sequence::print_val_by_rows (ostream & *os*, string *space* = "", bool *print_decl_p* = true, bool *print_row_numbers* = true) [virtual]

Prints each row on its own line with a row number.

This is a special output method for sequences. Uses `print_one_row` with `print_row_num` True.

Parameters:

os The output stream on which to print the Sequence.

space The leading spaces of the output.

print_decl_p If TRUE, prints the declaration of the Sequence as well as its data.

print_row_numbers If TRUE, prints the row numbers next to the Sequence instances (rows).

Definition at line 860 of file Sequence.cc.

References print_decl, and print_one_row.

Referenced by print_val.

3.25.3.16 virtual bool Sequence::read (const string & *dataset*) `[pure virtual]`

Reads a single instance of a Sequence.

Returns:

TRUE on success, FALSE on failure, *or* the end of the Sequence.

Implements BaseType.

Referenced by read_row.

3.25.3.17 bool Sequence::read_row (int *row*, const string & *dataset*, DDS & *dds*, bool *ce_eval* = true) `[virtual]`

Read row number *row* into the Sequence. The values of the row are obtained using the members of the sequence. This method calls the overloaded Sequence::read() method to read each row. The rows are counted using by the object (see `get_row_number()`). If a selection expression has been supplied, rows are counted only if they satisfy that expression.

Note that we can only advance. It is not possible to back up (though there is nothing theoretically against it).

Used on the server side.

NB: The first row is row number zero. A Sequence with 100 rows will have row numbers 0 to 99.

Returns:

A boolean value, with TRUE indicating that read_row should be called again because there's more data to be read. FALSE indicates the end of the Sequence.

Parameters:

row The row number to read.

dataset A string, often a file name, used to refer to t he dataset.

dds A reference to the DDS for this dataset.

ce_eval If True, evaluate any CE, otherwise do not.

Returns:

True if there are more rows to read, False if the EOF was found.

Definition at line 536 of file Sequence.cc.

References DDS::eval_selection, read, BaseType::read_p, and set_read_p.

3.25.3.18 BaseTypeRow * Sequence::row_value (size_t *row*)

Get a whole row from the sequence.

Parameters:
row Get *row* from the sequence.

Returns:
A BaseTypeRow object (vector<BaseType *>).

Definition at line 366 of file Sequence.cc.

Referenced by var_value.

3.25.3.19 void Sequence::set_read_p (bool *state*) `[virtual]`

Sets the value of the read_p# flag.

Sets the value of the `read_p` flag. This flag is TRUE when the class instance contains a valid value, and FALSE before a valid value has been read. This is meant to be called from the `read()` function. Data is ready to be sent when *both* the `_send_p` and `_read_p` flags are set to TRUE.

Parameters:
state The logical state to set the `read_p` flag.

Reimplemented from BaseType.

Definition at line 254 of file Sequence.cc.

References BaseType::set_read_p.

Referenced by read_row.

3.25.3.20 void Sequence::set_row_number_constraint (int *start*, int *stop*, int *stride* = 1)

Set the start, stop and stride for a row-number type constraint. This should be used only when the sequence is constrained using the bracket notation (which supplies start, stride and stop information). If omitted, the stride defaults to 1.

Parameters:
start The starting row number. The first row is row zero.

stop The eding row number. The 20th row is row 19.

stride The stride. A stride of two skips every other row.

Definition at line 740 of file Sequence.cc.

3.25.3.21 void Sequence::set_send_p (bool *state*) `[virtual]`

Sets the value of the `send_p` flag. This function is meant to be called from `serialize()`. Data is ready to be sent when *both* the `_send_p` and `_read_p` flags are set to TRUE.

Parameters:

state The logical state to set the `send_p` flag.

Reimplemented from BaseType.

Definition at line 245 of file Sequence.cc.

References BaseType::set_send_p.

3.25.3.22 string Sequence::toString () `[virtual]`

Write out the object's internal fields in a string. To be used for debugging when regular inspection w/ddd or gdb isn't enough.

Returns:

A string which shows the object's internal stuff.

Reimplemented from BaseType.

Definition at line 187 of file Sequence.cc.

References BaseType::toString.

3.25.3.23 unsigned int Sequence::val2buf (void ∗ *val*, bool *reuse* = false) `[virtual]`

Loads class data.

Store the value pointed to by *val* in the object's internal buffer. This function does not perform any checks, so users must be sure that the thing pointed to can actually be stored in the object's buffer. For example, an array cannot easily be fit into the data buffer for an Int32 object.

Only simple objects (Int, Float, Byte, and so on) and arrays and lists of these simple objects may be stored using this function. To put data into more complex constructor functions, use the functions provided by that class. For example, use the `var()` and `add_var()` members of the Grid class to manipulate data in that class.

Parameters:

val A pointer to the data to be inserted into the class data buffer.

reuse A boolean value, indicating whether the class internal data storage can be reused or not. If this argument is TRUE, the class buffer is assumed to be large enough to hold the incoming data, and it is *not* reallocated. If FALSE, new storage is allocated. If the internal buffer has not been allocated at all, this argument has no effect. This is currently used only in the Vector class.

Returns:

The size (in bytes) of the information copied from *val*.

See also:

Grid , Vector::val2buf

Implements BaseType.

Definition at line 771 of file Sequence.cc.

3.25.3.24 BaseType ∗ Sequence::var (const string & *name*, btp_stack & *s*) `[virtual]`

Returns a pointer to a member of a constructor class.

This version of var(...) searches for *name* and returns a pointer to the BaseType object if found. It uses the same search algorithm as above when *exact_match* is false. In addition to returning a pointer to the variable, it pushes onto *s* a BaseType pointer to each constructor type that ultimately contains *name*.

\note{The BaseType implementation always returns null. }

Deprecated:

Parameters:

name Find the variable whose name is *name*.

s Record the path to *name*.

Returns:

A pointer to the named variable.

Reimplemented from BaseType.

Definition at line 293 of file Sequence.cc.

References BaseType::name.

3.25.3.25 BaseType * Sequence::var (const string & *name*, bool *exact_match* = true, btp_stack * *s* = 0) [virtual]

Returns a pointer to a member of a constructor class.

Returns a pointer to the contained variable in a composite class. The composite classes are those made up of aggregated simple data types. Array, Grid, and Structure are composite types, while Int and Float are simple types. This function is only used by composite classes. The BaseType implementation always returns null.

Several of the subclasses overload this function with alternate access methods that make sense for that particular data type. For example, the Array class defines a `*var(int i)` method that returns the ith entry in the Array data, and the Structure provides a `*var(Pix p)` function using a pseudo-index to access the different members of the structure.

Parameters:

name The name of the class member.

exact_match True if only interested in variables whose full names match `name` exactly. If false, returns the first variable whose name matches `name`. For example, if `name` is x and point.x is a variable, then var("x", false) would return a `BaseType` pointer to point.x. If `exact_match` was `true` then `name` would need to be "point.x" for `var` to return that pointer. This feature simplifies constraint expressions for datasets which have complex, nested, constructor variables.

Returns:

A pointer to the member named in the *name* argument. If no name is given, the function returns the first (only) variable. For example, an Array has only one variable, while a Structure can have many.

Reimplemented from BaseType.

Definition at line 303 of file Sequence.cc.

References BaseType::name.

Referenced by element_count, is_linear, and width.

3.25.3.26 BaseType * Sequence::var_value (size_t *row*, size_t *i*)

Get the BaseType pointer to the i^{th} variable of *row*.

Parameters:

row Read from *row* in the sequence.

i Return the i^{th} variable from *row*.

Returns:
A BaseType which holds the variable and its value.

See also:
number_of_rows

Definition at line 402 of file Sequence.cc.

References row_value.

3.25.3.27 BaseType * Sequence::var_value (size_t *row*, const string & *name*)

Get the BaseType pointer to the named variable of a given row.

Parameters:
row Read from *row* in the sequence.

name Return *name* from *row*.

Returns:
A BaseType which holds the variable and its value.

See also:
number_of_rows

Definition at line 379 of file Sequence.cc.

References BaseType::name, and row_value.

Referenced by print_one_row.

3.25.3.28 unsigned int Sequence::width () `[virtual]`

Returns the size of the class instance data.

Return the number of bytes that are required to hold the instance's value. In the case of simple types such as Int32, this is the size of one Int32 (four bytes). For a String or Url type, `width()` returns the number of bytes needed for a `String *` variable, not the bytes needed for all the characters, since that value cannot be determined from type information alone. For Structure, and other constructor types size() returns the number of bytes needed to store pointers to the C++ objects.

Implements BaseType.

Definition at line 449 of file Sequence.cc.

References first_var, next_var, var, and BaseType::width.

3.25.4 Member Data Documentation

3.25.4.1 bool Sequence::deserialize

Deserialize (read from the network) the entire Sequence.

This method used to read a single row at a time. Now the entire sequence is read at once. The method used to return True to indicate that more data needed to be deserialized and False when the sequence was completely read. Now it simply returns true. This might seem odd, but it is because all of the other implementation of this method return true when the XDR readers succeed and False otherwise. Making this method return true breaks existing software the least.

Parameters:

source An XDR stream.

dds A DataDDS from which to read.

reuse Passed to child objects when they are deserialized.

Exceptions:

Error if a sequence stream marker cannot be read.

InternalErr if the `dds` param is not a DataDDS.

Returns:

A return value of false indicates that an EOS ("end of Sequence") marker was found, while a value of true indicates that there are more rows to be read.

Definition at line 635 of file Sequence.cc.

The documentation for this class was generated from the following files:

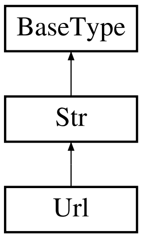
- Sequence.h
- Sequence.cc

3.26 Str Class Reference

Holds character string data.

`#include <Str.h>`

Inheritance diagram for Str::



Public Methods

- Str (const string &n="")
- virtual ~**Str** ()
- **Str** (const Str &copy_from)
- Str & **operator=** (const Str &rhs)
- virtual BaseType * **ptr_duplicate** ()=0
- virtual unsigned int width ()

  Returns the size of the class instance data.

- unsigned int **length** ()
- virtual bool serialize (const string &dataset, DDS &dds, XDR *sink, bool ce_eval=true)

  Move data to the net.

- virtual bool deserialize (XDR *source, DDS *dds, bool reuse=false)

  Receive data from the net.

- virtual bool read (const string &dataset)=0

  Reads data into a local buffer.

