art TUTORIAL

The Anne E. Norrick Lecture Series

Adam Lister

30th april, 2021

The Elephant in the Room

The number of times you google the “***art***” or “***art event***” will be high.

Why did they name the framework *art?* It’s *definitely* so they could call themselves *artists*.

The Elephant in the Room

The number of times you google the “***art***” or “***art event***” will be high.

Why did they name the framework *art?* It’s *definitely* so they could call themselves *artists*.

WORSE: There is a lightweight framework for reading art files. It’s called **Gallery***. The number of times I’ve searched for “***art gallery***”...

I **cannot overstate** how much I hate these names, but we’re stuck with it, so here goes.

*not covered in this tutorial

Pt I: Introduction

Introduction

- ➔ I'm definitely not an *art* expert, but I've got a couple years experience using it, and can make it do what I want *most* of the time
- ➔ This is going to be a bit code heavy, but that's the nature of software talks… sorry!
- ➔ There's going to be a bit of hands-on workshopping so **have your terminals open!**
  - ◆ Help can be found at **#art-tut-april-2021** on slack
- ➔ I'll try and split this up sensibly so that there's time for you to stretch/grab coffee and to work through the exercises!
- ➔ I recommend you open up these slides in your browser - there's some commands you'll probably want to copy/paste

What is an *art*?

art is an **event processing framework**

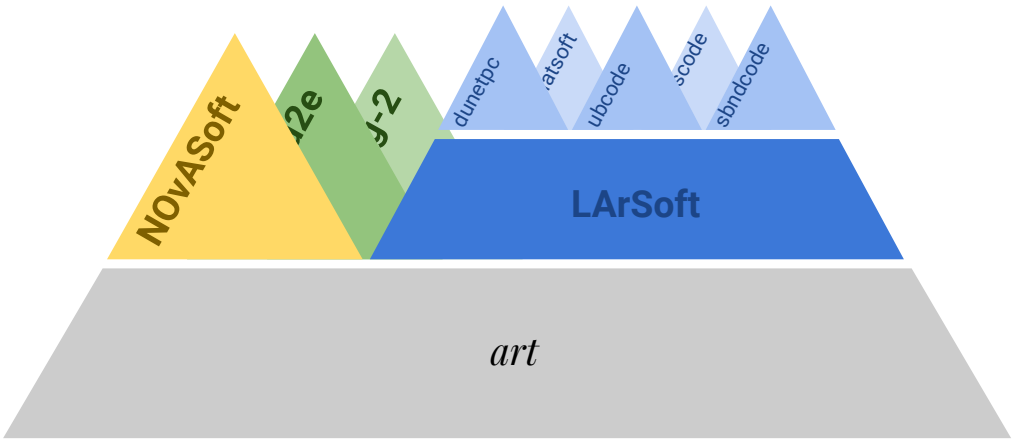
*What does **that** mean?* It's basically an event loop, but *fancy*

It's how NOvA performs

- Detector Simulation
- Reconstruction & Calibration
- CAF Making

It's worthwhile you being comfortable with *art*

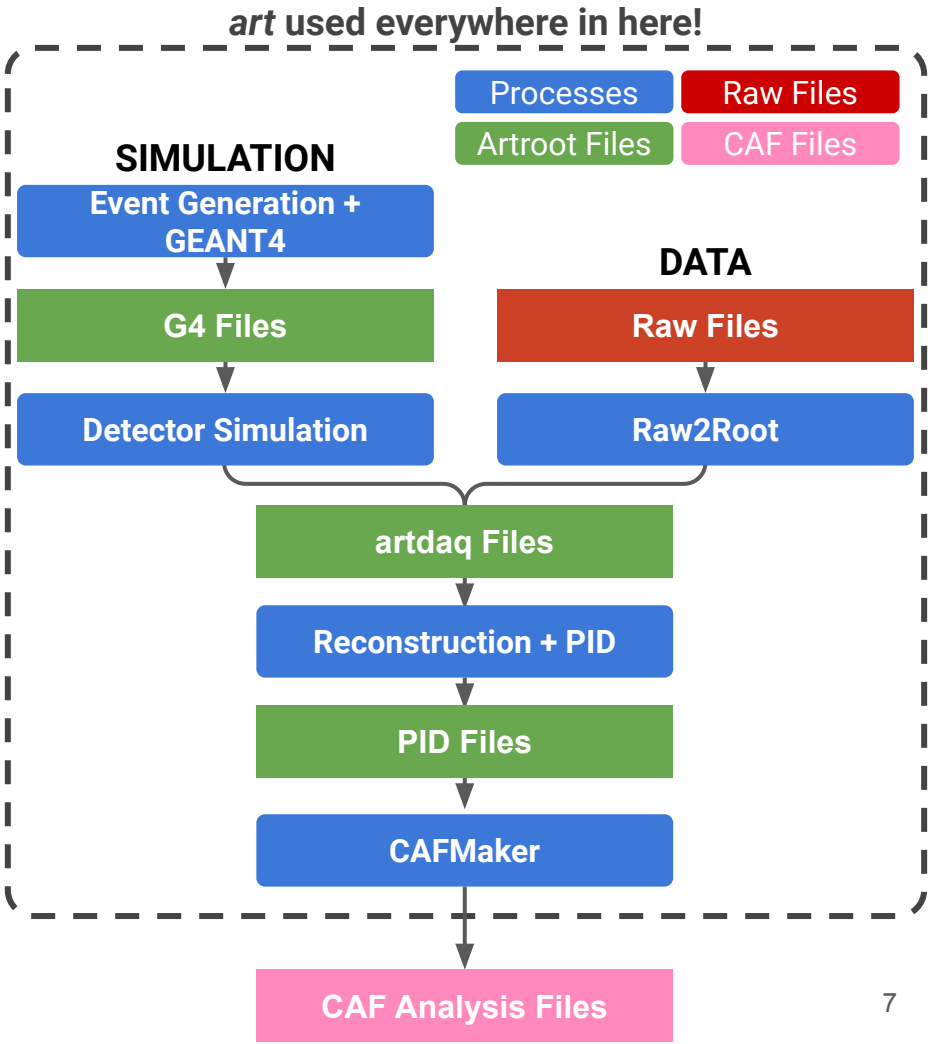
- Gain knowledge of your experiment!
- A large chunk of the FNAL world uses it!



The *art* Mountain Range

Where Does NOvA Use *art*?

There's not a lot of places it doesn't!



Where Does NOvA Use *art*?

There's not a lot of places it doesn't!

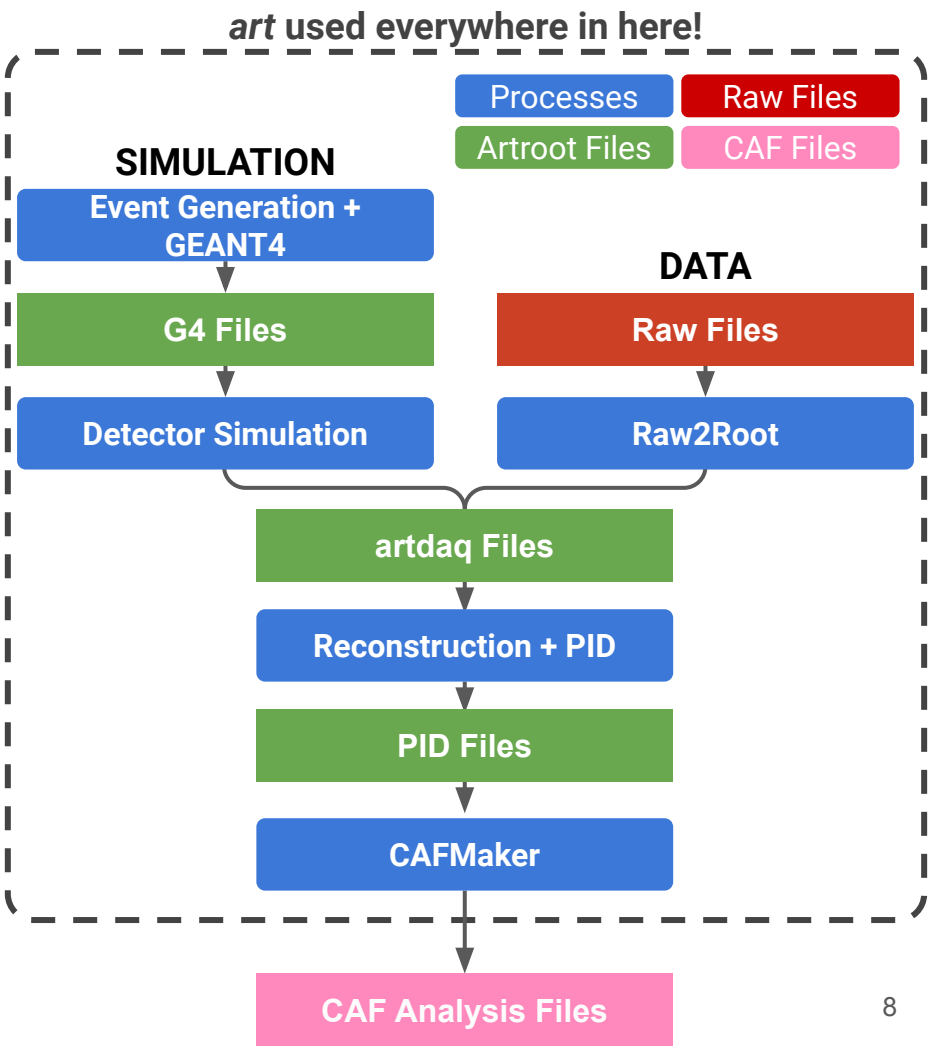
In any **process** we're mostly reading in **artroot files** and making more **artroot files** with more *stuff* added.

➔ By convention, these files just end in ".root"

Notable exceptions:

➔ *art* can read **.raw data files**, and make them into **artroot files**
- ("Raw2Root" in NOvA, "Swizzling" in LAr-World)

➔ *art* is used to read artroot files and make your familiar analysis-level **CAF files**



If you don't have the `setup_nova` script in your ~/.bashrc, it's in the backup slides

Test File!

For this tutorial I've set aside one test file that we can use to take a look at

SAM Definition: `alister1_art_tutorial_April2020`

If we all try to access this at the same time, we're going to have a bad time, so I suggest you copy the file in this definition to your `/nova/ana/users/${USER}/` area.

