Writing your First Analyzer

IDK^th October 2023

9th UK LArTPC Software & Analysis Workshop

Henry Lay & Isobel Mawby
h.lay@lancaster.ac.uk &
i.mawby1@lancaster.ac.uk
#larsoft_analysis

Overview & aims of this session

- Learn how to do some physics with the reconstructed events you produced
    - Don’t worry if you didn’t manage to make the files, I’ll point you to some we’ve made
- Learn how to access the reconstructed neutrino information
    - There is a generic procedure for accessing almost all of the neutrino information you have in every file you’ve made this week
- We’ll look at:
    - Reconstruction objects produced by Pandora and downstream reconstruction
    - Associations of these objects to higher-level information
    - Take your time & try to understand everything you do
- Hopefully we’ll be able to make some plots

Thanks to all who have given this tutorial over the last few years, these slides have been (very marginally) adapted from those previous versions.

Side note

- We have included what will probably be far too much to achieve in these sessions
- But hopefully it’s all structured clearly enough that you can continue with the exercises in your own time
- So please don’t worry if you don’t make it hugely far through this tutorial, there’s supposed to be too much content
- If you are reading these slides as a PDF, you might prefer to look at the Google Slides link explicitly, as some code blocks render better there

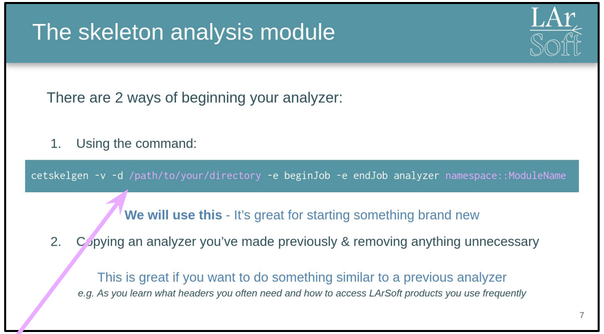
Slide Structure

‘New Topic’ Slide

1. The **Analyzer** Skeleton

6

‘Lecture’ Slide



The skeleton analysis module

There are 2 ways of beginning your analyzer:

1. Using the command:

```
cetskelgen -v -d /path/to/your/directory -e beginJob -e endJob analyzer namespace::ModuleName
```

We will use this - It’s great for starting something brand new

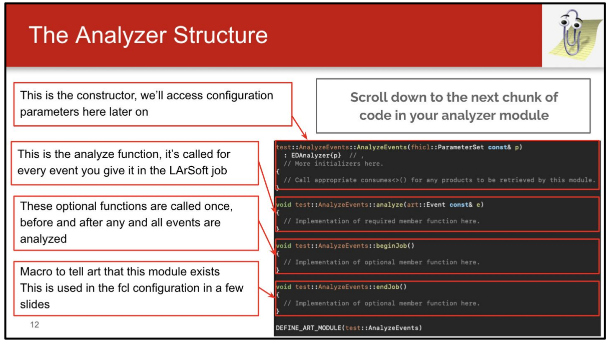
2. Copying an analyzer you’ve made previously & removing anything unnecessary

This is great if you want to do something similar to a previous analyzer

e.g. As you learn what headers you often need and how to access LArSoft products you use frequently

7

‘Exercise’ Slide



The Analyzer Structure

This is the constructor, we’ll access configuration parameters here later on

Scroll down to the next chunk of code in your analyzer module

This is the analyze function, it’s called for every event you give it in the LArSoft job

These optional functions are called once, before and after any and all events are analyzed

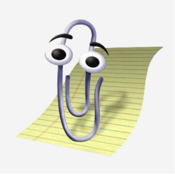
Macro to tell art that this module exists This is used in the fcl configuration in a few slides

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

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

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

void test::AnalyzeEvents::endJob()
{
  // Implementation of optional member function here.
}

DEFINE_ART_MODULE(test::AnalyzeEvents)
```

12

The pink text indicates places where you need to replace the line with your personal version.

The helpers around the room are here to be your (less sassy) clippy…



1. The **Analyzer** Skeleton

The skeleton analysis module

There are 2 ways of beginning your analyzer:

1. Using the command:

```
cetskelgen -v -d /path/to/your/directory -e beginJob -e endJob analyzer namespace::ModuleName
```

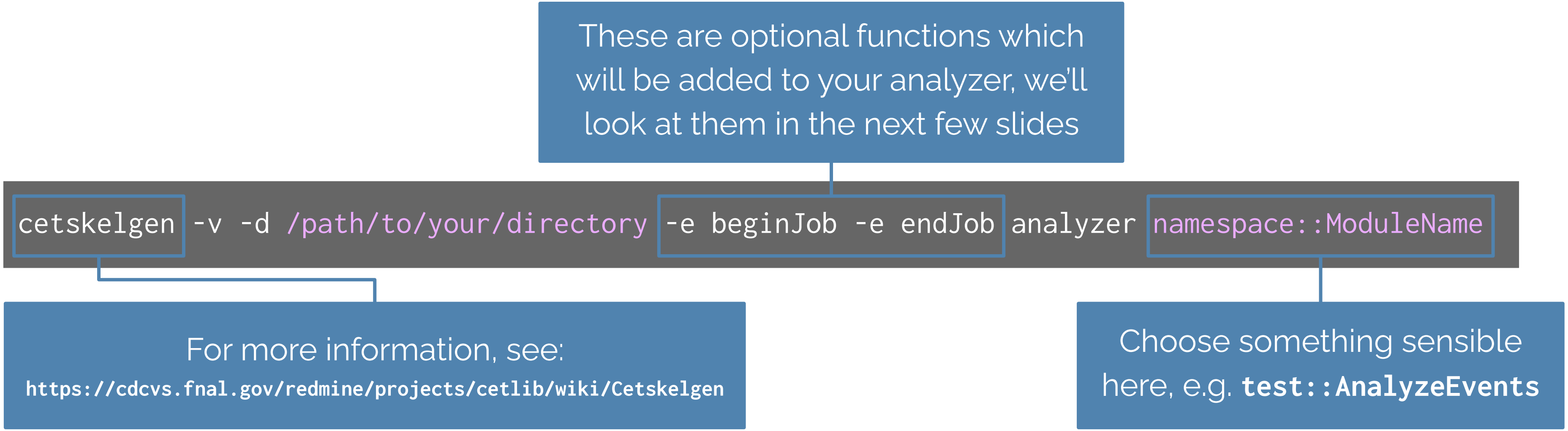
We will use this - It’s great for starting something brand new

2. Copying an analyzer you’ve made previously & removing anything unnecessary

This is great if you want to do something similar to a previous analyzer

e.g. As you learn what headers you often need and how to access LArSoft products you use frequently

cetskelgen



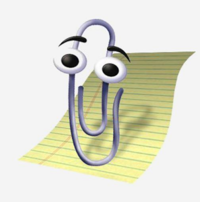
These are optional functions which will be added to your analyzer, we'll look at them in the next few slides

```
cetskelgen -v -d /path/to/your/directory -e beginJob -e endJob analyzer namespace::ModuleName
```

For more information, see:
`https://cdcvs.fnal.gov/redmine/projects/cetlib/wiki/Cetskelgen`

Choose something sensible here, e.g. **`test::AnalyzeEvents`**

Let’s do it!



If you are using a fresh terminal you will need to setup again:

```
source /cvmfs/sbnd.opensciencegrid.org/products/sbnd/setup_sbnd.sh
source /PATH/TO/YOUR/BUILD/AREA/localProducts*/setup
mrbslp
```

1. Navigate here:

We’ve put the CMakeLists.txt and build.sh files here…

