

Using Shared Libraries

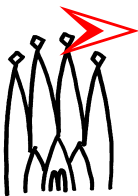
- The traditional way
- How to build shared images?
- Benefits
- Process configuration
- Fortran

M.Frank LHCb/CERN



The traditional way - comparison

- The past:
 - 1 Executable
 - Many object files/libraries, all linked into one executable **at link time**
- Gaudi:
 - 1 Executable
 - Very few object files/libraries linked in
 - Process acquires it's "personality" **at run-time**
 - Dynamic linkage of images
 - Execution of code within these images
- Keep possibility to link statically



What type of libraries exist?

➤ **Static** libraries

- Code to be **duplicated for each image**
- Build using CMT as known
- Result archive library

➤ **Implicitly linked** libraries

- Equivalent to tradition shared libraries
- Public implementation code: Base classes
- On the link step use CMT macro:

macro XXXXshlibflags "\$ (libraryshr_linkopts) ..."



GAUDI

M.Frank LHCb/CERN



What type of libraries exists?

➤ **Explicitly linked** libraries

➤ Loaded on demand component libraries

➤ Private implementation code

➤ Single entry point

- extern "C" std::list<const IFactory&>& getFactoryEntries()
- Access to all implemented factories

➤ On the link step use CMT macro:

```
macro XXXXshlibflags "$$(componentshr_linkopts) ..."
```



Benefits

- Process depends on configuration
 - PATH
 - LD_LIBRARY_PATH
- Modules become manageable
 - Minimize dependencies
- Upgrades are simple
 - No link step is involved
- Efficient use of resources
 - No unused ballast carried around



Managing the Environment

- Unix: cmt config; source setup.(c)sh
- WNT: use CMT add-in for Developer Studio

The screenshot shows the Microsoft Developer Studio interface with the CMT add-in. The title bar reads "Gaudi - Microsoft Developer Studio - [D:\...\Kernel\Bootstrap...". The menu bar includes File, Edit, View, Insert, Project, Build, Tools, Window, and Help. The toolbar contains several icons, including a CMT icon (a house with a question mark) and a CMT icon (a house with a question mark). The main window displays a project tree on the left with "Gaudi files" and "GaudiBase" folders. The central editor shows C++ code with comments like "// Declare fact" and "FactoryTable::F". The bottom status bar shows "Process Environment" and a command prompt window with the following text:

```
-----  
Process Environment  
-----  
CMT> SetEnvironment CLASSPATH = .;Z:\P32\jdk  
CMT> SetEnvironment LD_LIBRARY_PATH = S:\LHC  
CMT> SetEnvironment PATH = S:\LHCb\CMT\v1r3\  
-----
```

Green callout boxes with arrows point to various elements:

- Set Environment**: Points to the CMT icon in the toolbar.
- Show Environment**: Points to the CMT icon in the toolbar.
- Show CMT macros**: Points to the CMT icon in the toolbar.
- Build projects and workspace**: Points to the CMT icon in the toolbar.
- CMT dialog**: Points to the CMT icon in the toolbar.
- Display CMT help**: Points to the CMT icon in the toolbar.
- Output**: Points to the command prompt window at the bottom.



