Datatype Basics

Contents:

What is a Datatype?

A datatype is a collection of datatype properties which provide complete information for data conversion to or from that datatype.

Datatypes in HDF5 can be grouped as follows:

- **Pre-Defined Datatypes:** These are datatypes that are created by HDF5. They are actually opened (and closed) by HDF5, and can have a different value from one HDF5 session to the next.
- **Derived Datatypes:** These are datatypes that are created or derived from the pre-defined datatypes. Although created from pre-defined types, they represent a category unto themselves. An example of a commonly used derived datatype is a string of more than one character.

Pre-Defined Datatypes

The properties of pre-defined datatypes are:

- Pre-defined datatypes are opened and closed by HDF5.
- A pre-defined datatype is a handle and is *NOT PERSISTENT*. Its value can be different from one HDF5 session to the next.
- Pre-defined datatypes are Read-Only.
- As mentioned, other datatypes can be derived from pre-defined datatypes.

There are two types of pre-defined datatypes, *standard (file)* and *native*:

- **STANDARD**

  A standard (or file) datatype can be:
    - **Atomic:** A datatype which cannot be decomposed into smaller datatype units at the API level.
      The atomic datatypes are: `integer`, `float`, `string` (1-character), `date and time`, `bitfield`, `reference`, `opaque`
    - **Composite:** An aggregation of one or more datatypes.
      Composite datatypes include: `array, variable length, enumeration, compound datatypes`

      Array, variable length, and enumeration datatypes are defined in terms of a single atomic datatype, whereas a compound datatype is a datatype composed of a sequence of datatypes.

  Notes:

  - Standard pre-defined datatypes are the **SAME** on all platforms.
  - They are the datatypes that you see in an HDF5 file.
  - They are typically used when *creating* a dataset.

- **NATIVE**

  Native pre-defined datatypes are used for memory operations, such as reading and writing. They are **NOT THE SAME** on different platforms. They are similar to C type names, and are aliased to the appropriate HDF5 standard pre-defined datatype for a given platform.

For example, when on an Intel based PC, H5T_NATIVE_INT is aliased to the standard pre-defined type, H5T_STD_I32LE. On a MIPS machine, it is aliased to H5T_STD_I32BE.

Notes:

- Native datatypes are **NOT THE SAME** on all platforms.
- Native datatypes simplify memory operations (read/write). The HDF5 library automatically converts as needed.
- Native datatypes are **NOT** in an HDF5 File. The standard pre-defined datatype that a native datatype corresponds to is what you will see in the file.

The following table shows the native types and the standard pre-defined datatypes they correspond to. (Keep in mind that HDF5 can convert between datatypes, so you can specify a buffer of a larger type for a dataset of a given type. For example, you can read a dataset that has a short datatype into a long integer buffer.)

Fig. 1 *Some HDF5 pre-defined native datatypes and corresponding standard (file) type*

C Type	HDF5 Memory Type	HDF5 File Type *
Integer:		
int	`H5T_NATIVE_INT`	`H5T_STD_I32BE or H5T_STD_I32LE`
short	`H5T_NATIVE_SHORT`	`H5T_STD_I16BE or H5T_STD_I16LE`
long	`H5T_NATIVE_LONG`	`H5T_STD_I32BE, H5T_STD_I32LE, H5T_STD_I64BE or H5T_STD_I64LE`
long long	`H5T_NATIVE_LLONG`	`H5T_STD_I64BE or H5T_STD_I64LE`
unsigned int	`H5T_NATIVE_UINT`	`H5T_STD_U32BE or H5T_STD_U32LE`
unsigned short	`H5T_NATIVE_USHORT`	`H5T_STD_U16BE or H5T_STD_U16LE`
unsigned long	`H5T_NATIVE_ULONG`	`H5T_STD_U32BE, H5T_STD_U32LE, H5T_STD_U64BE or H5T_STD_U64LE`
unsigned long long	`H5T_NATIVE_ULLONG`	`H5T_STD_U64BE or H5T_STD_U64LE`
Float:		
float	`H5T_NATIVE_FLOAT`	`H5T_IEEE_F32BE or H5T_IEEE_F32LE`
double	`H5T_NATIVE_DOUBLE`	`H5T_IEEE_F64BE or H5T_IEEE_F64LE`