```
cd $MRB_SOURCE/sbndcode/sbndcode/Workshop/Analysis
```

2. Type the cetskelgen command:

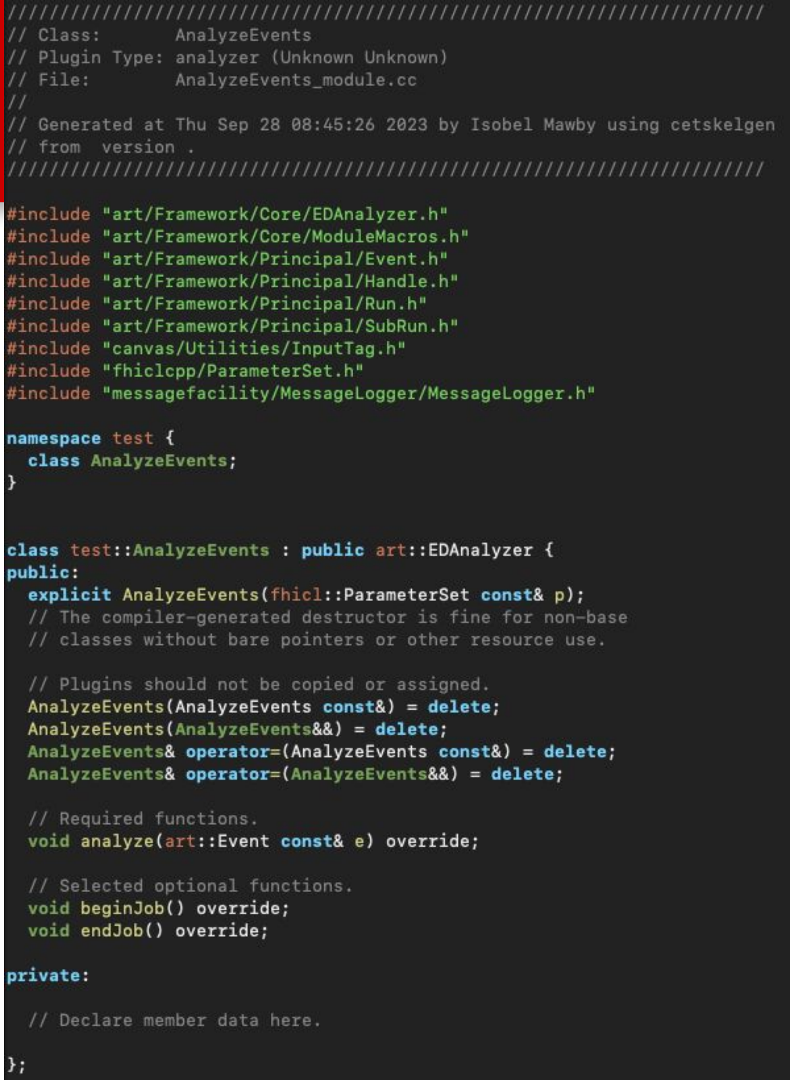
The full stop tells `cetskelgen` to place the analysis module in the current directory

```
cetskelgen -v -d . -e beginJob -e endJob analyzer test::AnalyzeEvents
```

What did we create?

- You should now find a file called **`AnalyzeEvents_module.cc, this is your analyzer!`**
- Open this!
- The top section should look something like the snippet on the right

(but most likely with a less ugly colour theme, apologies…)



```
////////////////////////////////////////////////////////////////////////
// Class:       AnalyzeEvents
// Plugin Type: analyzer (Unknown Unknown)
// File:        AnalyzeEvents_module.cc
//
// Generated at Thu Sep 28 08:45:26 2023 by Isobel Mawby using cetskelgen
// from  version .
////////////////////////////////////////////////////////////////////////

#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"

namespace test {
  class AnalyzeEvents;
}


class test::AnalyzeEvents : public art::EDAnalyzer {
public:
  explicit AnalyzeEvents(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.
  AnalyzeEvents(AnalyzeEvents const&) = delete;
  AnalyzeEvents(AnalyzeEvents&&) = delete;
  AnalyzeEvents& operator=(AnalyzeEvents const&) = delete;
  AnalyzeEvents& operator=(AnalyzeEvents&&) = delete;

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

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

private:

  // Declare member data here.

};
```

The Analyzer Structure

This is some information to explain what's in the file to someone who might want to use it

Or just for your forgetful, future self

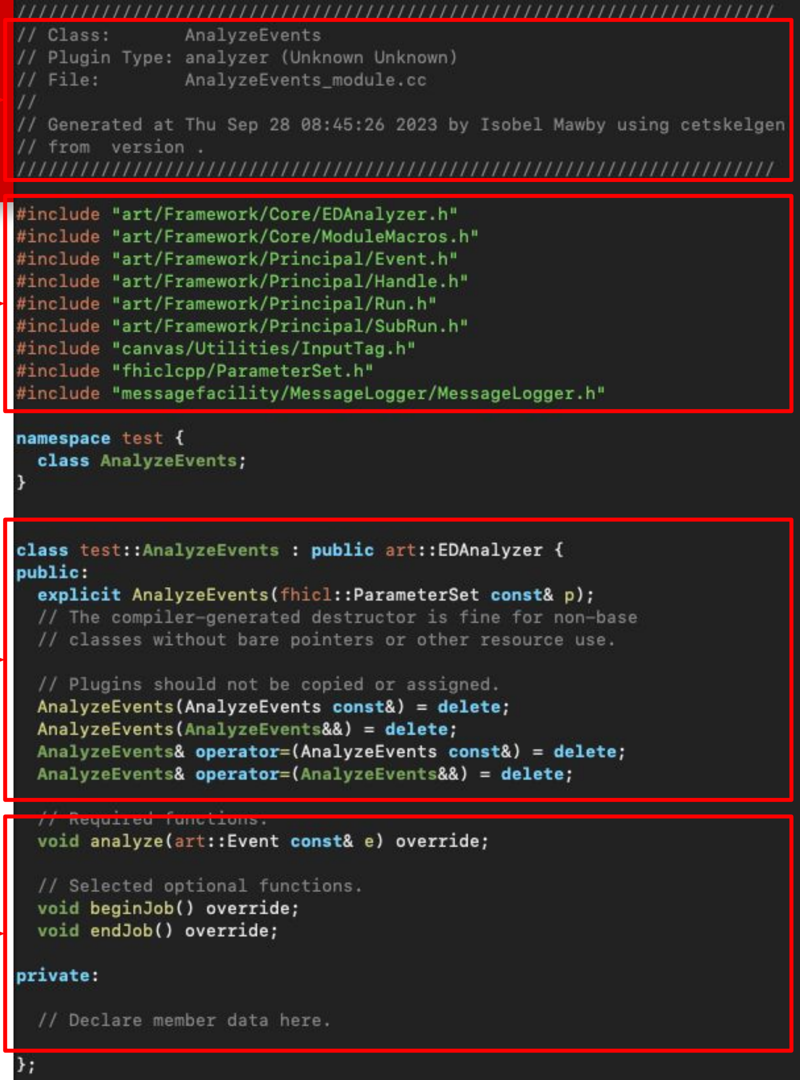
These are the default headers which should hopefully allow the empty analyzer to build

You'll add to these later!

Setting up the class you've just created

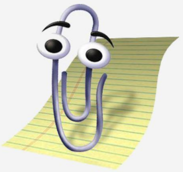
You shouldn't need to touch these

These are the functions you're going to modify for the analysis



```
////////////////////////////////////////////////////////////////////////
// Class:       AnalyzeEvents
// Plugin Type: analyzer (Unknown Unknown)
// File:        AnalyzeEvents_module.cc
//
// Generated at Thu Sep 28 08:45:26 2023 by Isobel Mawby using cetskelgen
// from  version .
////////////////////////////////////////////////////////////////////////

#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"

namespace test {
  class AnalyzeEvents;
}


class test::AnalyzeEvents : public art::EDAnalyzer {
public:
  explicit AnalyzeEvents(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.
  AnalyzeEvents(AnalyzeEvents const&) = delete;
  AnalyzeEvents(AnalyzeEvents&&) = delete;
  AnalyzeEvents& operator=(AnalyzeEvents const&) = delete;
  AnalyzeEvents& operator=(AnalyzeEvents&&) = delete;

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

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

private:

  // Declare member data here.

