Using Shared Libraries

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

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 aquires 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) ...”

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

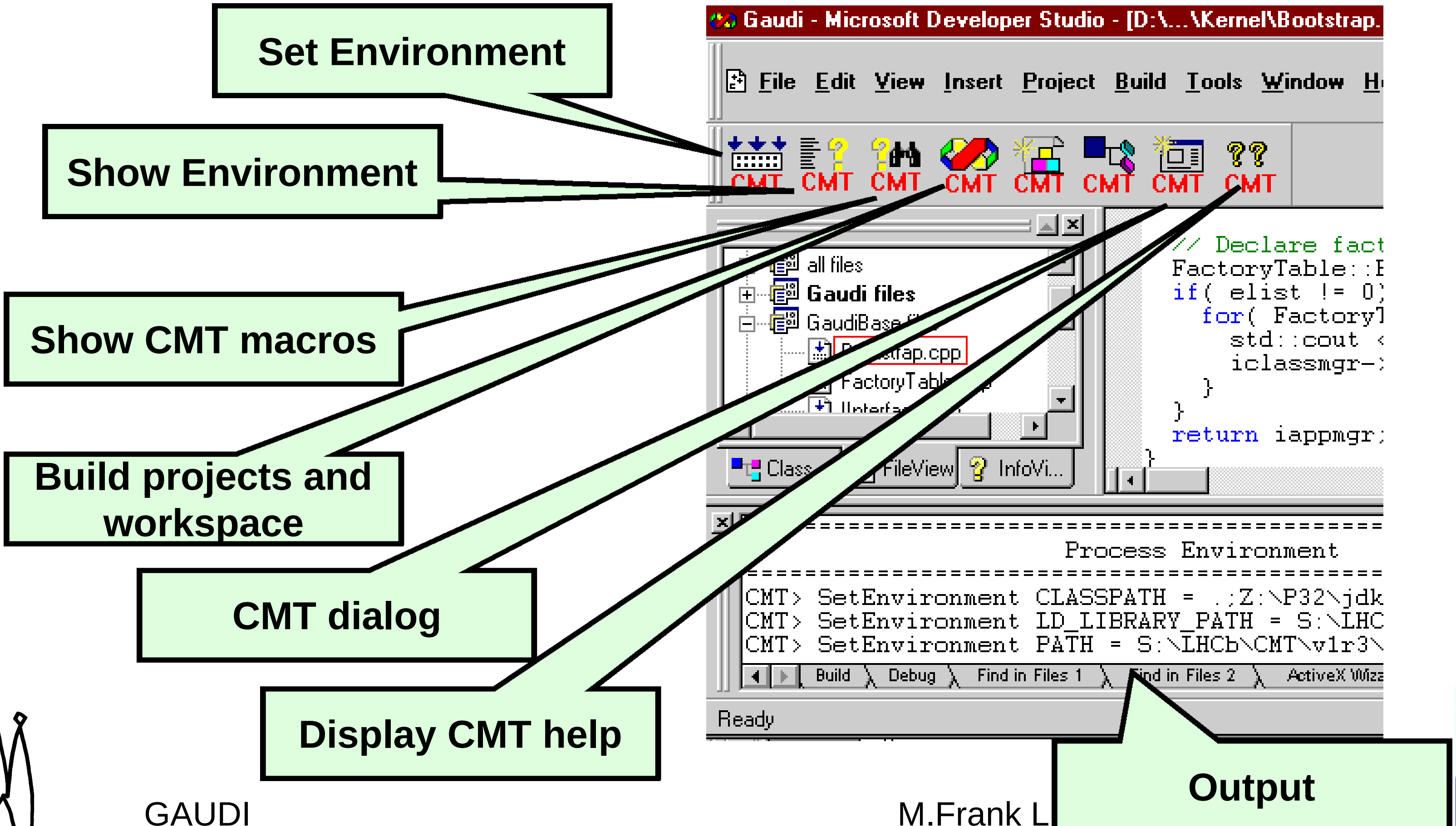
➢ Modules become manageable
  - Minimize dependencies

➢ Upgrades are simple
  - No link step is involved

➢ Efficient use of resources
  - No unused ballast carried around

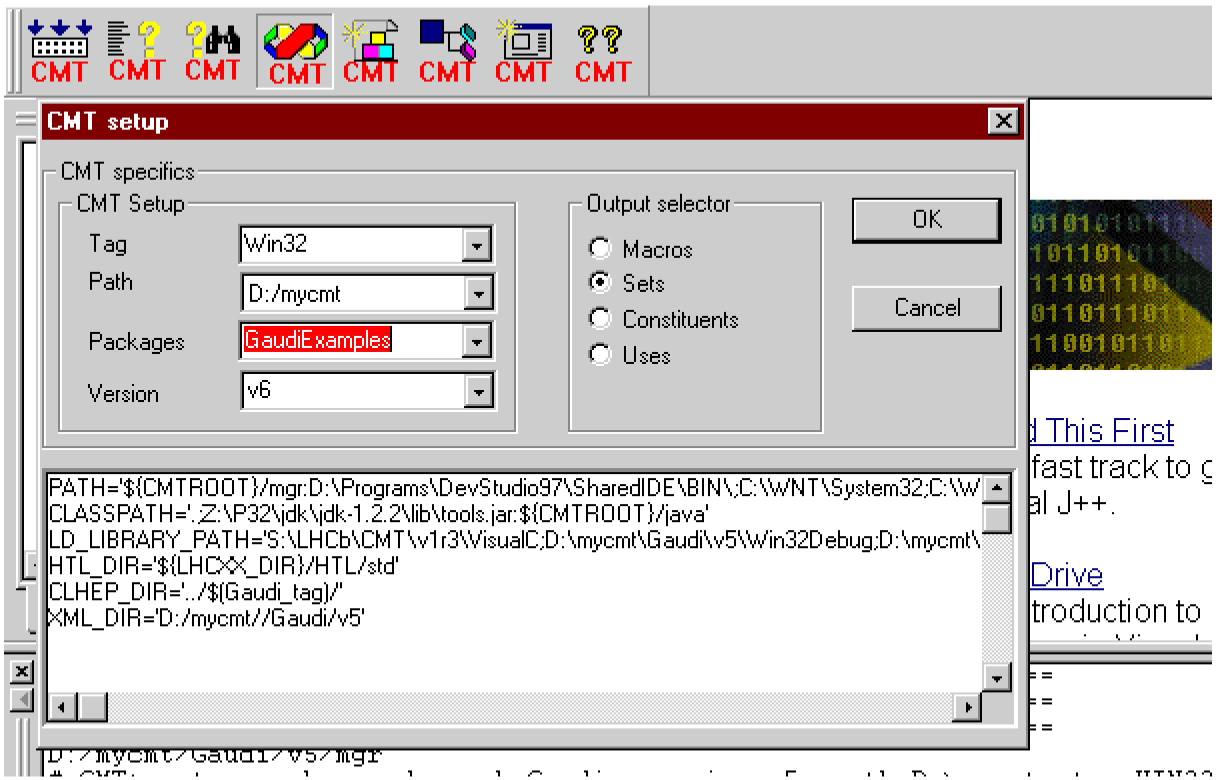
Managing the Environment

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



Managing the Environment

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



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

How to make objects persistent

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

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.OutStream = { "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

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