4

Accessing Event Data

Gaudi Framework Tutorial, April 2006

Schedule:	**Timing**	**Topic**
	20 minutes	Lecture
	20 minutes	Practice
	40 minutes	Total

Objectives

After completing this lesson, you should be able to:

- **Understand how objects are delivered to user algorithms.**
- **Retrieve objects from the event data store within an algorithm.**
- **Use relationships between objects.**
- **Specify event data input.**

Lesson Aim

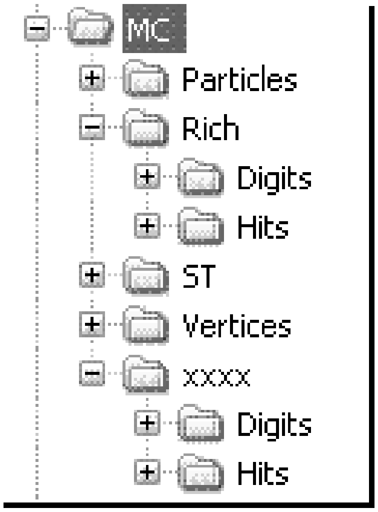
In the user algorithm, where the actual data processing takes place, these input data must be retrieved from the transient data store.

Running a Gaudi job usually means to process data from particle collisions. The concept how the data store delivers data to the user will be explained. You should be aware of the machinery behind in order to analyze failures.

Typically objects do not have a life on their own, but become powerful through their relationships. A typical example are generated Monte-Carlo particles which usually have an origin vertex and if they have finite lifetime also decay vertices. You will learn how to access these relationships.

These input data have to be specified to allow the job to access the requested data sets.

Event Data Reside In Data Store



Tree - similar to file system

Identification by path
”/Event/MC/Particles”
LHCb::MCParticleLocation::Default

Objects or

Containers of objects
KeyedContainer<Type>

4-3 Gaudi Framework Tutorial, April 2006

Event Data Reside In Data Store

Within Gaudi all event data reside in a data store.

- •Data and algorithms are separated.
- •Algorithms and data storage mechanisms are separated.

In other words, data have a transient and a persistent representation with not necessarily equal mapping. Opposite to having a single representation of “persistent capable” objects, this solution allows for optimisation depending on the demands of the chosen representation:

- •Persistent data are optimised in terms of persistent storage allocation, including e.g. data compression, minimisation of space used by bi-directional links
- •Transient data are optimised according to the required performance; this includes e.g. duplication of links, which are followed very often.

Data that either already have a persistent representation or that are intended to be written to a persistent medium reside in a transient data store, which acts like a library storing objects for the use of clients. These data stores are tree like entities, which can be browsed, just like a normal file system. Its full path uniquely identifies any transient representation of an object within a store. This browse capability is used to retrieve collections of objects to be made persistent. Of course the internals of the data store are not directly exposed to the algorithms, but rather hidden behind a service, the persistency service. This service acts as a secretary delivering objects to the client - if the at all possible.