};
```

The Analyzer Structure



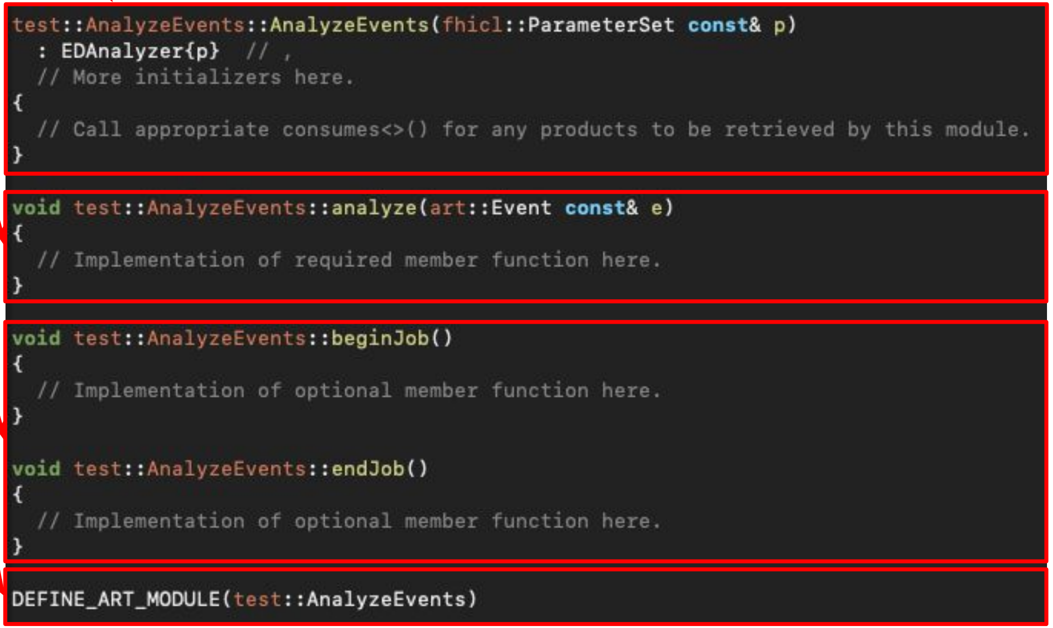
This is the constructor, we'll access configuration parameters here later on

This is the analyze function, it's called for every event you give it in the LArSoft job

These optional functions are called once, before and after any and all events are analyzed

Macro to tell art that this module exists
This is used in the fcl configuration in a few slides

Scroll down to the next chunk of code in your analyzer module

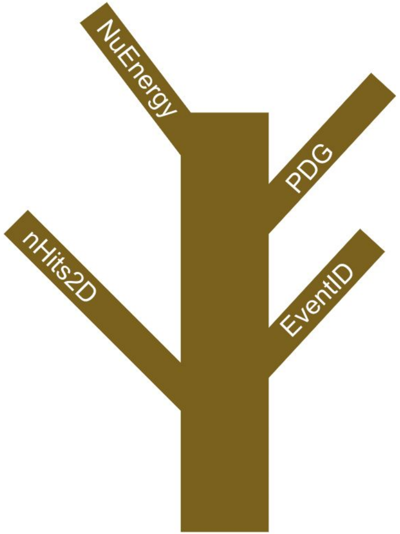


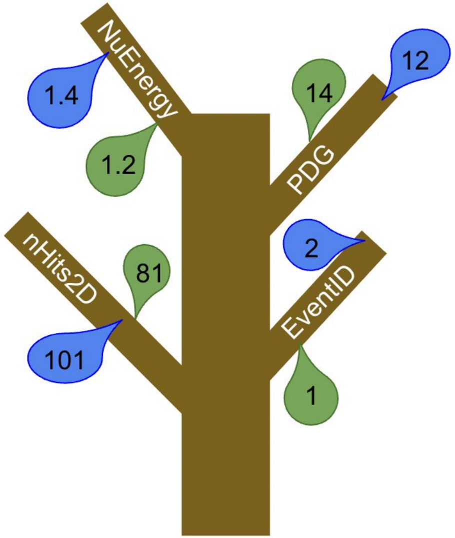
```
test::AnalyzeEvents::AnalyzeEvents(fhicl::ParameterSet const& p)
  : EDAnalyzer{p}  // ,
  // More initializers here.
{
  // Call appropriate consumes<>() for any products to be retrieved by this module.
}

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

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

void test::AnalyzeEvents::endJob()
{
  // Implementation of optional member function here.
}

DEFINE_ART_MODULE(test::AnalyzeEvents)
```

2. Obtaining Our First Analysis Information

Writing out Analysis Information

1) We're going to create a ROOT TTree to store our analysis information

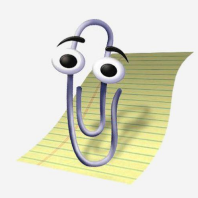




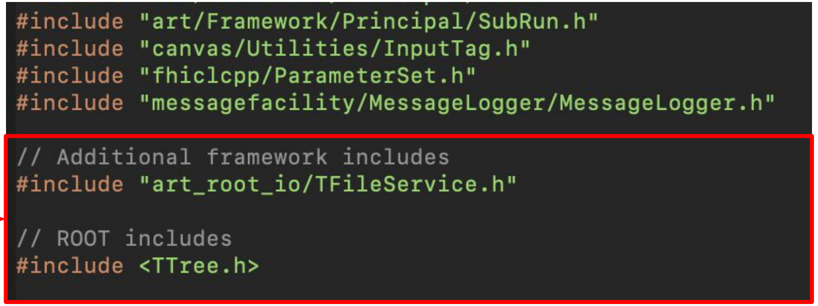
2) Then we will add to our tree, the 'Event ID' of our created events

https://root.cern.ch/doc/master/classTTree.html

Creating a TTree

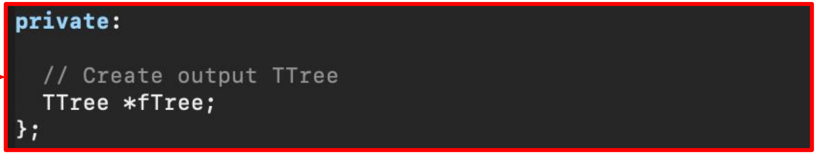


Add relevant LArSoft & ROOT headers



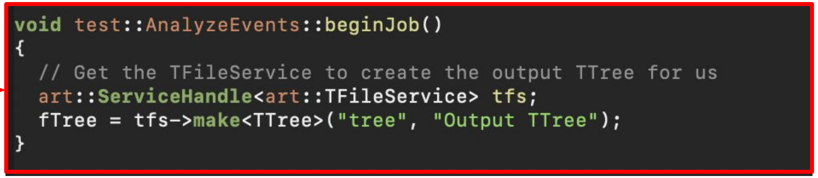
```
#include "art/Framework/Principal/SubRun.h"
#include "canvas/Utilities/InputTag.h"
#include "fhiclcpp/ParameterSet.h"
#include "messagefacility/MessageLogger/MessageLogger.h"