This is not a SAM tutorial, but to get at the file in the definition

Gives you the filename

```
setup_nova -r development -b maxopt
samweb list-files "defname:alister1_art_tutorial_April2020 "
cd /nova/ana/users/${USER}
ifdh_fetch <file_name>
export TESTFILE=$(readlink -f <file_name>)
echo $TESTFILE
```

This should print out the path for the file, saves us retyping it every time!

The *art* Data Format

```
root -l /nova/ana/users/${USER}
```

GAH! I have copied this file have opened it in root and see a bunch of TTrees! How do I find what I want?

Remember: These are artroot files! You *_can_* look at the data available to you by scanning the “Event” TTree, but there are ***a _lot_* of branches**, and you probably don’t want to browse them.

```
*.....................................................................*
*Br 3024 :sim::PhotonSignals_photrans__G4toArtdaq.obj.fPoissonLambda :  *
*        | Double_t fPoissonLambda[sim:                               *
*        | :PhotonSignals_photrans__G4toArtdaq.obj_]                  *
*Entries :      562 : Total  Size=   58861415 bytes  File Size  =    7379725 *
*Baskets :      562 : Basket Size=      16384 bytes  Compression=   7.97     *
*.....................................................................*
```

Don’t worry, there are some handy tools we can use to interrogate these files!

Event Dump

Here and for the rest of the tutorial, the input artroot file is the artroot file you copied to your local area earlier.

To get a summary of the data products in artroot files, we have the event dump FHiCL file, `eventdump.fcl`

What's a FHiCL file?!

It's just a configuration file which tells are *what to run* and *how to set it up*. *art* looks for fhicl files in your `FHICL_FILE_PATH` environment variable. Rather than trawling through every directory in that `PATH`, you can simply type `findfcl eventdump.fcl` (or any other fhicl file!)

Running this fhicl file, `art -c eventdump.fcl -n 1 -s $TESTFILE` results in something like the following:

```
Begin processing the 1st record. run: 13531 subRun: 13 event: 1 at 07-Apr-2021 22:47:32 CDT
PRINCIPAL TYPE: Event
PROCESS NAME..... | MODULE_LABEL..... | PRODUCT INSTANCE NAME | DATA PRODUCT TYPE..... | PRODUCT FRIENDLY TYPE..... | .SIZE
EventMixer..... | generator..... | ..... | std::vector<bsim::NuChoice>..... | bsim::NuChoices..... | ...36
EventMixer..... | singlemixer..... | ..... | std::vector<simb::GTruth>..... | simb::GTruths..... | ....?
EventMixer..... | singlemixer..... | ..... | std::vector<bsim::NuChoice>..... | bsim::NuChoices..... | ...34
EventMixer..... | singlemixer..... | ..... | std::vector<bsim::Dk2Nu>..... | bsim::Dk2Nus..... | ...34
EventMixer..... | TriggerResults..... | ..... | art::TriggerResults..... | art::TriggerResults..... | ....-
EventMixer..... | generator..... | ..... | std::vector<bsim::Dk2Nu>..... | bsim::Dk2Nus..... | ...36
EventMixer..... | singlemixer..... | ..... | std::vector<sim::Particle>..... | sim::Particles..... | ....?
EventMixer..... | singlemixer..... | ..... | std::vector<simb::MCTruth>..... | simb::MCTruths..... | ....?
EventMixer..... | genpremixer..... | ..... | sumdata::SpillData..... | sumdata::SpillData..... | ....-
EventMixer..... | geantgen..... | ..... | art::Assns<sim::Particle,simb::MCTruth,void>..... | sim::Particlesimb::MCTruthvoidart::Assns..... | .3595
EventMixer..... | g4premixer..... | ..... | std::vector<sim::TrueEnergy>..... | sim::TrueEnergys..... | ....?
EventMixer..... | g4premixer..... | ..... | art::Assns<sim::TrueEnergy,sim::Particle,void>..... | sim::Particlesim::TrueEnergyvoidart::Assns..... | ....?
```

Event Dump: A Closer Look

```
DATA PRODUCT TYPE.....
std::vector<bsim::NuChoice>.....
std::vector<simb::GTruth>.....
std::vector<bsim::NuChoice>.....
std::vector<bsim::Dk2Nu>.....
art::TriggerResults.....
std::vector<bsim::Dk2Nu>.....
std::vector<sim::Particle>.....
std::vector<simb::MCTruth>.....
sumdata::SpillData.....
art::Assns<sim::Particle,simb::MCTruth,void>.....
std::vector<sim::TrueEnergy>.....
art::Assns<sim::TrueEnergy,sim::Particle,void>.....
```

DATA PRODUCT TYPE

This is the first thing I usually look at, it's basically a list of each type of object in the event.

Here we see for example there's a vector of sim::Particle -type objects, a vector of bsim::NuChoice -tpye objects, etc.

```
Begin processing the 1st record. run: 13531 subRun: 13 event: 1 at 07-Apr-2021 22:47:32 CDT
PRINCIPAL TYPE: Event
PROCESS NAME..... | MODULE_LABEL..... | PRODUCT_INSTANCE_NAME | DATA PRODUCT TYPE..... | PRODUCT FRIENDLY TYPE..... | .SIZE
EventMixer..... | generator..... | ..... | std::vector<bsim::NuChoice>..... | bsim::NuChoices..... | ...36
EventMixer..... | singlemixer..... | ..... | std::vector<simb::GTruth>..... | simb::GTruths..... | ....?
EventMixer..... | singlemixer..... | ..... | std::vector<bsim::NuChoice>..... | bsim::NuChoices..... | ...34
EventMixer..... | singlemixer..... | ..... | std::vector<bsim::Dk2Nu>..... | bsim::Dk2Nus..... | ...34
EventMixer..... | TriggerResults..... | ..... | art::TriggerResults..... | art::TriggerResults..... | ....-
EventMixer..... | generator..... | ..... | std::vector<bsim::Dk2Nu>..... | bsim::Dk2Nus..... | ...36
EventMixer..... | singlemixer..... | ..... | std::vector<sim::Particle>..... | sim::Particles..... | ....?
EventMixer..... | singlemixer..... | ..... | std::vector<simb::MCTruth>..... | simb::MCTruths..... | ....?
EventMixer..... | genpremixer..... | ..... | sumdata::SpillData..... | sumdata::SpillData..... | ....-
EventMixer..... | geantgen..... | ..... | art::Assns<sim::Particle,simb::MCTruth,void>..... | sim::Particlesimb::MCTruthvoidart::Assns..... | .3595
EventMixer..... | g4premixer..... | ..... | std::vector<sim::TrueEnergy>..... | sim::TrueEnergys..... | ....?
EventMixer..... | g4premixer..... | ..... | art::Assns<sim::TrueEnergy,sim::Particle,void>..... | sim::Particlesim::TrueEnergyvoidart::Assns..... | ....?
```

Event Dump: A Closer Look

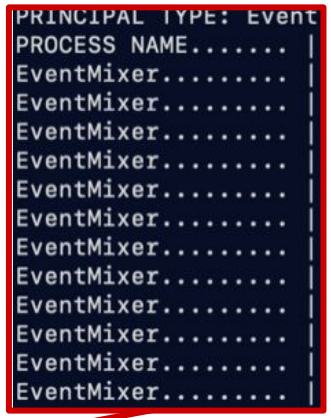
SIZE

This just tells you how many of each time of data product are in the event.

Entries with “?” generally means the data products have been “dropped” - ie removed from the event.

```
Begin processing the 1st record. run: 13531 subRun: 13 event: 1 at 07-Apr-2021 22:47:32 CDT
PRINCIPAL TYPE: Event
PROCESS NAME..... | MODULE_LABEL..... | PRODUCT INSTANCE NAME | DATA PRODUCT TYPE..... | PRODUCT FRIENDLY TYPE..... | .SIZE
EventMixer..... | generator..... | ..... | std::vector<bsim::NuChoice>..... | bsim::NuChoices..... | ...36
EventMixer..... | singlemixer..... | ..... | std::vector<simb::GTruth>..... | simb::GTruths..... | ....?
EventMixer..... | singlemixer..... | ..... | std::vector<bsim::NuChoice>..... | bsim::NuChoices..... | ...34
EventMixer..... | singlemixer..... | ..... | std::vector<bsim::Dk2Nu>..... | bsim::Dk2Nus..... | ...34
EventMixer..... | TriggerResults..... | ..... | art::TriggerResults..... | art::TriggerResults..... | ....–
EventMixer..... | generator..... | ..... | std::vector<bsim::Dk2Nu>..... | bsim::Dk2Nus..... | ...36
EventMixer..... | singlemixer..... | ..... | std::vector<sim::Particle>..... | sim::Particles..... | ....?
EventMixer..... | singlemixer..... | ..... | std::vector<simb::MCTruth>..... | simb::MCTruths..... | ....?
EventMixer..... | genpremixer..... | ..... | sumdata::SpillData..... | sumdata::SpillData..... | ....–
EventMixer..... | geantgen..... | ..... | art::Assns<sim::Particle,simb::MCTruth,void>..... | sim::Particlesimb::MCTruthvoidart::Assns..... | .3595
EventMixer..... | g4premixer..... | ..... | std::vector<sim::TrueEnergy>..... | sim::TrueEnergys..... | ....?
EventMixer..... | g4premixer..... | ..... | art::Assns<sim::TrueEnergy,sim::Particle,void>..... | sim::Particlesim::TrueEnergyvoidart::Assns..... | ....?
```

Event Dump: A Closer Look



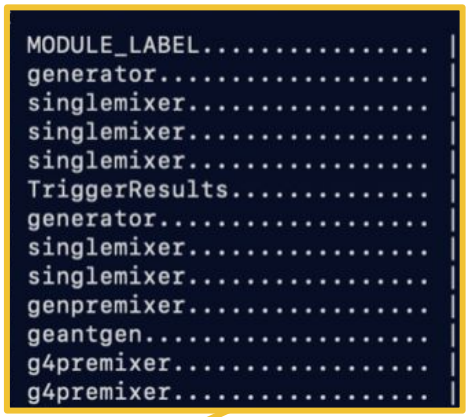
PROCESS NAME

The process name is a label related to the FHiCL file you run.