F90 Type	HDF5 Memory Type	HDF5 File Type *
integer	`H5T_NATIVE_INTEGER`	`H5T_STD_I32(8,16)BE or H5T_STD_I32(8,16)LE`
real	`H5T_NATIVE_REAL`	`H5T_IEEE_F32BE or H5T_IEEE_F32LE`
double-precision	`H5T_NATIVE_DOUBLE`	`H5T_IEEE_F64BE or H5T_IEEE_F64LE`

***** Note that the HDF5 File Types listed are those that are most commonly created. The file type created depends on the compiler switches and platforms being used. For example, on the Cray an integer is 64-bit, and using `H5T_NATIVE_INT` (C)

or `H5T_NATIVE_INTEGER` (F90) would result in an `H5T_STD_I64BE` file type.

The following code is an example of when you would use *standard* pre-defined datatypes vs. *native* types:

```
#include "hdf5.h"

   main() {

      hid_t       file_id, dataset_id, dataspace_id;
      herr_t      status;
      hsize_t     dims[2]={4,6};
      int         i, j, dset_data[4][6];

      for (i = 0; i < 4; i++)
           for (j = 0; j < 6; j++)
            dset_data[i][j] = i * 6 + j + 1;

      file_id = H5Fcreate ("dtypes.h5", H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);

      dataspace_id = H5Screate_simple (2, dims, NULL);

      dataset_id = H5Dcreate (file_id, "/dset", H5T_STD_I32BE, dataspace_id,
                              H5P_DEFAULT);

      status = H5Dwrite (dataset_id, H5T_NATIVE_INT, H5S_ALL, H5S_ALL,
                         H5P_DEFAULT, dset_data);

      status = H5Dclose (dataset_id);

      status = H5Fclose (file_id);
   }
```

By using the native types when reading and writing, the code that reads from or writes to a dataset can be the same for different platforms.

Can native types also be used when creating a dataset? Yes. However, just be aware that the resulting datatype in the file will be one of the standard pre-defined types and may be different than expected.

What happens if you do not use the correct native datatype for a standard (file) datatype? Your data may be incorrect or not what you expect.

Derived Datatypes

ANY pre-defined datatype can be used to derive user-defined datatypes.

To create a datatype derived from a pre-defined type:

- Make a copy of the pre-defined datatype:
  `tid = H5Tcopy (H5T_STD_I32BE);`
- *Change* the datatype.

There are numerous datatype functions that allow a user to alter a pre-defined datatype. See String below for a simple example.

Refer to the Datatype Interface in the HDF5 Reference Manual. Example functions are *H5Tset_size* and *H5Tset_precision*.

Specific Datatypes

On the Examples by API page under Datatypes you will find many example programs for creating and reading datasets with different datatypes.

Below is additional information on some of the datatypes. See the Examples by API page for examples of these datatypes.

Array Datatype vs Array Dataspace

H5T_ARRAY is a datatype, and it should not be confused with the dataspace of a dataset. The dataspace of a dataset can consist of a regular array of elements. For example, the datatype for a dataset could be an atomic datatype like integer, and the dataset could be an N-dimensional appendable array, as specified by the dataspace. See H5S_CREATE and H5S_CREATE_SIMPLE for details.

Unlimited dimensions and subsetting are not supported when using the H5T_ARRAY datatype.

The H5T_ARRAY datatype was primarily created to address the simple case of a compound datatype when all members of the compound datatype are of the same type and there is no need to subset by compound datatype members. Creation of such a datatype is more efficient and I/O also requires less work, because there is no alignment involved.

Array Datatype

The array class of datatypes, `H5T_ARRAY`, allows the construction of true, homogeneous, multi-dimensional arrays. Since these are homogeneous arrays, each element of the array will be of the same datatype, designated at the time the array is created.

Users may be confused by this datatype, as opposed to a dataset with a simple atomic datatype (eg. integer) that is an array. See Array Datatype vs Array Dataspace for more information.

Arrays can be nested. Not only is an array datatype used as an element of an HDF5 dataset, but the elements of an array datatype may be of any datatype, including another array datatype.

Array datatypes ***cannot be subdivided for I/O***; the entire array must be transferred from one dataset to another.

Within certain limitations, outlined in the next paragraph, array datatypes may be *N*-dimensional and of any dimension size. ***Unlimited dimensions, however, are not supported***. Functionality similar to unlimited dimension arrays is available through the use of variable-length datatypes.

