



Managed by Fermi Research Alliance, LLC for the U.S. Department of Energy Office of Science

Session 14:

Inventing a New Data Product

Rob Kutschke

art and LArSoft Course

August 5, 2015

In this Session You will Learn

- The mechanics of turning a class into a data product
 - Once this has been done you use it like any other data product.
- There is extensive documentation on the [art wiki](#)

What Should be in A Data Product

- Classes and structs that are mostly “just data”
 - C++ primitive types plus `std::string`
 - Plus aggregates of these: for example a `std::vector` of them
- No pointers or references
 - Later you will learn about `art::Ptr` and `art::Assns`.
- No c++11 constructs
- Member functions should know about other classes
 - Limited exceptions
 - The general solution is the [facade pattern](#)
- `std::map` will work but it’s very inefficient.
 - For some purposes there is a candidate replacement:
 - `cet::map_vector<T>`, behaves as a sparsely populated vector.
 - If you think you want a `std::map`, ask the *art* team.

Some Jargon

- **Persistent Representation**
 - The way that the bits in your data are arranged when it lives in a file, on disk or tape.
- **Transient Representation**
 - The way that the bits in your data are arranged when it lives in memory.
- **Persistency or Persistency Mechanism**
 - The software that moves your data between memory and disk
 - *art* uses ROOT to play this role
 - You have already seen two modules that do this
 - **RootInput** and **RootOutput**
 - These are plugins, much like your other modules
 - You may write your own plugins to do persistency.

ROOT Dictionaries

- How do you turn a class into a data product?
 - Create two files `classes_def.xml` and `classes.h`
 - Edit the `CMakeLists.txt` file to tell it to make ROOT dictionaries
 - When you run `buildtool`, it will find the instructions in the `CMakeLists.txt` file and it will send `classes_def.xml` and `classes.h` to a tool called `genreflex`, which comes with ROOT
 - `genreflex` is the tool that makes the ROOT dictionary
 - It makes two files with suffixes, `_dict.so` and `_map.so`.
- When you want to get or put a data product, *art* and ROOT use the dictionaries to transform between transient and persistent representations.

This Example uses Files in Two Directories

- The directory art-workbook/SimpleDataProducts
 - This holds the data product you will use, **EventSummary**, plus the start of a second data product that you will finish, **TrackSummary**.
 - It also has the classes_def.xml and classes.h files.
 - And a CMakeFiles.txt
- The directory art-workbook/UsingSimpleDataProducts
 - Summary_module.cc
 - Creates the EventSummary data product
 - ReadSummary_module.cc
 - Reads the EventSummary data product
 - Some .fcl files to run the examples

SimpleDataProducts/EventSummary.h

```
namespace tex{
  class EventSummary {
  public:
    EventSummary();
#ifdef __GCCXML__
    int nTracks()const { return nPositive_ + nNegative_; }
    int nPositive() const { return nPositive_; }
    int nNegative() const { return nNegative_; }
    void increment( int q);
#endif // __GCCXML__
  private:
    int nPositive_;
    int nNegative_;
  };
#ifdef __GCCXML__
  std::ostream& operator<<(std::ostream& ost,
                          const tex::EventSummary& sum );
#endif // __GCCXML__
}
```

SimpleDataProducts/EventSummary.h

- This class holds the number of positively and negatively charged reconstructed tracks that are found in one event.
 - Not very interesting but rich enough for a first example
- For the persistency mechanism to work properly, `genreflex` only needs to see:
 - The data members; the default constructor
- All other data members may be hidden from `genreflex`
 - This will speed up dictionary generation; may be important for large experiments
 - This is what the `#ifndef ... #endif` construction does.
 - The price is that if you use these dictionaries to use with interactive root, you will be missing a lot of functionality
 - If you need this, just remove the `#ifndef` and `#endif` macros.

SimpleDataProducts/classes_def.xml

```
<lcgdict>
  <class name="tex::EventSummary" classVersion="10" />
  <class name="art::Wrapper<tex::EventSummary>" />
</lcgdict>
```

- If EventSummary has data members that are classes or structs, you must add lines to declare them
 - Not needed if that class is declared in another dictionary
 - For example, primitive types, CLHEP::Hep3Vector, and CLHEP::HepLorentzVector are declared in dictionaries generated by *art*
- Only need the art::Wrapper for the data product, not for its constituents.
- Talk about version control later.