- virtual unsigned int val2buf (void *val, bool reuse=false)

  Loads class data.

- virtual unsigned int buf2val (void **val)

  Reads the class data.

- virtual void print_val (ostream &os, string space="", bool print_decl_p=true)

  Prints the value of the variable.

- virtual bool ops (BaseType *b, int op, const string &dataset)

  The class relational operators.

Protected Attributes

- string **_buf**

3.26.1 Detailed Description

Holds character string data.

See also:
BaseType , Url

Definition at line 51 of file Str.h.

3.26.2 Constructor & Destructor Documentation

3.26.2.1 Str::Str (const string & *n* = "")

The Str constructor requires only the name of the variable to be created. The name may be omitted, which will create a nameless variable. This may be adequate for some applications.

Parameters:
n A string containing the name of the variable to be created.

Definition at line 49 of file Str.cc.

3.26.3 Member Function Documentation

3.26.3.1 unsigned int Str::buf2val (void ** *val*) [virtual]

Reads the class data.

Reads the class data into the memory referenced by *val*. The caller must allocate enough storage to *val* to hold the class data. If *val* is NULL, however, memory will be allocated by this function with `new()`. Even if the memory is allocated this way, the caller is responsible for deallocating that memory. Array and List values for simple types are stored as C would store an array.

Parameters:

val A pointer to a pointer to the memory into which the class data will be copied. If the value pointed to is NULL, memory will be allocated to hold the data, and the pointer value modified accordingly. The calling program is responsible for deallocating the memory indicated by this pointer.

Returns:

The size (in bytes) of the information copied to *val*.

Implements BaseType.

Definition at line 134 of file Str.cc.

3.26.3.2 bool Str::deserialize (XDR * *source*, DDS * *dds*, bool *reuse* = false) [virtual]

Receive data from the net.

Receives data from the network connection identified by the `source` parameter. The data is put into the class data buffer according to the input `dds`.

This function is only used on the client side of the DODS client/server connection.

Parameters:

source A valid XDR pointer to the process connection to the net. This is generally created with a call to `new_xdrstdio()`.

dds The Data Descriptor Structure object corresponding to this dataset. See *The DODS User Manual* for information about this structure. This would have been received from the server in an earlier transmission.

reuse A boolean value, indicating whether the class internal data storage can be reused or not. If this argument is TRUE, the class buffer is assumed to be large enough to hold the incoming data, and it is *not* reallocated. If FALSE, new storage is allocated. If the internal buffer has not been allocated at all, this argument has no effect.

Returns:

Always returns TRUE.

Exceptions:

Error when a problem reading from the XDR stream is found.

See also:

DDS

Implements BaseType.

Definition at line 110 of file Str.cc.

3.26.3.3 bool Str::ops (BaseType ∗ *b*, int *op*, const string & *dataset*) [virtual]

The class relational operators.

This function contains the relational operators used by the constraint expression evaluator in the DDS class. Each class that wants to be able to evaluate relational expressions must overload this function. The implementation in BaseType returns false and prints an error message.

The *op* argument refers to a table generated by bison from the constraint expression parser. Use statements like the following to correctly interpret its value:

```
switch (op) {
case EQUAL:
return i1 == i2;
case NOT_EQUAL:
return i1 != i2;
case GREATER:
return i1 > i2;
case GREATER_EQL:
return i1 >= i2;
case LESS:
return i1 < i2;
case LESS_EQL:
return i1 <= i2;
case REGEXP:
cerr << "Regular expressions not valid for integer values" << endl;
return false;
default:
cerr << "Unknown operator" << endl;
return false;
}
```

This function is used by the constraint expression evaluator.

Parameters:

b The value with which the instance value is to be compared.

op An integer index indicating which relational operator is implied. Choose one from the following: `EQUAL`, `NOT_EQUAL`, `GREATER`, `GREATER_EQL`, `LESS`, `LESS_EQL`, and `REGEXP`.

dataset The name of the dataset from which the instance's data has come (or is to come).

Returns:
The boolean value of the comparison.

Reimplemented from BaseType.

Definition at line 181 of file Str.cc.

References BaseType::read, read, BaseType::read_p, and BaseType::type.

3.26.3.4 void Str::print_val (ostream & *os*, string *space* = "", bool *print_decl_p* = true) `[virtual]`

Prints the value of the variable.

Prints the value of the variable, with its declaration. This function is primarily intended for debugging DODS applications. However, it can be overloaded and used to do some useful things. Take a look at the asciival and writeval clients, both of which overload this to output the values of variables in different ways.

NB: This function uses C++'s iostream to handle creating the print representations of simple type variables. For floating point numbers this is set to six digits of precision by default. If you want more precision (IEEE 64-bit floats have 15 digits of precision, 32-bit floats have 8), use the setprecision() I/O manipulator.

Parameters:

os The output stream on which to print the value.

space This value is passed to the `print_decl()` function, and controls the leading spaces of the output.

print_decl_p A boolean value controlling whether the variable declaration is printed as well as the value.

Implements BaseType.

Definition at line 170 of file Str.cc.

References BaseType::print_decl.

3.26.3.5 virtual bool Str::read (const string & *dataset*) `[pure virtual]`

Reads data into a local buffer.

Put the data into a local buffer so that it may be sent to a client. This operation involves reading data from whatever source (often a local disk), and filling out the fields in the data type class. This is the heart of the DODS DAP Class operation. Much of the work of implementing a new DODS server API consists in creating the `read()` functions to read various data types.

Note that this function is only for DODS servers. It has no use on the client side of a DODS client/server connection. The DODS client and server communicate their data with `serialize()` and `deserialize()`.

This method is not implemented for the BaseType class, nor for its children. However, it should be implemented for the specialized children of those classes. For example, it is not implemented for the Float64 class, but does exist for the NCFloat64 class, specialized to read data from local netCDF files.

This method should be implemented to throw Error when it encounters an unrecoverable error.

For an example of use, see the netCDF library classes. The netCDF library is part of the DODS source distribution, and can be found under `/export/home/tomfool/DODS/src/nc-dods`.

In some sub-classes, such as Array or Grid, the `read()` function must explicitly take into account constraint information stored with the class data. For example, the Grid::read method will be called when only one component of the Grid is to be sent. Your implementation of Grid::read should check send_p() for each member of the Grid before reading that member to avoid reading data into memory that won't be sent (and thus is not needed in memory).

Returns:

The function returns a boolean value, with TRUE indicating that read() should be called again because there's more data to read, and FALSE indicating there's no more data to read. Note that this behavior is necessary to properly handle variables that contain Sequences. WRONG! (9/5/2001 jhrg) Sequences now are read in one shot. The return value of this method should always be false.

Parameters:

dataset A string naming the dataset from which the data is to be read. The meaning of this string will vary among data APIs.

See also:

BaseType

Implements BaseType.

Referenced by ops, and serialize.

3.26.3.6 bool Str::serialize (const string & *dataset*, DDS & *dds*, XDR ∗ *sink*, bool *ce_eval* = true) [virtual]

Move data to the net.

Sends the data from the indicated (local) dataset through the connection identified by the *sink* parameter. If the data is not already incorporated into the DDS object, read the data from the dataset.

This function is only used on the server side of the client/server connection, and is generally only called from the DDS::send() function. It has no BaseType implementation; each datatype child class supplies its own implementation.

Parameters:

dataset The (local) name of dataset to be read.

dds The Data Descriptor Structure object corresponding to this dataset. See *The DODS User Manual* for information about this structure.

sink A valid XDR pointer generally created with a call to `new_xdrstdio()`. This typically routes data to a TCP/IP socket.

ce_eval A boolean value indicating whether to evaluate the DODS constraint expression that may accompany this dataset. The constraint expression is stored in *dds*.

Returns:

This method always returns true. Older versions used the return value to signal success or failure.

Exceptions:

InternalErr .

Error .

See also:

DDS

Implements BaseType.

Definition at line 85 of file Str.cc.

References DDS::eval_selection, read, and BaseType::read_p.

3.26.3.7 unsigned int Str::val2buf (void ∗ *val*, bool *reuse* = false) [virtual]

Loads class data.

Store the value pointed to by *val* in the object's internal buffer. This function does not perform any checks, so users must be sure that the thing pointed to can actually be stored in the object's buffer. For example, an array cannot easily be fit into the data buffer for an Int32 object.

Only simple objects (Int, Float, Byte, and so on) and arrays and lists of these simple objects may be stored using this function. To put data into more complex constructor functions, use the functions provided by that class. For example, use the `var()` and `add_var()` members of the Grid class to manipulate data in that class.

Parameters:

val A pointer to the data to be inserted into the class data buffer.

reuse A boolean value, indicating whether the class internal data storage can be reused or not. If this argument is TRUE, the class buffer is assumed to be large enough to hold the incoming data, and it is *not* reallocated. If FALSE, new storage is allocated. If the internal buffer has not been allocated at all, this argument has no effect. This is currently used only in the Vector class.

Returns:

The size (in bytes) of the information copied from *val*.

See also:

Grid , Vector::val2buf

Implements BaseType.

Definition at line 155 of file Str.cc.

3.26.3.8 unsigned int Str::width () `[virtual]`

Returns the size of the class instance data.

Return the number of bytes that are required to hold the instance's value. In the case of simple types such as Int32, this is the size of one Int32 (four bytes). For a String or Url type, `width()` returns the number of bytes needed for a `String *` variable, not the bytes needed for all the characters, since that value cannot be determined from type information alone. For Structure, and other constructor types size() returns the number of bytes needed to store pointers to the C++ objects.

Implements BaseType.

Definition at line 79 of file Str.cc.

The documentation for this class was generated from the following files:

- Str.h
- Str.cc

3.27 StrCmp Class Template Reference

`#include <Operators.h>`

Static Public Methods

- bool **eq** (T1 v1, T2 v2)
- bool **ne** (T1 v1, T2 v2)
- bool **gr** (T1 v1, T2 v2)
- bool **ge** (T1 v1, T2 v2)
- bool **lt** (T1 v1, T2 v2)
- bool **le** (T1 v1, T2 v2)
- bool **re** (T1 v1, T2 v2)

3.27.1 Detailed Description

template<class T1, class T2> class StrCmp< T1, T2 >

Compare two string types. This class is one implementation of the comparison policy used by rops.

See also:
rops

Definition at line 109 of file Operators.h.