// Additional framework includes
#include "art_root_io/TFileService.h"

// ROOT includes
#include <TTree.h>
```

Declare TTree



```
private:

  // Create output TTree
  TTree *fTree;
};
```

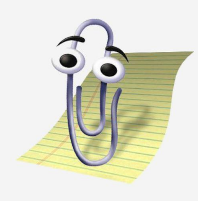
Create your TTree



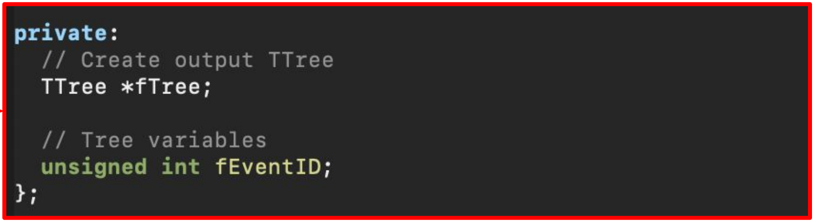
```
void test::AnalyzeEvents::beginJob()
{
  // Get the TFileService to create the output TTree for us
  art::ServiceHandle<art::TFileService> tfs;
  fTree = tfs->make<TTree>("tree", "Output TTree");
}
```

Note: The order represents their locations in the file

Writing Out a Variable



Declare event-based variables



```
private:
  // Create output TTree
  TTree *fTree;

  // Tree variables
  unsigned int fEventID;
};
```

Access our event ID from the LArSoft event we’re analysing & fill the TTree

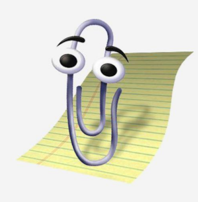
```
void test::AnalyzeEvents::analyze(art::Event const& e)
{
  // Set the event ID
  fEventID = e.id().event();

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

Add branches for the variables we want to fill

```
void test::AnalyzeEvents::beginJob()
{
  // Get the TFileService to create the output TTree for us
  art::ServiceHandle<art::TFileService> tfs;
  fTree = tfs->make<TTree>("tree", "Output TTree");

  // Add branches to TTree
  fTree->Branch("eventID", &fEventID);
}
```

Note: The order represents their locations in the file

Running the analysis module

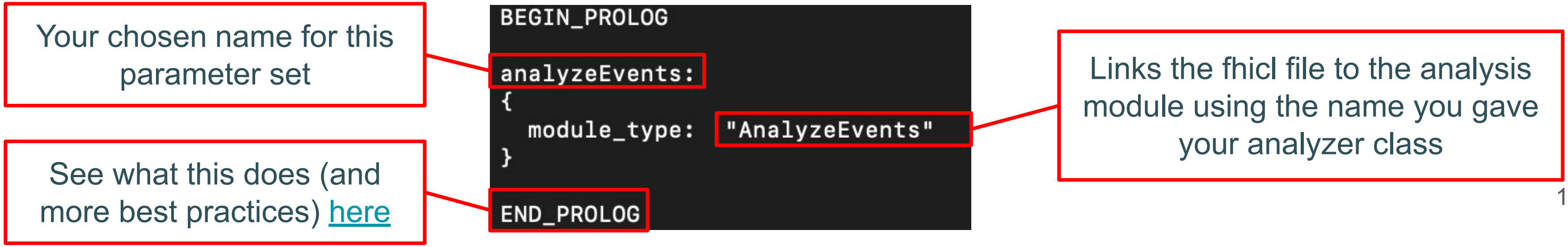


In order to be able to run the analyzer, we now need to write 2 fhicl files

- The first will configure our analysis - **an include fcl**
    - This is where we point the analyzer to the objects/parameters we want to access from the input files (this will make more sense soon…)
- The second will be used to run our analysis **- a run/job fcl**
    - This links together the configuration file and the analysis module

---

Fhicl 1: Configuring the analyzer. Create a file, e.g. `analysisConfig.fcl` & fill it with this:



Your chosen name for this parameter set

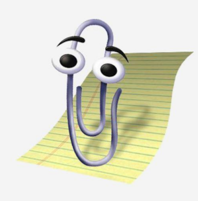
See what this does (and more best practices) here

```
BEGIN_PROLOG

analyzeEvents:
{
  module_type:  "AnalyzeEvents"
}

END_PROLOG
```

Links the fhicl file to the analysis module using the name you gave your analyzer class

Fhicl 2: Running the module

Create another file, e.g.
`run_analyzeEvents.fcl`
& fill it with this:



Include your analyzer configuration fhicl

Name this process

Must not include any underscores

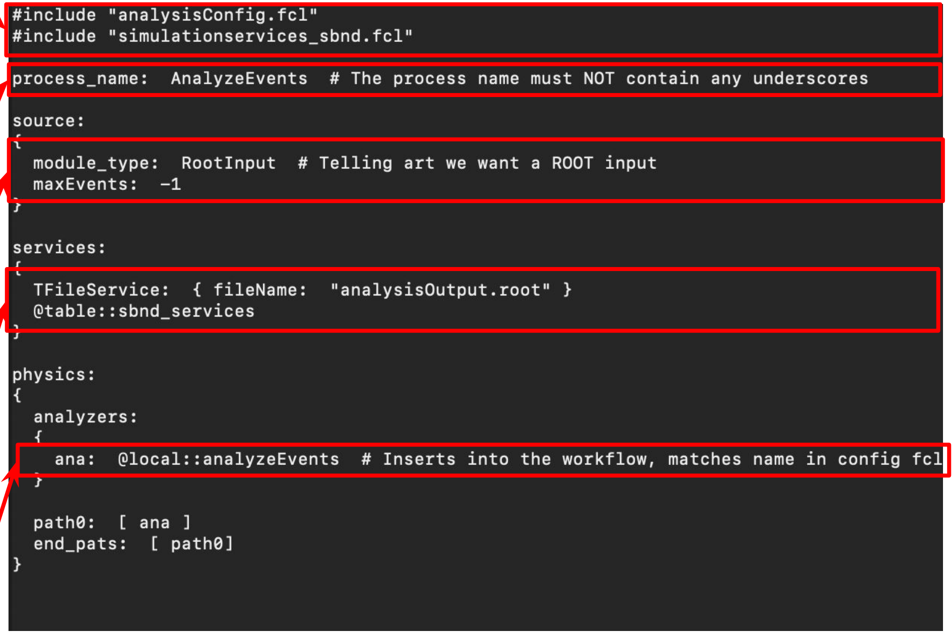
Tell it to expect a ROOT input file

Output filename

This is a default, and can be changed on the command line

ana sets our module **`analyzeEvents`** as part of the workflow

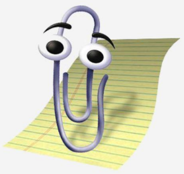
Note, this matches the name in the configuration fcl file



```
#include "analysisConfig.fcl"
#include "simulationservices_sbnd.fcl"

process_name:  AnalyzeEvents  # The process name must NOT contain any underscores

source:
{
  module_type:  RootInput  # Telling art we want a ROOT input
  maxEvents:  -1
}

services:
{
  TFileService:  { fileName:  "analysisOutput.root" }
  @table::sbnd_services
}

physics:
{
  analyzers:
  {
    ana:  @local::analyzeEvents  # Inserts into the workflow, matches name in config fcl
  }

  path0:  [ ana ]
  end_pats:  [ path0]
}
```

Let’s try running it…

Pre-made reconstructed events

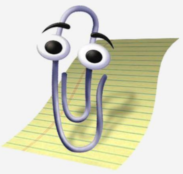


Haven’t made a reconstruction file? Don’t panic!

