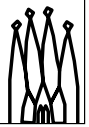
3

From XML to C++ Objects



Gaudi Framework Tutorial, 2001

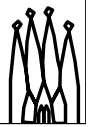
This part of the tutorial deals with XML converters that converts the XML persistent representation into C++ objects in the transient data store. It will be followed by a small exercise, aiming at converting the geometry we defined in the previous lesson into C++ and reaching it from an algorithm. We will also learn how to use userParameters and userParameterVectors to customize a bit the detector elements.

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

Objectives

After completing this lesson, you should be able to do the following:

- **Access the geometry in the C++ algorithms**
- **Use userParameters(Vector) to customize detector elements**



3-2 Gaudi Advanced Tutorial, 2001

Lesson Aims

• Access the geometry in the C++ objects : the different existing objects in C++ and their interfaces were actually presented in the first lesson of this tutorial. We only explain here how to get them.

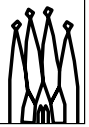
• Use userParameter and userParameterVector to customize detector elements. This is a special feature that allows the user to define his own set of parameters inside detector elements.

Accessing the Geometry in C++

- **automatic, by defining a SmartDataPtr :**

```
SmartDataPtr<IDetectorElement>
  cave(detSvc(),"/dd/Structure/LHCb" );
if (!cave) { /* error message */ }
```

- **do not forget to test the pointer, the conversion is processed at this time**

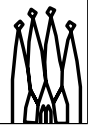


Accessing the geometry in C++

As in the whole Gaudi software, you access transient stores by using SmartDataPtr. Once more, you should test them in order to launch the retrieval of the data (and the conversion from XML into C++ in our case).

Reading XML Files

- **Converters used to build C++ objets from XML**
- **One converter per object type**
  - XmlDetectorElementCnv
  - XmlLVolumeCnv
  - XmlMixtureCnv
  - XmlMuonStationCnv
  - …
  - XmlMySubDetCnv
- **almost 1 to 1 mapping between XML elements and C++ objects**



Reading XML files

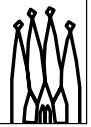
In order to build the C++ objects from the transient store, Gaudi provides a set of converters that maps XML elements into C++ objects.

There is one converter per C++ object type. Some of them are XmlDetectorElementCnv, XmlLVolumeCnv and XmlMixtureCnv which convert respectively detector elements, logical volumes and materials. Some more specific converters can be defined when necessary like XmlMuonStationCnv for Muon stations or the XmlMySubDetCnv that we will define in a further exercise.

As you can see, the converters map XML elements to C++ objects in an almost one to one mapping.

First Summary

- We are able to reach the geometry description from the C++ transient world
- Everything is transparent for the C++ user, there is no need to know it comes from XML
- BUT we have no way to extend the schema and especially to add specific parameters to a detector element



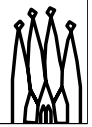
First summary

So far, we are able to describe geometry and reach it from the C++ world, using the standard converters provided by Gaudi. The use of these converters is actually fully transparent for the user.

The only problem here is that the user has no way to extend the schema and especially to add specific parameters into detector elements.

Specializing Detector Elements

1. adding userParameter(vector)s to default DetectorElements
2. extending and specializing the DetectorElement object in C++, using userParameters in XML
3. extending XML DTD and writing a dedicated converter



Specializing Detector Elements

There are mainly three ways of specializing detector elements.

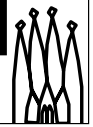
1. the first and less complicated one is to add userParameters to the detector element in the XML code. This will be detailed in the end of this lesson.
2. the second is to extend and specialize the DetectorElement object in C++. This allows to add new members and methods. The initialization of this new object uses then the userParameters defined previously. This will be detailed in the next lesson.
3. the last and most complicated way is to extend the XML DTD to allow specific XML elements and store complex information. This will need to write a dedicated converter and will be detailed in the next lesson.

Specializing by using userParameter[Vector]

- **Two elements :**
  <userParameter> and <userParameterVector>
- **3 string attributes : name, type and comment**
- **One value given as text**

```
<userParameter
      comment="blablabla"
      name="description"
      type="string">
  Calibration channels
</userParameter>
```

```
<userParameterVector
      name="NbChannels"
      type="int"
      comment="blabla">
  530  230
  570 270
</userParameterVector>
```



Specializing by using userParameter and userParameterVector

These are two elements of the LHCb structure DTD that allow the user to add his own parameters to a given detector element. These elements have three attributes defining the parameter :

- name : this will be the only way to retrieve the parameter in C++
- type : this has no restriction but only int, double and string are recognized. All other types are treated as strings.
- comment : you are free to put here a small explanation of the meaning and usage of the parameter

The value of the parameter is the value of the element itself, which is the text appearing between the opening and the closing tag. In the case of a vector, the different values should be separated by spaces and/or carriage return only. If the type of a parameter is int or double, the value will be computed using the expression evaluator of the XmlCnvSvc. Thus parameters (I mean the one defined via the element <parameter>, not user parameters), units and mathematical functions can be safely used.

C++ API for userParameters (1)

Methods on DetectorElement for userParameters :

- **`string userParameterAsString (string name)`**
- **`double userParameterAsDouble (string name)`**
- **`int userParameterAsInt (string name)`**

The equivalent exist for userParameterVectors

```
std::string description = elem->userParameterAsString ("description");
std::vector<int> channelNbs = elem->userParameterVectorAsInt ("NbChannels");

log << MSG::INFO << description << " : ";
for (std::vector<int>::iterator it = channelNbs.begin();
      it != channelNb.end();
      it++)
  log << *it;
log << endreq;
```

C++ API for userParameters

Since the userParameters and userParameterVectors are defined in the LHCb DTD, there are converted by the default converters and can be retrieved in the C++ world using the following API (these are all methods of the DetectorElement object, of course) :

- string userParameterAsString (string name) : gets the value of a given parameter, as a string. Actually returns the exact string that was in the XML code.
- double userParameterAsDouble (string name) : gets the double value of a given parameter. This is computed from the string value, by calling the expression evaluator. Note that this method only deals with parameters of type int and double. Thus it will tell you there is no such parameter if it is called for a parameter of another type.
- int userParameterAsInt (string name) : gets the integer value of a given parameter. This is computed from the string value, by calling the expression evaluator. Note that this method only deals with parameters of type int. Thus it will tell you there is no such parameter if it is called for a parameter of another type.