The documentation for this class was generated from the following file:

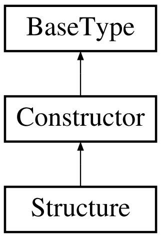
- Operators.h

3.28 Structure Class Reference

Holds a structure (aggregate) type.

`#include <Structure.h>`

Inheritance diagram for Structure::



Public Methods

- Structure (const string &n="")
- Structure (const Structure &rhs)
- virtual ~**Structure** ()
- Structure & **operator=** (const Structure &rhs)
- virtual BaseType ∗ **ptr_duplicate** ()=0
- virtual int element_count (bool leaves=false)

    Count the members of constructor types.

- virtual bool **is_linear** ()
- virtual void set_send_p (bool state)
- virtual void set_read_p (bool state)

    Sets the value of the read_p# flag.

- virtual unsigned int width ()

    Returns the size of the class instance data.

- virtual bool serialize (const string &dataset, DDS &dds, XDR ∗sink, bool ce_eval=true)

    Move data to the net.

- virtual bool deserialize (XDR ∗source, DDS ∗dds, bool reuse=false)

    Receive data from the net.

- virtual bool read (const string &dataset)=0

    Reads data into a local buffer.

- virtual unsigned int val2buf (void ∗val, bool reuse=false)
- virtual unsigned int buf2val (void ∗∗val)
- virtual BaseType ∗ var (const string &name, bool exact_match=true, btp_stack ∗s=0)

    Returns a pointer to a member of a constructor class.

- virtual BaseType ∗ var (const string &name, btp_stack &s)

    Returns a pointer to a member of a constructor class.

- virtual void add_var (BaseType ∗bt, Part part=nil)
- Pix first_var ()
- void next_var (Pix &p)
- BaseType ∗ var (Pix p)
- virtual void print_decl (ostream &os, string space="", bool print_semi=true, bool constraint_info=false, bool constrained=false)

    Print an ASCII representation of the variable structure.

- virtual void print_val (ostream &os, string space="", bool print_decl_p=true)

    Prints the value of the variable.

- virtual void print_all_vals (ostream &os, XDR ∗src, DDS ∗dds, string space="", bool print_decl_p=true)
- virtual bool check_semantics (string &msg, bool all=false)

    Compare an object's current state with the sematics of its type.

3.28.1 Detailed Description

Holds a structure (aggregate) type.

This data type is used to hold a collection of related data types, in a manner roughly corresponding to a C structure. The member types can be simple or compound types, and can include other Structures.

The DODS structure is defined as a singly-linked list. This means that Structure elements can be accessed either by name, with the `var()` function, or by their position in the list, either with the overloaded version of `var()`, or the combination of the `first_var()` and `next_var()` functions.

The `val2buf()` and `buf2val()` functions only return the size of the structure. To read parts of a DODS Structure into an application program, use the `buf2val()` function of the element of the Structure in question.

Note that the predicate-setting functions `set_send_p()` and `set_read_p()` set their flags for the Structure as well as for each of the Structure's member elements.

Similar to C, you can refer to members of Structure elements with a "." notation. For example, if the Structure has a member Structure called "Tom" and Tom has a member Float32 called "shoe_size", then you can refer to Tom's shoe size as "Tom.shoe_size".

Definition at line 89 of file Structure.h.

3.28.2 Constructor & Destructor Documentation

3.28.2.1 Structure::Structure (const string & *n* = "")

The Structure constructor requires only the name of the variable to be created. The name may be omitted, which will create a nameless variable. This may be adequate for some applications.

Parameters:

n A string containing the name of the variable to be created.

Definition at line 67 of file Structure.cc.

3.28.2.2 Structure::Structure (const Structure & *rhs*)

The Structure copy constructor.

Definition at line 72 of file Structure.cc.

3.28.3 Member Function Documentation

3.28.3.1 void Structure::add_var (BaseType ∗ *bt*, Part *part* = nil) `[virtual]`

Adds an element to a Structure.

Parameters:

bt A pointer to the DODS type variable to add to this Structure.

part defaults to nil

Reimplemented from BaseType.

Definition at line 153 of file Structure.cc.

References BaseType::ptr_duplicate, and BaseType::set_parent.

3.28.3.2 unsigned int Structure::buf2val (void ** *val*) `[virtual]`

Returns the size of the structure.

Implements BaseType.

Definition at line 223 of file Structure.cc.

3.28.3.3 bool Structure::check_semantics (string & *msg*, bool *all* = false) `[virtual]`

Compare an object's current state with the sematics of its type.

This function checks the class instance for internal consistency. This is important to check for complex constructor classes. For BaseType, an object is semantically correct if it has both a non-null name and type.

For example, an Int32 instance would return FALSE if it had no name or no type defined. A Grid instance might return FALSE for more complex reasons, such as having Map arrays of the wrong size or shape.

This function is used by the DDS class, and will rarely, if ever, be explicitly called by a DODS application program. A variable must pass this test before it is sent, but there may be many other stages in a retrieve operation where it would fail.

Returns:

Returns FALSE when the current state violates some aspect of the type semantics, TRUE otherwise.

Parameters:

msg A returned string, containing a message indicating the source of any problem.

all For complex constructor types (Grid, Sequence, Structure), this flag indicates whether to check the sematics of the member variables, too.

See also:

DDS::check_semantics

Reimplemented from BaseType.

Definition at line 450 of file Structure.cc.

References BaseType::check_semantics, BaseType::name, and BaseType::type_name.

3.28.3.4 bool Structure::deserialize (XDR * *source*, DDS * *dds*, bool *reuse* = false) `[virtual]`

Receive data from the net.

Receives data from the network connection identified by the `source` parameter. The data is put into the class data buffer according to the input `dds`.

This function is only used on the client side of the DODS client/server connection.

Parameters:

source A valid XDR pointer to the process connection to the net. This is generally created with a call to `new_xdrstdio()`.

dds The Data Descriptor Structure object corresponding to this dataset. See *The DODS User Manual* for information about this structure. This would have been received from the server in an earlier transmission.

reuse A boolean value, indicating whether the class internal data storage can be reused or not. If this argument is TRUE, the class buffer is assumed to be large enough to hold the incoming data, and it is *not* reallocated. If FALSE, new storage is allocated. If the internal buffer has not been allocated at all, this argument has no effect.

Returns:

Always returns TRUE.

Exceptions:

Error when a problem reading from the XDR stream is found.

See also:

DDS

Implements BaseType.

Definition at line 204 of file Structure.cc.

References BaseType::deserialize, first_var, next_var, and var.

3.28.3.5 int Structure::element_count (bool *leaves* = false) `[virtual]`

Count the members of constructor types.

Return a count of the total number of variables in this variable. This is used to count the number of variables held by a constructor variable - for simple type and vector variables it always returns 1.

For compound data types, there are two ways to count members. You can count the members, or you can count the simple members and add that to the count of the compound members. For example, if a Structure contains an Int32 and another Structure that itself contains two Int32 members, the element count of the top-level structure could be two (one Int32 and one Structure) or three (one Int32 by itself and two Int32's in the subsidiary Structure). Use the *leaves* parameter to control which kind of counting you desire.

Returns:

Returns 1 for simple types. For compound members, the count depends on the *leaves* argument.

Parameters:

leaves This parameter is only relevant if the object contains other compound data types. If FALSE, the function counts only the data variables mentioned in the object's declaration. If TRUE, it counts the simple members, and adds that to the sum of the counts for the compound members. This parameter has no effect for simple type variables.

Reimplemented from BaseType.

Definition at line 98 of file Structure.cc.

References BaseType::element_count, first_var, next_var, and var.

3.28.3.6 Pix Structure::first_var ()

Returns the pseudo-index (Pix) of the first structure element.

Definition at line 322 of file Structure.cc.

Referenced by deserialize, element_count, print_all_vals, serialize, and width.

3.28.3.7 void Structure::next_var (Pix & *p*)

Increments the input index to point to the next element in the structure.

Definition at line 333 of file Structure.cc.

Referenced by deserialize, element_count, print_all_vals, serialize, and width.

3.28.3.8 void Structure::print_all_vals (ostream & *os*, XDR ∗ *src*, DDS ∗ *dds*, string *space* = "", bool *print_decl_p* = true) `[virtual]`

Prints the Structure and all elements of any Sequences contained within.

See also:

Sequence::print_all_vals

Definition at line 406 of file Structure.cc.

References BaseType::deserialize, first_var, next_var, print_decl, BaseType::print_val, print_val, BaseType::type, and var.

3.28.3.9 void Structure::print_decl (ostream & *os*, string *space* = " ", bool *print_semi* = true, bool *constraint_info* = false, bool *constrained* = false) [virtual]

Print an ASCII representation of the variable structure.

Write the variable's declaration in a C-style syntax. This function is used to create textual representation of the Data Descriptor Structure (DDS). See *The DODS User Manual* for information about this structure.

A simple array declaration might look like this:

```
Float64 lat[lat = 180];
```

While a more complex declaration (for a Grid, in this case), would look like this:

```
Grid {
ARRAY:
Int32 sst[time = 404][lat = 180][lon = 360];
MAPS:
Float64 time[time = 404];
Float64 lat[lat = 180];
Float64 lon[lon = 360];
} sst;
```

Parameters:

os The output stream on which to print the declaration.

space Each line of the declaration will begin with the characters in this string. Usually used for leading spaces.

print_semi A boolean value indicating whether to print a semicolon at the end of the declaration.

constraint_info A boolean value indicating whether constraint information is to be printed with the declaration. If the value of this parameter is TRUE, `print_decl()` prints the value of the variable's `send_p()` flag after the declaration.

constrained If this boolean value is TRUE, the variable's declaration is only printed if is the `send_p()` flag is TRUE. If a constraint expression is in place, and this variable is not requested, the `send_p()` flag is FALSE.

See also:

DDS , DDS::CE

Reimplemented from BaseType.

Definition at line 351 of file Structure.cc.

References BaseType::name, BaseType::send_p, and BaseType::type_name.

Referenced by print_all_vals, and print_val.

3.28.3.10 void Structure::print_val (ostream & *os*, string *space* = "", bool *print_decl_p* = true) `[virtual]`

Prints the value of the variable.

Prints the value of the variable, with its declaration. This function is primarily intended for debugging DODS applications. However, it can be overloaded and used to do some useful things. Take a look at the asciival and writeval clients, both of which overload this to output the values of variables in different ways.

