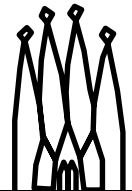


3

Job Options and Printing

Gaudi Framework Tutorial, April 2006

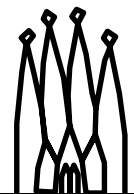


Schedule:	Timing	Topic
	15 minutes	Lecture
	20 minutes	Practice
	35 minutes	Total

Objectives

After completing this lesson, you should be able to:

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



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.

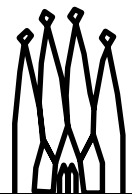
Also a few standard job options will be introduced.

Job Options

- **Job is steered by “cards” file**
- **Options are not directly accessed**
- **Access through IJobOptionsSvc interface**
 - **Details hidden from users by the framework**

3-3

Gaudi Framework Tutorial, April 2006



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 uses the same mechanism, by reading in one or more Job Options files. These files were foreseen as a temporary solution. In the long run, all job options might be stored in a database, which would facilitate manipulation of e.g. different production settings. Another future development that is foreseen is to code the job options in python.

The job options are accessed by the framework using a special service, which exposes the *IJobOptionsSvc* interface. Because of this separation, only the service will need to be changed if the options are moved to a database or to python scripts.

Job Options: Data Types

Primitives

- **bool, char, short, int, long, long long, float, double, std::string**

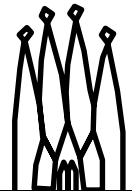
- And unsigned char, short, int, long, long long

Arrays of primitives

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

3-4

Gaudi Framework Tutorial, April 2006




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 GaudiAlgorithm {  
private:  
    std::string m_partName;  
    ...  
};
```

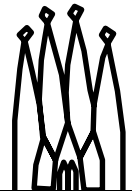
 LHCb convention

Declare the property in the Constructor, and initialize it with a default value

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

3-5

Gaudi Framework Tutorial, April 2006



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 job options file

- **File path is first argument of executable**

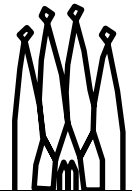
```
../$CMTDEB/Main.exe ../options/jobOptions.opts
```

- **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, April 2006

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 enclosed 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 of the *instance*, not at the level of the class.

Job Options: Conventions

Many algorithms need many options

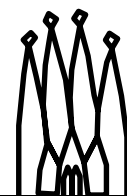
– Options go along with code

- New code release may need different options
- Must be configurable with cmt

– Need for conventions

3-7

Gaudi Framework Tutorial, April 2006



Job Options: Conventions

When talking about large applications such as a reconstruction program, it is clear that many different algorithms are involved.

There are two broad categories of options:

1. Options to configure the job (Sequence of algorithms to be executed, DLLs to be loaded)
2. Options to configure the *properties* of algorithms and tools (e.g. cuts, tuning parameters)

For ease of maintenance, the two types of options should be kept logically separated.

The job configuration options should be released in a job options file together with the code (depending on complexity this can be in a component package, or in the package defining a package group or a project)

The default values of the algorithm and tool properties should be in the code (declareProperty initializer) – only options different from the default need to be put in an external file (e.g. when several instances of an algorithm or tool are needed). Such a file, if needed, should be released in the same package as the algorithm.

In future, all job options files released with a project may be copied to a single installation directory. It will aid the migration if the file names of job options files are unique (jobOptions.opts is NOT a good name!)

LHCb conventions

LHCb applications organize sequencing of algorithms, then take specific options from corresponding algorithms package

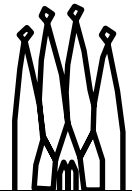
```
ApplicationMgr.DLLs += { "STAlgorithms" };
ApplicationMgr.TopAlg += {
    "MCSTDepositCreator/MCITDepositCreator" };
#include "$STALGORITHMSROOT/options/itDigi.opts"
```

Default values for the options should be hard-coded. Included options files modify these defaults.

```
MCITDepositCreator.tofVector = {25.9, 28.3, 30.5};
ToolSvc.STSignalToNoiseToolIT.conversionToADC = 0.0015;
```

3-8

Gaudi Framework Tutorial, April 2006



LHCb: Conventions

LHCb applications configure the job by loading the necessary component libraries (ApplicationMgr.DLLs) and by setting up the processing sequences for the algorithms. Any algorithm specific options are delegated to the algorithm packages.

The name of the processing sequences are set by convention. They consist of a program *phase* ("Digi") and the sub-system ("IT"), followed by "Seq".

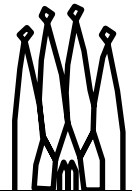
Job Options You Must Know

<code>ApplicationMgr.EvtMax</code>	<code><integer></code>
<code>ApplicationMgr.DLLs</code>	<code><Array of string></code>
<code>ApplicationMgr.TopAlg</code>	<code><Array of string></code>

- **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

3-9

Gaudi Framework Tutorial, April 2006



Job Options You Must Know

These options are essential in any job. During the tutorial other options will be introduced as well, which you should add to this list to be kept in (brain-)memory.

Job options printout

Contents of job options files is printed out when Gaudi starts.

- Control printing during processing:

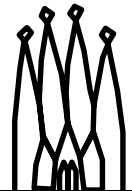
```
#pragma print off // Do not print options defined after this
#pragma print on  // Switch back on
```

- Print a single sorted list of all modified options:

```
#printOptions
```

3-10

Gaudi Framework Tutorial, April 2006



By default, Gaudi prints out the contents of the job options files as it processes them. The printing can be controlled by adding the following directives to job options files:

```
#pragma print off
```

Switches off printing of all options defined after this directive (can be nested)

```
#pragma print on
```

Switches on printing of all options defined after this directive (can be nested)

```
#printOptions
```

Do not print contents of options files during their processing. Instead, print a sorted list of all modified options at the end of the job options processing

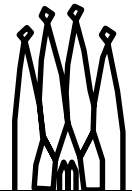
Printing

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.

3-11

Gaudi Framework Tutorial, April 2006



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.

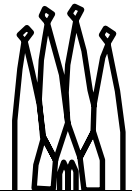
Printing - MsgStream

Using the MsgStream class

- Usable like `std::cout`
- Allows for different levels of printing
 - `MSG::VERBOSE` (=1)
 - `MSG::DEBUG` (=2)
 - `MSG::INFO` (=3)
 - `MSG::WARNING` (=4)
 - `MSG::ERROR` (=5)
 - `MSG::FATAL` (=6)
 - `MSG::ALWAYS` (=7)
- Record oriented
- Allows to define severity level per object instance

3-12

Gaudi Framework Tutorial, April 2006



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:

Verbose, Debug, Informational, Warning, Error and Fatal levels. Always is reserved for informational messages that should always be printed.

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

Send to predefined message stream

```
info() << "PDG particle ID of " << m_partName  
      << " is " << m_partID << endmsg;  
  
err() << "Cannot retrieve properties for particle "  
      << m_partName << endmsg;
```

Print error and return bad status

```
return Error("Cannot retrieve particle properties");
```

Formatting with format("string", vars)

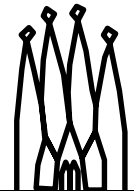
```
debug() << format("E: %8.3f GeV", energy ) << endmsg;
```

Set printlevel in job options

```
MessageSvc.OutputLevel    = 5; // MSG::ERROR  
MySvc.OutputLevel        = 4; // MSG::WARNING  
MyAlgorithm.OutputLevel  = 3; // MSG::INFO
```

3-13

Gaudi Framework Tutorial, April 2006



MsgStream - Usage

The GaudiAlgorithm and GaudiTool base classes hide the technicalities of creating MsgStream. Simply use the verbose(), debug(), info(), warning(), err(), fatal(), always() methods as in the examples above, passing the values to dump and the end-of-record stream modifier endmsg.

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: if you set the OutputLevel to 2 (debug), the base class will print the value of all the algorithm Properties during initialization

Caveat: The GaudiAlgorithm and GaudiTool base classes were only introduced recently. In looking at older code, you will come across explicit usage of the MsgStream as shown below. This should now be avoided:

Add Header file

```
#include "GaudiKernel/MsgStream.h"
```

Create object and print

```
MsgStream log(msgSvc(), name());  
log << MSG::INFO << "Hello world!" << endmsg;
```

Hands On: DecayTreeAlgorithm

Introduce a property

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

Print value in DecayTreeAlgorithm using

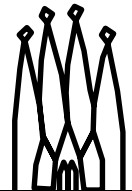
- accessors to `MsgStream` class
- several severity levels

Add algorithm instance to top alg list

- Name: `B0_Decays`

3-14

Gaudi Framework Tutorial, April 2006



Hands On

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

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: If you have time left...

Extend for printout of D^0 decays

- **Re-use the existing implementation**

Play with printing directives:

#pragma print off

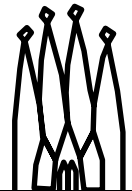
#pragma print on

#printOptions



Hands On: DecayTreeAlgorithm.h

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



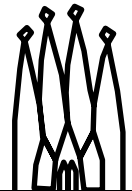
Hands On: DecayTreeAlgorithm.cpp

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

StatusCode DecayTreeAlgorithm::initialize() {
    debug() << "Decay Particle:" << m_partName
            << "Number of daughter generations in printout:"
            << m_depth
            << endmsg;
}
```

3-17

Gaudi Framework Tutorial, April 2006

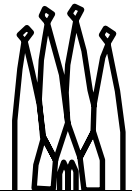


In fact, this is a bit of a stupid example. As you will have noticed, the GaudiAlgorithm base class already prints the value of all its properties if the message level is MSG::DEBUG

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      = 3;
```



Hands On: add D0 decays

```
// 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      = 3;

// Add D0 decay algorithm to list of top level algorithms
ApplicationMgr.TopAlg += {"DecayTreeAlgorithm/D0_Decays"};

// Setup of D0 decay algorithm
D0_Decays.DecayParticle    = "D0";
```

