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

Session 13:
A Simple Producer Module

Rob Kutschke

art and LArSoft Course

August 5, 2015

Preliminaries

- Thanks to everyone who filled out the surveys!
- If you have not already done so, we hope that you can find some time today to fill out:
  - For Monday: http://goo.gl/forms/TWTHjuVkG1
  - For Tuesday: http://goo.gl/forms/EePqePbZNJ

In this Session You will Learn

- A 4 part mantra for writing a producer module that produces one data product
  - A producer may produce as many data products as it wants
  - Just repeat the steps
- How to get data products with instance names from the event

Part 1: Inherit from `art::EDProducer`

```
#include "art/Framework/Core/EDProducer.h”

namespace tex {
  class Prod : public art::EDProducer {
  public:
    explicit Prod(fhicl::ParameterSet const& pset);
    void produce( art::Event& event) override;
  };
}
```

- Analyzer modules inherit from `art::EDAnalyzer`
- The called-every-event member function is called produce, not analyze.

Part 2: non-const Arguments

```
void produce( art::Event& event) override;
```

```
void analyze( art::Event const& event) override;
```

- Analyzer modules have only const access to the event.
- Producer modules have full access to the event
  - They may add new data products
  - They may NOT modify existing data products
- Similarly for arguments of beginRun/endRun etc

Part 3: Call produces() in the Constructor

```
#include "toyExperiment/MCDataProducts/IntersectionCollection.h”

tex::Prod::Prod(fhicl::ParameterSet const& ){
  produces<IntersectionCollection>();
}
```

- `produces` is a member function found somewhere in the inheritance tree
  - It tells art what types this module is allowed to produce
  - It must be called in the constructor
- The template argument is the data type that will be produced.
  - data type = name of class, struct, typedef

Part 4: Get/Do/Put In the produce Member Function

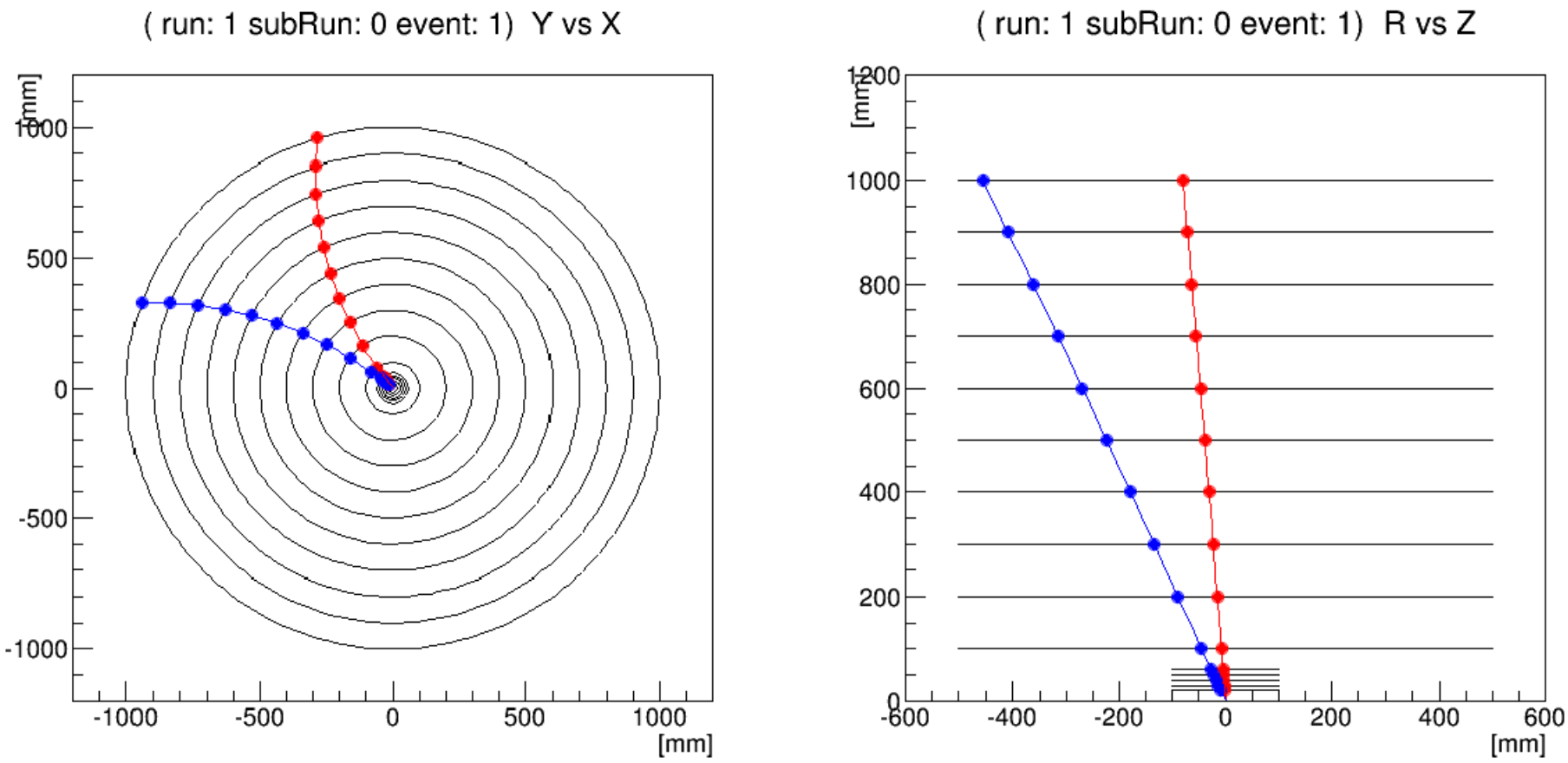
```
// Default construct an empty data product
auto output = std::make_unique<IntersectionCollection>();

 //  … do the work to fill the data product

// Give the data product to the event
event.put( std::move(output) );
```

- The type of `output` is `std::unique_ptr<IntersectionColllection>`
- There is no need to use `new`!