NB: This function uses C++'s iostream to handle creating the print representations of simple type variables. For floating point numbers this is set to six digits of precision by default. If you want more precision (IEEE 64-bit floats have 15 digits of precision, 32-bit floats have 8), use the setprecision() I/O manipulator.

Parameters:

os The output stream on which to print the value.

space This value is passed to the `print_decl()` function, and controls the leading spaces of the output.

print_decl_p A boolean value controlling whether the variable declaration is printed as well as the value.

Implements BaseType.

Definition at line 378 of file Structure.cc.

References print_decl.

Referenced by print_all_vals.

3.28.3.11 virtual bool Structure::read (const string & *dataset*) `[pure virtual]`

Reads data into a local buffer.

Put the data into a local buffer so that it may be sent to a client. This operation involves reading data from whatever source (often a local disk), and filling out the fields in the data type class. This is the heart of the DODS DAP Class operation. Much of the work of implementing a new DODS server API consists in creating the `read()` functions to read various data types.

Note that this function is only for DODS servers. It has no use on the client side of a DODS client/server connection. The DODS client and server communicate their data with `serialize()` and `deserialize()`.

This method is not implemented for the BaseType class, nor for its children. However, it should be implemented for the specialized children of those classes. For example, it is not implemented for the Float64 class, but does exist for the NCFloat64 class, specialized to read data from local netCDF files.

This method should be implemented to throw Error when it encounters an unrecoverable error.

For an example of use, see the netCDF library classes. The netCDF library is part of the DODS source distribution, and can be found under `/export/home/tomfool/DODS/src/nc-dods`.

In some sub-classes, such as Array or Grid, the `read()` function must explicitly take into account constraint information stored with the class data. For example, the Grid::read method will be called when only one component of the Grid is to be sent. Your implementation of Grid::read should check send_p() for each member of the Grid before reading that member to avoid reading data into memory that won't be sent (and thus is not needed in memory).

Returns:

The function returns a boolean value, with TRUE indicating that read() should be called again because there's more data to read, and FALSE indicating there's no more data to read. Note that this behavior is necessary to properly handle variables that contain Sequences. WRONG! (9/5/2001 jhrg) Sequences now are read in one shot. The return value of this method should always be false.

Parameters:

dataset A string naming the dataset from which the data is to be read. The meaning of this string will vary among data APIs.

See also:

BaseType

Implements BaseType.

Referenced by serialize.

3.28.3.12 bool Structure::serialize (const string & *dataset*, DDS & *dds*, XDR * *sink*, bool *ce_eval* = true) `[virtual]`

Move data to the net.

Sends the data from the indicated (local) dataset through the connection identified by the *sink* parameter. If the data is not already incorporated into the DDS object, read the data from the dataset.

This function is only used on the server side of the client/server connection, and is generally only called from the DDS::send() function. It has no BaseType implementation; each datatype child class supplies its own implementation.

Parameters:

dataset The (local) name of dataset to be read.

dds The Data Descriptor Structure object corresponding to this dataset. See *The DODS User Manual* for information about this structure.

sink A valid XDR pointer generally created with a call to `new_xdrstdio()`. This typically routes data to a TCP/IP socket.

ce_eval A boolean value indicating whether to evaluate the DODS constraint expression that may accompany this dataset. The constraint expression is stored in *dds*.

Returns:

This method always returns true. Older versions used the return value to signal success or failure.

Exceptions:

InternalErr .

Error .

See also:

DDS

Implements BaseType.

Definition at line 187 of file Structure.cc.

References DDS::eval_selection, first_var, next_var, read, BaseType::read_p, BaseType::send_p, BaseType::serialize, and var.

3.28.3.13 void Structure::set_read_p (bool *state*) `[virtual]`

Sets the value of the read_p# flag.

Sets the value of the `read_p` flag. This flag is TRUE when the class instance contains a valid value, and FALSE before a valid value has been read. This is meant to be called from the `read()` function. Data is ready to be sent when *both* the `_send_p` and `_read_p` flags are set to TRUE.

Parameters:

state The logical state to set the `read_p` flag.

Reimplemented from BaseType.

Definition at line 136 of file Structure.cc.

References BaseType::set_read_p.

3.28.3.14 void Structure::set_send_p (bool *state*) `[virtual]`

Sets the value of the `send_p` flag. This function is meant to be called from `serialize()`. Data is ready to be sent when *both* the `_send_p` and `_read_p` flags are set to TRUE.

Parameters:

state The logical state to set the `send_p` flag.

Reimplemented from BaseType.

Definition at line 125 of file Structure.cc.

References BaseType::set_send_p.

3.28.3.15 unsigned int Structure::val2buf (void * *val*, bool *reuse* = false) `[virtual]`

This function assumes that val contains values for all the elements of the structure in the order those elements are declared.

Returns:

Returns the size of the structure.

Implements BaseType.

Definition at line 216 of file Structure.cc.

3.28.3.16 BaseType * Structure::var (Pix *p*)

Returns a pointer to the *p*th element.

Definition at line 342 of file Structure.cc.

3.28.3.17 BaseType * Structure::var (const string & *name*, btp_stack & *s*) `[virtual]`

Returns a pointer to a member of a constructor class.

This version of var(...) searches for *name* and returns a pointer to the BaseType object if found. It uses the same search algorithm as above when *exact_match* is false. In addition to returning a pointer to the variable, it pushes onto *s* a BaseType pointer to each constructor type that ultimately contains *name*.

\note{The BaseType implementation always returns null. }

Deprecated:

Parameters:

name Find the variable whose name is *name*.

s Record the path to *name*.

Returns:

A pointer to the named variable.

Reimplemented from BaseType.

Definition at line 244 of file Structure.cc.

References BaseType::name.

3.28.3.18 BaseType ∗ Structure::var (const string & *name*, bool *exact* = true, btp_stack ∗ *s* = 0) `[virtual]`

Returns a pointer to a member of a constructor class.

Returns a pointer to the contained variable in a composite class. The composite classes are those made up of aggregated simple data types. Array, Grid, and Structure are composite types, while Int and Float are simple types. This function is only used by composite classes. The BaseType implementation always returns null.

Several of the subclasses overload this function with alternate access methods that make sense for that particular data type. For example, the Array class defines a `*var(int i)` method that returns the ith entry in the Array data, and the Structure provides a `*var(Pix p)` function using a pseudo-index to access the different members of the structure.

Parameters:

name The name of the class member.

exact_match True if only interested in variables whose full names match `name` exactly. If false, returns the first variable whose name matches `name`. For example, if `name` is x and point.x is a variable, then var("x", false) would return a `BaseType` pointer to point.x. If `exact_match` was `true` then `name` would need to be "point.x" for `var` to return that pointer. This feature simplifies constraint expressions for datasets which have complex, nested, constructor variables.

Returns:

A pointer to the member named in the *name* argument. If no name is given, the function returns the first (only) variable. For example, an Array has only one variable, while a Structure can have many.

Reimplemented from BaseType.

Definition at line 233 of file Structure.cc.

References BaseType::name.

Referenced by deserialize, element_count, print_all_vals, serialize, and width.

3.28.3.19 unsigned int Structure::width () [virtual]

Returns the size of the class instance data.

Return the number of bytes that are required to hold the instance's value. In the case of simple types such as Int32, this is the size of one Int32 (four bytes). For a String or Url type, `width()` returns the number of bytes needed for a `String *` variable, not the bytes needed for all the characters, since that value cannot be determined from type information alone. For Structure, and other constructor types size() returns the number of bytes needed to store pointers to the C++ objects.

Implements BaseType.

Definition at line 172 of file Structure.cc.

References first_var, next_var, var, and BaseType::width.

The documentation for this class was generated from the following files:

- Structure.h
- Structure.cc

3.29 SUCmp Class Template Reference

`#include <Operators.h>`

Static Public Methods

- bool **eq** (T1 v1, UT2 v2)
- bool **ne** (T1 v1, UT2 v2)
- bool **gr** (T1 v1, UT2 v2)
- bool **ge** (T1 v1, UT2 v2)
- bool **lt** (T1 v1, UT2 v2)
- bool **le** (T1 v1, UT2 v2)
- bool **re** (T1 v1, UT2 v2)

3.29.1 Detailed Description

template<class T1, class UT2> class SUCmp< T1, UT2 >

Compare two numerical types, the first one signed and the second unsigned. If the signed argument is negative, zero is used in the comparison. This class is one implementation of the comparison policy used by rops. This class is here to make writing the Byte::ops, ... member functions simpler. It is not necessary since the functions could twiddle the order of arguments to rops and use `USCmp`. Having this class make Byte:ops, ... simper to read and write.

See also:
Byte::ops , USCmp , Cmp , ops

Definition at line 90 of file Operators.h.

The documentation for this class was generated from the following file:

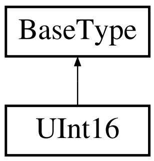
- Operators.h

3.30 UInt16 Class Reference

Holds an unisigned 16-bit integer.

`#include <UInt16.h>`

Inheritance diagram for UInt16::



Public Methods

- **UInt16** (const string &n="")
- virtual ~**UInt16** ()
- **UInt16** (const UInt16 &copy_from)
- UInt16 & **operator=** (const UInt16 &rhs)
- virtual BaseType * **ptr_duplicate** ()=0
- virtual unsigned int width ()

  Returns the size of the class instance data.

- virtual bool serialize (const string &dataset, DDS &dds, XDR *sink, bool ce_eval=true)

  Move data to the net.

- virtual bool deserialize (XDR *source, DDS *dds, bool reuse=false)

  Receive data from the net.

- virtual bool read (const string &dataset)=0

  Reads data into a local buffer.

- virtual unsigned int val2buf (void *val, bool reuse=false)

  Loads class data.

- virtual unsigned int buf2val (void **val)

  Reads the class data.

- virtual void print_val (ostream &os, string space="", bool print_decl_p=true)

  Prints the value of the variable.

- virtual bool ops (BaseType *b, int op, const string &dataset)

  The class relational operators.

Protected Attributes

- dods_uint16 **_buf**

Friends

- class Byte
- class Int16
- class Int32
- class UInt32
- class Float32
- class Float64

3.30.1 Detailed Description

Holds an unisigned 16-bit integer.

Definition at line 44 of file UInt16.h.

3.30.2 Member Function Documentation

3.30.2.1 unsigned int UInt16::buf2val (void ** *val*) `[virtual]`

Reads the class data.