There is a pre-made reconstruction file which can be found here:

`/home/share/october2023/reconstruction/output_detsim_numu_like_Reco1Reco2.root`

Compiling and running your code



First, we need to compile what you've written so far

From the `$MRB_SOURCE/sbndcode/sbndcode/Workshop/Analysis` directory:

```
source build.sh
```

This has each build command in one place, have a look to make sure you're comfortable with what it does before using it

Then (when successful) run your analyzer!

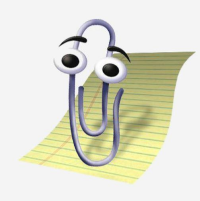
```
lar -c run_analyzeEvents.fcl -s /path/to/input/file.root -n 10
```

Let's just run over 10 events while we make sure things build.
We'll run on the whole sample later

Open the file in ROOT to investigate our output file...

```
root -l analysisOutput.root
```

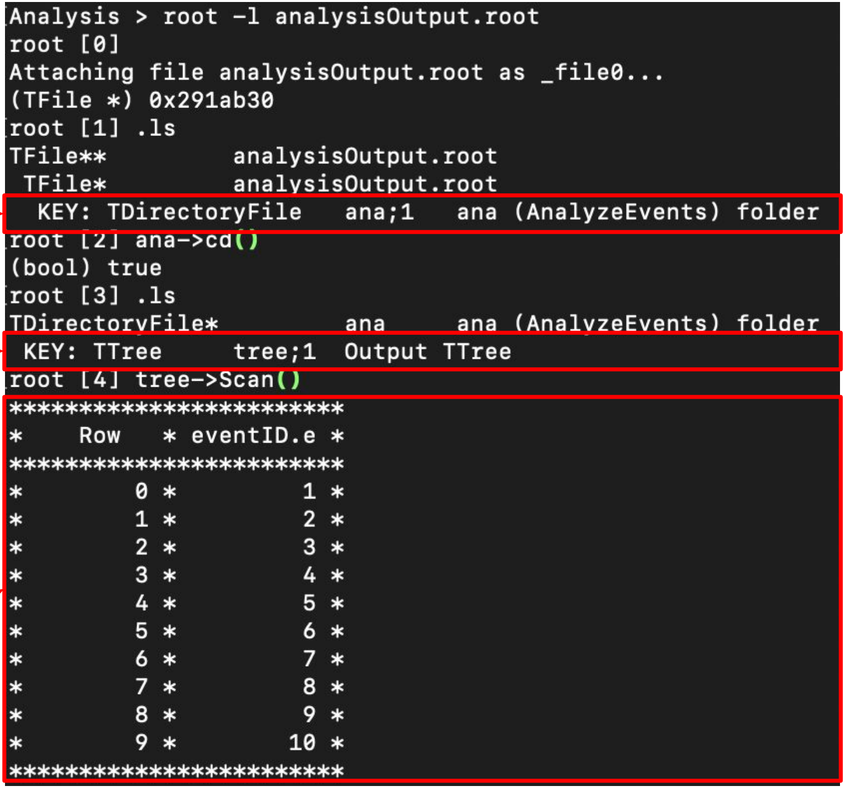
Looking at the output in ROOT



Here you can see that the name you gave to the analyzer in the fhicl run script is the name of your directory (**`ana`**): Open it with *cd*

You can see the output (T)Tree that we created, use *Scan* to view its contents (can also use Show(*entryNumber*), a TBrowser etc…)

Your **`tree`** exists and contains the **`eventIDs`**! Success! (hopefully)



```
Analysis > root -l analysisOutput.root
root [0]
Attaching file analysisOutput.root as _file0...
(TFile *) 0x291ab30
root [1] .ls
TFile**         analysisOutput.root
 TFile*         analysisOutput.root
  KEY: TDirectoryFile   ana;1   ana (AnalyzeEvents) folder
root [2] ana->cd()
(bool) true
root [3] .ls
TDirectoryFile*         ana     ana (AnalyzeEvents) folder
 KEY: TTree     tree;1  Output TTree
root [4] tree->Scan()
************************
*    Row   * eventID.e *
************************
*        0 *         1 *
*        1 *         2 *
*        2 *         3 *
*        3 *         4 *
*        4 *         5 *
*        5 *         6 *
*        6 *         7 *
*        7 *         8 *
*        8 *         9 *
*        9 *        10 *
************************
```

3. Accessing Data Products

A quick aside on how to access our reconstruction information, so we can obtain some cooler analysis information!

Accessing products from our files (1)

- Currently, just focused on EventID, but how do we access the information that we've added to the 'simulation/data' files e.g. in the Pandora stage?
- There are two ways the information is stored in these files:

```
std::vector<art::Ptr<recob::PFParticle>>

        {PFP_A,    PFP_B,     PFP_C}
```

1) As a vector of objects:

e.g. a vector of all PFParticles created by Pandora

```
{PFP_A → Vtx_B,
 PFP_B → Vtx_A,
 PFP_C → Vtx_C}
```

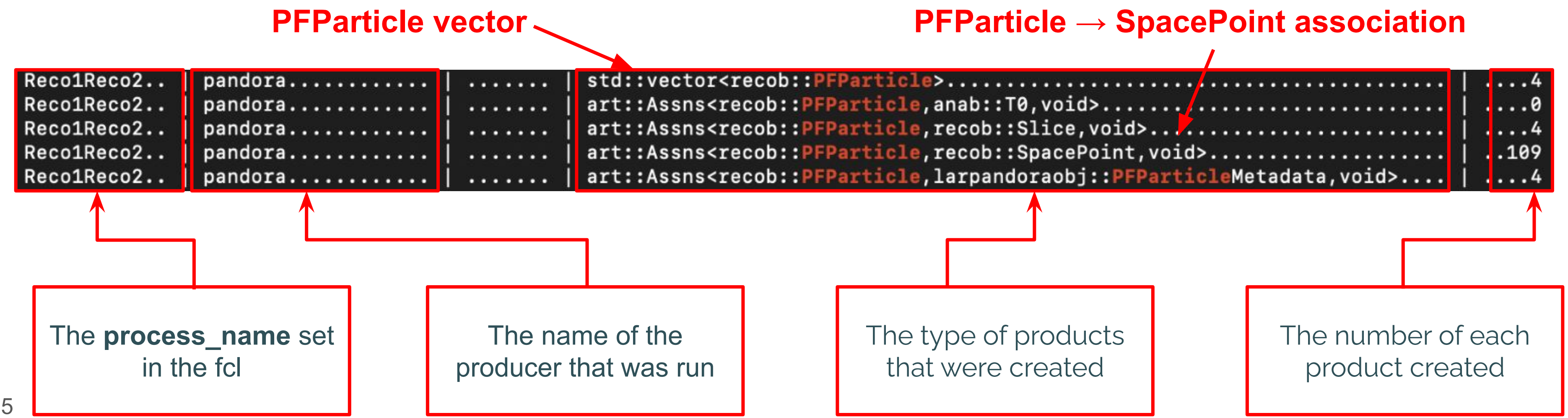
2) As associations:

e.g. links between PFParticles and their associated reconstructed vertex

Accessing products from our files (2)

- We can use eventdump.fcl to see what data products are saved in our ‘simulation/data’ files

```
lar -c eventdump.fcl whateverYourSimulationOrDataFileIsCalled.root -n 1
```