Each FHiCL file has a `process_name: thisprocess` line, which defines this label.

```
Begin processing the 1st record. run: 13531 subRun: 13 event: 1 at 07-Apr-2021 22:47:32 CDT
PRINCIPAL TYPE: Event
PROCESS NAME..... | MODULE_LABEL..... | PRODUCT INSTANCE NAME | DATA PRODUCT TYPE..... | PRODUCT FRIENDLY TYPE..... | .SIZE
EventMixer..... | generator..... | ..... | std::vector<bsim::NuChoice>..... | bsim::NuChoices..... | ...36
EventMixer..... | singlemixer..... | ..... | std::vector<simb::GTruth>..... | simb::GTruths..... | ....?
EventMixer..... | singlemixer..... | ..... | std::vector<bsim::NuChoice>..... | bsim::NuChoices..... | ...34
EventMixer..... | singlemixer..... | ..... | std::vector<bsim::Dk2Nu>..... | bsim::Dk2Nus..... | ...34
EventMixer..... | TriggerResults..... | ..... | art::TriggerResults..... | art::TriggerResults..... | ....-
EventMixer..... | generator..... | ..... | std::vector<bsim::Dk2Nu>..... | bsim::Dk2Nus..... | ...36
EventMixer..... | singlemixer..... | ..... | std::vector<sim::Particle>..... | sim::Particles..... | ....?
EventMixer..... | singlemixer..... | ..... | std::vector<simb::MCTruth>..... | simb::MCTruths..... | ....?
EventMixer..... | genpremixer..... | ..... | sumdata::SpillData..... | sumdata::SpillData..... | ....-
EventMixer..... | geantgen..... | ..... | art::Assns<sim::Particle,simb::MCTruth,void>..... | sim::Particlesimb::MCTruthvoidart::Assns..... | .3595
EventMixer..... | g4premixer..... | ..... | std::vector<sim::TrueEnergy>..... | sim::TrueEnergys..... | ....?
EventMixer..... | g4premixer..... | ..... | art::Assns<sim::TrueEnergy,sim::Particle,void>..... | sim::Particlesim::TrueEnergyvoidart::Assns..... | ....?
```

Event Dump: A Closer Look



MODULE LABEL

Each FHiCL file can run several *art modules*, this part tells you which module ran made the data product you're looking at

```
Begin processing the 1st record. run: 13521 subRun: 13 event: 1 at 07-Apr-2021 22:47:32 CDT
PRINCIPAL TYPE: Event
PROCESS NAME..... | MODULE_LABEL..... | PRODUCT INSTANCE NAME | DATA PRODUCT TYPE..... | PRODUCT FRIENDLY TYPE..... | .SIZE
EventMixer..... | generator..... | ..... | std::vector<bsim::NuChoice>..... | bsim::NuChoices..... | ...36
EventMixer..... | singlemixer..... | ..... | std::vector<simb::GTruth>..... | simb::GTruths..... | ....?
EventMixer..... | singlemixer..... | ..... | std::vector<bsim::NuChoice>..... | bsim::NuChoices..... | ...34
EventMixer..... | singlemixer..... | ..... | std::vector<bsim::Dk2Nu>..... | bsim::Dk2Nus..... | ...34
EventMixer..... | TriggerResults..... | ..... | art::TriggerResults..... | art::TriggerResults..... | ....-
EventMixer..... | generator..... | ..... | std::vector<bsim::Dk2Nu>..... | bsim::Dk2Nus..... | ...36
EventMixer..... | singlemixer..... | ..... | std::vector<sim::Particle>..... | sim::Particles..... | ....?
EventMixer..... | singlemixer..... | ..... | std::vector<simb::MCTruth>..... | simb::MCTruths..... | ....?
EventMixer..... | genpremixer..... | ..... | sumdata::SpillData..... | sumdata::SpillData..... | ....-
EventMixer..... | geantgen..... | ..... | art::Assns<sim::Particle,simb::MCTruth,void>..... | sim::Particlesimb::MCTruthvoidart::Assns..... | .3595
EventMixer..... | g4premixer..... | ..... | std::vector<sim::TrueEnergy>..... | sim::TrueEnergys..... | ....?
EventMixer..... | g4premixer..... | ..... | art::Assns<sim::TrueEnergy,sim::Particle,void>..... | sim::Particlesim::TrueEnergyvoidart::Assns..... | ....?
```

Ok, Just Before We Break For 15 Mins...

We're going to be looking at the Demo Package, so let's go ahead and get that set up. Perform the magic incantation...

```
cd /nova/app/users/${USER} &&
newrel -t development 2021-04-30-art-tut &&
cd 2021-04-30-art-tut &&
srt_setup -a &&
addpkg_svn -h Demo &&
novasoft_build -t
```

Pt II: FHiCL Files

FHiCL Files Introduction

FHiCL: **F**ermilab **Hi**erarchical **C**onfiguration **L**anguage

Using FHiCL files means you don't need to re-build every time you want to change configuration - do it on the fly!

Two types of fhicl files:

- ➔ **Top level "job" file:** These are the files you run, typically starts with a `process_name` and Have a `physics` block [more on this later]
- ➔ **configuration file:** You're not able to run these files, they just define a bunch of FHiCL Parameters

FHiCL parameters typically take the form `LABEL : VALUE` where "LABEL" is just a string, and "VALUE" can be any simple type (bool/int/float/double/string, etc), or a vector of simple types using `LABEL : [VALUE 1, VALUE 2, VALUE 3, ...]`

FHiCL Files Introduction: Job FHiCLs

```
#include "services.fcl"
#include "TutAnalyzer.fcl"

process_name: TutAna

services:
{
  # Load the service that manages root files for histograms.
  TFileService: { fileName: "tut_hist.root" closeFileFast: false }
  @table::standard_services
}

source:
{
  module_type: RootInput
}

# Define and configure some modules to do work on each event.
# First modules are defined; they are scheduled later.
# Modules are grouped by type.
physics:
{
  analyzers:
  {
    ana: @local::standard_tutanalyzer
  }

  tutana: [ ana ]

  end_paths:     [tutana]  #end_path are things that do not modify ar
}
```

The job fhicl file we're going to start by looking at is `Demo/tutanajob.fcl`

FHiCL Files Introduction: Job FHiCLs

```
#include "services.fcl"
#include "TutAnalyzer.fcl"

process_name: TutAna

services:
{
  # Load the service that manages root files for histograms.
  TFileService: { fileName: "tut_hist.root" closeFileFast: false }
  @table::standard_services
}

source:
{
  module_type: RootInput
}

# Define and configure some modules to do work on each event.
# First modules are defined; they are scheduled later.
# Modules are grouped by type.
physics:
{
  analyzers:
  {
    ana: @local::standard_tutanalyzer
  }

  tutana: [ ana ]

  end_paths:     [tutana]  #end_path are things that do not modify ar
}
```

Corresponds to the `PROCESS_NAME` from the `eventdump.fcl` output earlier

If your input artroot file has data products with a given process name, *art* will complain if you try to run a FHiCL file with the same process name, since it will think you're trying to overwrite those data products **(NOT allowed!)**

FHiCL Files Introduction: Job FHiCLs

```
#include "services.fcl"
#include "TutAnalyzer.fcl"

process_name: TutAna

services:
{
  # Load the service that manages root files for histograms.
  TFileService: { fileName: "tut_hist.root" closeFileFast: false }
  @table::standard_services
}

source:
{
  module_type: RootInput
}

# Define and configure some modules to do work on each event.
# First modules are defined; they are scheduled later.
# Modules are grouped by type.
physics:
{
  analyzers:
  {
    ana: @local::standard_tutanalyzer
  }

  tutana: [ ana ]

  end_paths:     [tutana]  #end_path are things that do not modify ar
}
```

SERVICES

Services are configured in the “services” block, and in our case, most are loaded from *services.fcl* using @table::standard_services [more later]

A service is *basically* a helper class that does *a thing*.

Some example services:

- **TFileService:** handles output to ROOT files when you want to save histograms/TTrees, etc.
- **Calibrator:** handles calibration
- **Geometry:** loads in gdml files, defines some fiducial volumes, etc.

FHiCL Files Introduction: Job FHiCLs

```
#include "services.fcl"
#include "TutAnalyzer.fcl"

process_name: TutAna

services:
{
  # Load the service that manages root files for histograms.
  TFileService: { fileName: "tut_hist.root" closeFileFast: false }
  @table::standard_services
}

source:
{
  module_type: RootInput
}

# Define and configure some modules to do work on each event.
# First modules are defined; they are scheduled later.
# Modules are grouped by type.
physics:
{
  analyzers:
  {
    ana: @local::standard_tutanalyzer
  }

  tutana: [ ana ]

  end_paths:     [tutana]  #end_path are things that do not modify ar
}
```

SOURCE

This can take two options for module_type

- **EmptyEvent:** This is used when generating new Monte Carlo events from scratch
- **RootInput:** When you're reading in an artroot file for downstream processing of generation

FHiCL Files Introduction: Job FHiCLs

```
#include "services.fcl"
#include "TutAnalyzer.fcl"

process_name: TutAna

services:
{
  # Load the service that manages root files for histograms.
  TFileService: { fileName: "tut_hist.root" closeFileFast: false }
  @table::standard_services
}

source:
{
  module_type: RootInput
}

# Define and configure some modules to do work on each event.
# First modules are defined; they are scheduled later.
# Modules are grouped by type.
physics:
{
  analyzers:
  {
    ana: @local::standard_tutanalyzer
  }

  tutana: [ ana ]

  end_paths:     [tutana]  #end_path are things that do not modify a
}
```

PHYSICS BLOCK

Three types of sub-blocks: *analyzers, producers, filters* [more later]

Each line in the sub-block defines a different module. In this case we're defining just the **TutAnalyzer** module, which is defined in **TutAnalyzer.fcl** and pulled in using @local::

```
standard_tutanalyzer:
{
  module_type: TutAnalyzer

  ProngLabel: "prod"
}
```

FHiCL Files Introduction: Job FHiCLs

```
#include "services.fcl"
#include "TutAnalyzer.fcl"

process_name: TutAna

services:
{
  # Load the service that manages root files for histograms.
  TFileService: { fileName: "tut_hist.root" closeFileFast: false }
  @table::standard_services
}

source:
{
  module_type: RootInput
}

# Define and configure some modules to do work on each event.
# First modules are defined; they are scheduled later.
# Modules are grouped by type.
physics:
{
  analyzers:
  {
    ana: @local::standard_tutanalyzer
  }

  tutana: [ ana ]

  end_paths:     [tutana]  #end_path are things that do not modify a
}
```