Reads the class data into the memory referenced by *val*. The caller must allocate enough storage to *val* to hold the class data. If *val* is NULL, however, memory will be allocated by this function with `new()`. Even if the memory is allocated this way, the caller is responsible for deallocating that memory. Array and List values for simple types are stored as C would store an array.

Parameters:

val A pointer to a pointer to the memory into which the class data will be copied. If the value pointed to is NULL, memory will be allocated to hold the data, and the pointer value modified accordingly. The calling program is responsible for deallocating the memory indicated by this pointer.

Returns:

The size (in bytes) of the information copied to *val*.

Implements BaseType.

Definition at line 119 of file UInt16.cc.

References width.

3.30.2.2 bool UInt16::deserialize (XDR ∗ *source*, DDS ∗ *dds*, bool *reuse* = false) `[virtual]`

Receive data from the net.

Receives data from the network connection identified by the `source` parameter. The data is put into the class data buffer according to the input `dds`.

This function is only used on the client side of the DODS client/server connection.

Parameters:

source A valid XDR pointer to the process connection to the net. This is generally created with a call to `new_xdrstdio()`.

dds The Data Descriptor Structure object corresponding to this dataset. See *The DODS User Manual* for information about this structure. This would have been received from the server in an earlier transmission.

reuse A boolean value, indicating whether the class internal data storage can be reused or not. If this argument is TRUE, the class buffer is assumed to be large enough to hold the incoming data, and it is *not* reallocated. If FALSE, new storage is allocated. If the internal buffer has not been allocated at all, this argument has no effect.

Returns:

Always returns TRUE.

Exceptions:

Error when a problem reading from the XDR stream is found.

See also:

DDS

Implements BaseType.

Definition at line 92 of file UInt16.cc.

3.30.2.3 bool UInt16::ops (BaseType * *b*, int *op*, const string & *dataset*) [virtual]

The class relational operators.

This function contains the relational operators used by the constraint expression evaluator in the DDS class. Each class that wants to be able to evaluate relational expressions must overload this function. The implementation in BaseType returns false and prints an error message.

The *op* argument refers to a table generated by bison from the constraint expression parser. Use statements like the following to correctly interpret its value:

```
switch (op) {
case EQUAL:
return i1 == i2;
case NOT_EQUAL:
return i1 != i2;
case GREATER:
return i1 > i2;
case GREATER_EQL:
return i1 >= i2;
case LESS:
return i1 < i2;
case LESS_EQL:
return i1 <= i2;
case REGEXP:
cerr << "Regular expressions not valid for integer values" << endl;
return false;
default:
cerr << "Unknown operator" << endl;
return false;
}
```

This function is used by the constraint expression evaluator.

Parameters:

b The value with which the instance value is to be compared.

op An integer index indicating which relational operator is implied. Choose one from the following: `EQUAL`, `NOT_EQUAL`, `GREATER`, `GREATER_EQL`, `LESS`, `LESS_EQL`, and `REGEXP`.

dataset The name of the dataset from which the instance's data has come (or is to come).

Returns:

The boolean value of the comparison.

Reimplemented from BaseType.

Definition at line 148 of file UInt16.cc.

References BaseType::read, read, BaseType::read_p, and BaseType::type.

3.30.2.4 void UInt16::print_val (ostream & *os*, string *space* = "", bool *print_decl_p* = true) [virtual]

Prints the value of the variable.

Prints the value of the variable, with its declaration. This function is primarily intended for debugging DODS applications. However, it can be overloaded and used to do some useful things. Take a look at the asciival and writeval clients, both of which overload this to output the values of variables in different ways.

NB: This function uses C++'s iostream to handle creating the print representations of simple type variables. For floating point numbers this is set to six digits of precision by default. If you want more precision (IEEE 64-bit floats have 15 digits of precision, 32-bit floats have 8), use the setprecision() I/O manipulator.

Parameters:

os The output stream on which to print the value.

space This value is passed to the `print_decl()` function, and controls the leading spaces of the output.

print_decl_p A boolean value controlling whether the variable declaration is printed as well as the value.

Implements BaseType.

Definition at line 137 of file UInt16.cc.

References BaseType::print_decl.

3.30.2.5 virtual bool UInt16::read (const string & *dataset*) [pure virtual]

Reads data into a local buffer.

Put the data into a local buffer so that it may be sent to a client. This operation involves reading data from whatever source (often a local disk), and filling out the fields in the data type class. This is the heart of the DODS DAP Class operation. Much of the work of implementing a new DODS server API consists in creating the `read()` functions to read various data types.

Note that this function is only for DODS servers. It has no use on the client side of a DODS client/server connection. The DODS client and server communicate their data with `serialize()` and `deserialize()`.

This method is not implemented for the BaseType class, nor for its children. However, it should be implemented for the specialized children of those classes. For example,

it is not implemented for the Float64 class, but does exist for the NCFloat64 class, specialized to read data from local netCDF files.

This method should be implemented to throw Error when it encounters an unrecoverable error.

For an example of use, see the netCDF library classes. The netCDF library is part of the DODS source distribution, and can be found under `/export/home/tomfool/DODS/src/nc-dods`.

In some sub-classes, such as Array or Grid, the `read()` function must explicitly take into account constraint information stored with the class data. For example, the Grid::read method will be called when only one component of the Grid is to be sent. Your implementation of Grid::read should check send_p() for each member of the Grid before reading that member to avoid reading data into memory that won't be sent (and thus is not needed in memory).

Returns:

The function returns a boolean value, with TRUE indicating that read() should be called again because there's more data to read, and FALSE indicating there's no more data to read. Note that this behavior is necessary to properly handle variables that contain Sequences. WRONG! (9/5/2001 jhrg) Sequences now are read in one shot. The return value of this method should always be false.

Parameters:

dataset A string naming the dataset from which the data is to be read. The meaning of this string will vary among data APIs.

See also:

BaseType

Implements BaseType.

Referenced by ops, and serialize.

3.30.2.6 bool UInt16::serialize (const string & *dataset*, DDS & *dds*, XDR * *sink*, bool *ce_eval* = true) `[virtual]`

Move data to the net.

Sends the data from the indicated (local) dataset through the connection identified by the *sink* parameter. If the data is not already incorporated into the DDS object, read the data from the dataset.

This function is only used on the server side of the client/server connection, and is generally only called from the DDS::send() function. It has no BaseType implementation; each datatype child class supplies its own implementation.

Parameters:

dataset The (local) name of dataset to be read.

dds The Data Descriptor Structure object corresponding to this dataset. See *The DODS User Manual* for information about this structure.

sink A valid XDR pointer generally created with a call to `new_xdrstdio()`. This typically routes data to a TCP/IP socket.

ce_eval A boolean value indicating whether to evaluate the DODS constraint expression that may accompany this dataset. The constraint expression is stored in *dds*.

Returns:

This method always returns true. Older versions used the return value to signal success or failure.

Exceptions:

InternalErr .

Error .

See also:

DDS

Implements BaseType.

Definition at line 74 of file UInt16.cc.

References DDS::eval_selection, read, and BaseType::read_p.

3.30.2.7 unsigned int UInt16::val2buf (void * *val*, bool *reuse* = false) `[virtual]`

Loads class data.

Store the value pointed to by *val* in the object's internal buffer. This function does not perform any checks, so users must be sure that the thing pointed to can actually be stored in the object's buffer. For example, an array cannot easily be fit into the data buffer for an Int32 object.

Only simple objects (Int, Float, Byte, and so on) and arrays and lists of these simple objects may be stored using this function. To put data into more complex constructor functions, use the functions provided by that class. For example, use the `var()` and `add_var()` members of the Grid class to manipulate data in that class.

Parameters:

val A pointer to the data to be inserted into the class data buffer.

reuse A boolean value, indicating whether the class internal data storage can be reused or not. If this argument is TRUE, the class buffer is assumed to be

large enough to hold the incoming data, and it is *not* reallocated. If FALSE, new storage is allocated. If the internal buffer has not been allocated at all, this argument has no effect. This is currently used only in the Vector class.

Returns:
The size (in bytes) of the information copied from *val*.

See also:
Grid , Vector::val2buf

Implements BaseType.

Definition at line 103 of file UInt16.cc.

References width.

3.30.2.8 unsigned int UInt16::width () `[virtual]`

Returns the size of the class instance data.

Return the number of bytes that are required to hold the instance's value. In the case of simple types such as Int32, this is the size of one Int32 (four bytes). For a String or Url type, `width()` returns the number of bytes needed for a `String *` variable, not the bytes needed for all the characters, since that value cannot be determined from type information alone. For Structure, and other constructor types size() returns the number of bytes needed to store pointers to the C++ objects.

Implements BaseType.

Definition at line 68 of file UInt16.cc.

Referenced by buf2val, and val2buf.

3.30.3 Friends And Related Function Documentation

3.30.3.1 friend class Byte `[friend]`

This class allows Byte, ..., Float64 access to `_buf` to simplify and speed up the relational operators.

NB: According to Stroustrup it does not matter where (public, private or protected) friend classes are declared.

Definition at line 50 of file UInt16.h.

The documentation for this class was generated from the following files:

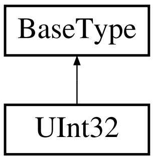
- UInt16.h
- UInt16.cc

3.31 UInt32 Class Reference

Holds a 32-bit unsigned integer.

`#include <UInt32.h>`

Inheritance diagram for UInt32::



Public Methods

- **UInt32** (const string &n="")
- virtual ~**UInt32** ()
- **UInt32** (const UInt32 &copy_from)
- UInt32 & **operator=** (const UInt32 &rhs)
- virtual BaseType * **ptr_duplicate** ()=0
- virtual unsigned int width ()

  Returns the size of the class instance data.

- virtual bool serialize (const string &dataset, DDS &dds, XDR *sink, bool ce_-eval=true)

  Move data to the net.

- virtual bool deserialize (XDR *source, DDS *dds, bool reuse=false)

  Receive data from the net.

- virtual bool read (const string &dataset)=0

  Reads data into a local buffer.

- virtual unsigned int val2buf (void *val, bool reuse=false)

  Loads class data.

- virtual unsigned int buf2val (void **val)

  Reads the class data.

- virtual void print_val (ostream &os, string space="", bool print_decl_p=true)

    Prints the value of the variable.

- virtual bool ops (BaseType *b, int op, const string &dataset)

    The class relational operators.