PFParticle vector

PFParticle → SpacePoint association

```
Reco1Reco2.. | pandora........... | ....... | std::vector<recob::PFParticle>.................................. | ....4
Reco1Reco2.. | pandora........... | ....... | art::Assns<recob::PFParticle,anab::T0,void>..................... | ....0
Reco1Reco2.. | pandora........... | ....... | art::Assns<recob::PFParticle,recob::Slice,void>................. | ....4
Reco1Reco2.. | pandora........... | ....... | art::Assns<recob::PFParticle,recob::SpacePoint,void>............ | ..109
Reco1Reco2.. | pandora........... | ....... | art::Assns<recob::PFParticle,larpandoraobj::PFParticleMetadata,void>.... | ....4
```

The **process_name** set in the fcl

The name of the producer that was run

The type of products that were created

The number of each product created

Accessing Vectors (the technical details)

- In our analyzer, let’s say that we want to obtain the vector of slices

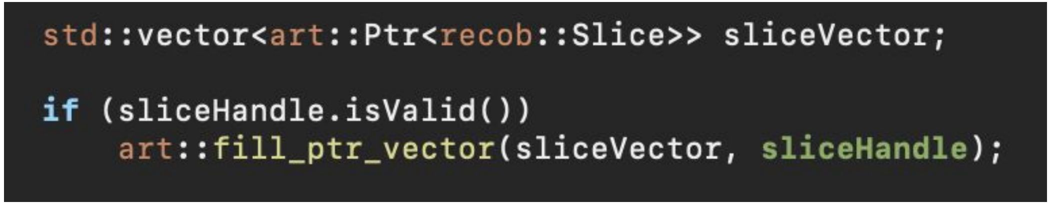
- We first need to set up the **data object handle**, consider this to be the link between your code and the object vector in the simulation/data files

```
art::ValidHandle<std::vector<recob::Slice>> sliceHandle = e.getValidHandle<std::vector<recob::Slice>>("pandora");
```

the type of object we’re after

the name of the producer that created it (see previous slide)

- After we check that our handle is valid, we can now retrieve the vector in our code



```
std::vector<art::Ptr<recob::Slice>> sliceVector;

if (sliceHandle.isValid())
    art::fill_ptr_vector(sliceVector, sliceHandle);
```

Accessing Associations (Technical Details)

- Say that, in our analyser, we want to obtain the vector of PFParticles connected to a given slice

Slice → **PFParticle Vector**

Some Isobel Jargon: ‘Considered Object’ ‘Associated Object Vector’

- We first initialise a **FindManyP** object, consider this to be a link between your code and the associations of a given object vector (in this case, the vector in which our considered slice lives)

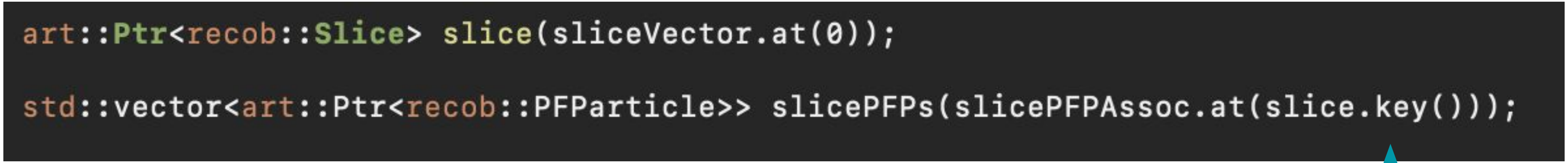
```
// Get associations between slices and pfparticles
art::FindManyP<recob::PFParticle> slicePFPAssoc(sliceHandle, e, "pandora");
```

our handle to the object vector

the name of the producer that created the association

Accessing Associations (Technical Details)

- To get the PFParticles associated to a particular slice, in this case the first slice in *sliceVector*
- We then do:



```
art::Ptr<recob::Slice> slice(sliceVector.at(0));

std::vector<art::Ptr<recob::PFParticle>> slicePFPs(slicePFPAssoc.at(slice.key()));
```

HEY ISOBEL/HENRY! What's that key function about?

What’s the key function about?

- Every `art::Ptr<...>` has a key function
- It returns the index of the ‘pointed to’ object in the vector in which it lives, and is used to identify the connected associations

Consider:

```
std::vector<art::Ptr<recob::Slice>> isobelsAwesomeSliceVector = {sliceA, sliceB, sliceC};
```

Then:

```
sliceA->key() == 0     sliceB->key() == 1     sliceC->key() == 2
```

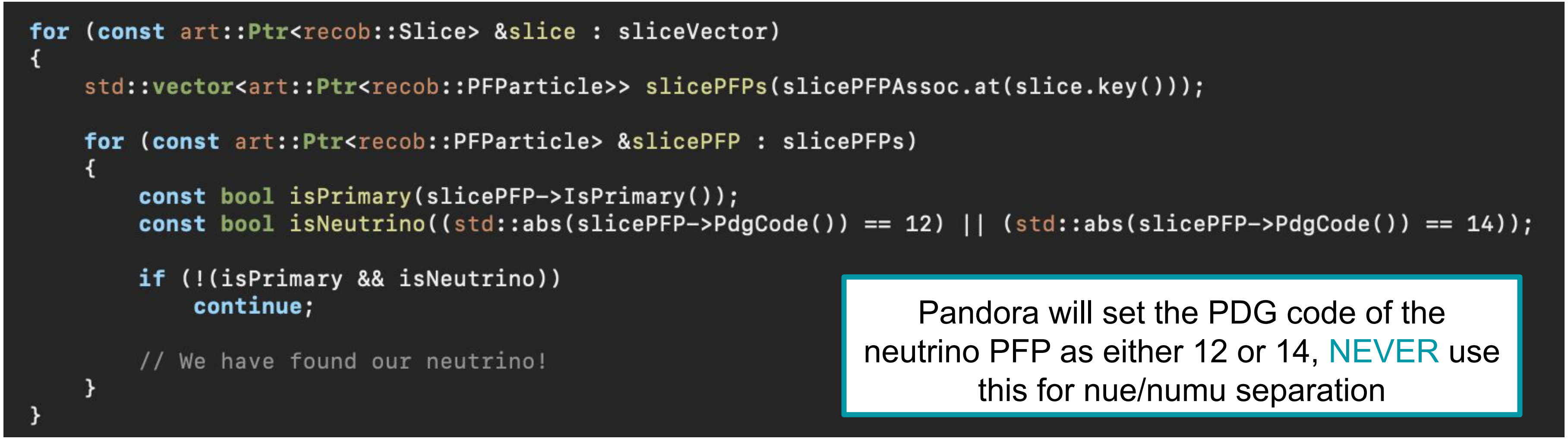
So, to get the PFParticle vector associated with sliceC, we’d do:

```
std::vector<art::Ptr<recob::PFParticle>> slicePFPs = slicePFPAssoc.at(sliceC.key());
```

4. Investigating our Neutrino Hierarchy

Obtaining the Neutrino Hierarchy

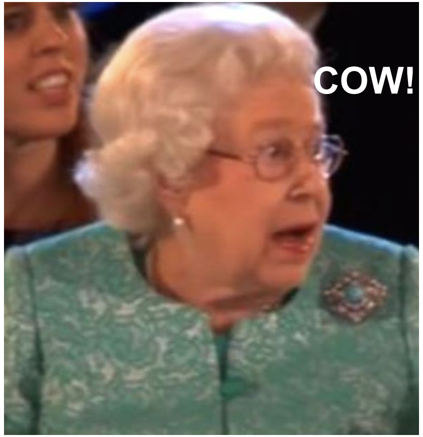
- In an experiment with background cosmic rays (like SBND), our reconstruction output will consist of slices, some containing cosmic-like hierarchies, others neutrino-like hierarchies.
- IN OUR OPINION, the best way to obtain the PFParticles of a neutrino hierarchy is:



```
for (const art::Ptr<recob::Slice> &slice : sliceVector)
{
    std::vector<art::Ptr<recob::PFParticle>> slicePFPs(slicePFPAssoc.at(slice.key()));

    for (const art::Ptr<recob::PFParticle> &slicePFP : slicePFPs)
    {
        const bool isPrimary(slicePFP->IsPrimary());
        const bool isNeutrino((std::abs(slicePFP->PdgCode()) == 12) || (std::abs(slicePFP->PdgCode()) == 14));

        if (!(isPrimary && isNeutrino))
            continue;

        // We have found our neutrino!
    }
}
```