Containers: e.g. KeyedContainer

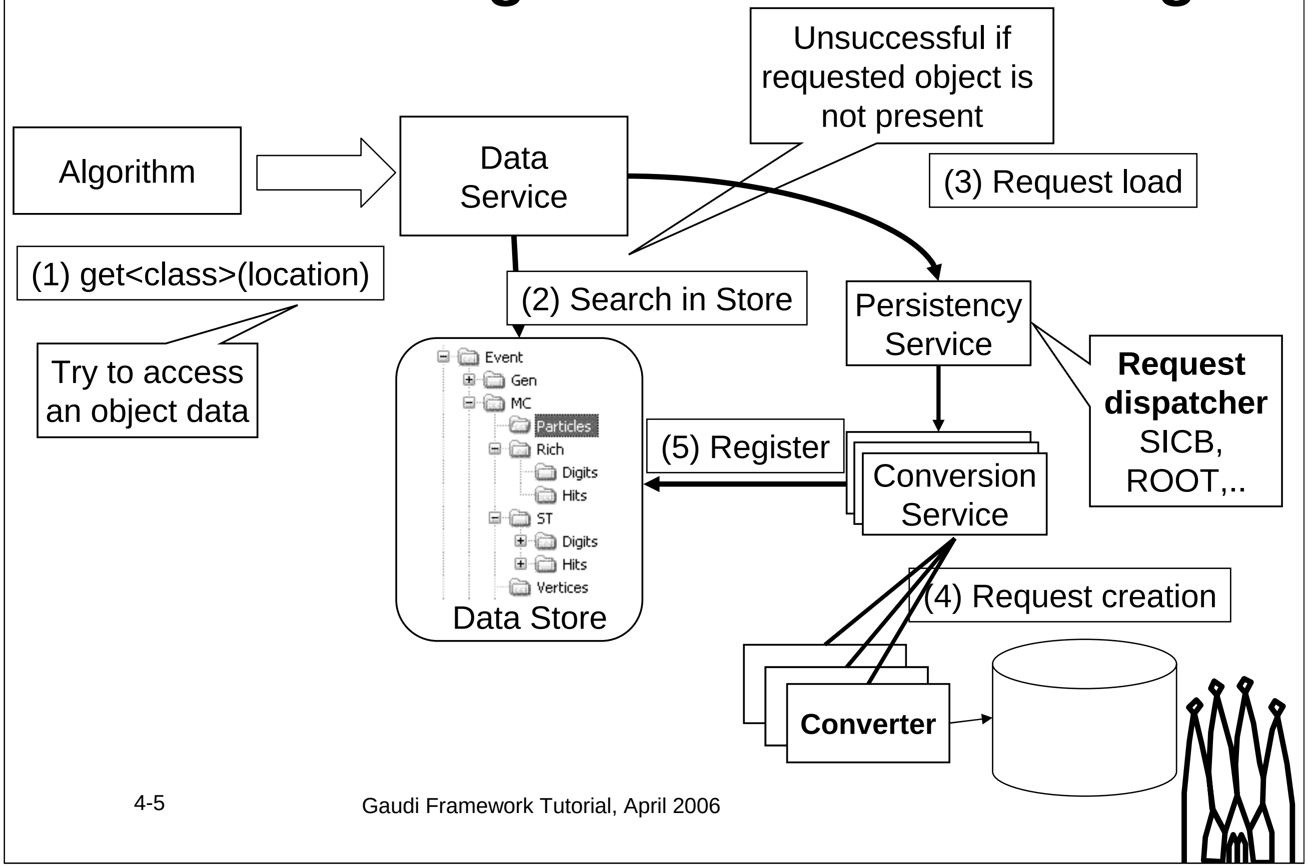
- **Templated class**
  - *Usually hidden to end-user code by typedefs in header file*

```
typedef KeyedContainer<MCParticle, Containers::HashMap> MCParticles;
```

- **Iteration like any STL vector**

```
LHCb::MCParticles* parts; //note LHCb:: namespace
...
LHCb::MCParticles::const_iterator i;
for( i=parts->begin(); i != parts->end(); i++ ) {
  info() << (*i)->particleID().pid() << endmsg;
}
```

Understanding Data Stores: Loading



4-5 Gaudi Framework Tutorial, April 2006

Understanding Data Stores: Loading

Whenever a client requests an object from the data service the following sequence is invoked:

- •The data service searches in the data store whether the transient representation of the requested objects already exists. If the object exists, a reference is returned and the sequence ends here.
- •Otherwise the request is forwarded to the persistency service. The persistency service dispatches the request to the appropriate conversion service capable of handling the specified storage technology. The selected conversion service uses a set of data converters - each capable of creating the transient representation of the specified object type from its persistent data.
- •The data converter accesses the persistent data store, creates the transient object and returns it to the conversion service.
- •The conversion service registers the object with the data store, the sequence completes and the object is returned to the client. Once registered with the corresponding data store, the object knows about its hosting service.
- •A recent possibility is to declare an algorithm to create the data on demand if it cannot be loaded. This DataOnDemandSvc is beyond the scope of this tutorial

Caveats

Consider Objects on the store as READ-ONLY

- **Do not modify existing objects!**
- **Do not destroy existing objects!**

Never, never delete an object which is registered to the store

- **It's not yours!**
- **It will only screw up others!**

4-6 Gaudi Framework Tutorial, April 2006

Caveats

Once an object is registered to the data store you should no longer consider it as yours. In particular changing containers is an absolute *don't.*

The data store also manages objects. An object created with *new* uses system memory. Once an object is registered to the data store, the data store is responsible for calling once and only once the corresponding *delete* operator. Deleting objects twice typically results in an access violation.

Although it is possible, you should never unregister an existing object for a simple reason:

•You never know who holds a reference to this object
(and typically there is no way to find this out). All these references will be invalid.

Data Access In GaudiAlgorithm

```
const LHCb::MCHeader* evt = get<LHCb::MCHeader>(
                    LHCb::MCHeaderLocation::Default );

// No need to test a return code, this method throws
// an exception if data is not found
```


Data Access In Algorithms

The GaudiAlgorithm and GaudiTool base classes have templated methods to simplify access to the event and detector data. Note that the get method throws an exception if the data is not found, which is caught by the base class. This exploits a convention of the LHCb data model: containers must be always present, even if they are empty (e.g. if no tracks found in a minimum bias event) – so the absence of a container, when it is expected, is an error. If you know that the data you are looking for may not be there, check first the existence with bool GaudiAlgorithm::exist<T>() method

In older code, instead of get<class-type> methods, you may see requests to the eventSvc() to return a SmartDataPtr<class-type>, which is then cast to a normal C++ pointer whose validity must be checked.