Protected Attributes

- dods_uint32 **_buf**

Friends

- class Byte
- class Int16
- class UInt16
- class Int32
- class Float32
- class Float64

3.31.1 Detailed Description

Holds a 32-bit unsigned integer.

See also:
BaseType

Definition at line 45 of file UInt32.h.

3.31.2 Member Function Documentation

3.31.2.1 unsigned int UInt32::buf2val (void ** *val*) `[virtual]`

Reads the class data.

Reads the class data into the memory referenced by *val*. The caller must allocate enough storage to *val* to hold the class data. If *val* is NULL, however, memory will be allocated by this function with `new()`. Even if the memory is allocated this way, the caller is responsible for deallocating that memory. Array and List values for simple types are stored as C would store an array.

Parameters:

val A pointer to a pointer to the memory into which the class data will be copied. If the value pointed to is NULL, memory will be allocated to hold the data, and the pointer value modified accordingly. The calling program is responsible for deallocating the memory indicated by this pointer.

Returns:

The size (in bytes) of the information copied to *val*.

Implements BaseType.

Definition at line 116 of file UInt32.cc.

References width.

3.31.2.2 bool UInt32::deserialize (XDR ∗ *source*, DDS ∗ *dds*, bool *reuse* = false) `[virtual]`

Receive data from the net.

Receives data from the network connection identified by the `source` parameter. The data is put into the class data buffer according to the input `dds`.

This function is only used on the client side of the DODS client/server connection.

Parameters:

source A valid XDR pointer to the process connection to the net. This is generally created with a call to `new_xdrstdio()`.

dds The Data Descriptor Structure object corresponding to this dataset. See *The DODS User Manual* for information about this structure. This would have been received from the server in an earlier transmission.

reuse A boolean value, indicating whether the class internal data storage can be reused or not. If this argument is TRUE, the class buffer is assumed to be large enough to hold the incoming data, and it is *not* reallocated. If FALSE, new storage is allocated. If the internal buffer has not been allocated at all, this argument has no effect.

Returns:

Always returns TRUE.

Exceptions:

Error when a problem reading from the XDR stream is found.

See also:

DDS

Implements BaseType.

Definition at line 88 of file UInt32.cc.

3.31.2.3 bool UInt32::ops (BaseType * *b*, int *op*, const string & *dataset*) [virtual]

The class relational operators.

This function contains the relational operators used by the constraint expression evaluator in the DDS class. Each class that wants to be able to evaluate relational expressions must overload this function. The implementation in BaseType returns false and prints an error message.

The *op* argument refers to a table generated by bison from the constraint expression parser. Use statements like the following to correctly interpret its value:

```
switch (op) {
case EQUAL:
return i1 == i2;
case NOT_EQUAL:
return i1 != i2;
case GREATER:
return i1 > i2;
case GREATER_EQL:
return i1 >= i2;
case LESS:
return i1 < i2;
case LESS_EQL:
return i1 <= i2;
case REGEXP:
cerr << "Regular expressions not valid for integer values" << endl;
return false;
default:
cerr << "Unknown operator" << endl;
return false;
}
```

This function is used by the constraint expression evaluator.

Parameters:

b The value with which the instance value is to be compared.

op An integer index indicating which relational operator is implied. Choose one from the following: `EQUAL`, `NOT_EQUAL`, `GREATER`, `GREATER_EQL`, `LESS`, `LESS_EQL`, and `REGEXP`.

dataset The name of the dataset from which the instance's data has come (or is to come).

Returns:

The boolean value of the comparison.

Reimplemented from BaseType.

Definition at line 145 of file UInt32.cc.

References BaseType::read, read, BaseType::read_p, and BaseType::type.

3.31.2.4 void UInt32::print_val (ostream & *os*, string *space* = "", bool *print_decl_p* = true) `[virtual]`

Prints the value of the variable.

Prints the value of the variable, with its declaration. This function is primarily intended for debugging DODS applications. However, it can be overloaded and used to do some useful things. Take a look at the asciival and writeval clients, both of which overload this to output the values of variables in different ways.

NB: This function uses C++'s iostream to handle creating the print representations of simple type variables. For floating point numbers this is set to six digits of precision by default. If you want more precision (IEEE 64-bit floats have 15 digits of precision, 32-bit floats have 8), use the setprecision() I/O manipulator.

Parameters:

os The output stream on which to print the value.

space This value is passed to the `print_decl()` function, and controls the leading spaces of the output.

print_decl_p A boolean value controlling whether the variable declaration is printed as well as the value.

Implements BaseType.

Definition at line 134 of file UInt32.cc.

References BaseType::print_decl.

3.31.2.5 virtual bool UInt32::read (const string & *dataset*) `[pure virtual]`

Reads data into a local buffer.

Put the data into a local buffer so that it may be sent to a client. This operation involves reading data from whatever source (often a local disk), and filling out the fields in the data type class. This is the heart of the DODS DAP Class operation. Much of the work of implementing a new DODS server API consists in creating the `read()` functions to read various data types.

Note that this function is only for DODS servers. It has no use on the client side of a DODS client/server connection. The DODS client and server communicate their data with `serialize()` and `deserialize()`.

This method is not implemented for the BaseType class, nor for its children. However, it should be implemented for the specialized children of those classes. For example,

it is not implemented for the Float64 class, but does exist for the NCFloat64 class, specialized to read data from local netCDF files.

This method should be implemented to throw Error when it encounters an unrecoverable error.

For an example of use, see the netCDF library classes. The netCDF library is part of the DODS source distribution, and can be found under `/export/home/tomfool/DODS/src/nc-dods`.

In some sub-classes, such as Array or Grid, the `read()` function must explicitly take into account constraint information stored with the class data. For example, the Grid::read method will be called when only one component of the Grid is to be sent. Your implementation of Grid::read should check send_p() for each member of the Grid before reading that member to avoid reading data into memory that won't be sent (and thus is not needed in memory).

Returns:

The function returns a boolean value, with TRUE indicating that read() should be called again because there's more data to read, and FALSE indicating there's no more data to read. Note that this behavior is necessary to properly handle variables that contain Sequences. WRONG! (9/5/2001 jhrg) Sequences now are read in one shot. The return value of this method should always be false.

Parameters:

dataset A string naming the dataset from which the data is to be read. The meaning of this string will vary among data APIs.

See also:

BaseType

Implements BaseType.

Referenced by ops, and serialize.

3.31.2.6 bool UInt32::serialize (const string & *dataset*, DDS & *dds*, XDR ∗ *sink*, bool *ce_eval* = true) `[virtual]`

Move data to the net.

Sends the data from the indicated (local) dataset through the connection identified by the *sink* parameter. If the data is not already incorporated into the DDS object, read the data from the dataset.

This function is only used on the server side of the client/server connection, and is generally only called from the DDS::send() function. It has no BaseType implementation; each datatype child class supplies its own implementation.

Parameters:

dataset The (local) name of dataset to be read.

dds The Data Descriptor Structure object corresponding to this dataset. See *The DODS User Manual* for information about this structure.

sink A valid XDR pointer generally created with a call to `new_xdrstdio()`. This typically routes data to a TCP/IP socket.

ce_eval A boolean value indicating whether to evaluate the DODS constraint expression that may accompany this dataset. The constraint expression is stored in *dds*.

Returns:

This method always returns true. Older versions used the return value to signal success or failure.

Exceptions:

InternalErr .

Error .

See also:

DDS

Implements BaseType.

Definition at line 70 of file UInt32.cc.

References DDS::eval_selection, read, and BaseType::read_p.

3.31.2.7 unsigned int UInt32::val2buf (void * *val*, bool *reuse* = false) `[virtual]`

Loads class data.

Store the value pointed to by *val* in the object's internal buffer. This function does not perform any checks, so users must be sure that the thing pointed to can actually be stored in the object's buffer. For example, an array cannot easily be fit into the data buffer for an Int32 object.

Only simple objects (Int, Float, Byte, and so on) and arrays and lists of these simple objects may be stored using this function. To put data into more complex constructor functions, use the functions provided by that class. For example, use the `var()` and `add_var()` members of the Grid class to manipulate data in that class.

Parameters:

val A pointer to the data to be inserted into the class data buffer.

reuse A boolean value, indicating whether the class internal data storage can be reused or not. If this argument is TRUE, the class buffer is assumed to be

large enough to hold the incoming data, and it is *not* reallocated. If FALSE, new storage is allocated. If the internal buffer has not been allocated at all, this argument has no effect. This is currently used only in the Vector class.

Returns:
The size (in bytes) of the information copied from *val*.

See also:
Grid , Vector::val2buf

Implements BaseType.

Definition at line 99 of file UInt32.cc.

References width.

3.31.2.8 unsigned int UInt32::width () `[virtual]`

Returns the size of the class instance data.

Return the number of bytes that are required to hold the instance's value. In the case of simple types such as Int32, this is the size of one Int32 (four bytes). For a String or Url type, `width()` returns the number of bytes needed for a `String *` variable, not the bytes needed for all the characters, since that value cannot be determined from type information alone. For Structure, and other constructor types size() returns the number of bytes needed to store pointers to the C++ objects.

Implements BaseType.

Definition at line 64 of file UInt32.cc.

Referenced by buf2val, and val2buf.

3.31.3 Friends And Related Function Documentation

3.31.3.1 friend class Byte `[friend]`

This class allows Byte, ..., Float64 access to `_buf` to simplify and speed up the relational operators.

NB: According to Stroustrup it does not matter where (public, private or protected) friend classes are declared.

Definition at line 51 of file UInt32.h.

The documentation for this class was generated from the following files:

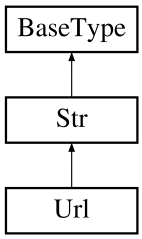
- UInt32.h
- UInt32.cc

3.32 Url Class Reference

Holds an internet address (URL).

`#include <Url.h>`

Inheritance diagram for Url::



Public Methods

- Url (const string &n="")
- virtual ~**Url** ()
- virtual BaseType * **ptr_duplicate** ()=0

Protected Attributes

- string **_buf**

Friends

- class Str

3.32.1 Detailed Description

Holds an internet address (URL).

See also:
BaseType , Str

Definition at line 40 of file Url.h.

3.32.2 Constructor & Destructor Documentation

3.32.2.1 Url::Url (const string & *n* = "")

The URL constructor requires only the name of the variable to be created. The name may be omitted, which will create a nameless variable. This may be adequate for some applications.

Parameters:
n A string containing the name of the variable to be created.