Remember the toy Experiment ….



Intersections shown as filled circles.

Aside: The Naming of Data Products

- Each data product has a name that has 4 fields, separated by underscores. The order is:
  - The name of the data type (in a "friendly" format).
  - Module label of the module that created it
  - Instance name
  - Process name of the process that created it
- The instance name is used if a producer makes more than one data product of the same type.
  - It is legal for an instance name to be an empty string
- The data product name is used as the name of the TBranch that holds the data product in the ROOT file.

There are two Data Products Used in this example

- In the jobs that created the input files used by the workbook, the module label `detsim` produced two data products:
  - All of the intersection objects for the inner 5 layers
  - All of the intersection objects for the outer 10 layers
  - Both have the same data type, tex::IntersectionCollection
  - Both produced by the same module label and process name
  - The two data products have instance names of inner and outer.
- In an `art::InputTag`: "module_label:instance_name"
  - For example:

```
innerTag : "detsim:inner"
outerTag : "detsim:outer"
```

See for Yourself

- (If needed, login and setup everything)
- In your build window look at the data products in the input file:

```
art -c fcl/FileDumper/fileDumperFriendly.fcl
   -s inputFiles/input01.art
```

- This is one long line, not two short ones.
- There is a discussion about this file in FirstProducer/README

FirstProducer/ConcatenateIntersections1_module.cc

```
class ConcatenateIntersections1 : public art::EDProducer {
 public:

   explicit ConcatenateIntersections1(fhicl::ParameterSet const& pset);
   void produce( art::Event& event) override;

 private:

   // Input tags for the two input data products.
   art::InputTag innerTag_;
   art::InputTag outerTag_;
};
```

- As usual, input tags for are initialized in the c'tor, by getting them from the parameter set.

Create, Fill and Put the Ouptut Data Product

```
void tex::ConcatenateIntersections1::produce( art::Event& event){

  auto inner = event.getValidHandle<IntersectionCollection>(innerTag_);
  auto outer = event.getValidHandle<IntersectionCollection>(outerTag_);

  // Create empty data product and reserve the required size.
  auto output = std::make_unique<IntersectionCollection>();
  output->reserve(inner->size()+outer->size());

  // Fill the data product
  output->insert( output->end(), inner->begin(), inner->end() );
  output->insert( output->end(), outer->begin(), outer->end() );

  // Add the product to the event
  event.put( std::move(output) );
}
```

Run the Exercise

- Run it:

```
art -c fcl/FirstProducer/producer1.fcl
```

- Look at the data products in the output file:

```
art -c fcl/FileDumper/fileDumperFriendly.fcl
  -s output/concatenateIntersections1.art
```

- Read the discussion in FirstProducer/README

After event.put() the unique_ptr is no longer valid

- After the call to:

```
event.put( std::move(output) );
```

- The variable `output` is invalid.
  - It no longer points at anything
  - This is a security feature: *art* makes it difficult to modify a data product after you have given it to the event.
- This is illustrated in:
  - FirstProducer/ConcatenateIntersections2_module.cc
  - To run it:

```
art -c fcl/FirstProducer/producer2.fcl
```

  - Then read the discussion in the README

Errors

- One producer can make many different data products.
- In the c'tor there needs to be a call to `produces<T>` for each type that you plan to produce.
  - If you produce many data products of the same type, you only need one call for that type.
- If you call event.put on a data product of a type for which there is no all to `produces<T>`, art will throw an exception and attempt graceful shutdown.

A bug to fix:

- Look at the code in
  - FirstProducer/ConcatenateIntersections3_module.cc
  - Run it with:

```
art -c fcl/FirstProducer/producer3.fcl
```

- This produces a run-time error
- Find and fix it
- The answer is in the README

Using art::Event::getManyByType<T>

- There is another way to get the two input collections from the event.
- You can ask art to give you a vector of Handles to all data products of a given type.
- This is illustrated in
  - FirstProducer/ConcatenateIntersections4_module.cc
  - Run it with:

```
art -c fcl/FirstProducer/producer4.fcl
```

Using and Producing in-Run Data Products

- FirstProducer/ConcatenateIntersections5_module.cc
  - Shows how to create a data product that lives in the run object
  - Follow the same pattern with Run -> SubRun to create a data product that lives in the SubRun object.
  - To run it:

```
art -c fcl/FirstProducer/producer5.fcl
```

  - Read the discussion in the README

Questions so Far?

Get Started

- Go to your source directory
- Read art-workbook/FirstProducer/README
- Read section 0 first. It tells you how to update your git repository and rebuild before continuing.

Backup Slides: