3

Printing and Job Options

Schedule:

Timing	Topic
15 minutes	Lecture
20 minutes	Practice
35 minutes	Total

Objectives

After completing this lesson, you should be able to:

- **Know how to print.**
- **Know how to implement job options in an algorithm.**
- **Know basic job options.**

Lesson Aim

Printing obviously helps to understand what is going on inside a program. Although it can never replace a real debugger it is essential e.g. in batch application to know why certain actions failed. In the following printing in Gaudi is introduced and the usage is shown.

Any Gaudi job is steered by a setup file called the *job options*. It will be shown how you can customize an algorithm with properties. These properties allow you to change the behavior at run-time and allow for more flexibility.

Also a few standard job options will be introduced.

Job Options

- Job is steered by “cards” file
- Temporary: Database on the long run
- Options are not directly accessed
- Access through IJobOptionsSvc interface

Job Options

Typically every analysis job is steered by a *cards* file. *Cards* historically were *real cards*, meaning punch cards used to pass parameters from some input device to the program.

Gaudi for the time being uses the same mechanism. However, in the long run this is seen to be not sufficient: Typically they are only used to override default values - in the future these options will likely be stored in a database. This will then allow to access all properties of algorithms e.g. for official productions.

For this reason the additional options are only accessed using a special service, which exposes the *IJobOptionsSvc* interface. Because of this separation it will only be this service, which needs to be changed.

Job Options: Data Types

Primitives

- bool, double, long, std::string

Arrays of primitives

- std::vector<bool>, std::vector<double>...

Job Options: Data Types

Objects like algorithms and services can retrieve options of several data types from the job option file. These are primitive options like bools, doubles etc. and arrays of those.

Using Job Options

Declare property variable as data member

```
class DecayTreeAlgorithm : public Algorithm  {
private:
  std::string m_partName;
   ...
};
```

Declare the property in the Constructor

```
DecayTreeAlgorithm:: DecayTreeAlgorithm( <args> )
<initialization>
{
  declareProperty( "DecayParticle", m_partName = "B0");
}
```

3-5 Gaudi Framework Tutorial, 2001

Using Job Options

Optional parameters of an algorithm are part of the algorithm itself. In C++ they are typically implemented as member variables.

However, the framework must be made aware, that a given algorithm has a certain property and that the value of this property may be changed.

Property defaults may sometimes be useful. However, if a default value can not ensure proper behavior, it may be better to require external input.

Set Job Options

Set options in joboptions file

- File path is first argument of executable
- C++ like syntax
- Example

```
B0_Decays.DecayParticle = “B0”;
D0_Decays.DecayParticle = “D0”;
```

- Object name (Instance not class)
- Property name
- Property value

3-6 Gaudi Framework Tutorial, 2001

Set Job Options

The job options file itself is passed to the executable as the first argument.

The job options have C++ like syntax. This means in particular

•A property of an algorithm is addressed using the following syntax:
 <object-name>.<option-name> = <value>;

•Any option is terminated by a semi-colon (;).

•Strings are encolsed in double quotes (“value”).

•Arrays of options are enclosed in curly brackets. Example: SomeAlg.SomeOpt = {1, 2, 3, 5, 6};

•Job options are assigned to an object according to the name if the *instance,* not at the level of the class.

Job Options: Conventions

Many algorithms need many options

- **Options go along with code**
  - New code release may need different options
  - Must be configurable with cmt
- **Need for conventions**
  - Look how Brunel organizes options

Job Options: Conventions

When talking about large applications such as a reconstruction program, it is clear that many different algorithms are involved. Currently in SICB there is one big area, where all these options are stored in form of cdf files (dbase). Because of this complete separation this area changes version very often.

Typically the release of new code goes along with changed options. These options, must be configurable with cmt. For example if you want to work on a selected package, it is very likely that you will also change the required options without having to check out yet another package and at the end worry how your changes will affect existing code. For this reason the same conventions were adopted.

To have a coherent picture over the organization of job options, the Brunel approach should be adopted.

Brunel: Conventions

Brunel will then include the options according to the detector

- **$CALOALGSROOT/options/Brunel.opts**

```
BrunelDigiCALOSeq.Members += { …
      "CaloDigitisationAlgorithm/EcalDigi",
      "CaloDigitisationAlgorithm/HcalDigi"   };

#include "$CALOALGSROOT/options/EcalDigi.opts”
#include "$CALOALGSROOT/options/HcalDigi.opts"
```

Brunel: Conventions

Brunel - by convention - expects an option file in a directory called “<detector>Algs”; the corresponding environment variable, which also contains the proper version number is set up by cmt. The required file name is called Brunel.opts.

Within this option file each subdetector can the set up the assigned sequences for each phase. An example is given above for the calorimeters.

Job Options You Must Know

```
#include "$STDOPTS/Common.opts"
ApplicationMgr.EvtMax           <integer>
ApplicationMgr.DLLs             <Array of string>
ApplicationMgr.TopAlg           <Array of string>
```

- **Standard Configuration**
- **Maximal number of events to execute**
- **Component libraries to be loaded**
- **Top level algorithms: "Type/Name"**
  "DecayTreeAlgorithm/B0_Decays"
  This also defines the execution schedule

Job Options You Must Know

There is a standard set of job options, which must be supplied for *every* Gaudi task. A set of standard options, which depends on the Gaudi release for LHCb should always be included.

