Introduction to *gallery*

Marc Paterno

14 August 2017

What is *gallery*?

Definition

gallery is a (UPS) product that provides libraries that support the reading of event data from *art*/ROOT data files outside of the *art* event-processing framework executable.

All the bits in red are important.

- *gallery* comes as a binary install; you are not building it.
- *art* is a framework, *gallery* is a library.
- When using *art*, you write libraries that “plug into” the framework. When using *gallery*, you write a main program that uses libraries.
- When using *art*, the framework provides the event loop. When using *gallery*, you write your own event loop.
- *art* comes with a powerful and safe (but complex) build system. With *gallery*, you provide your own build system.

What does *gallery* do?

- gallery provides access to event data in *art*/ROOT files outside the *art* event processing framework executable:
  - without the use of `EDProducer`s, `EDAnalyzer`s, *etc.*, thus
  - without the facilities of the framework (*e.g.* callbacks for runs and subruns, *art* services, writing of *art*/ROOT files, access to non-event data).
- You can use *gallery* to write:
  - compiled C++ programs,
  - ROOT macros,
  - Using PyROOT, Python scripts.
- You can invoke any code you want to compile against and link to. Be careful to avoid introducing binary incompatibilities.

When should I use *gallery*?

- If you want to use either Python or interactive ROOT to access *art*/ROOT data files.
- If you do not want to use framework facilities, because you do not need the abilities they provide, and only need to access event data.
- If you want to create an interactive program that allows random navigation between events in an *art*/ROOT data file (e.g., an event display).

When should you not use *gallery*?

- When you need to use framework facilities (run data, subrun data, metadata, services, *etc.*)
- When you want to put something into the `Event`. For the *gallery* `Event`, you can not do so. For the *art* `Event`, you do so to communicate the product to another module, or to write it to a file. In *gallery*, there are no (framework!) modules, and *gallery* can not write an *art*/ROOT file.
- If your only goal is an ability to build a smaller system than your experiment's infrastructure provides, you might be interested instead in using the build system *studio*: `https://cdcvs.fnal.gov/redmine/projects/studio/wiki`. You can use *studio* to write an *art* module, and compile and link it, without (re)building any other code.

A well-structured `main` program skeleton

```cpp
int main(int argc, char** argv) {
  using namespace std;
  using gallery::Event;
  vector<string> filenames(argv+1, argv+argc);
  // create histograms, etc. here
  for (Event e(filenames); !e.atEnd(); e.next())
  {
    // call your analysis functions here
  }
}
```

Demonstration of *gallery*

- Make yourself a new top-level directory. Do not put this directory under the one you have used for other tutorials.
- Go into the directory.
- Make the DUNE software available for setup. Note the following command should be on one line; it is split only to fit on this slide:

  ```
  source /cvmfs/dune.opensciencegrid.org/products
  /dune/setup_dune.sh
  ```

- The demonstration code is available at https://github.com/marcpaterno/gallery-demo.
- Clone it, and go into the newly-created directory:

  ```
  git clone https://github.com/marcpaterno/gallery-demo.git
  cd gallery-demo
  ```

Demonstration of *gallery* (continued)

- Look at the `demo-setup` script, which sets up your environment.
- Source the `demo-setup` script: `source demo-setup`
- Build the `demo.cc` program using CMake, as instructed on the GitHub page specified above.
- Run the `demo` program on an input file (the name will be provided by the session organizers):

  ```
  ./demo input-file
  ```