The equivalent methods exist for userParameterVectors. The only difference is that they return vectors of strings, doubles or ints. Here are the signatures :

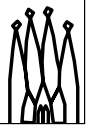
- vector<string> userParameterVectorAsString (string name);
- vector<double> userParameterVectorAsDouble (string name);
- vector<int> userParameterVectorAsInt (string name);

C++ API for userParameters (2)

More methods :

- `double userParameter (string name)`
- `vector<string> userParameters()`
- `string userParameterType (string name)`
- `string userParameterComment (string name)`

- `vector<double> userParameterVector (string name);`
- `vector<string> UserParameterVectors();`
- `string userParameterVectorType (string name);`
- `string userParameterVectorComment (string name);`



C++ API for userParameters

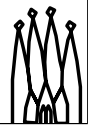
Here are some more methods that allow to access the type and comment associated with a parameter or the list of defined parameters :

- double userParameter (string name) : equivalent to userParameterAsDouble.
- vector<string> userParameters() : returns a vector of the defined parameters.
- string userParameterType (string name) : gets the type of a given parameter. Actually returns the exact string that was given for this attribute in the XML code.
- string userParameterComment (string name) : gets the comment of a given parameter. Actually returns the exact string that was given for this attribute in the XML code.

The equivalent methods exist for userParameterVectors. Here are the signatures :

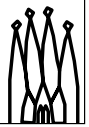
- vector<double> userParameterVector (string name);
- vector<string> userParameterVectors();
- string userParameterVectorType (string name);
- string userParameterVectorComment (string name);

Exercise 2



Exercise Goal

- **Add the number of channels of each detector element together with a short description of these**
- **Write an algorithm that is able to display the geometry we described and the data concerning the channels in each detector element**



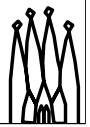
Exercise goal

The goal of the second exercise is to extend the description of mySubDet we did in the first exercise and to reach it from the C++ world.

The extension consists in adding information on the number of channels in the XML definition of detector elements.

How to Start

- **reuse what you did in the previous exercise or copy data/DDDBSolution1 into data/DDDB**
- **copy the src2 directory into src**
  - **it includes empty files for the algorithm (called AccessGeoAlgorithm)**



How to start

This is what you have to do in order to start working on this second exercise :

- if you did not succeed in the first exercise, you should start with a valid geometry by copying the data/DDDBSolution1 directory into data/DDDB :

  mv data/DDDB data/myDDDB

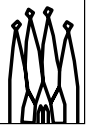
  cp –r data/DDDBSolution1 data/DDDB
- then, you must copy the src2 directory into src :

  cp –r src2 src

This provides empty files for the AccessGeoAlgorithm you will have to write

Hints for the XML

- **Just define two userParameter per detector elements :**
  - **one with name “description” and type string**
  - **the other with name “nbChannels” and type int**
- **Once you define two of them on the first element, just cut and paste them 4 times and modify the values**



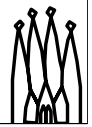
Hints for the XML

The only modification to do in the XML you wrote in the first exercise is to add two userParameter elements per detector elements (maybe not the main one), one called description and the other called NbChannels. Feel free to change the names, of course.

One trick is that you should define these parameters on one detector element first and copy and paste them in every other. You can then change the values and the names and comments are already ok.

Hints for the Algorithm

- **There are only 5 places where you should add code in the prepared file, after the big comments**
- **Don’t try to do all at once, you should be able to compile without filling the 3 display methods**
- **To compile your algorithm, go to the cmt directory and issue a `make` command**



Hints for the algorithm

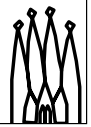
The algorithm is called AccessGeoAlgorithm. The .h and .cpp files are given but the interesting parts are empty and you should fill these blanks.

There are 5 places where code should be added. You can find them easily due to the big comments explaining what to add, just before each blank. Note that you are not obliged to fill all blanks for your first try. The display methods can be left empty for a first compilation.

Once you are done, at least with the initialize method, you can type `make` to compile your algorithm.

How to Run Your Algorithm

- **`source $LHCBHOME/scripts/tutorial.csh`** **(if new shell)**
- **`cd tutorial`**
- **`getpack Tutorial/Main v3`**
- **`cd Tutorial/Main/v3/cmt`**
- **uncomment "use Detector …" in the requirements file**
- **`cmt config`**
- **`source setup.csh`**
- **`make`**
- **`../i386_linux22/Main.exe ../../../Detector/v3/options/AccessGeo.opts`**



How to run your algorithm

In order to run you algorithm, you need to drive it from an executable. This is provided in tutorials by the Tutorial/Main package. Thus, here is what you should do :

- if you open a new shell for running
  source $LHCBHOME/scripts/tutorial.csh
- then go to the tutorial directory
  cd tutorial
- get the Main package, configure it and compile it :
  getpack Tutorial/Main v3
  cd Tutorial/Main/v3/cmt
  uncomment "use Detector …" in the requirements file
  cmt config
  source setup.csh
  make
- run the main program :
  ../i386_linux22/Main.exe ../../../Detector/v3/options/AccessGeo.opts