PHYSICS BLOCK

Three types of sub-blocks: *analyzers, producers, filters* [more later]

Each line in the sub-block defines a different module. In this case we're defining just the **tutanalyzer** module, which is defined in **TutAnalyzer.fcl**

The list of analyzers to run is defined in tutana (but this can be named anything)

`end_paths`: contains any analyzer paths
`trigger_paths`: contains any producer/filter paths

Local/Table

Both `@local::` and `@table::` are used for fetching configurations from some other FHiCL file. The only difference is that `@local::` includes the `{}`. **Example:**

If you have a configuration file, `MyInclude.fcl`

```
my_default_paramset:
{
 my_string_1: "test"
 my_int_1: 1
}
```

Either of these are a valid way to include the configuration!

```
#include "MyInclude.fcl"
my_local_set: @local::my_default_paramset
```

```
#include "MyInclude.fcl"
my_table_set:
{
 @table::my_default_paramset
}
```

Ok, Let's Put This To Use

Running this FHiCL file on the test file,

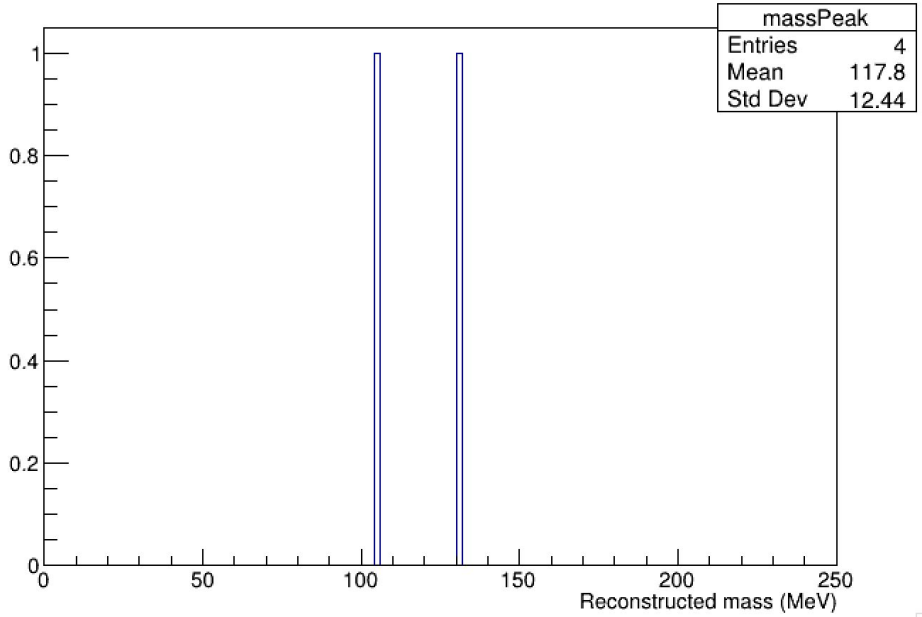
```
art -c tutanajob.fcl $TESTFILE
```

We see that we get an root file output,

`tut_hist.root`

Notice this name corresponds to what was configured in the FHiCL file using the TFileService!

Opening up this root file in root, we see the plot on the right. Ok, not so interesting… but success! 🎉

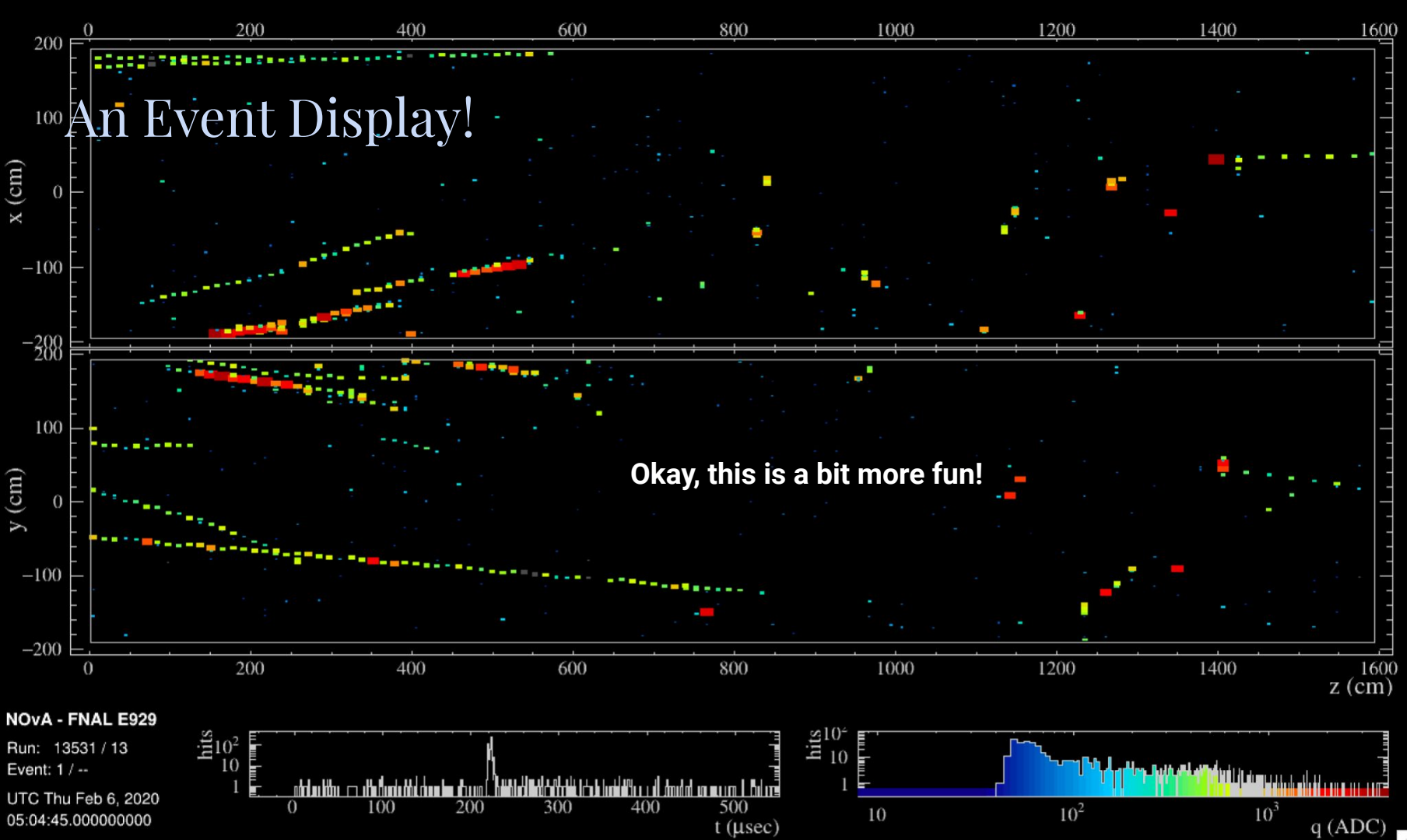


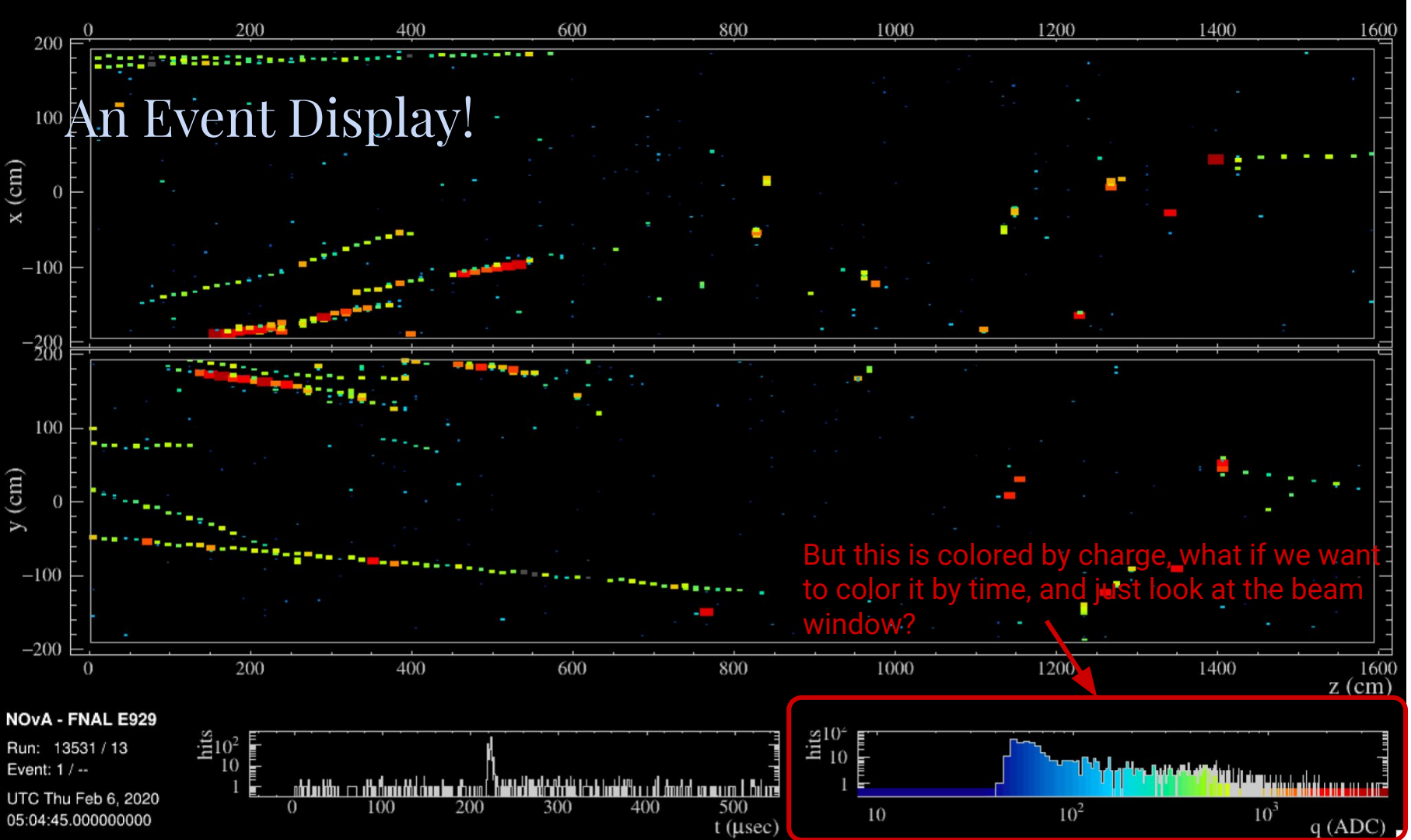
Let’s Do Something More Fun

The Event Display is also run with *art*, and you can get to it by just running

```
art -c evd.fcl $TESTFILE
```

The result is...


