StorageHelpers: A Format Independent Storage Specification

C. D. Jones[1], *D. Riley*[1], *M. Lohner*[1]

Cornell University,USA

Abstract

The CLEO III data access system is designed to be input/output data format independent. This allows us to use the most appropriate storage technology for each type of data and for each use of the system (e.g. reconstruction versus analysis). But if we have N storage formats and M data items to store, we potentially have to deal with N*M storage specifications. To avoid this problem, we developed StorageHelpers which are C++ classes that encapsulate the schema and compression information of an object you wish to store. StorageHelpers allow us to efficiently store objects for any format without any modifications to the class being stored. We have tried to make it easy for a user to write a StorageHelper by using the standard C++ iostream notation and by creating a C++ header file parser which will write most of the StorageHelper code automatically. Using a StorageHelper is also easy, all the user has to do is link with the library that contains the object they want to read or write.

Keywords: storage, CLEO, format independent

1 Introduction

The CLEO III data access system is designed to be independent of the input/output data format. This allows us to use the most appropriate storage technology for each type of data and for each use of the system (e.g. reconstruction versus analysis). But if we have N storage formats and M data items to store, we potentially have to deal with N*M storage specifications. To avoid this problem, we developed StorageHelpers which are C++ classes that encapsulate the schema and compression information of an object you wish to store. StorageHelpers allow us to efficiently store objects for any format without any modifications to the class being stored. StorageHelpers also allow specification of lossy compression hints, but this subject will not be covered in this paper because of length constraints.

We decided not to use a data description language (DDL) to specify storage because

- users do not have to learn a new language to create their storable objects
- it allows users to have non-trivial transformations from memory objects to stored objects
- maintenance is easier since no need to create DDL to storage format code generators

2 Design

The core of the design of StorageHelpers is:

- StorageHelpers register themselves with a StorageManager
- data sources and sinks retrieve the appropriate StorageHelpers from the StorageManager
- sources and sinks use StorageHelpers to read and write the data.

2.1 Overview of CLEO III data access

All CLEO III data is accessed through an object called the Frame [1]. Conceptually, the Frame holds all information necessary to understand the CLEO detector at an instant in time. The data within the Frame are grouped by Records, where a Record holds data that have the same life-time. For example, the Event Record holds all data associated with that particular event such as the hits in the drift chamber and the tracks that were found. To access the data, a user calls the templated function `extract()` with arguments specifying the Record that contains the data and a variable able to hold the data.

The extract call does not actually get the data from the Record, because the Record does not directly hold the data. Instead, a Record holds a data Proxy. So the extract call retrieves the Proxy that is associated with the requested data and then calls the `get()` method of the Proxy to actually obtain the data. Using Proxies allows us to do the work of obtaining a piece of data only if that data has been requested.

A data source (e.g. a binary file or an Objectivity database) must supply Proxies for the data the source holds. When the source's Proxy's `get()` method is called, the Proxy must retrieve the data from the source. For example, a Proxy for a binary file could read the data from a memory buffer that was filled by the source. A data sink must call `extract` for each piece of data that is to be stored. Therefore the StorageHelper system must

- provide Proxies which can read from any source and then create the proper objects
- provide ProxyWriters which can extract the data from a Record and then format the data to be written out by any sink

2.2 Serialization

We need a way for the StorageHelpers to be able to communicate with any data source and sink. We chose to make every data source and sink look like a standard C++ iostream. Therefore to read data from a source, a StorageHelper is given a SourceStream object and similarly to write data to a sink a StorageHelper is given a SinkStream.

To allow us determine the schema of our data when writing out to a data sink, the user must pass both the data field's value and the data field's name in the call. E.g.

```
void MyDataStorageHelper::implementStore( SinkStream& iSink,
                                          const MyData& iData ) {
   iSink << SMField<int>( ‘‘a’’, iData.a() )
         << SMField<double>(‘‘b’’, iData.b() ); }
```

which can be simplified by using a templated function to determine the template argument of `SMField<>` by the return type of `iData.a()` and `iData.b()`

```
iSink << sm_field( ‘‘a’’, iData.a() ) << sm_field(‘‘b’’, iData.b() );
```

Further simplification can be obtained by using a C pre-processor macro to build the Field's name from the name of the member function

```
iSink << SM_VAR(iData, a ) << SM_VAR(iData, b );
```

When reading an object back from a source, only the data value is returned

```
MyData* MyDataStorageHelper::deliver( SourceStream& iSource ) {
  int a;   double b;
  iSource >> a >> b;
  return new MyData( a, b ); }
```

2.3 Containers of Objects

It is often the case that an object will internally hold other objects, particularly containers of other objects. StorageHelpers are designed to handle any container that provides iterators similar to those of the standard C++ library. To write out a container of pointers to objects one would do

```
iSink << sm_field(‘‘contents’’, sm_contents(iData.begin(),
                                            iData.end()));
```

To read the container back we use a Standard C++ Library insert iterator

```
vector<MyData*> container;
back_insert_iterator<vector<MyData*> > insertIterator(container);
iSource >> sm_make_contents( insertIterator);
```

The storage and retrieval of data in a container is done by calling the StorageHelper for the data type in the container and having that StorageHelper do the actual work.

