6

Creating Objects and Writing Data

Schedule:	Timing	Topic
	20 minutes	Lecture
	25 minutes	Practice
	45 minutes	Total

Objectives

After completing this lesson, you should be able to:

- **Design data store objects.**
- **Implement standard persistency.**
- **Read and write your own mini-DSTs.**

Lesson Aim

For physics data processing it is typically not sufficient to use only existing objects. For example when building a reconstruction program new objects have to be created, which hold the reconstruction information such as the result of track fits etc.

The aim of this presentation is to show the few key issues to be considered when designing new objects. It will also be shown how a simple persistency mechanism can be implemented, which allows to write small user defined mini-dsts.

Two Types Of Objects

Identifiable objects

- Access by name: “/Event”, “/Event/MC” etc.
- ObjectVector<MCParticle>: “/Event/MC/MCParticles”

Non-identifiable objects

- Container is identifiable
- 5th. MCParticle in ObjectVector<MCParticle>

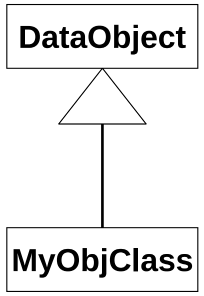
Two Types of Objects

Typically Gaudi itself knows about 2 types of objects:

•Identifiable objects. These are the atomic units known to the data store. They can be individually retrieved from the data store.

•Non identifiable objects typically are aggregated into containers such as the ObjectVector, which in turn is identifiable.

Design of Identifiable Objects



Inherit from DataObject

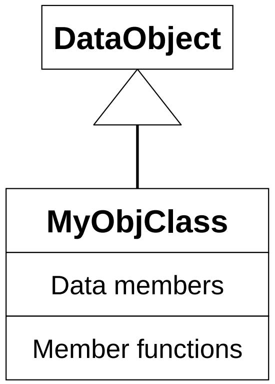
- Data store objects must implement a basic functionality
- Class understood by the data store

Design of Identifiable Objects

The data requires from each object a certain functionality. The most important one is the ability to properly delete the object. For this reason each object on the store must inherit from the class DataObject.

Another functionality of the data object is the capability of browsing the next layer of objects. Like in a unix file system you can browse the directory without actually touching any of the files.

Design of Identifiable Objects



Override:
virtual const CLID& clID() const;
static const CLID& classID()
(Persistent type information)

! CLID must be unique !

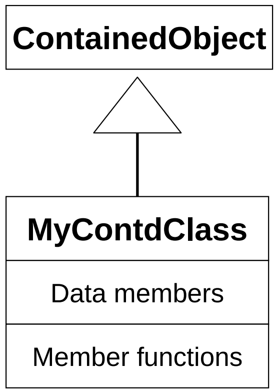
Design of Identifiable Objects

Since a normal DataObject is not sufficient for physics you have to attach data to it. This is done in the sub-class. To access these data and/or manipulate the data or present them in the requested form to the algorithm using this object member functions are needed.

C++ has no intrinsic persistency mechanism. Although there is some run-time type information (RTTI) available, this information cannot be used for persistency. For this reason a class identifier was invented, the CLID. Hence each class *must* override the corresponding access functions. The CLID is used by Gaudi to decide which converter must be used in order to make the object persistent and to create an object in memory.

If the class evolves e.g. when you add additional data fields, which cannot be re-calculated from existing persistent object data, you *must* use a new CLID.

Non-Identifiable Objects



Same rules
Replace DataObject with ContainedObject
Override:
virtual const CLID& clID() const;
static const CLID& classID()
(Persistent type information)

Non- Identifiable Objects

All requirements of DataObjects are also valid for ContainedObjects. Do not forget to reserve an unused CLID for the object class.

Data Persistency

- **Data conversion mechanism**
  - Transient -> Persistent … Persistent -> Transient
- **Generic converters**
  - Object serialization as done by Java, Root, MFC, etc.
  - Converters come (nearly) for free
- **Specific converters**
  - Real work
  - Allows optimization
    - Persistent world: minimize I/O, use DBase features
    - Transient world: optimize navigation

Data Persistency

The data conversion mechanism in Gaudi must solve the problem to first write object from memory to disk and later be able to read these objects back.

There are two possibilities to achieve this:

•A generic conversion mechanism, which uses object serialization as it is known from Java, Root or the Microsoft foundation class library. This sort of serialization is simple to implement and the converters come nearly for free. However, this mechanism can only make limited use of optimization e.g. if the object could easily be recreated from other, redundant data residing in the transient data store.

•The other possibility is to write a specialized converter. This involves real work, because then the converter must be written by hand. However, there are benefits:

•A specialized converter allows to better minimize I/O (pack doubles into short by reducing the dynamic range, recalculate certain redundant data etc.).

•A specialized converter could take advantage of the underlying database engine.

•In the transient world such a converter could also add additional data for improved navigation.

Note:

Generic converters do not create “native” ROOT objects. Interactive access to individual data members is currently not possible using ROOT.

Data Serialization

- **Data Serialization: Transient->Persistent**

```
long                 m_num;
float                m_px, m_py, m_pz;
SmartRef<MCParticle> m_mcTruth;

StreamBuffer& MyObj::serialize(StreamBuffer& s) const {
  DataObject::serialize(s);
  s << m_num << m_px << m_py << m_pz
    << m_mcTruth(this);
  return s;
}
```

Base class

Member data

Note: !! Latest now your class needs a unique CLID !!

Data Serialization

When making an object persistent data serialization maps the object’s data to a flat byte stream, which then can be written to disk. This task is handled by the StreamBuffer object.

Primitive data items such as integers, numbers and the like are very simple to write. The problem arises when writing pointers. Pointers have the disadvantage that they point to some location of the machine’s memory. This depends on the machine, the load, the number of tasks etc. This is also the real application of the SmartRef object: SmartRefs allow in a rather anonymous way to convert a pointer to an object in the datastore into information which can be written to disk.

Note:

The output is a const member function of the object, because writing data should not change the object itself!

Data Serialization

- Data Serialization: Persistent->Transient
- It's just the reverse

```
StreamBuffer& MyObj::serialize(StreamBuffer& s) {
  DataObject::serialize(s);
  s >> m_num >> m_px >> m_py >> m_pz
    >> m_mcTruth(this);
  return s;
}
```

Base class

Member data

Data Serialization

When reading objects from disk the reverse actions are performed. After reading the data from the StreamBuffer, the object should be in the same state as before writing. This is exactly true for primitive data items. References are not in the same state. Whereas a reference did contain a pointer to a C++ object in memory, it now only contains the recipe how to get hold of this object when needed. This however is equivalent.

Generic Converters

- Declare Converter
- Requires empty constructor

```
#include "GaudiDb/DbGenericConverter.h"

#include "MyDataObject.h" // Of type DataObject
_ImplementConverter(MyDataObject)
```

See: GaudiDb/DbGenericConverter.h what this does

Generic Converter

The last thing needed by Gaudi to read and write data is the converter itself. This converter can be instantiated using a macro. Because such a converter must create objects of the requested type when reading back already written objects, one of the object constructors must have an empty signature:

```
MyDataObject()  {
}
```

You can have any number of other constructors as well.

Generic Container Converters

- Declare Container Converter
- Requires empty constructor

```
#include "GaudiKernel/ObjectVector.h”
#include "GaudiKernel/ObjectList.h”
#include "GaudiDb/DbContainerConverter.h"


#include ”MyContainedObject.h”
_ImplementContainerConverters(MCParticle)
```

See: GaudiDb/DbContainerConverter.h what this does

The Remaining Machinery

Setup in the job options

```
ApplicationMgr.DLLs      += { "GaudiDb", "GaudiRootDb" };
ApplicationMgr.ExtSvc    += {"DbEventCnvSvc/RootEvtCnvSvc"};
ApplicationMgr.OutStream = { "RootDst" };
EventPersistencySvc.CnvServices += { "RootEvtCnvSvc" };
RootEvtCnvSvc.DbType = "ROOT";

//  Setup for ROOT I/O System
RootDst.ItemList  = { "/Event#999" };
RootDst.Output    = "DATAFILE='RootDst.root' TYP='ROOT'";
```

The Remaining Machinery

The rest of the job is done in the job options.

•Gaudi must be instructed to load the additional code and create an additional service used to write root objects.

•A output stream must be defined, which takes care of writing a specified list of objects to the output destination.

•The output destination must be defined.

The two non highlighted statements must be added for the internal setup.

Reading Objects

Setup in the job options

```
ApplicationMgr.DLLs     += { "GaudiDb", "GaudiRootDb" };
ApplicationMgr.ExtSvc   += {"DbEventCnvSvc/RootEvtCnvSvc"};

EventPersistencySvc.CnvServices += { "RootEvtCnvSvc" };
RootEvtCnvSvc.DbType = "ROOT";
```

```
//  Setup data source
EventSelector.Input =
      {"DATAFILE='RootDst.root' TYP='ROOT' OPT='READ'" };
```

Reading Objects

For reading most of the previously described options remain.

Different is the definition of the input file.

Hands On

- **Create an EventSummary**
  - contains e.g. #of MC particles
  - highest Pt
  - ...
- **Register it on the store /Event/Summary**
- **Write the object to a ROOT file**

Hands On

In this exercise we try to invent a new object where we intend to store event summary informaton. This could contain e.g. the number of MC particles and the maximal transverse momentum. This object must then be registered to the data store.

Once registered the object should be written to a root file.

EventSummary.h

```
static const CLID& CLID_EventSummary = 199;
class EventSummary : public DataObject   {
public:
  EventSummary()   : m_ptMax(0.0)  {}
  virtual ~EventSummary()  {}
  // methods to access m_px etc.
  float ptMax() { return m_ptMax; }
  void setPtMax(float pt) {…}
  virtual const CLID& clID()  const;
  + classID(), serialize()
private:
  float                m_ptMax;
};
```

6-15 Gaudi Framework Tutorial, 2001

Implement Converter

EventSummaryCnv.cpp

– Don’t forget to declare the factory in xxx_load.cpp

```
#include "GaudiDb/DbGenericConverter.h”

#include ”EventSummary.h”
_ImplementConverter(EventSummary)
```

6-16 Gaudi Framework Tutorial, 2001

Register The Object

VisibleEnergyAlgorithm.cpp

```
#include "GaudiKernel/IDataProviderSvc.h”
#include ”EventSummary.h”
StatusCode VisibleEnergyAlgorithm::execute()  {
  EventSummary* summary = new EventSummary();
  … fill in data …
  status = eventSvc()->registerObject(
                  “/Event/Summary”, summary);
  if ( !status.isSuccess() )  { ! Error !
    delete summary;  error message etc.
  }
}
```

6-17 Gaudi Framework Tutorial, 2001

Update Job Options

```
ApplicationMgr.DLLs     += {"GaudiDb", "GaudiRootDb" };
ApplicationMgr.ExtSvc   += {"DbEventCnvSvc/RootEvtCnvSvc"};
ApplicationMgr.OutStream = {"RootDst" };
RootEvtCnvSvc.DbType     = "ROOT";
EventPersistencySvc.CnvServices += { "RootEvtCnvSvc" };

//  Setup for ROOT I/O System
RootDst.ItemList  = {"/Event#1","/Event/EventSummary#1"};
RootDst.Output    = "DATAFILE='RootDst.root' TYP='ROOT'";
```

6-18 Gaudi Framework Tutorial, 2001