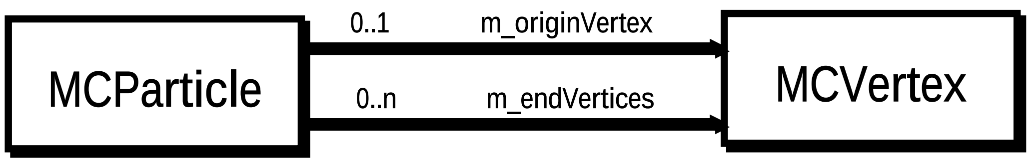
This is now hidden from users but it is useful to understand the underlying data access mechanism. The SmartDataPtr class can be thought of as a normal C++ pointer having a constructor, and is used in the same way as a normal C++ pointer. It is a “smart” pointer because it allows to access objects in the data store. The SmartDataPtr checks whether the requested object is present in the transient store and loads it if necessary. It uses the data service to get hold of the requested object and deliver it to the user.

Note:

Before the object is delivered to the user, a type check is performed. This ensures that the type in the data store actually is the same as specified by the user.

Relationships Between Objects

- **Objects have relationships between each other**



- **MCVertices are also connected to MCParticles, but this we neglect here...**

Relationships Between Objects

Objects do not only have a life on their own, but become powerful through their relationships. A typical example are generated Monte-Carlo particles which usually have an origin vertex and if they have finite lifetime decay vertices.

The relationships can have different multiplicity: 0, 1 or many. Normally these relationships are implemented either as pointers or as arrays of pointers. However, this has the disadvantage, that these pointers cannot be made persistent because the next time the program starts the referred object could be located in a completely different part of the memory. Where this will be is unpredictable: it depends on the number of users currently logged in, the number of tasks running etc.

For this reason Gaudi uses another mechanism, which allows on one hand persistency, but on the other hand is the usage sufficiently similar to a raw pointer.

Implementing Relationships

- **0..1 Relationships can be implemented using a *SmartRef<class-type>***
- **Usage similar to a normal pointer**
- **0..n Relationships are implemented using a *SmartRefVector<class-type>* …an array of *SmartRef<class-type>***
- **Allows for late data loading**
  - Only loaded when SmartRef is dereferenced

Implementing Relationships

The (Gaudi-)equivalent of the pointer between objects on the data store are SmartRefs. Similar to the SmartDataPtr this is as well a template class. When de-referencing (e.g. using the operator -> ()) the object behind is requested from the datastore and delivered to the user.

0..many relationships are implemented using arrays of these objects, or to be precise a SmartRefVector. This vector behaves like a std::vector from the STL library.

This allows for late object loading only when the pointer actually is used, but though have a consistent view.

Using These Relationships

```
class MCParticle  {
  ...
  SmartRef<LHCb::MCVertex> m_originVertex;
  ...
  LHCb::MCVertex* originVertex()  {
    return m_originVertex;
  }
};
```

See Doxygen code documentation

…load data and trigger conversion

Using the Relationships

Ideally the SmartRefs are not visible outside the class.

In the above example the Smartref is automatically converted to the corresponding pointer which actually is returned to the client. The client only sees the raw C++ pointer. Possible side-effects from using the SmartRef directly are not propagated.

SmartRef, pointers, references

```
LHCb::MCParticle* p = new LHCb::MCParticle();
SmartRef<LHCb::MCParticle> ref;
```

```
ref = p;
```
Assignment from raw pointer

```
p = ref;
```
Assignment to raw pointer

```
Gaudi::LorentzVector& m_A  = p->momentum();
Gaudi::LorentzVector& m_B  = ref->momentum();
Gaudi::LorentzVector m_slow = ref->momentum();
```


Using SmartRef<type>

The usage of the SmartRefs and raw pointers is interchangeable. You can assign the pointer to the smart reference as well as the reverse.

The overloaded “->” operator allows the same usage like a pointer.

Note:

The “&” at the bottom makes a big difference:

Gaudi::LorentzVector& m_A and m_B are aliases to the object behind, whereas Gaudi::LorentzVector m_slow is a *copy* of the particle’s 4-momentum. Both is absolutely perfect C++ code. However, not paying attention to details like this can easily account for very efficiently executing code and very poor performance. Usually in this case the language is claimed to be “bad”, whereas in practice it’s the programmers fault.