2.4 Registration

We want to be able to register the appropriate StorageHelpers with the StorageManager without any special action by the user. Essentially we want to figure out which StorageHelpers to register just by determining which extract calls the user has made. We accomplish this by using the linker and global object initialization.

We explicitly instantiate a particular instance of the extract templated function in one source file. In this same source file we put a static instance of the StorageHelper class we want to register. For our example of the MyData class, we would have a file called T_MyData.cc which contains the instantiation of extract for MyData and has a static variable of type MyDataStorageHelper called s_MyDataStorageHelper. If a user has called extract for a MyData object, they will need to link to the library that has the T_MyData.o object file. By linking to T_MyData.o they automatically get the static variable s_MyDataStorageHelper. When the code is executed, the first thing that happens is that all static variables are initialized. When s_MyDataStorageHelper's constructor is run, it registers s_MyDataStorageHelper with the StorageManager. Therefore just by calling extract and linking to the correct library, the correct StorageHelper is registered.

2.5 Sinks and Sources

Sinks create the ProxyWriters needed to store the objects by using all the StorageHelpers that have been registered. To work with StorageHelpers, a class which inherits from SinkStream must be created which takes the serialized object data and writes it out to the Sink.

Sources use only those StorageHelpers which correspond to data types that are available in the Source. It does not matter if there are some data types in the Source that do not have StorageHelpers because the absence of a registered StorageHelpers means nothing if the job will not need to read that data type. To work with StorageHelpers, a class which inherits from SourceStream must be created which can read back the object data from the Source and return it in a serialized manner.

Although StorageHelpers work best with Sinks and Sources that are naturally serialized (e.g. binary files) it is possible to work with non-serialized Sinks and Sources, e.g. an object database. To accomplished this,

- generate a schema by querying the StorageHelpers for the type and name of each object's field
- have a code generator use the schema to generate a special SinkStream/SourceStream for each type to write/read

Every time a StorageHelper changes, the above procedure must be repeated.

3 Creating StorageHelpers

We want users to use StorageHelpers to store their own types of data in addition to storing the official reconstructed objects. To make this practical, it must be fairly easy to write a StorageHelper especially for simple classes. Therefore to aid in the construction of StorageHelpers we have created a code generator which creates a skeleton of a StorageHelper by parsing a class's header file.

The parser looks for the first non trivial constructor defined in the class declaration. Once found, the parser takes the types and names of the constructor's arguments. The auto-generator then assumes that the arguments' have the same name and return type as a member function of the class and uses these names to write the storage code.

As an example, we will use the simple class defined below.

```
class MyData {
    public:
      MyData( int a, double b );
      int a() const;
      double b() const;
      ... };
```

The parser will find the constructor `MyData( int a, double b )` and will assume that the class has functions `int a() const` and `double b() const`. After parsing, the auto-generator will produce the following code (which has been simplified for illustration purposes).

```
void MyDataStorageHelper::implementStore( SinkStream& iSink,
                                          const MyData& iData ) {
   iSink << SM_VAR( iData, a ) << SM_VAR( iData, b ); }

MyData* MyDataStorageHelper::deliver( SourceStream& iSource ) {
  int a ;
  double b;
  iSource >> a >> b;
  return new MyData( a, b ); }
```

For most purposes, the auto-generated code will be correct but the user can modify the code.

4 Conclusion

The core StorageHelper code has been written, but we are only just beginning to have users write their StorageHelpers. Our very limited sample shows that our auto-generation works very well for simple classes (which are the majority) but we need to improve our documentation on how to handle more advanced cases. However we have found it relatively easy to write Sources/Sinks for storing to a simple ASCII text format and to an Objectivity/DB which is storing the objects as binary blobs [2].

References

1 C.D.Jones, M.Lohner , "CLEO's User Centric Data Access System", CHEP2000, Padova, Spring 2000.

2 M.Lohner, C.D.Jones, D.Riley, "CLEO III Data Storage", CHEP2000, Padova, Spring 2000.