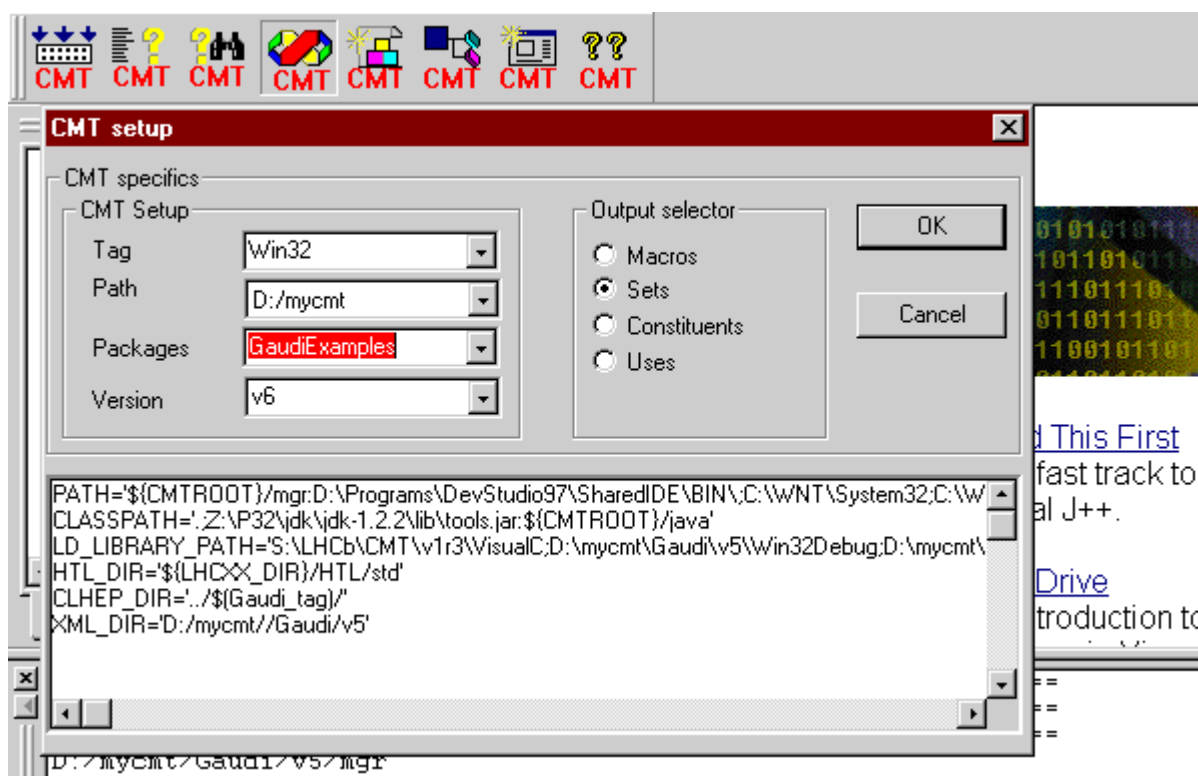
GAUDI

M.Frank L



Managing the Environment

- CMT add-in determines CMT parameters from active project
- If there is no active project: Dialog is invoked



GAUDI

M.Frank LHCb/CERN



Side Remarks

➤ Fortran

- Must be linked directly into the executable
I.e. the traditional way
 - Problem of duplication of common blocks

- Do not even think of linking Fortran to shared images
- unless you know what you are doing





GAUDI

M.Frank LHCb/CERN



How to make objects persistent

- What must be provided
- The musts
- I won't bore with internals
- See Rio example in GaudiExamples

M.Frank LHCb/CERN



Storing objects

- From using Sicb data we know
 - Writing converters is a pain for starters
 - ...but cannot be avoided
- Automate conversion procedure as far as possible
 - Use data serialization mechanism
 - inspired by Java / MFC / ROOT



Provide Class Identifier

```
class MyObject : public DataObject / ContainedObject      {  
...  
// Retrieve reference to class definition structure  
virtual const CLID& clID() const      { return MyObject::classID(); }  
static const CLID& classID()          { return CLID_MyObject;   }  
...  
};
```

- Nothing new
- Needed by the “generic converter” and “generic object factory”
- Mandatory to handle inhomogeneous containers



Provide Data Serializers

```
StreamBuffer& MyObject::serialize( StreamBuffer& s ) const {  
    DataObject::serialize(s);  
    return s << m_event << m_run << m_time;  
}
```

Writing

```
StreamBuffer& MyObject::serialize( StreamBuffer& s ) {  
    DataObject::serialize(s);  
    return s >> m_event >> m_run >> m_time;  
}
```

Reading

➤ Accepted data types

- Primitives: int, float, ...
- Smart references: e.g. SmartRef<MCParticle>



Provide Factories

➤ Object factory for

➤ Contained objects

```
static const ContainedObjectFactory<MyTrack>      s_MyTrackFactory;  
const IFactory& MyTrackFactory =                  s_MyTrackFactory;
```

➤ Object container e.g. ObjectVector<MyTrack>

```
static const DataObjectFactory<ObjectVector<MyTrack> >  
                                                s_MyTrackVectorFactory;  
const IFactory& MyTrackVectorFactory =          s_MyTrackVectorFactory;
```

➤ Converter Factory

```
static const DbUserCnvFactory<ObjectVector<MyTrack> > s_CnvFactory;  
const ICnvFactory& MyTrackCnvFactory =          s_CnvFactory;
```



Job options

➤ Persistency setup

// Application Mgr

```
ApplicationMgr.ExtSvc += { "DbEventCnvSvc/RootDbEvtCnvSvc" };  
ApplicationMgr.DLLs += { "DbCnvImp", "DbConverters", "RootDb" };  
ApplicationMgr.OutputStream = { "RootDst" };
```

// Output Stream

```
RootDst.ItemList = { "/Event/MyTracks#1" };  
RootDst.EvtDataSvc = "EventDataSvc";  
RootDst.EvtConversionSvc = "RootDbEvtCnvSvc";  
RootDst.OutputFile = "resultEx.root";
```

// Persistency/Conversion service setup:

```
EventPersistencySvc.CnvServices = { "RootDbEvtCnvSvc",  
                                     "SicbEventCnvSvc" };  
RootDbEvtCnvSvc.DbTypeName = "RootDb::OODataBase";  
RootDbEvtCnvSvc.FddbName = "RioFederation";
```



Conclusions

- It should be simple to make objects persistent
- Generic converters and object factories are easy to implement
- Total overhead < 20 LOC per class

➤ Output to	WNT	Linux
➤ RIO	X	X
➤ Objectivity/DB (not supported)	X	
➤ ODBC / RDBMS (not supported)	X	