The others are listed below. During the tutorial other options will be introduced as well, which you should add to this list to be kept in (brain-)memory.

Printing

Why not use std::cout, std::cerr, ... ?

- **Yes, it prints, but**
  - **Do you always want to print to the log file?**
  - **How can you connect std::cout to the message window of an event display?**
  - **You may want to switch on/off printing at several levels just for one given algorithm, service etc.**

Printing

Print statements are a very useful way to document checkpoints within a running program. C++ by itself implements three standard output streams, which in practice all go to the terminal output:

•std::cout, the standard output destination

•std::cerr, for logging errors

•std::clog, for debugging

These printout destinations however have some disadvantages

•They all go to log files, a more fine grained specification of the destination is not possible.

•Although possible it is e.g. not too obvious how to redirect output properly e.g. to an error logger display in the online environment.

•You may want to switch on debug printing

  •For the algorithm/service you want to debug and you do not want to get flooded by all the printouts of other algorithms

  •You want to globally adjust the level of severity for printout.

To summarize, there are quite some reasons why the standard printing may not be entirely adequate.

Printing - MsgStream

Using the MsgStream class

- Usable like std::cout
- Allows for different levels of printing

– MSG::DEBUG	(=1)
– MSG::INFO	(=2)
– MSG::WARNING	(=3)
– MSG::ERROR	(=4)
– MSG::FATAL	(=5)

- Record oriented
- Allows to define severity level per object instance

3-11 Gaudi Framework Tutorial, 2001

Printing - MsgStream

The alternative to using the default print streams defined by C++ is a Gaudi extension, the MsgStream. The usage of this class should be the same as for the standard streams. The MsgStream however, allows to specify more fine grained severity levels:

•Debug, Informational, warning, error and fatal levels.

Secondly, printout of the MsgStream class is record oriented, not line oriented like for the C++ output streams. Standard output streams print whenever a *newline* character appears. The MsgStream prints on the occurrence of an end-record specifier. A record may contain several lines of output.

MsgStream objects allow to define the severity level based on the name of an object instance. This feature allows to enable printouts for one single algorithm while suppressing extensive printout for others.

MsgStream - Usage

Add Header file

```
#include “GaudiKernel/MsgStream.h”
```

Create object and print

```
MsgStream log(msgSvc(), name());
log << MSG::INFO << “Hello world!” << endreq;
```

Set printlevel in job options

```
MessageSvc.OutputLevel   = 4; // MSG::ERROR
MySvc.OutputLevel        = 3; // MSG::WARNING
MyAlgorithm.OutputLevel  = 2; // MSG::INFO
```

3-12 Gaudi Framework Tutorial, 2001

MsgStream - Usage

In order to use the MsgStream class you first have to include the appropriate header file.

Then you must create the object for printing. The constructor of a MsgStream object takes two arguments: a reference to the MessageSvc, and an identifier, which is typically the algorithm’s name.

For printing itself, you must then first pass the level of the current record, the values to dump and finally the end-of-record stream modifier.

In the job options you can then specify the output level for your printout. In this example general printout is only done for messages with a severity ERROR or higher. However, for the service instance “MySvc” also warning messages will be printed and for the algorithm “MyAlgorithm” even informational messages.

Note:

•The name specified in the job options and the name of MsgStream object you use *must* be the same.

•MsgStream objects should only be used locally, wherever you need them. Do not use them as member variables etc.

Hands On: DecayTreeAlgorithm

Introduce a property

- std::string called “DecayParticle”
- long   called “DecayDepth”

Print value in DecayTreeAlgorithm using

- MsgStream class
- several severity levels

Add algorithm instance to top alg list

- Name: B0_Decays

Hands On

You will introduce properties to the DecayTreeAlgorithm. This algorithm has an empty implementation we have already build.

Then these properties will be printed when the algorithm is initialized. This requires that the algorithm is instantiated, so it must be added to the list of top level algorithms.

Hands On: DecayTreeAlgorithm.h

```
class DecayTreeAlgorithm : public Algorithm {
private:
  /// Name of the particle to be analysed
  std::string           m_partName;
  /// Integer property to set the depth of printout
  long                  m_depth;
...
};
```

Hands On: DecayTreeAlgorithm.cpp

```
 DecayTreeAlgorithm::DecayTreeAlgorithm(
    const std::string& nam, ISvcLocator* pSvcLocator)
 :   Algorithm(nam, pSvcLocator)
 {
   declareProperty( "DecayParticle", m_partName   = "B0” );
   declareProperty( "DecayDepth",    m_depth      =  2   );
 }

StatusCode DecayTreeAlgorithm::initialize()   {
  MsgStream log ( msgSvc(), name() );
  log << MSG::DEBUG << “Decay Particle:” << m_partName
      << “Number of daughter generations in printout:”
      << m_depth
      << endreq;
}
```

Hands On: B0DecayTree.opts

```
// Add B0 decay algorithm to list of top level algorithms
ApplicationMgr.TopAlg += {"DecayTreeAlgorithm/B0_Decays"};

// Setup of B0 decay algorithm
B0_Decays.DecayParticle    = "B0";
B0_Decays.DecayDepth       =  2;
```

Hands On: If you have time left...

Extend for printout of D^0 decays

- **Re-use the existing implementation**