An Event Display!
Okay, this is a bit more fun!
x (cm)
y (cm)
z (cm)
NOvA - FNAL E929
Run: 13531 / 13
Event: 1 / --
UTC Thu Feb 6, 2020
05:04:45.000000000
hits
t (μsec)
hits
q (ADC)


An Event Display!
x (cm)
y (cm)
z (cm)
But this is colored by charge, what if we want to color it by time, and just look at the beam window?
NOvA - FNAL E929
Run: 13531 / 13
Event: 1 / --
UTC Thu Feb 6, 2020
05:04:45.000000000
hits
t (μsec)
hits
q (ADC)

Remember: you can find FHiCL files using `findfcl`!

You may need to .qqqqqqqqq to get out of EVD

Modifying the Event Display

Let's look at `evd.fcl`:

```
findfcl evd.fcl
```

Guess we want `RawDrawingOptions`, and guess it's probably in `evd_services.fcl`

So copy both `evd.fcl` and `evd_services.fcl` to your local area so that we can modify them.

```
#include "evd_services.fcl"
#include "services.fcl"
#include "BackTracker.fcl"

process_name: EVD


services:
{
  RandomNumberGenerator: {} #ART native random number generator
  @table::core_services

  # EVD services
  PlotDrawingOptions:        @local::standard_plotdrawingopt
  GeometryDrawingOptions:    @local::standard_geomdrawingopt
  SimulationDrawingOptions:  @local::standard_simdrawingopt
  RawDrawingOptions:         @local::standard_rawdrawingopt
  RecoDrawingOptions:        @local::standard_recodrawingopt
  ScanOptions:               @local::standard_scanopt
  SliceNavigator:            @local::standard_slicenavigator
  Colors:                    @local::standard_colors
  EventDisplay:              @local::standard_evd
}
services.BackTracker: @local::standard_backtracker
```

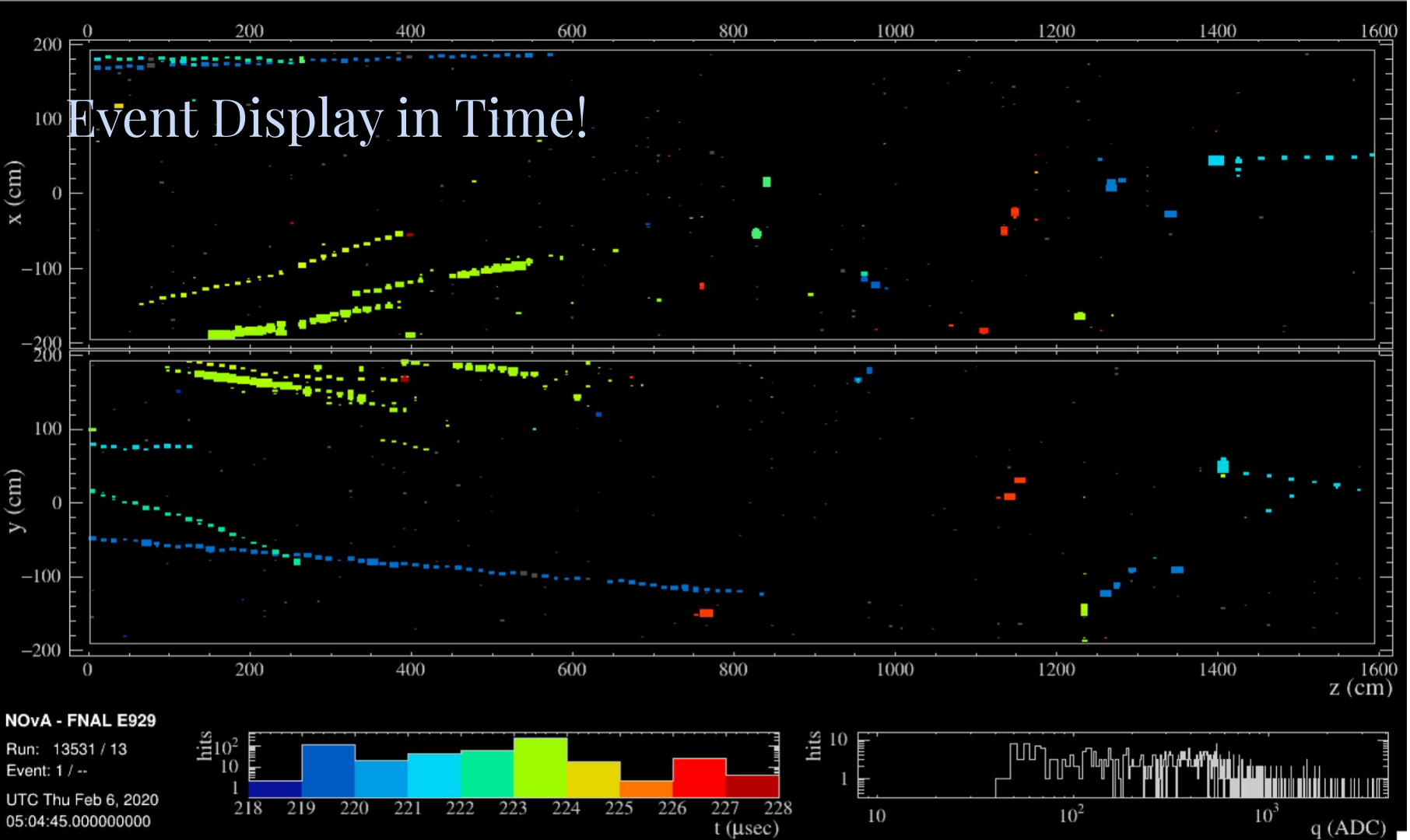
Looking in evd_services.fcl...

```
# Configure the raw drawing options
standard_rawdrawingopt:
{
 Color:              {val:0                    gui:"rb:by charge,by time"      doc:"Use color to show time or charge of hits?"}
 WhichHits:          {val:0                    gui:"rb:RAW,CAL"                doc:"Draw raw hits or calibrated hits?"}
 WhichQ:             {val:0                    gui:"rb:PE,PECOR"               doc:"Which calibrated charge to show?"}
 RawDrawingOpt:      {val:2                    gui:"cb:mask bad channels,scale hits by charge,suppress ghosted hits, ghost dimmed hits" doc:"How to draw hits"}
 ScaleFactor:        {val:1.3                  gui:"sl:0,5"                     doc:"Scale factor to apply to hits"}
 TimeRange:          {val:[-50,550]            gui:"sl:-50,550"                doc:"Time range in units of usec"}
 TimeBinSize:        {val:1                    gui:"sl:0.05,10"                doc:"Time bin size in units of ns"}
 TimeAutoZoomTruth:  {val:0                    gui:"rb:off,on"                 doc:"Auto-zoom to truth times of interest"}
 RawDigitsModules:   {val:["daq"] gui:"te"     gui:"lbm:daq,mrcc,mre,another"  doc:"Modules producing raw hits"}
 CellHitsModules:    {val:["calhit"] gui:"te"  gui:"lbm:calhit,another"      doc:"Backup CellHit source"}
 ADCRange:           {val:[8,4096]             gui:"sl:1,4096"                 doc:"ADC histogram range"}
 ADCBinSize:         {val:4                    gui:"sl:1,128"                  doc:"ADC bin size"}
 Hit3DStyle:         {val:0x03                 gui:"cb:boxes,towers,crossings" doc:"How to render hits on 3D display"}
 THistogram:         {val:1                    gui:"rb:off,on"                 doc:"Draw time histogram or not?"}

 RawDigitsModulesAdd:     {val:[] gui:"te" doc:"Additional raw digits modules"}
 CellHitsModulesAdd:      {val:[] gui:"te" doc:"Additional CellHit modules"}
}
```

We can spot the two options here which are what we want:
`Color: { val: 0 }`, and `TimeRange: { val: [-50, 550] }`

Let's switch to `Color: {val: 1… }` and `TimeRange: {val: [218, 228] … }`, and then re-run the art command...


Event Display in Time!
x (cm)
y (cm)
z (cm)
NOvA - FNAL E929
Run: 13531 / 13
Event: 1 / --
UTC Thu Feb 6, 2020
05:04:45.000000000
hits
t (μsec)
hits
q (ADC)

An Easier Way...

One cool thing about FHiCL files is that you can override configuration options from the top level “job” FHiCL file!

So rather than going into `evd_services.fcl` you can simply add these lines to the top level `evd.fcl` FHiCL file (so long as you’ve made a local copy!)

```
services.RawDrawingOptions.Color.val: 1
services.RawDrawingOptions.TimeRange.val: [218, 228]
```

and you’ll accomplish the same thing as what we just did!

Useful Tidbits Summary for FHiCL

`findfcl` lives in `novaproduction` and is handy to find FHiCL files in your `FHICL_FILE_PATH` environment variable

Want to print the full configuration?

```
fhicl-dump <input FHiCL file>
```

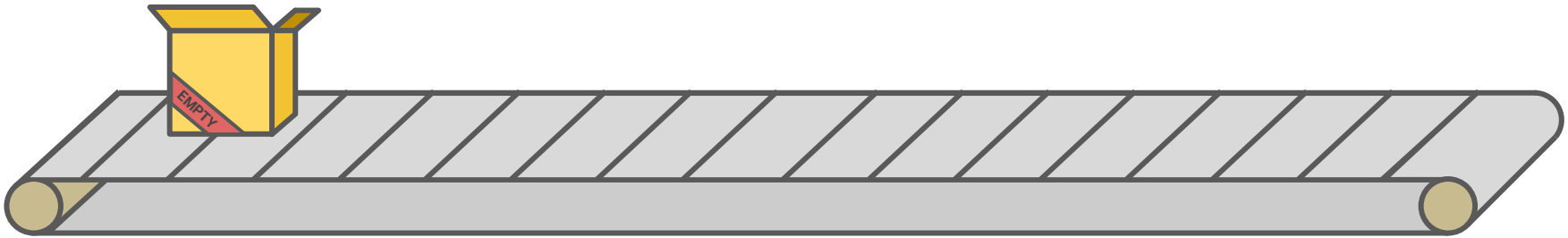
`-a` option will append a comment to every line listing which fhicl file the configuration is set in

Want to see what fhicl configuration made an artroot file?

```
config_dumper <input_artroot_file>
```

Pt III: Writing *art* Modules

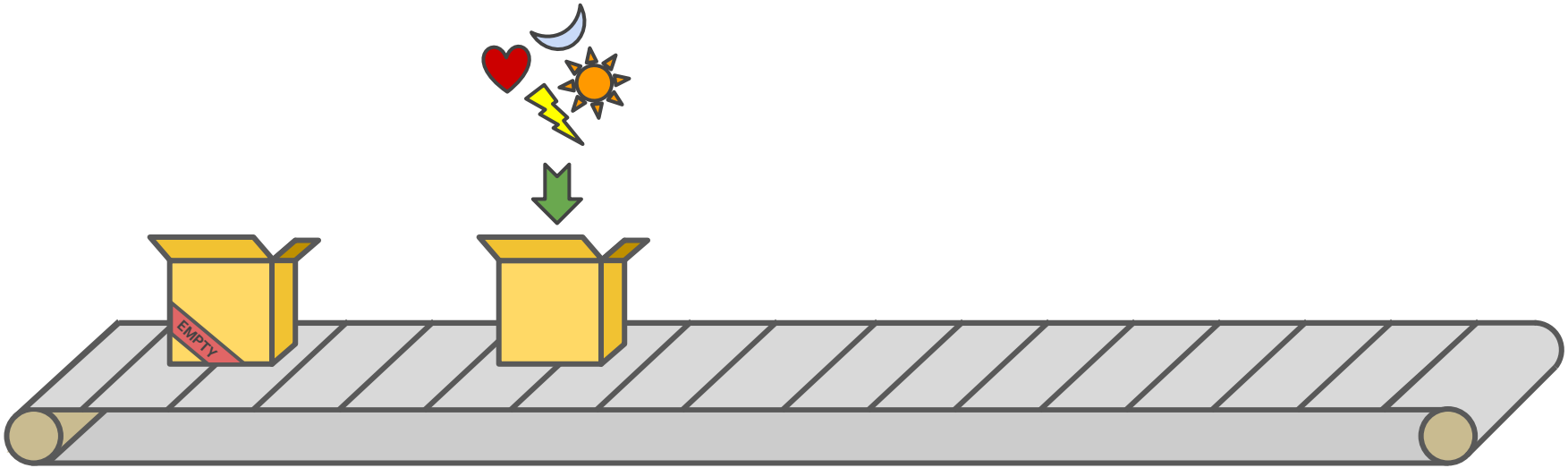
OK, But Like, What Does an *art* Module Actually Do?



Empty Event

Idea for these slides stolen shamelessly from an excellent MicroBooNE tutorial by Andy Mastbaum

OK, But Like, What Does an *art* Module Actually Do?

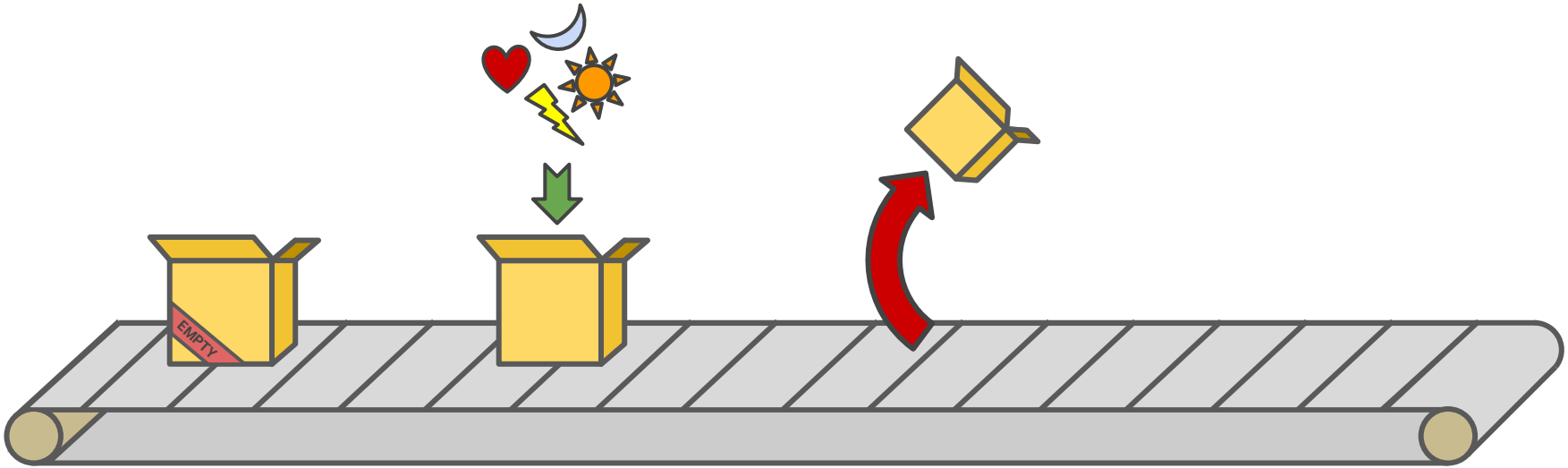


Empty Event

Producer Module

Add data products
(“stuff”) to the event

OK, But Like, What Does an *art* Module Actually Do?



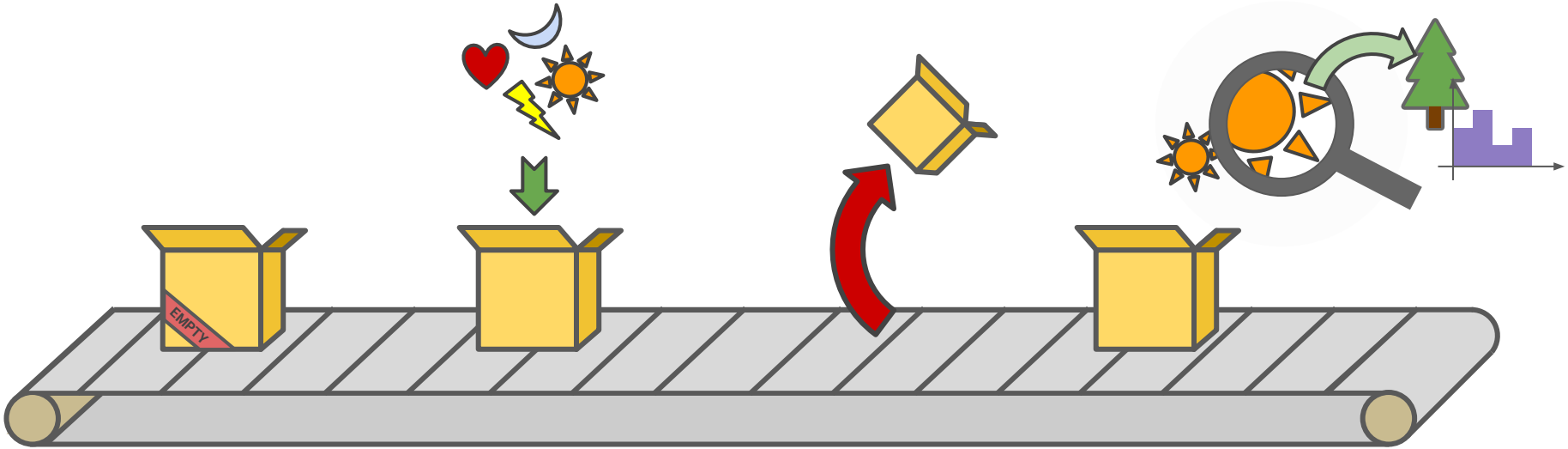
Empty Event

Producer Module
Add data products (“stuff”) to the event

Filter Module
Remove certain events that meet some defined criteria

Idea for these slides stolen shamelessly from an excellent MicroBooNE tutorial by Andy Mastbaum

OK, But Like, What Does an *art* Module Actually Do?



Empty Event

Producer Module
Add data products (“stuff”) to the event

Filter Module
Remove certain events that meet some defined criteria

Analyzer Module
Inspect data products, make ROOT trees or histograms

Idea for these slides stolen shamelessly from an excellent MicroBooNE tutorial by Andy Mastbaum

How To Build An *art* Module

CLASS DEFINITION
Declare methods, data members, etc

CONSTRUCTOR
Anything that needs run *before* the event loop, typically read in configuration etc

ANALYZE/PRODUCE/FILTER
Every module has one of the above. This is your event loop.
"For every event do *these things*"

OPTIONAL METHOD

OPTIONAL METHOD

⋮ ⋮

OPTIONAL METHOD

Where c++ best practice is to separate out code into `.cxx` and `.h` files, modules are single files that end in `_module.cc`.

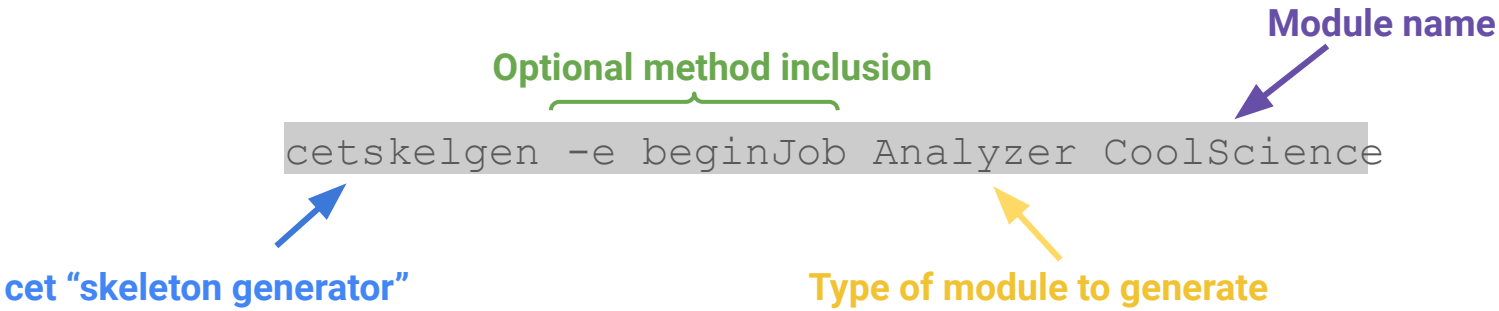
The three main ingredients are the **class definition**, **initializer**, and the **loop method**.

Other useful functions that have a special meaning in art:

- **beginJob/endJob:**
  run code in at beginning/end of job
- **beginRun/endRun:**
  run code at the beginning/end of every run
- **beginSubRun/endSubRun:**
  run code at the beginning/end of every subRun

Generating an *art* Module

As you can imagine, there's a lot of boilerplate that is the same between different modules. There's a handy helper tool to generate all of the stuff you need



Please keep the name CoolScience - it'll make our lives easier in a few slides!

CoolScience_module.cc

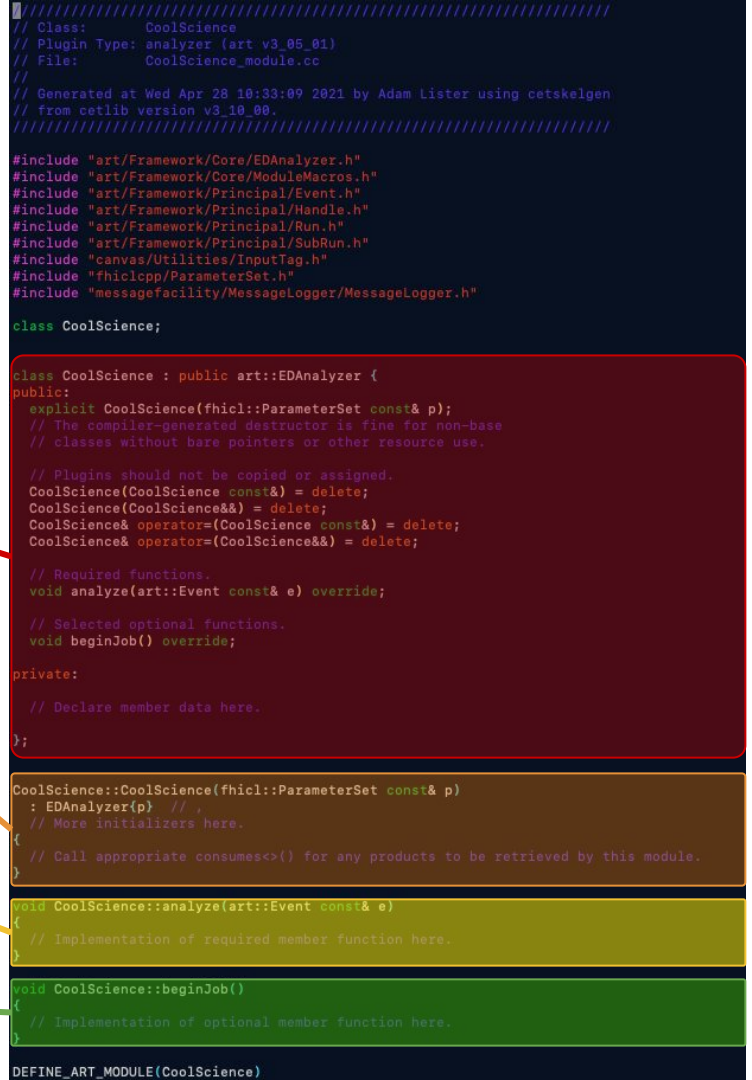
Opening up the module we just generated, you can see all of the sections that make up the art module!

CLASS DEFINITION
Declare methods, data members, etc

CONSTRUCTOR
Anything that needs run *before* the event loop, typically read in configuration etc

ANALYZE/PRODUCE/FILTER
Every module has one of the above. This is your event loop.
"For every event do *these things*"

OPTIONAL METHOD



```
//////////////////////////////////////////////////////////////////////
// Class:       CoolScience
// Plugin Type: analyzer (art v3_05_01)
// File:        CoolScience_module.cc
//
// Generated at Wed Apr 28 10:33:09 2021 by Adam Lister using cetskelgen
// from cetlib version v3_10_00.
//////////////////////////////////////////////////////////////////////

#include "art/Framework/Core/EDAnalyzer.h"
#include "art/Framework/Core/ModuleMacros.h"
#include "art/Framework/Principal/Event.h"
#include "art/Framework/Principal/Handle.h"
#include "art/Framework/Principal/Run.h"
#include "art/Framework/Principal/SubRun.h"
#include "canvas/Utilities/InputTag.h"
#include "fhiclcpp/ParameterSet.h"
#include "messagefacility/MessageLogger/MessageLogger.h"

class CoolScience;

class CoolScience : public art::EDAnalyzer {
public:
  explicit CoolScience(fhicl::ParameterSet const& p);
  // The compiler-generated destructor is fine for non-base
  // classes without bare pointers or other resource use.

  // Plugins should not be copied or assigned.
  CoolScience(CoolScience const&) = delete;
  CoolScience(CoolScience&&) = delete;
  CoolScience& operator=(CoolScience const&) = delete;
  CoolScience& operator=(CoolScience&&) = delete;

  // Required functions.
  void analyze(art::Event const& e) override;

  // Selected optional functions.
  void beginJob() override;

private:

  // Declare member data here.

};

CoolScience::CoolScience(fhicl::ParameterSet const& p)
  : EDAnalyzer{p}  // ,
  // More initializers here.
{
  // Call appropriate consumes<>() for any products to be retrieved by this module.
}

void CoolScience::analyze(art::Event const& e)
{
  // Implementation of required member function here.
}

void CoolScience::beginJob()
{
  // Implementation of optional member function here.
}

DEFINE_ART_MODULE(CoolScience)
```

Let’s Make this do Something!

In the next few slides I’ll be stepping through each section and showing which lines of code to add.

In total there’s only 10 lines of new code - but if you’re having trouble getting something to work you can copy & paste from the backup slides - or ask for help in **#art-tut-april-2021** on slack!

Let's Make this do Something!

First we need to know what we want to look at. For this example , we're going to look at the different Prongs, so let's include this:

```
#include "RecoBase/Prong.h"
```

To find the right paths to include, I generally go to the **dOxygen** for the product I want to use and look at the top line.

```
#include "/cvmfs/nova-development.opensciencegrid.org/novasoft/releases/N21-04-27/RecoBase/Prong.h"
```

This is the bit you want

The dOxygen also has a list of all of the methods that you can call for a given data product.

Modifications – Class Definition

```
class CoolScience : public art::EDAnalyzer {
public:
  explicit CoolScience(fhicl::ParameterSet const& p);
  // The compiler-generated destructor is fine for non-base
  // classes without bare pointers or other resource use.

  // Plugins should not be copied or assigned.
  CoolScience(CoolScience const&) = delete;
  CoolScience(CoolScience&&) = delete;
  CoolScience& operator=(CoolScience const&) = delete;
  CoolScience& operator=(CoolScience&&) = delete;

  // Required functions.
  void analyze(art::Event const& e) override;

  // Selected optional functions.
  void beginJob() override;

private:

  // Declare member data here.
  std::string fProngLabel;

};
```

In the **class definition**, we're going to add any data members we want.

In this case we're going to want to read the *module label* from the FHiCL configuration, so we know we're going to want a string.

Modifications - Constructor

```
CoolScience::CoolScience(fhicl::ParameterSet const& p)
  : EDAnalyzer{p}  // ,
  // More initializers here.
{
  // Call appropriate consumes<>() for any products to be retrieved by this module.
  fProngLabel = p.get<std::string>("ProngLabel", "elasticarmshs");
}
```

Type to cast the parameter as

Label of the `LABEL : VALUE` pair from FHiCL file

Optional default value in case ProngLabel not defined in FHiCL file

The **Constructor** is where we actually read the FHiCL file, so let's assign the label to the string we defined in the **Class Definition.**

This is done with `fhicl::ParameterSet::get()` - it just parses the FHiCL file, and casts the value of the `LABEL : VALUE` pair to the type we specify.

Modifications – Analyze

CLASS DEFINITION
CONSTRUCTOR
ANALYZE
OPTIONAL METHOD

```
void CoolScience::analyze(art::Event const& e)
{
  // Implementation of required member function here.
  art::Handle<std::vector<rb::Prong> > prongHandle;
  e.getByLabel(fProngLabel, prongHandle);
  std::vector< art::Ptr< rb::Prong > > prongPtrVector;
  art::fill_ptr_vector(prongPtrVector, prongHandle);

  for (art::Ptr<rb::Prong> prong : prongPtrVector){

    MF_LOG_VERBATIM("CoolScience")
      << "prong has length " << prong->TotalLength();

  }
}
```

There are a number of ways to access data products, but this gets us to something you'll be more used to seeing. See **here** for details.

The first four lines just access the data product we want.

Data products are accessed using **art::Handle**'s

➔ basically a container with some additional methods for safety & getting more information

`e.getByLabel` gets the data product using the ProngLabel we defined, and fills the Handle.

The last two highlighted lines then just convert this into a vector of **art::Ptr**'s

➔ Think of this like a regular ptr, but with some additional methods

Modifications - Analyze #2

CLASS DEFINITION
CONSTRUCTOR
ANALYZE
OPTIONAL METHOD

```
void CoolScience::analyze(art::Event const& e)
{
  // Implementation of required member function here.
  art::Handle<std::vector<rb::Prong> > prongHandle;
  e.getByLabel(fProngLabel, prongHandle);
  std::vector< art::Ptr< rb::Prong > > prongPtrVector;
  art::fill_ptr_vector(prongPtrVector, prongHandle);

  for (art::Ptr<rb::Prong> prong : prongPtrVector){

    MF_LOG_VERBATIM("CoolScience")
      << "prong has length " << prong->TotalLength();

  }
}
```

This part is just looping over the vector and printing out the prong lengths

`MF_LOG_VERBATIM` is used rather than `std::cout` because it means that you can configure it through the Message Facility (another *art* service), but here it's just acting like a typical `cout` statement.

There's a bunch of different output streams in [MessageLogger.h](), I find the ones I use most are

`MF_LOG_VERBATIM` - print this (basically `cout`)
`MF_LOG_DEBUG` - only show in debug builds

Now, Running!