The maximum number of dimensions, i.e., the maximum rank, of an array datatype is specified by the HDF5 library constant `H5S_MAX_RANK`. The minimum rank is 1 (one). All dimension sizes must be greater than 0 (zero).

One array datatype may only be converted to another array datatype if the number of dimensions and the sizes of the dimensions are equal and the datatype of the first array's elements can be converted to the datatype of the second array's elements.

Array datatype APIs

There are three functions that are specific to array datatypes: one, H5T_ARRAY_CREATE, for creating an array datatype, and two, H5T_GET_ARRAY_NDIMS and H5T_GET_ARRAY_DIMS for working with existing array datatypes.

Creating

The function H5T_ARRAY_CREATE creates a new array datatype object. Parameters specify

- the base datatype of each element of the array,
- the rank of the array, i.e., the number of dimensions,
- the size of each dimension, and
- the dimension permutation of the array, i.e., whether the elements of the array are listed in C or FORTRAN order. (**Note:** The permutation feature is not implemented in Release 1.4.)

Working with existing array datatypes

When working with existing arrays, one must first determine the the rank, or number of dimensions, of the array.

The function H5T_GET_ARRAY_NDIMS returns the rank of a specified array datatype.

In many instances, one needs further information. The function H5T_GET_ARRAY_DIMS retrieves the permutation of the array and the size of each dimension. (**Note:** The permutation feature is not implemented in Release 1.4.)

Compound

Properties of compound datatypes. A compound datatype is similar to a struct in C or a common block in Fortran. It is a collection of one or more atomic types or small arrays of such types. To create and use of a compound datatype you need to refer to various *properties* of the data compound datatype:

- It is of class *compound.*
- It has a fixed total *size*, in bytes.
- It consists of zero or more *members* (defined in any order) with unique names and which occupy non-overlapping regions within the datum.
- Each member has its own *datatype*.
- Each member is referenced by an *index number* between zero and N-1, where N is the number of members in the compound datatype.
- Each member has a *name* which is unique among its siblings in a compound datatype.
- Each member has a fixed *byte offset*, which is the first byte (smallest byte address) of that member in a compound datatype.
- Each member can be a small array of up to four dimensions.

Properties of members of a compound datatype are defined when the member is added to the compound type and cannot be subsequently modified.

Defining compound datatypes. Compound datatypes must be built out of other datatypes. First, one creates an empty compound datatype and specifies its total size. Then members are added to the compound datatype in any order.

Member names. Each member must have a descriptive name, which is the key used to uniquely identify the member within the compound datatype. A member name in an HDF5 datatype does not necessarily have to be the same as the name of the corresponding member in the C struct in memory, although this is often the case. Nor does one need to define all members of the C struct in the HDF5 compound datatype (or vice versa).

Offsets. Usually a C struct will be defined to hold a data point in memory, and the offsets of the members in memory will be the offsets of the struct members from the beginning of an instance of the struct. The library defines the macro to compute the offset of a member within a struct:
`HOFFSET(s,m)`
This macro computes the offset of member *m* within a struct variable *s*.

Here is an example in which a compound datatype is created to describe complex numbers whose type is defined by the `complex_t` struct.

```
typedef struct {
   double re;   /*real part */
   double im;   /*imaginary part */
} complex_t;

complex_t tmp;  /*used only to compute offsets */
hid_t complex_id = H5Tcreate (H5T_COMPOUND, sizeof tmp);
H5Tinsert (complex_id, "real", HOFFSET(tmp,re),
           H5T_NATIVE_DOUBLE);
H5Tinsert (complex_id, "imaginary", HOFFSET(tmp,im),
           H5T_NATIVE_DOUBLE);
```

Reference

There are two types of Reference datatypes in HDF5:

- Reference to objects
- Reference to a dataset region

Reference to objects

In HDF5, objects (i.e. groups, datasets, and named datatypes) are usually accessed by name. There is another way to access stored objects -- by reference.

An object reference is based on the relative file address of the object header in the file and is constant for the life of the object. Once a reference to an object is created and stored in a dataset in the file, it can be used to dereference the object it points to. References are handy for creating a file index or for grouping related objects by storing references to them in one dataset.