Pandora will set the PDG code of the neutrino PFP as either 12 or 14, NEVER use this for nue/numu separation

The Neutrino Hierarchy in LArSoft



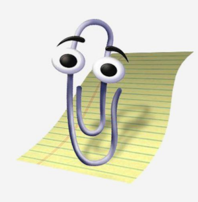
```
queeniePFP->Self() == 11
queeniePFP->Parent() == kPFParticlePrimary
queeniePFP->Daughters() == {1, 5, 7, 9}
```

```
didILeaveTheOvenOnPFP->Self() == 5
didILeaveTheOvenOnPFP->Parent() == 11
didILeaveTheOvenOnPFP->Daughter() == {4, 13}
```

```
willIAmPFP->Self() == 13
willIAmPFP->Parent() == 5
willIAmPFP->Daughters() == {3, 10, 6}
```

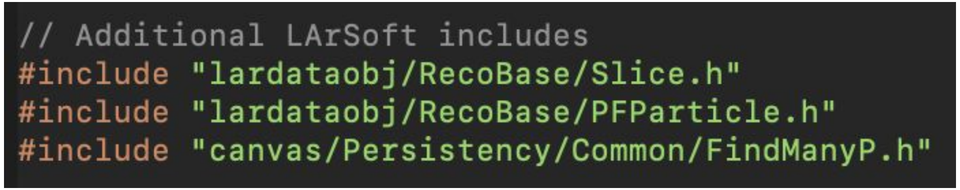
```
curiousGeorgePFP->Self() == 3
curiousGeorgePFP->Parent() == 13
curiousGeorgePFP->Daughters() == {}
```

Implementing Neutrino Hierarchy Variables (1)



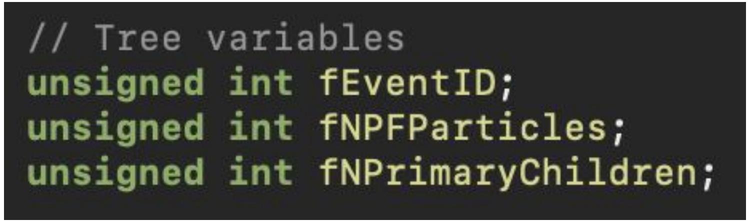
- Let’s ‘calculate’ some neutrino hierarchy variables, and add them to our tree!

1) First, we’ll need some new includes:



```
// Additional LArSoft includes
#include "lardataobj/RecoBase/Slice.h"
#include "lardataobj/RecoBase/PFParticle.h"
#include "canvas/Persistency/Common/FindManyP.h"
```

2) Create new member variables, and connect them to our (T)Tree



```
// Tree variables
unsigned int fEventID;
unsigned int fNPFParticles;
unsigned int fNPrimaryChildren;
```

```
// Add branches to TTree
fTree->Branch("eventID", &fEventID);
fTree->Branch("nPFParticles", &fNPFParticles);
fTree->Branch("nPrimaryChildren", &fNPrimaryChildren);
```

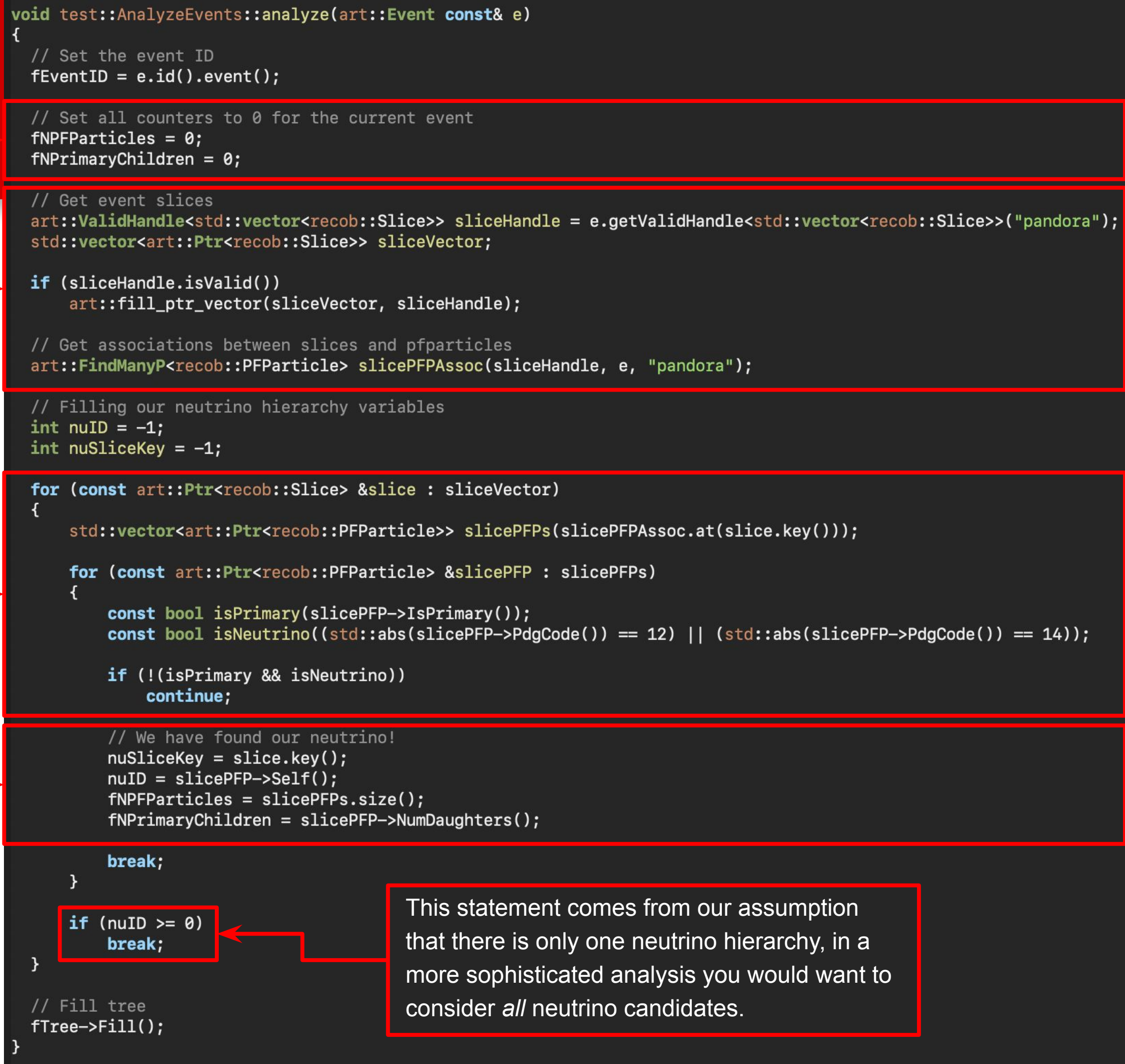
3) Calculate the neutrino hierarchy variables

Initialise our neutrino hierarchy variables to zero at the start of every event

Get the reconstructed slices in the event and the PFParticle associations

Loop through the slices until we find the neutrino PFParticle (here, we assume that, across all slices, there is only one neutrino candidate - this isn’t normally the case!)