Definition at line 28 of file Url.cc.

References BaseType::set_type.

3.32.3 Friends And Related Function Documentation

3.32.3.1 friend class Str `[friend]`

This enables instances of Str to access _buf in this class.

Definition at line 46 of file Url.h.

The documentation for this class was generated from the following files:

- Url.h
- Url.cc

3.33 USCmp Class Template Reference

```
#include <Operators.h>
```

Static Public Methods

- bool **eq** (UT1 v1, T2 v2)
- bool **ne** (UT1 v1, T2 v2)
- bool **gr** (UT1 v1, T2 v2)
- bool **ge** (UT1 v1, T2 v2)
- bool **lt** (UT1 v1, T2 v2)
- bool **le** (UT1 v1, T2 v2)
- bool **re** (UT1 v1, T2 v2)

3.33.1 Detailed Description

template<class UT1, class T2> class USCmp< UT1, T2 >

Compare two numerical types, the first one unsigned and the second signed. If the signed argument is negative, zero is used in the comparison. This class is one implementation of the comparison policy used by rops.

See also:
rops , SUCmp , Cmp

Definition at line 59 of file Operators.h.

The documentation for this class was generated from the following file:

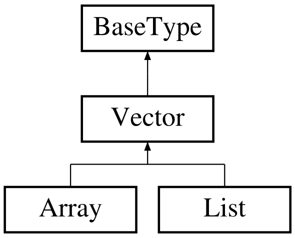
- Operators.h

3.34 Vector Class Reference

Holds a one-dimensional collection of DODS data types.

```
#include <Vector.h>
```

Inheritance diagram for Vector::



Public Methods

- Vector (const string &n=””, BaseType *v=0, const Type &t=dods_null_c)

  The Vector constructor.

- Vector (const Vector &rhs)
- virtual ~**Vector** ()
- Vector & **operator=** (const Vector &rhs)
- virtual BaseType * **ptr_duplicate** ()=0
- virtual int element_count (bool leaves)

  Count the members of constructor types.

- virtual void set_send_p (bool state)

  Indicates that the data is ready to send.

- virtual void set_read_p (bool state)

  Indicates that the data is ready to send.

- virtual unsigned int width ()

  Returns the width of the data, in bytes.

- virtual int length ()
- virtual void set_length (int l)

- virtual bool serialize (const string &dataset, DDS &dds, XDR *sink, bool ce_eval=true)

  Serialize a Vector.

- virtual bool deserialize (XDR *source, DDS *dds, bool reuse=false)

  Receive data from the net.

- virtual bool read (const string &dataset)=0

  Reads data into a local buffer.

- virtual unsigned int val2buf (void *val, bool reuse=false)

  Reads data into the Vector buffer.

- virtual unsigned int buf2val (void **val)
- void set_vec (unsigned int i, BaseType *val)

  Sets element i *to value* val.

- void vec_resize (int l)
- virtual BaseType * var (const string &name="", bool exact_match=true)
- virtual BaseType * var (const string &name, btp_stack &s)
- virtual BaseType * var (unsigned int i)
- virtual void add_var (BaseType *v, Part p=nil)

  Add the BaseType pointer to this constructor type instance.

- virtual void print_decl (ostream &os, string space="", bool print_semi=true, bool constraint_info=false, bool constrained=false)

  Print an ASCII representation of the variable structure.

- virtual void print_val (ostream &os, string space="", bool print_decl_p=true)

  Prints the value of the variable.

- virtual bool check_semantics (string &msg, bool all=false)

  Compare an object's current state with the sematics of its type.

Protected Methods

- void **_duplicate** (const Vector &v)

3.34.1 Detailed Description

Holds a one-dimensional collection of DODS data types.

Holds a one-dimensional array of DODS data types. This class takes two forms, depending on whether the elements of the vector are themselves simple or compound objects. This class contains common functionality for the List and Array classes, and should rarely be used directly.

When each element of the class is a simple data type, the Vector is implemented as a simple array of C types, rather than as an array of BaseType data types. A single private "template" BaseType instance (`_var`) is used to hold information in common to all the members of the array. The template is also used as a container to pass values back and forth to an application program, as in `var()`.

If the elements of the vector are themselves compound data types, the array is stored as a vector of BaseType pointers (see the DODS class **BaseTypePtrVec**). The template is still used to hold information in common to all the members of the array, but is not used to pass information to and from the application program.

See also:
BaseType , List , Array

Definition at line 64 of file Vector.h.

3.34.2 Constructor & Destructor Documentation

3.34.2.1 Vector::Vector (const string & *n* = "", BaseType ∗ *v* = 0, const Type & *t* = dods_null_c)

The Vector constructor.

The Vector constructor requires the name of the variable to be created, and a pointer to an object of the type the Vector is to hold. The name may be omitted, which will create a nameless variable. The template object may not be omitted.

Parameters:

- ***n*** A string containing the name of the variable to be created.
- ***v*** A pointer to a variable of the type to be included in the Vector.
- ***t*** The type of the resulting Vector object, from the Type enum list. There is no DODS Vector object, so all uses of this method will be from the List or Array classes. This defaults to `dods_null_c`.

See also:
Type

Definition at line 74 of file Vector.cc.

References BaseType::set_parent.

3.34.2.2 Vector::Vector (const Vector & *rhs*)

The Vector copy constructor.

Definition at line 83 of file Vector.cc.

3.34.3 Member Function Documentation

3.34.3.1 void Vector::add_var (BaseType * *v*, Part *p* = nil) `[virtual]`

Add the BaseType pointer to this constructor type instance.

Propagate the name of the BaseType instance to this instance. This ensures that variables at any given level of the DDS table have unique names (i.e., that Arrays do not have their default name "").

NB: Part p defaults to nil for this class

Reimplemented from BaseType.

Definition at line 775 of file Vector.cc.

References BaseType::name, BaseType::ptr_duplicate, BaseType::set_name, BaseType::set_parent, and BaseType::type_name.

3.34.3.2 unsigned int Vector::buf2val (void ** *val*) `[virtual]`

Copies data from the Vector buffer. This function assumes that *val* points to an array large enough to hold N instances of the 'C' representation of the element type. In the case of a Vector containing compound elements, this function assumes that *val* points to an array large enough to hold N instances of the DODS class used to represent that type.

Use this function only with Vectors containing simple DODS types. See `set_vec()` to access members of Vectors containing compound types.

Returns:
The number of bytes used to store the array.

Parameters:

val A pointer to a pointer to the memory into which the class data will be copied. If the value pointed to is NULL, memory will be allocated to hold the data, and the pointer value modified accordingly. The calling program is responsible for deallocating the memory indicated by this pointer.

See also:

Vector::set_vec

Implements BaseType.

Definition at line 671 of file Vector.cc.

References length, BaseType::type, BaseType::width, and width.

3.34.3.3 bool Vector::check_semantics (string & *msg*, bool *all* = false) [virtual]

Compare an object's current state with the sematics of its type.

This function checks the class instance for internal consistency. This is important to check for complex constructor classes. For BaseType, an object is semantically correct if it has both a non-null name and type.

For example, an Int32 instance would return FALSE if it had no name or no type defined. A Grid instance might return FALSE for more complex reasons, such as having Map arrays of the wrong size or shape.

This function is used by the DDS class, and will rarely, if ever, be explicitly called by a DODS application program. A variable must pass this test before it is sent, but there may be many other stages in a retrieve operation where it would fail.

Returns:

Returns FALSE when the current state violates some aspect of the type semantics, TRUE otherwise.

Parameters:

msg A returned string, containing a message indicating the source of any problem.

all For complex constructor types (Grid, Sequence, Structure), this flag indicates whether to check the sematics of the member variables, too.

See also:

DDS::check_semantics

Reimplemented from BaseType.

Reimplemented in Array.

Definition at line 827 of file Vector.cc.

References BaseType::check_semantics.

3.34.3.4 bool Vector::deserialize (XDR * *source*, DDS * *dds*, bool *reuse* = false) [virtual]

Receive data from the net.

Receives data from the network connection identified by the `source` parameter. The data is put into the class data buffer according to the input `dds`.

This function is only used on the client side of the DODS client/server connection.

Parameters:

source A valid XDR pointer to the process connection to the net. This is generally created with a call to `new_xdrstdio()`.

dds The Data Descriptor Structure object corresponding to this dataset. See *The DODS User Manual* for information about this structure. This would have been received from the server in an earlier transmission.

reuse A boolean value, indicating whether the class internal data storage can be reused or not. If this argument is TRUE, the class buffer is assumed to be large enough to hold the incoming data, and it is *not* reallocated. If FALSE, new storage is allocated. If the internal buffer has not been allocated at all, this argument has no effect.

Returns:

Always returns TRUE.

Exceptions:

Error when a problem reading from the XDR stream is found.

See also:

DDS

Implements BaseType.

Definition at line 446 of file Vector.cc.

References length, BaseType::ptr_duplicate, set_length, BaseType::type, BaseType::type_name, vec_resize, BaseType::width, width, and BaseType::xdr_coder.

3.34.3.5 int Vector::element_count (bool *leaves*) [virtual]

Count the members of constructor types.

Return a count of the total number of variables in this variable. This is used to count the number of variables held by a constructor variable - for simple type and vector variables it always returns 1.

For compound data types, there are two ways to count members. You can count the members, or you can count the simple members and add that to the count of the compound members. For example, if a Structure contains an Int32 and another Structure that itself contains two Int32 members, the element count of the top-level structure could be two (one Int32 and one Structure) or three (one Int32 by itself and two Int32's in the subsidiary Structure). Use the *leaves* parameter to control which kind of counting you desire.

Returns:

Returns 1 for simple types. For compound members, the count depends on the *leaves* argument.

Parameters:

leaves This parameter is only relevant if the object contains other compound data types. If FALSE, the function counts only the data variables mentioned in the object's declaration. If TRUE, it counts the simple members, and adds that to the sum of the counts for the compound members. This parameter has no effect for simple type variables.

Reimplemented from BaseType.

Definition at line 123 of file Vector.cc.

References BaseType::element_count, and var.

3.34.3.6 int Vector::length () `[virtual]`

Returns the number of elements in the vector. Note that some child classes of Vector use the length of -1 as a flag value.

See also:

Array::append_dim

Definition at line 300 of file Vector.cc.

Referenced by buf2val, deserialize, List::member, List::null, print_val, serialize, Array::update_length, val2buf, and width.