Creating and storing references to objects

The following steps are involved in creating and storing file references to objects:

1. Create the objects or open them if they already exist in the file.
2. Create a dataset to store the objects' references, by specifying H5T_STD_REF_OBJ as the datatype
3. Create and store references to the objects in a buffer, using H5R_CREATE.
4. Write a buffer with the references to the dataset, using H5D_WRITE with the H5T_STD_REF_OBJ datatype.

Reading references and accessing objects using references

The following steps are involved:

1. Open the dataset with the references and read them. The `H5T_STD_REF_OBJ` datatype must be used to describe the memory datatype.
2. Use the read reference to obtain the identifier of the object the reference points to using H5R_DEREFERENCE.
3. Open the dereferenced object and perform the desired operations.
4. Close all objects when the task is complete.

Reference to a dataset region

A dataset region reference points to a dataset selection in another dataset. A reference to the dataset selection (region) is constant for the life of the dataset.

Creating and storing references to dataset regions

The following steps are involved in creating and storing references to a dataset region:

1. Create a dataset to store the dataset region (selection), by passing in H5T_STD_REF_DSETREG for the datatype when calling H5D_CREATE.
2. Create selection(s) in existing dataset(s) using H5S_SELECT_HYPERSLAB and/or H5S_SELECT_ELEMENTS.
3. Create reference(s) to the selection(s) using H5R_CREATE and store them in a buffer.
4. Write the references to the dataset regions in the file.
5. Close all objects.

Reading references to dataset regions

The following steps are involved in reading references to dataset regions and referenced dataset regions (selections).

1. Open and read the dataset containing references to the dataset regions. The datatype `H5T_STD_REF_DSETREG` must be used during read operation.
2. Use H5R_DEREFERENCE to obtain the dataset identifier from the read dataset region reference.

   OR

   Use H5R_GET_REGION to obtain the dataspace identifier for the dataset containing the selection from the read dataset region reference.
3. With the dataspace identifier, the H5S interface functions, H5S_GET_SELECT_*, can be used to obtain information about the selection.
4. Close all objects when they are no longer needed.

The dataset with the region references was read by H5D_READ with the `H5T_STD_REF_DSETREG` datatype specified.

The read reference can be used to obtain the dataset identifier by calling H5R_DEREFERENCE or by obtaining obtain spacial information (dataspace and selection) with the call to H5R_GET_REGION.

The reference to the dataset region has information for both the dataset itself and its selection. In both functions:

1. The first parameter is an identifier of the dataset with the region references.
2. The second parameter specifies the type of reference stored. In this example, a reference to the dataset region is stored.
3. The third parameter is a buffer containing the reference of the specified type.

This example introduces several H5S_GET_SELECT_* functions used to obtain information about selections:

Function	Description
H5S_GET_SELECT_NPOINTS	Returns the number of elements in the hyperslab
H5S_GET_SELECT_HYPER_NBLOCKS	Returns the number of blocks in the hyperslab
H5S_GET_SELECT_HYPER_BLOCKLIST	Returns the "lower left" and "upper right" coordinates of the blocks in the hyperslab selection
H5S_GET_SELECT_BOUNDS	Returns the coordinates of the "minimal" block containing a hyperslab selection
H5S_GET_SELECT_ELEM_NPOINTS	Returns the number of points in the element selection
H5S_GET_SELECT_ELEM_NPOINTS	Returns the coordinates of the element selection

String

A simple example of creating a derived datatype is using the string datatype, H5T_C_S1 (H5T_FORTRAN_S1) to create strings of more than one character. Strings can be stored as either fixed or variable length, and may have different rules for padding of unused storage:

Fixed Length 5-character String Datatype:

```
hid_t strtype;                     /* Datatype ID */
herr_t status;

strtype = H5Tcopy (H5T_C_S1);
status = H5Tset_size (strtype, 5); /* create string of length 5 */
```

Variable Length String Datatype:

```
strtype = H5Tcopy (H5T_C_S1);
status = H5Tset_size (strtype, H5T_VARIABLE);
```

The ability to derive datatypes from pre-defined types allows users to create any number of datatypes, from simple to very complex.

As the term implies, variable length strings are strings of varying lengths. They are stored internally in a heap, potentially impacting efficiency in the following ways:

1. Heap storage requires more space than regular raw data storage.
2. Heap access generally reduces I/O efficiency because it requires individual read or write operations for each data element rather than one read or write per dataset or per data selection.
3. A variable length dataset consists of pointers to the heaps of data, not the actual data. Chunking and filters, including compression, are not available for heaps.