Specify Event Data Input

```
EventSelector.Input = {
  “DATAFILE=‘a_filename’ [Spec]”
[,“DATAFILE=‘another_filename’ [Spec]”]
 };
```

- **Event data input is specified in the *job options***
- **[Spec] is an array of qualified strings: *KEY1=‘VALUE1’ … KEYn=‘VALUEn’***
- **Several files can be specified, separated by a comma**

Specify Event Data Input

Event Data Input is specified in the job options and is a property of the EventSelector. The property is a vector of qualified strings of the form

“KEY1=‘VALUE1’ … KEY2=‘VALUE2’ “, // Specification of the first input

“ ….” // Next input, etc

Note:

- A *key* refers to an individual information necessary to open the specified input source.
- A *value* is the information content corresponding to this key.
- *Values* are enclosed within **single quotes** (‘).
- One data input stream is enclosed between **double quotes** (“). The input stream is specified through at least one *key-value* pair.
- The square brackets [] above are indicating optional arguments, they are not part of the syntax

Specify POOL Event Input

```
EventSelector.Input = {"DATAFILE='PFN:castor:
/castor/cern.ch/grid/lhcb/production/DC06/v1-lumi2/00001284/SIM/00001284_00000001_1.sim'
 TYP='POOL_ROOTTREE' OPT='READ'"
}
```

- **PFN: keyword tells POOL this is a *physical file name***
- **castor: keyword selects data transfer protocol (rootd in this case)**
  omit for a disk file
- **OPT='READ' is read only file**

4-13 Gaudi Framework Tutorial, April 2006

Specify POOL Event Data Input

Files are specified by the key *DATAFILE* followed by the file name.

For disk files, the file name can be relative or absolute to the execution directory:

'PFN:./myfile.dst'

The available official datasets can be found in the LHCb bookkeeping database using the corresponding web page: http://lhcb-comp.web.cern.ch/lhcb-comp/bookkeeping. Once you have selected a dataset, you can ask the web interface to create the data cards for you.

Hands On: Print B^0 Decays

- **Use particle property service**

```
m_ppSvc = svc<IParticlePropertySvc>( "ParticlePropertySvc", true );
```

  – See “User Guide” chapter 11.5 for usage

- **Retrieve MCParticles from event store**
  – LHCb::MCParticleLocation::Default

- **Filter out the B^0 particles**
  – MCParticle::particleID().pid() is PDG particle code

- **For each B^0 Loop over all decay vertices and print the decay particles**
  – recursion ?

Hands On: Print B^0 Decays

In the following tutorial we will try to extract from the MC truth the B^0 particles and try to print the entire decay tree. The required actions are the following:

•Filter out all B^0 particles.

•For each B^0 loop over all decay vertices and print the daughter particles.

•If the daughters have decay vertices themselves, recurse the second step.

DecayTreeAlgorithm.cpp: Add Headers

```
// Using Particle properties
#include "GaudiKernel/IParticlePropertySvc.h"
#include "GaudiKernel/ParticleProperty.h"

// Accessing data:
#include "Event/MCParticle.h"
#include "Event/MCVertex.h"
```

Using IParticlePropertySvc

```
...DecayTreeAlgorithm.h...
  IParticlePropertySvc* m_ppSvc;
  std::string           m_partName;
  int                   m_partID;
...DecayTreeAlgorithm::initialize()...
  m_ppSvc = svc<IParticlePropertySvc>( "ParticlePropertySvc",
                                       true );
  ParticleProperty* partProp = m_ppSvc->find( m_partName );
  if ( 0 == partProp )   {  // You have to handle the error!
  }
  m_partID = partProp->pdgID();
```

Hands On: Retrieve and loop over MCParticles

```
LHCb::MCParticles* parts = get<LHCb::MCParticles>(
               LHCb::MCParticleLocation::Default );

LHCb::MCParticles::const_iterator i;
for(i=parts->begin();i != parts->end(); i++ ) {
  if ( (*i)->particleID().pid() == m_partID ){
    printDecayTree( *i );
  }
}
```

New member function

Hands On: Print Decays

For each selected particle:

- **Loop over decay vertices**
- **Print all daughters**
  - **If daughters have decay vertices**
  - **recurse**
- **If you run out of time, just print some particle property**

Loop Over Decay Vertices

```
const SmartRefVector<LHCb::MCVertex>& decays = mother->endVertices();
SmartRefVector<LHCb::MCVertex>::const_iterator ivtx;
for ( ivtx = decays.begin(); ivtx != decays.end(); ivtx++ )  {
  const SmartRefVector<LHCb::MCParticle> daughters=(*ivtx)->products();
  SmartRefVector<LHCb::MCParticle>::const_iterator idau;
  for( idau = daughters.begin(); idau != daughters.end(); idau++ ) {
    printDecayTree( depth+1, prefix+" |", *idau );
  }
}
```

Solution

- In src.decaytree directory of Tutorial/Components package
- To try this solution and start next exercise from it:

Uncomment Tutorial 1 options in $MAINROOT/options/jobOptions.opts

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

Move your modified files if you want to keep them

```
cp ../src.decaytree/*.* .
cd ../cmt
gmake
$MAINROOT/$CMTCONFIG/Main.exe $MAINROOT/options/jobOptions.opts
```