I don’t think there’s a lot to gain from having you write a FHiCL file (for these simple cases, I almost always copy/paste another one and just replace the relevant parts), so I’ve written one for you.

```
ifdh cp /pnfs/nova/persistent/users/alister1/immutable/coolsciencejob.fcl\
$SRT_PRIVATE_CONTEXT/Demo/
```

`cd` to the top of your test release and re-build using `novasoft_build -t`

Now run:

```
art -c coolsciencejob.fcl -n 1 $TESTFILE
```

And you should see some output printing the lengths of the prongs in the first event!

Now The Real Deal – Let's Make A TTree!

Say we want to not only print the values, but make some distributions - we can do that!

To make an output root file with a TTree in it, we're going to use the TFileService.

First up, add some more includes to the top of your `CoolScience` module:

```
#include "art_root_io/TFileService.h"
#include "TTree.h"
```

Modifications - Class Definition

```
class CoolScience : public art::EDAnalyzer {
public:
  explicit CoolScience(fhicl::ParameterSet const& p);
  // The compiler-generated destructor is fine for non-base
  // classes without bare pointers or other resource use.

  // Plugins should not be copied or assigned.
  CoolScience(CoolScience const&) = delete;
  CoolScience(CoolScience&&) = delete;
  CoolScience& operator=(CoolScience const&) = delete;
  CoolScience& operator=(CoolScience&&) = delete;

  // Required functions.
  void analyze(art::Event const& e) override;

  // Selected optional functions.
  void beginJob() override;

private:

  // Declare member data here.
  art::ServiceHandle<art::TFileService> tfs;
  std::string fProngLabel;
  double prongLength;
  TTree* tree;

};
```

In the class definition, we're going to add some more variables, and instantiate the TFileService, which comes in as an art::ServiceHandle.

CLASS DEFINITION
CONSTRUCTOR
ANALYZE
OPTIONAL METHOD

Modifications - beginJob

Now we're actually going to use the beginJob method we added to our module!

This is done with `tfs->make< T >` , where `T` is the type of object you want to put in the file, and the arguments in the parentheses are the arguments you'd usually give to the ROOT constructor for that type of object. From the ROOT constructor for the TTree,

TTree (const char *name, const char *title, Int_t splitlevel=99, TDirectory *dir=gDirectory)

Normal tree constructor. More...

we can see that the options are the name and title for the TTree.

We then define the tree branch in the same way you would in a ROOT script.

```
void CoolScience::beginJob()
{
  // Implementation of optional member function here.
  tree = tfs->make< TTree >("coolsci_tree", "Cool Science Analysis Tree");
  tree->Branch("prongLength", &prongLength);
}
```

Modifications – Analyze

CLASS DEFINITION
CONSTRUCTOR
ANALYZE
OPTIONAL METHOD

```
void CoolScience::analyze(art::Event const& e)
{
  // Implementation of required member function here.
  art::Handle<std::vector<rb::Prong> > prongHandle;
  e.getByLabel(fProngLabel, prongHandle);
  std::vector< art::Ptr< rb::Prong > > prongPtrVector;
  art::fill_ptr_vector(prongPtrVector, prongHandle);

  for (art::Ptr<rb::Prong> prong : prongPtrVector){

    prongLength = prong->TotalLength();
    MF_LOG_VERBATIM("CoolScience")
      << "prong has length " << prongLength;

    tree->Fill();
  }
}
```

We then fill the tree in the same way that we would in a regular ROOT script!

… and we're done! Let's run this!

Here We Go Again...

Re build, `novasoft_build -t`, and now run over all events!

```
art -c coolsciencejob.fcl $TESTFILE -T
/nova/ana/users/${USER}/coolscience.root
```

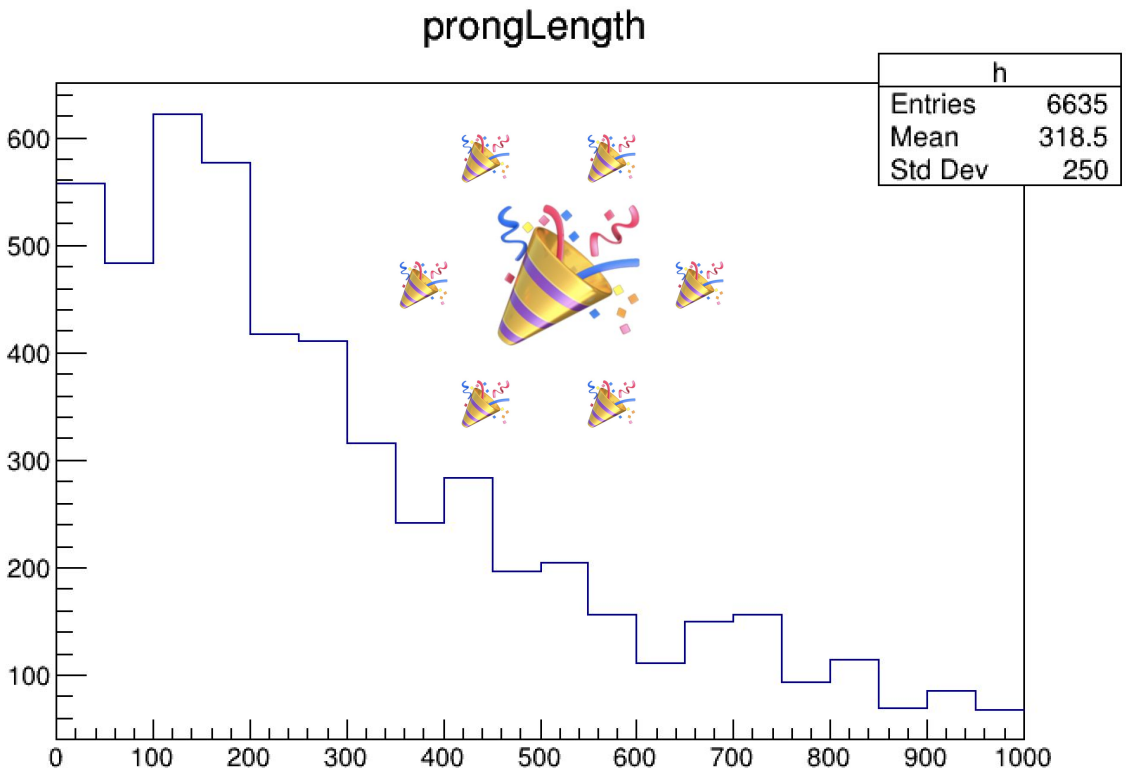
-T option redirects the output of the root file to somewhere else

We are good citizens and store our files in the /nova/ana area!

If that all worked, you should have an output file in your `/nova/ana/users/${USER}` area: `coolscience.root`. Open it up and…

```
root -l /nova/ana/users/${USER}/coolscience.root
TTree* tree = (TTree*)_file0->Get("ana/coolsci_tree")
tree->Draw("prongLength >> h(20, 0, 1000)")
```

After All That



Summarising All That...

- ➔ I've just shown you a (relatively) simple example of how to pull out prong information from an artroot file in novasoft using an analyzer module
  - ◆ This isn't limited to prongs though! Example: want to look at vertices? Get a handle to the rb::Vertex objects! Anything you can see in the eventdump is fair game!
  - ◆ This isn't limited to novasoft! Everything here is generic to *art*.
- ➔ This has only really covered analyzer modules, producer modules and filter modules are more complicated (but only a little) - look for examples in the repository if you want to write them!
- ➔ There are more complicated topics in art - the one you're most likely to come across are *associations*. Basically "Get me all of the hits associated with this prong!".
  - ◆ See **this page on the nova wiki**, or again, look at examples!
- ➔ Thanks for dialling in! I hope you learned something!

Some Other Places To Look

- ➔ art framework on NOvA wiki
- ➔ art wiki
- ➔ NOvASoft dOxygen
- ➔ Justin Vasel's 2018 YN art tutorial
- ➔ Justin Vasel's 2020 YN art tutorial
- ➔ Gavin's coding conventions [obey, or else!]

Backup

setup_nova

Add this function

```
function setup_nova {
       source /cvmfs/nova.opensciencegrid.org/novasoft/slf6/novasoft/setup/setup_nova.sh "$@"
       Setup_fnal_security
}
```

to your `~/.bashrc` file, and then `source ~/.bashrc`

CoolScience_module - Class Definition

```
class CoolScience : public art::EDAnalyzer {
public:
  explicit CoolScience(fhicl::ParameterSet const& p);
  // The compiler-generated destructor is fine for non-base
  // classes without bare pointers or other resource use.

  // Plugins should not be copied or assigned.
  CoolScience(CoolScience const&) = delete;
  CoolScience(CoolScience&&) = delete;
  CoolScience& operator=(CoolScience const&) = delete;
  CoolScience& operator=(CoolScience&&) = delete;

  // Required functions.
  void analyze(art::Event const& e) override;

  // Selected optional functions.
  void beginJob() override;

private:

  // Declare member data here.
  std::string fProngLabel;
  art::ServiceHandle<art::TFileService> tfs;
  double prongLength;
  TTree* tree;

};
```

CoolScience_module - Constructor

```
CoolScience::CoolScience(fhicl::ParameterSet const& p)
  : EDAnalyzer{p}  // ,
  // More initializers here.
{
  // Call appropriate consumes<>() for any products to be retrieved by this module.
  fProngLabel = p.get<std::string>("ProngLabel", "elasticarmshs");
}
```

CoolScience_module - analyze

```
void CoolScience::analyze(art::Event const& e)
{
  // Implementation of required member function here.
  art::Handle<std::vector<rb::Prong>> prongHandle;
  e.getByLabel(fProngLabel, prongHandle);
  std::vector<art::Ptr<rb::Prong>> prongPtrVector;
  art::fill_ptr_vector(prongPtrVector, prongHandle);

  for (art::Ptr<rb::Prong> prong : prongPtrVector){

    prongLength = prong->TotalLength();
    MF_LOG_VERBATIM("CoolScience")
      << "prong has length" << prong->TotalLength();

    tree->Fill();
  }

}
```

CoolScience_module - beginJob

```
void CoolScience::beginJob()
{
  // Implementation of optional member function here.
  tree = tfs->make<TTree>("coolsci_tree", "Cool Science Analysis Tree");
  tree->Branch("prongLength", &prongLength);
}
```