See Section 6.6.1 Strings in the HDF5 User's Guide, for more information on how fixed and variable length strings are stored.

Variable Length

Variable-length (VL) datatypes are sequences of an existing datatype (atomic, VL, or compound) which are not fixed in length from one dataset location to another. In essence, they are similar to C character strings -- a sequence of a type which is pointed to by a particular type of *pointer* -- although they are implemented more closely to FORTRAN strings by including an explicit length in the pointer instead of using a particular value to terminate the sequence.

VL datatypes are useful to the scientific community in many different ways, some of which are listed below:

- Ragged arrays: Multi-dimensional ragged arrays can be implemented with the last (fastest changing) dimension being ragged by using a VL datatype as the type of the element stored. (Or as a field in a compound datatype.)
- Fractal arrays: If a compound datatype has a VL field of another compound type with VL fields (a *nested* VL datatype), this can be used to implement ragged arrays of ragged arrays, to whatever nesting depth is required for the user.
- Polygon lists: A common storage requirement is to efficiently store arrays of polygons with different numbers of vertices. VL datatypes can be used to efficiently and succinctly describe an array of polygons with different numbers of vertices.
- Character strings: Perhaps the most common use of VL datatypes will be to store C-like VL character strings in dataset elements or as attributes of objects.
- Indices: An array of VL object references could be used as an index to all the objects in a file which contain a particular sequence of dataset values. Perhaps an array something like the following:
```
Value1: Object1, Object3,  Object9
            Value2: Object0, Object12, Object14, Object21, Object22
            Value3: Object2
            Value4: <none>
            Value5: Object1, Object10, Object12
                .
                .
```
- Object Tracking: An array of VL dataset region references can be used as a method of tracking objects or features appearing in a sequence of datasets. Perhaps an array of them would look like:
```
Feature1: Dataset1:Region,  Dataset3:Region,  Dataset9:Region
            Feature2: Dataset0:Region,  Dataset12:Region, Dataset14:Region,
                      Dataset21:Region, Dataset22:Region
            Feature3: Dataset2:Region
            Feature4: <none>
            Feature5: Dataset1:Region,  Dataset10:Region, Dataset12:Region
                .
                .
```

Variable-length datatype memory management

With each element possibly being of different sequence lengths for a dataset with a VL datatype, the memory for the VL datatype must be dynamically allocated. Currently there are two methods of managing the memory for VL datatypes: the standard C malloc/free memory allocation routines or a method of calling user-defined memory management routines to allocate or free memory. Since the memory allocated when reading (or writing) may be complicated to release, an HDF5 routine is provided to traverse a memory buffer and free the VL datatype information without leaking memory.

Variable-length datatypes cannot be divided

VL datatypes are designed so that they cannot be subdivided by the library with selections, etc. This design was chosen due to the complexities in specifying selections on each VL element of a dataset through a selection API that is easy to understand. Also, the selection APIs work on dataspaces, not on datatypes. At some point in time, we may want to create a way for dataspaces to have VL components to them and we would need to allow selections of those VL regions, but that is beyond the scope of this document.

What happens if the library runs out of memory while reading?

It is possible for a call to `H5Dread` to fail while reading in VL datatype information if the memory required exceeds that which is available. In this case, the `H5Dread` call will fail gracefully and any VL data which has been allocated prior to the memory shortage will be returned to the system via the memory management routines detailed below. It may be possible to design a *partial read* API function at a later date, if demand for such a function warrants.

Strings as variable-length datatypes

Since character strings are a special case of VL data that is implemented in many different ways on different machines and in different programming languages, they are handled somewhat differently from other VL datatypes in HDF5.

HDF5 has native VL strings for each language API, which are stored the same way on disk, but are exported through each language API in a natural way for that language. When retrieving VL strings from a dataset, users may choose to have them stored in memory as a native VL string or in HDF5's `hvl_t` struct for VL datatypes.

VL strings may be created in one of two ways: by creating a VL datatype with a base type of `H5T_NATIVE_ASCII`, `H5T_NATIVE_UNICODE`, etc., or by creating a string datatype and setting its length to `H5T_VARIABLE`. The second method is used to access native VL strings in memory. The library will convert between the two types, but they are stored on disk using different datatypes and have different memory representations.