SimpleDataProducts/classes.h

```
#include "art-workbook/SimpleDataProducts/EventSummary.h"
#include "art/Persistency/Common/Wrapper.h"

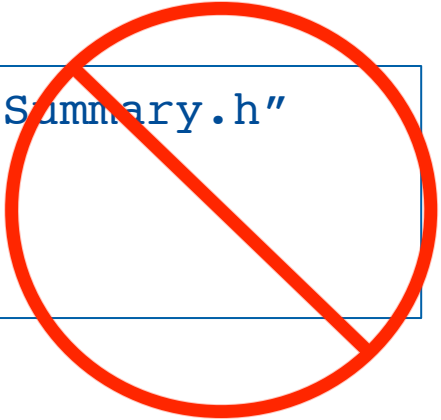
namespace {
  struct dictionary {
    art::Wrapper<tex::EventSummary> sum;
  };
}
```

- Every data product present in classes_def.xml must be `#include'd` (either directly or indirectly).
- You need an entry in the struct for every type that is a templated type
- This includes all of the `art::Wrapper<T>` types
- The data member names have no meaning but must be unique within the struct.
- The name of the struct has no meaning but must be unique with the file;

Late Breaking News

- If you look in the classes.h in your repository it actually looks like:

```
#include "art-workbook/SimpleDataProducts/EventSummary.h"  
#include "art/Persistency/Common/Wrapper.h"  
  
template class art::Wrapper<tex::EventSummary>;
```



- This pattern is now deprecated.
- We did not catch it on time to make the change for this course.
 - Update your repository in a few days and it will be fixed
- The new technique is more robust (there are some weird cases in which the other pattern fails that work correctly here; don't ask ...).

SimpleDataProducts/CMakeLists.txt

```
art_make(  
  LIB_LIBRARIES  
    ${ART_PERSISTENCY_PROVENANCE}  
    ${ART_PERSISTENCY_COMMON}  
    ${ART_UTILITES}  
    ${CETLIB}  
    ${CLHEP}  
  DICT_LIBRARIES  
    art-workbook_SimpleDataProducts  
    ${ART_PERSISTENCY_CORE}  
    ${ART_PERSISTENCY_PROVENANCE}  
    ${ART_PERSISTENCY_COMMON}  
    ${ART_UTILITES}  
    ${CETLIB}  
)
```

- See next page

SimpleDataProducts/CMakeLists.txt

- This tells the build system to
 - Compile all of the .cc files
 - Link them into a shared library
 - [lib/libart-workbook_SimpleDataProducts.so](#)
 - (relative to your build directory)
 - The cmake variable **LIB_LIBRARIES** describes the link list needed when the shared library is link.
 - Run genreflex
 - The cmake variable **DICT_LIBRARIES** describes the link list needed when the dictionary and map shared libraries are linked.
 - The library made in the second step is part of this link list.
 - The “lib/lib” part of the .so name is not needed in the link list

What Happens if you Change the Data Members

- This called **schema evolution**
- Often ROOT is smart enough to do the right thing automatically.
- But sometimes it is not. In that case you need to write some code to help ROOT out.
 - This is much, much easier if you have carefully maintained version numbers of each class in your `classes_def.xml` file.
 - It is very, very hard to retro-fit.
 - Although `classVersion` tags are optional, it's best to use them.
- `cetbuildtools` and `mrB` have resources to help with the automatic maintenance of `classVersion` numbers.
 - I don't know how to use them so they are not illustrated here
 - Your first stop is the [art wiki](#); if not, try Chris Green
- You can also maintain them by hand.

Questions so Far?

Get Started

- Go to your source directory
- Follow the instructions in
 - UsingSimpleDataProducts/README

Backup Slides:

SimpleDataProducts/EventSummary.cc

```
tex::EventSummary::EventSummary() :
    nPositive_(0),
    nNegative_(0){
}

void tex::EventSummary::increment( int q){
    if ( q > 0 ){
        ++nPositive_;
    } else{
        ++nNegative_;
    }
}
```

SimpleDataProducts/EventSummary.cc

```
std::ostream& tex::operator<<(std::ostream& ost,
                             const tex::EventSummary& summary){
    ost << "( Event Summary: Tracks: Postive: "
        << summary.nPositive()
        << " Negative: "
        << summary.nNegative()
        << ")";

    return ost;
}
```

- This is the piece that allows you to do:

```
EventSummary summary;
// Fill it up
std::cout << summary << std::endl;
```