3.34.3.7 void Vector::print_decl (ostream & *os*, string *space* = " ", bool *print_semi* = true, bool *constraint_info* = false, bool *constrained* = false) `[virtual]`

Print an ASCII representation of the variable structure.

Write the variable's declaration in a C-style syntax. This function is used to create textual representation of the Data Descriptor Structure (DDS). See *The DODS User Manual* for information about this structure.

A simple array declaration might look like this:

```
Float64 lat[lat = 180];
```

While a more complex declaration (for a Grid, in this case), would look like this:

```
Grid {
ARRAY:
Int32 sst[time = 404][lat = 180][lon = 360];
MAPS:
Float64 time[time = 404];
Float64 lat[lat = 180];
Float64 lon[lon = 360];
} sst;
```

Parameters:

os The output stream on which to print the declaration.

space Each line of the declaration will begin with the characters in this string. Usually used for leading spaces.

print_semi A boolean value indicating whether to print a semicolon at the end of the declaration.

constraint_info A boolean value indicating whether constraint information is to be printed with the declaration. If the value of this parameter is TRUE, `print_decl()` prints the value of the variable's `send_p()` flag after the declaration.

constrained If this boolean value is TRUE, the variable's declaration is only printed if is the `send_p()` flag is TRUE. If a constraint expression is in place, and this variable is not requested, the `send_p()` flag is FALSE.

See also:

DDS , DDS::CE

Reimplemented from BaseType.

Reimplemented in Array.

Definition at line 790 of file Vector.cc.

References BaseType::print_decl, BaseType::send_p, BaseType::type_name, and var.

Referenced by print_val.

3.34.3.8 void Vector::print_val (ostream & *os*, string *space* = "", bool *print_decl_p* = true) `[virtual]`

Prints the value of the variable.

Prints the value of the variable, with its declaration. This function is primarily intended for debugging DODS applications. However, it can be overloaded and used to do some useful things. Take a look at the asciival and writeval clients, both of which overload this to output the values of variables in different ways.

NB: This function uses C++'s iostream to handle creating the print representations of simple type variables. For floating point numbers this is set to six digits of precision by default. If you want more precision (IEEE 64-bit floats have 15 digits of precision, 32-bit floats have 8), use the setprecision() I/O manipulator.

Parameters:

os The output stream on which to print the value.

space This value is passed to the `print_decl()` function, and controls the leading spaces of the output.

print_decl_p A boolean value controlling whether the variable declaration is printed as well as the value.

Implements BaseType.

Reimplemented in Array.

Definition at line 801 of file Vector.cc.

References length, print_decl, BaseType::print_val, and var.

3.34.3.9 virtual bool Vector::read (const string & *dataset*) `[pure virtual]`

Reads data into a local buffer.

Put the data into a local buffer so that it may be sent to a client. This operation involves reading data from whatever source (often a local disk), and filling out the fields in the data type class. This is the heart of the DODS DAP Class operation. Much of the work of implementing a new DODS server API consists in creating the `read()` functions to read various data types.

Note that this function is only for DODS servers. It has no use on the client side of a DODS client/server connection. The DODS client and server communicate their data with `serialize()` and `deserialize()`.

This method is not implemented for the BaseType class, nor for its children. However, it should be implemented for the specialized children of those classes. For example, it is not implemented for the Float64 class, but does exist for the NCFloat64 class, specialized to read data from local netCDF files.

This method should be implemented to throw Error when it encounters an unrecoverable error.

For an example of use, see the netCDF library classes. The netCDF library is part of the DODS source distribution, and can be found under `/export/home/tomfool/DODS/src/nc-dods`.

In some sub-classes, such as Array or Grid, the `read()` function must explicitly take into account constraint information stored with the class data. For example, the Grid::read method will be called when only one component of the Grid is to be sent. Your implementation of Grid::read should check send_p() for each member of the Grid before reading that member to avoid reading data into memory that won't be sent (and thus is not needed in memory).

Returns:

The function returns a boolean value, with TRUE indicating that read() should be called again because there's more data to read, and FALSE indicating there's no more data to read. Note that this behavior is necessary to properly handle variables that contain Sequences. WRONG! (9/5/2001 jhrg) Sequences now are read in one shot. The return value of this method should always be false.

Parameters:

dataset A string naming the dataset from which the data is to be read. The meaning of this string will vary among data APIs.

See also:

BaseType

Implements BaseType.

Implemented in Array.

Referenced by serialize.

3.34.3.10 bool Vector::serialize (const string & *dataset*, DDS & *dds*, XDR ∗ *sink*, bool *ce_eval* = true) `[virtual]`

Serialize a Vector.

This uses the BaseType member xdr_coder to encode each element of a cardinal array. See Sun's XDR manual. For Arrays of Str and Url types, send the element count over as a prefix to the data so that deserialize will know how many elements to read.

NB: Arrays of cardinal types must already be in BUF (in the local machine's representation) *before* this call is made.

Implements BaseType.

Definition at line 344 of file Vector.cc.

References DDS::eval_selection, length, read, BaseType::read_p, BaseType::type, BaseType::width, and BaseType::xdr_coder.

3.34.3.11 void Vector::set_length (int *l*) `[virtual]`

Sets the length of the vector. This function does not allocate any new space.

Definition at line 312 of file Vector.cc.

Referenced by Array::clear_constraint, deserialize, Array::reset_constraint, and Array::update_length.

3.34.3.12 void Vector::set_read_p (bool *state*) `[virtual]`

Indicates that the data is ready to send.

This function sets the `read_p` flag for both the Vector itself and its element template. This does not matter much when the Vector contains simple data types, but does become significant when the Vector contains compound types.

Reimplemented from BaseType.

Definition at line 157 of file Vector.cc.

References BaseType::set_read_p.

3.34.3.13 void Vector::set_send_p (bool *state*) `[virtual]`

Indicates that the data is ready to send.

This function sets the `send_p` flag for both the Vector itself and its element template. This does not matter much when the Vector contains simple data types, but does become significant when the Vector contains compound types.

Reimplemented from BaseType.

Definition at line 144 of file Vector.cc.

References BaseType::set_send_p.

3.34.3.14 void Vector::set_vec (unsigned int *i*, BaseType * *val*)

Sets element *i* to value *val*.

Sets an element of the vector to a given value. If the type of the input and the type of the Vector do not match, an error condition is returned.

Use this function only with Vectors containing compound DODS types. See `buf2val()` to access members of Vectors containing simple types.

NOTE: The memory allocated by this function should be freed using delete, *not* delete[]!

Returns:

void was a type mismatch.

Parameters:

i The index of the element to be changed.

val A pointer to the value to be inserted into the array.

See also:

Vector::buf2val

Definition at line 743 of file Vector.cc.

References BaseType::type, and vec_resize.

3.34.3.15 unsigned int Vector::val2buf (void ∗ *val*, bool *reuse* = false) [virtual]

Reads data into the Vector buffer.

Copies data into the class instance buffer. This function assumes that the input *val* indicates memory which contains, in row major order, enough elements of the correct type to fill the array. For an array of a cardinal type the memory is simply copied in whole into the Vector buffer. For compound types, the subsidiary `val2buf` is called `length()` times on each successive piece of *val*.

Returns:

The number of bytes used by the array.

Parameters:

val A pointer to the input data.

reuse A boolean value, indicating whether the class internal data storage can be reused or not. If this argument is TRUE, the class buffer is assumed to be large enough to hold the incoming data, and it is *not* reallocated. If FALSE, new storage is allocated. If the internal buffer has not been allocated at all, this argument has no effect.

Implements BaseType.

Definition at line 579 of file Vector.cc.

References length, BaseType::ptr_duplicate, BaseType::type, vec_resize, BaseType::width, and width.

3.34.3.16 BaseType ∗ Vector::var (unsigned int *i*) [virtual]

Returns a pointer to the specified Vector element. For Vectors containing simple data types, the element returned will be a copy of the indicated element. For compound types, the return pointer will indicate the element itself.

Parameters:

i The index of the desired Vector element. Zero indicates the first element of the Vector.

Returns:

A pointer to a BaseType class instance containing the value of the indicated element.

See also:

BaseType::var

Definition at line 232 of file Vector.cc.

References BaseType::type, BaseType::val2buf, and BaseType::width.

3.34.3.17 BaseType * Vector::var (const string & *name*, btp_stack & *s*) `[virtual]`

This version of var(...) searches for *name* and returns a pointer to the BaseType object if found. It uses the same search algorithm as above when *exact_match* is false. In addition to returning a pointer to the variable, it pushes onto *s* a BaseType pointer to each constructor type that ultimately contains *name*.

Parameters:

name Find the variable whose name is *name*.

s Record the path to *name*.

Returns:

A pointer to the named variable.

Reimplemented from BaseType.

Definition at line 202 of file Vector.cc.

References BaseType::is_constructor_type, BaseType::name, and BaseType::var.

3.34.3.18 BaseType * Vector::var (const string & *name* = "", bool *exact_match* = true) `[virtual]`

Returns a copy of the template array element. If the Vector contains simple data types, the template will contain the value of the last vector element accessed with the {\tt Vector::var(int i)} function, if any. If no such access has been made, or if the Vector contains compound data types, the value held by the template instance is undefined.

Note that the parameter *exact_match* is not used by this mfunc.

Parameters:

name The name of the variabe to find.

exact_match Unused.

Returns:

A pointer to the BaseType if found, otherwise null.

See also:

Vector::var

Definition at line 177 of file Vector.cc.

References BaseType::is_constructor_type, BaseType::name, and BaseType::var.

Referenced by Grid::check_semantics, element_count, List::member, print_decl, Array::print_decl, and print_val.

3.34.3.19 void Vector::vec_resize (int *l*)

Resizes a Vector. If the input length is greater than the current length of the Vector, new memory is allocated (the Vector moved if necessary), and the new entries are appended to the end of the array and padded with Null values. If the input length is shorter, the tail values are discarded.

Definition at line 326 of file Vector.cc.

Referenced by deserialize, set_vec, and val2buf.

3.34.3.20 unsigned int Vector::width () `[virtual]`

Returns the width of the data, in bytes.

Returns the number of bytes needed to hold the *entire* array. This is equal to `length()` times the width of each element.

Implements BaseType.

Definition at line 283 of file Vector.cc.

References length, and BaseType::width.

Referenced by buf2val, deserialize, and val2buf.

The documentation for this class was generated from the following files:

- Vector.h
- Vector.cc

Index