Multi-byte character representations, such as UNICODE or *wide* characters in C/C++, will need the appropriate character and string datatypes created so that they can be described properly through the datatype API. Additional conversions between these types and the current ASCII characters will also be required.

Variable-width character strings (which might be compressed data or some other encoding) are not currently handled by this design. We will evaluate how to implement them based on user feedback.

Variable-length datatype APIs

Creation

VL datatypes are created with the `H5Tvlen_create()` function as follows:

type_id = H5Tvlen_create(hid_t base_type_id);

The base datatype will be the datatype that the sequence is composed of, characters for character strings, vertex coordinates for polygon lists, etc. The base datatype specified for the VL datatype can be of any HDF5 datatype, including another VL datatype, a compound datatype, or an atomic datatype.

Querying base datatype of VL datatype

It may be necessary to know the base datatype of a VL datatype before memory is allocated, etc. The base datatype is queried with the `H5Tget_ super()` function, described in the H5T documentation.

Querying minimum memory required for VL information

It order to predict the memory usage that `H5Dread` may need to allocate to store VL data while reading the data, the `H5Dget_vlen_size()` fun ction is provided:

herr_t H5Dvlen_get_buf_size(hid_t dataset_id, hid_t type_id, hid_t space_id, hsize_t *size)

This routine checks the number of bytes required to store the VL data from the dataset, using the `space_id` for the selection in the dataset on disk and the `type_id` for the memory representation of the VL data in memory. The `*size` value is modified according to how many bytes are required to store the VL data in memory.

Specifying how to manage memory for the VL datatype

The memory management method is determined by dataset transfer properties passed into the `H5Dread` and `H5Dwrite` functions with the dataset transfer property list.

Default memory management is set by using `H5P_DEFAULT` for the dataset transfer property list identifier. If `H5P_DEFAULT` is used with `H5Dread`

, the system `malloc` and `free` calls will be used for allocating and freeing memory. In such a case, `H5P_DEFAULT` should also be passed as the property list identifier to `H5Dvlen_reclaim`.

The rest of this subsection is relevant only to those who choose *not* to use default memory management.

The user can choose whether to use the system `malloc` and `free` calls or user-defined, or custom, memory management functions. If user-defined memory management functions are to be used, the memory allocation and free routines must be defined via `H5Pset_vlen_mem_manager()`, as follows:

herr_t H5Pset_vlen_mem_manager(hid_t plist_id, H5MM_allocate_t alloc, void *alloc_info, H5MM_free_t free, void *free_info)

The `alloc` and `free` parameters identify the memory management routines to be used. If the user has defined custom memory management routines, `alloc` and/or `free` should be set to make those routine calls (i.e., the name of the routine is used as the value of the parameter); if the user prefers to use the system's `malloc` and/or `free`, the `alloc` and `free` parameters, respectively, should be set to `NULL`

The prototypes for the user-defined functions would appear as follows:

typedef void *(*H5MM_allocate_t)(size_t size, void *info) ; typedef void (*H5MM_free_t)(void *mem, void *free_info) ;

The `alloc_info` and `free_info` parameters can be used to pass along any required information to the user's memory management routines.

In summary, if the user has defined custom memory management routines, the name(s) of the routines are passed in the `alloc` and `free` parameters and the custom routines' parameters are passed in the `alloc_info` and `free_info` parameters. If the user wishes to use the system `malloc` and `free` functions, the `alloc` and/or `free` parameters are set to `NULL` and the `alloc_info` and `free_info` parameters are ignored.

Recovering memory from VL buffers read in

The complex memory buffers created for a VL datatype may be reclaimed with the `H5Dvlen_reclaim()` function call, as follows:

herr_t H5Dvlen_reclaim(hid_t type_id, hid_t space_id, hid_t plist_id, void *buf);

The `type_id` must be the datatype stored in the buffer, `space_id` describes the selection for the memory buffer to free the VL datatypes within, `plist_id` is the dataset transfer property list which was used for the I/O transfer to create the buffer, and `buf` is the pointer to the buffer to free the VL memory within. The VL structures (`hvl_t`) in the user's buffer are modified to zero out the VL information after it has been freed.

If nested VL datatypes were used to create the buffer, this routine frees them from the bottom up, releasing all the memory without creating memory leaks.