Fermilab

Session 7: More Module Interface

Rob Kutschke

art and LArSoft Course

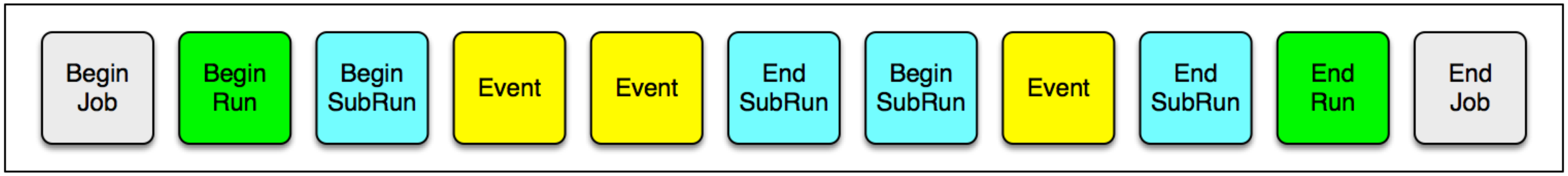
August 4, 2015

Welcome to Day 2!

- Yesterday, you:
  - Followed the site specific setup procedure
    - `source /products/course_setup.sh`
  - Source window: cloned a repository and checked out a branch
  - Build window: built and ran code
- How to continue after logging out and back in:
  - See Chapter 11 of the art workbook writeup (2 pages)
    - Follow the site specific setup procedure.
    - Open source and build windows
    - `source` one setup script in each of the source and build windows
  - Continue to work on the previous exercise or start a new one.
  - (Note the two meanings of "source"; is it clear?)

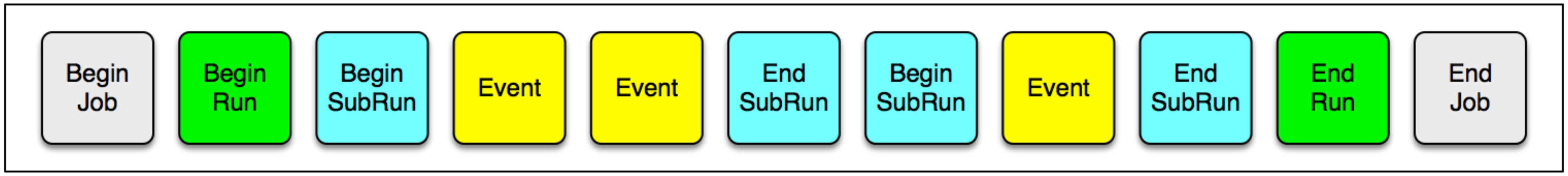
Recap: The Event Loop

- Your experiment groups events into runs and subruns
  - Your experiment the meaning of a run or subrun
  - Art provides bookkeeping tools to help manage them
- A short *art* job might see the following:



- A longer *art* job might see many runs, many subruns per run and many events per subrun.
- If I read all of my data to choose very rare but very interesting events (a sparse skim), I might have many runs and subruns with zero events!
- *art* can manage both situations

Recap: The `analyze` Member Function



```
namespace tex {
  class First : public art::EDAnalyzer {
  public:
    explicit First (fhicl::ParameterSet const& );
    void analyze   (art::Event const& event     ) override;
  };
}
```

- `analyze` is called once for every event.
- `art::Event` is an `art::EventID` plus data products
- `Art::EventID` 3 parts: run, subrun and event numbers.

New With the First Part of this Exercise:

```
class Optional : public art::EDAnalyzer {
 public:

   explicit Optional(fhicl::ParameterSet const& );
   void beginJob    () override;
   void beginRun    ( art::Run const&    run    ) override;
   void beginSubRun( art::SubRun const& subRun ) override;
   void analyze     ( art::Event const&  event  ) override;

 };
```

- A module may choose to define member functions that *art* will call at start of the job, at the start of each run and at the start of each subrun.
- You will also see the `endJob, endRun` and `endSubRun` member functions.

`art::Run` and `art::SubRun` objects:

```
void beginJob    () override;
void beginRun    ( art::Run const&     run    ) override;
void beginSubRun( art::SubRun const& subRun ) override;
void analyze     ( art::Event const&  event  ) override;
```

- `art::Event`
  - An `art::EventID` plus a collection of data products.
- `art::Run`
  - An `art::RunID` plus a collection of data products.
- `art::SubRun`
  - An `art::SubRunID` plus a collection of data products.
- `art::SubRun`ID
  - has 2 parts: run and subrun numbers
- `art::Run`ID
  - has 1 part: run number

beginJob vs Constructor

- Both are called once at the start of job.
- What tasks should be done in each?
  - Always initialize member data in the constructor
    - Prefer initializer list over initialization in the body of the c’tor
  - Some other operations must be done in the constructor
    - These will be described as you encounter them.
  - Other advice:
    - Your experiment may have a policy – ask!
    - One choice is to do as much as possible in the constructor.
    - My choice: create histogram, ntuple and TTree objects at `beginJob`, `beginRun` or `beginSubRun`, never in the constructor.
      - In my mind this separates the “computing infrastructure” work from the physics work.

Tracer

- *art* has a command line option `--trace`

```
art –c file.fcl  --trace
```

- This tells art to print an informational message just before and just after every call to user supplied code
  - And just before and after some of its own internal operations.
- You can use this to see if art is calling your code at the times when you expect it to be called.
- If you don't understand what art is doing, this is one of the tools you can use to help understand.
- You will use this option in this exercise.

Module Hygiene

- Did you remember to use override?
- When you look at the example code, you will see that does not provide a destructor. Because the destructor has no work to do, the compiler supplied destructor will do the right thing
  - If it will do the right thing, let the compiler write it for you

Questions so Far?

Hints on Navigating the Giant PDF file

- Title page
- Blank page
- List of Chapters (3 pages long)
- Detailed Table of Contents (16 pages long)
- Everything is internally hyperlinked:
  - Page numbers in the TOC, and index
  - Table, Listing, Figure and Section cross-references
  - Configure your browser to highlight hyperlinks.
- Many PDF browsers have previous and next buttons
  - MAC Safari
    - Back: Apple-[
    - Forward: Apple-]

Fermilab

Get Started

- Start to work on Chapter 13 (Exercise 3) in the are workbook writeup
  - https://web.fnal.gov/project/ArtDoc/Shared%20Documents/art-documentation.pdf

- My Powerpoint is flakey.
- If the above link fails or if it display pdf as text, try:
  - https://web.fnal.gov/project/ArtDoc/SitePages/documentation.aspx
  - Under latest releases, click on the document with the highest version number.
- If both links fail, mouse in the url.

Backup Slides:

Data Products

- See section 3.6.4 of the art workbook writeup.
- The unit of event-data that is managed by *art*
  - More precisely by art::Event
- Examples:
  - Raw data is often one data product per sub-system
  - Each module in the reconstruction chain will create one or more data products.
    - Unpacked hits for each subsystem
    - Reconstructed tracks, showers, jets, electrons, muons ….
    - Reconstructed neutrino interactions
      - Sometimes called "events", just to create more confusion …
  - The simulation chain will create many data products

The Assembly Line Metaphor

- *art* is like an assembly line
- The `art::Event` is the product being built
- Each function in each module is a work station along the line
- *art*'s job is to make sure that the product (the `art::Event`) gets to each work station (functions supplied by modules) in the right order.