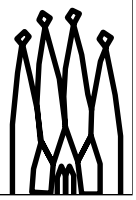
6

Creating Objects and Writing Data

Gaudi Framework Tutorial, April 2006

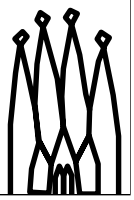


Schedule:	**Timing**	**Topic**
	20 minutes	Lecture
	25 minutes	Practice
	45 minutes	Total

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.

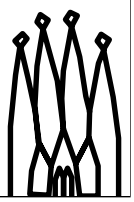
Two Types Of Objects

Identifiable objects

- Access by name: "/Event/MC/Particles" or LHCb::MCParticleLocation::Default
- Typically:
  - a KeyedContainer (e.g. of MCParticles)
  - a DataObject (e.g. "/Event/MC/Header")

Non-identifiable objects

- 5th. MCParticle in MCParticles
- MCParticle with key = 25



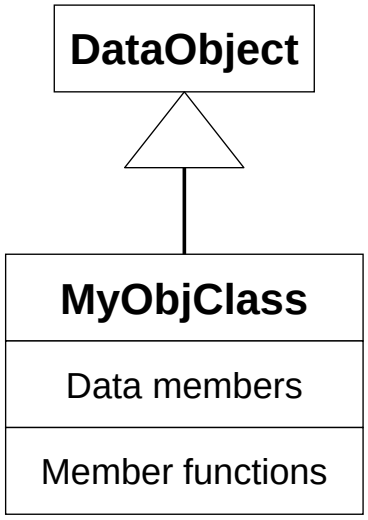
Two Types of Objects

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.

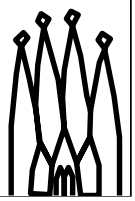
•Non identifiable objects typically are aggregated into containers such as the KeyedContainer, which in turn is identifiable. An object within the container can be accessed via the container – e.g. the 5th object, or the object with a specific "Key"

Design of Identifiable Objects



Inherit from DataObject

- **Data store objects must implement a basic functionality**
- **Class understood by the data store**
- **Have a unique CLID**



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.

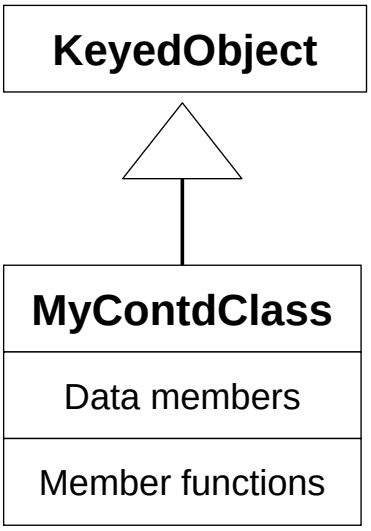
Another functionality of the data object is the capability of browsing the next layer of objects. Like in a unix file system you can browse the directory without actually touching any of the files.

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* provide a unique CLID

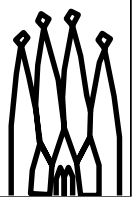
If the class evolves e.g. when you add additional data fields, which cannot be re-calculated from existing persistent object data, you *must* use a new CLID.

Non-Identifiable Objects



Same rules

Replace DataObject with KeyedObject



Non- Identifiable Objects

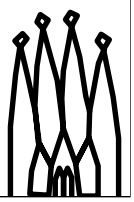
All requirements of DataObjects are also valid for contained objects. There are various types of contained objects, the most commonly used in LHCb is the KeyedObject, which resides in a KeyedContainer. It is not identifiable directly from the data store, by via its key in he container.

Do not forget to reserve an unused CLID for the object class.

Gaudi Object Description

Set of tools to describe event data classes

- **Description in XML**
- **Generates .h plus dictionaries**
  - Including all setters and getters
  - Including all infrastructure needed for persistency



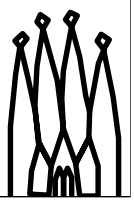
Gaudi Object Description

In order to facilitate writing persistent capable classes, and to ensure uniformity, a tool is provided for describing the classes in a high level language (Xml) from which the header files and dictionaries are generated.

Here we give only a brief introduction to this tool.

G.O.D. Xml file

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE gdd SYSTEM "gdd.dtd">
<gdd>
  <package name="TutKernel">
     <class author="Marco Cattaneo"
      desc="Example of a simple keyedObject"
      name="VertexInfo"
      location="MC/PrimaryVertices"
      id="12345"
     >
       &KeyedObject;
       <base name="KeyedObject&lt;int&gt;"/>
       <import name="Event/MCVertex"/>
       <attribute
        desc="Vertex multiplicity"
        name="multiplicity“ type="long" init="0"
       />
    </class>
  </package>
</gdd>
```

Example Xml file for Gaudi Object Description

Xml files consist of a set of tags, following some basic rules:

- All tags must have an opening and closing tag:
  <tag> Some data </tag>
  <tag Some data />
- Document has a DOCTYPE, described by a DTD
- Possible tags are described in the dtd, and contain name/value pairs

In this example we define a class “VertexInfo” with classID 12345. It inherits from KeyedObject, and has one attribute “multiplicity”.

There are many more tags possible. This is described in the more advanced event model lesson.

Note on conventions:

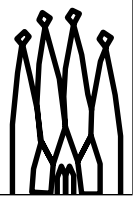
GaudiObjDesc will put header files generated from the Xml in the sub-directory /Event. This is an exception to the standard convention that all include files reside in a sub-directory with the same name as the package, and is done so that all event model includes will be found on the same path (#include “Event/xxx.h”) regardless of the package.

Adding objects to the Transient Event Store

In order to write out an object, you first have to add it to the T.E.S:

```
// Create a container for the VertexInfo objects
  LHCb::VertexInfos* infoCont = new LHCb::VertexInfos();
// Declare the VertexInfo to the transient store
  put( infoCont, LHCb::VertexInfoLocation::Default );

// Loop over vertices to identify primary vertices
{ ...
  // Make new VertexInfo object, insert in container
    LHCb::VertexInfo* newInfo = new LHcb::VertexInfo();
    newInfo->setMultiplicity( daughters.size() );
    infoCont->insert( newInfo );
}
```



Adding objects to the transient store

Objects to be added to the TES must be created with **new**. Once these objects have been **put** on the transient store, they are *owned* by the TES, who will take care of deleting them when they are no longer needed (at the end of the event)

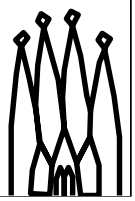
To be added to the transient store, the object must inherit from DataObject – in this case VertexInfos is a KeyedContainer for the VertexInfo objects described in the Xml example of the previous slide

The VertexInfo objects themselves are KeyedObjects that can be put into the TES by adding them to a KeyedContainer. Objects to be added to containers are created with **new** and are owned by the container, who will take care of deleting them.

It is a matter of choice whether you fill the container first and then **put** it on the transient store, or the other way round. The advantage of first putting on the store, then filling, is that you don't need to worry about deleting the objects you have just created with new, in the case of alternate return before the **put** statement

Data Persistency

- **Data conversion mechanism**
  - Transient -> Persistent … Persistent -> Transient
- **Generic converters**
  - Object serialization by Root
  - Converters come for free. They use information from the dictionaries that are automatically generated by G.O.D.
- **Specific converters**
  - Possible, but well beyond the scope of this tutorial
  - Allow optimization in transient and/or persistent world



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:

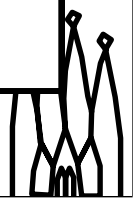
•A generic conversion mechanism, which uses object serialization (e.g. from POOL/Root). This sort of serialization uses generic converters and class specific dictionaries that are generated automatically by the G.O.D. tools

•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, for example it could allow to better minimize I/O (pack doubles into short by reducing the dynamic range, recalculate certain redundant data etc.). In the transient world such a converter could also add additional data for improved navigation.

The Remaining Machinery

Setup in the job options

```
// Services and dictionaries for reading/writing event data
#include "$STDOPTS/simDicts.opts"
// Additional dictionary for user defined classes
PoolDbCacheSvc.Dlls += {“TutKernelDict" };
// Define the data to write out and the output file
ApplicationMgr.OutStream = { "RootDst" };
RootDst.ItemList  = { "/Event#1",
                      "/Event/MC#1",
                      "/Event/MC/Header#1",
                      "/Event/MC/PrimaryVertices#1“ };
RootDst.Output    = "DATAFILE='RootDst.root'
                     TYP='POOL_ROOTTREE' OPT='REC'";
```

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 read and write objects (GaudiPoolDbRoot.opts).

•It must be told the list of dictionaries to load that describe all the classes that will be read or written (PoolDicts.opts for standard LHCb event classes, ComponentsDict for the class we will create in this tutorial)

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

•The output file must be defined, with the appropriate technology TYPe (POOL_ROOTTREE) and access rights (RECreate for read/write).

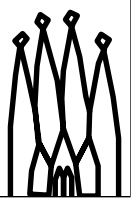
The two lines highlighted are needed also for reading LHCb data

You also need to add some magic lines in the requirements file, to activate the GOD machinery and generate the dictionaries, these are the six lines that are commented in the requirements file of Tutorial/Components package. These lines are created automatically by emacs if you edit a new requirements file in a package whose name contains the string “Event”.

Hands on: VertexInfo.xml

```
cd ~/cmtuser
getpack Tutorial/TutKernel
cd Tutorial/TutKernel/v1r0
```

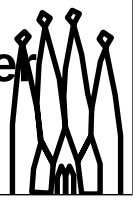
- **Look at the file in xml directory**
  - **Understand it and adapt it to your needs**
- **Look at the cmt/requirements file**
  - **Understand it**
  - **Use it to create the VertexInfo.h header file**
  - **Look at the generated header file**



Hands On: write persistent data

```
cd ~/cmtuser/Tutorial/Components/v7r0/src
```

- **Modify VisibleEnergy.cpp to:**
  - **create VertexInfo objects**
  - **Add them to a VertexInfos KeyedContainer**
  - **Register the VertexInfos on the transient event store at the default location**
    - See slide 6-8
- **Write the object to a ROOT file**
  - See slide 6-10
- **Browse the output file with the Root browser**



Hands On

In this exercise we try to invent a new object where we intend to store primary vertex information. As there may be several primary vertices in an event (pileup), we should have a container that can hold several such objects.

The VertexInfo could contain e.g. the number of MC particles, the total neutral energy of the vertex, and a reference to the MCVertex object.

The container must then be registered to the data store.

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

Solution

In src.data directory of Tutorial/Components package

To try this solution and start next exercise from it:

Uncomment Tutorial 3 options in $MAINROOT/options/jobOptions.opts

```
cd ~/cmtuser/Tutorial/Component/v7r0/src
```

Move your modified files if you want to keep them

```
cp ../src.data/*.* .
cd ../cmt
```

Uncomment `use TutKernel` directive in requirements

```
cmt broadcast gmake
source setup.csh
$MAINROOT/$CMTCONFIG/Main.exe $MAINROOT/options/jobOptions.opts
```