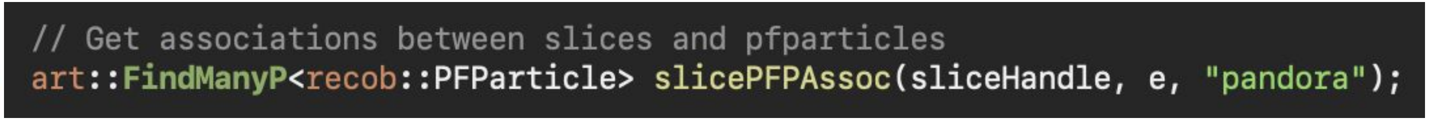
Fill the neutrino hierarchy variables, and note the neutrino ID (and the neutrino slice ID)



```
void test::AnalyzeEvents::analyze(art::Event const& e)
{
  // Set the event ID
  fEventID = e.id().event();

  // Set all counters to 0 for the current event
  fNPFParticles = 0;
  fNPrimaryChildren = 0;

  // Get event slices
  art::ValidHandle<std::vector<recob::Slice>> sliceHandle = e.getValidHandle<std::vector<recob::Slice>>("pandora");
  std::vector<art::Ptr<recob::Slice>> sliceVector;

  if (sliceHandle.isValid())
    art::fill_ptr_vector(sliceVector, sliceHandle);

  // Get associations between slices and pfparticles
  art::FindManyP<recob::PFParticle> slicePFPAssoc(sliceHandle, e, "pandora");

  // Filling our neutrino hierarchy variables
  int nuID = -1;
  int nuSliceKey = -1;

  for (const art::Ptr<recob::Slice> &slice : sliceVector)
  {
    std::vector<art::Ptr<recob::PFParticle>> slicePFPs(slicePFPAssoc.at(slice.key()));

    for (const art::Ptr<recob::PFParticle> &slicePFP : slicePFPs)
    {
      const bool isPrimary(slicePFP->IsPrimary());
      const bool isNeutrino((std::abs(slicePFP->PdgCode()) == 12) || (std::abs(slicePFP->PdgCode()) == 14));

      if (!(isPrimary && isNeutrino))
        continue;

      // We have found our neutrino!
      nuSliceKey = slice.key();
      nuID = slicePFP->Self();
      fNPFParticles = slicePFPs.size();
      fNPrimaryChildren = slicePFP->NumDaughters();

      break;
    }

    if (nuID >= 0)
      break;
  }

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

This statement comes from our assumption that there is only one neutrino hierarchy, in a more sophisticated analysis you would want to consider *all* neutrino candidates.

HARD CODING MODULE NAMES IS A VERY VERY VERY BAD IDEA!



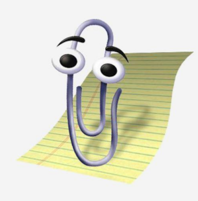
```
// Get associations between slices and pfparticles
art::FindManyP<recob::PFParticle> slicePFPAssoc(sliceHandle, e, "pandora");
```

```
// Get associations between slices and pfparticles
art::FindManyP<recob::PFParticle> slicePFPAssoc(sliceHandle, e, m_sliceLabel);
```

Save module names as member variables instead!

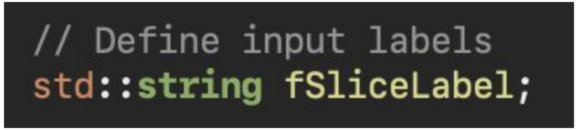
We'll see how to do this in the next few slides…

Implementing Neutrino Hierarchy Variables (4)



- We pass module names into our analyzer through the analysisConfig.fcl file:

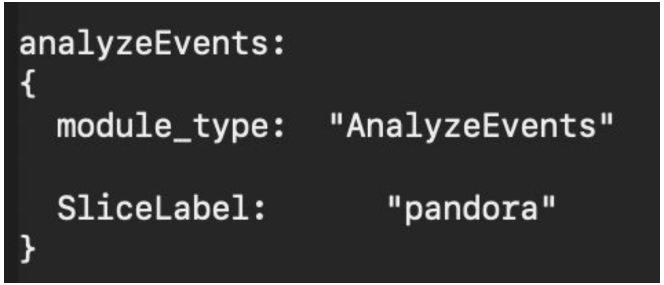
In your analyzer:



```
// Define input labels
std::string fSliceLabel;
```

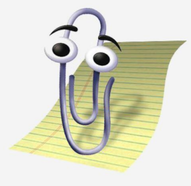
```
test::AnalyzeEvents::AnalyzeEvents(fhicl::ParameterSet const& p)
    : EDAnalyzer{p},
    fSliceLabel(p.get<std::string>("SliceLabel"))
{
  // Call appropriate consumes<>() for any products to be retrieved by this module.
}
```

In analysisConfig.fcl:



```
analyzeEvents:
{
   module_type:   "AnalyzeEvents"

   SliceLabel:        "pandora"
}
```

Fhicl configuration file linking & running



```
source build.sh
```

Compile changes

```
lar -c run_analyzeEvents.fcl -s /path/to/input/file.root -n 10
```

Run analyzer

```
root -l analysisOutput.root
```

Check output

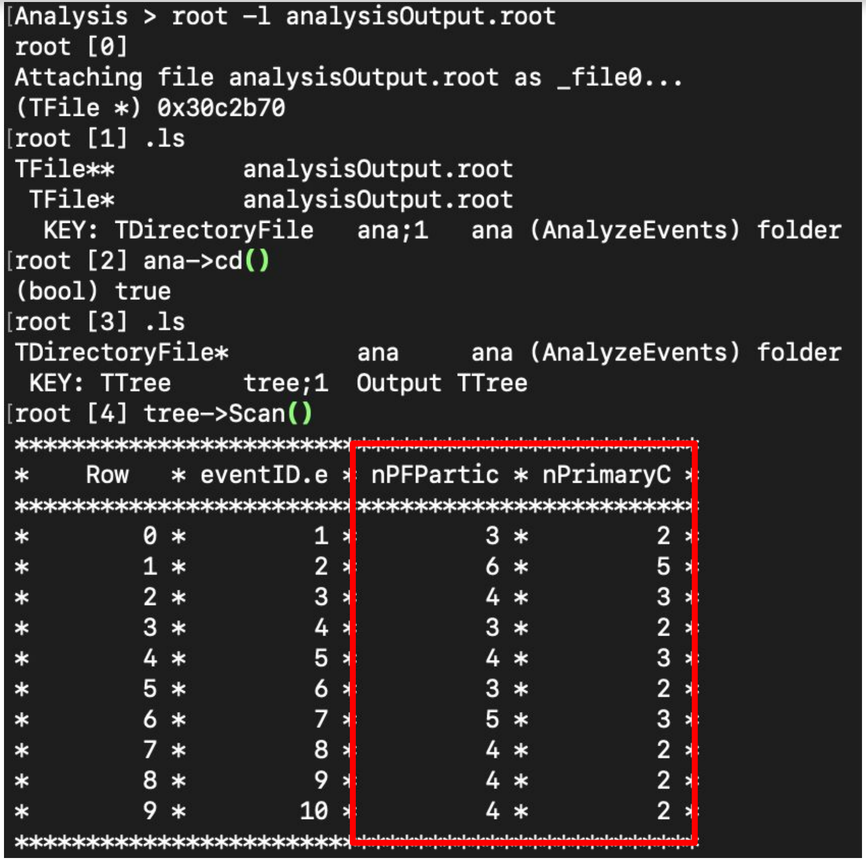
What our output looks like now

- Our (T)Tree should now have 2 new branches

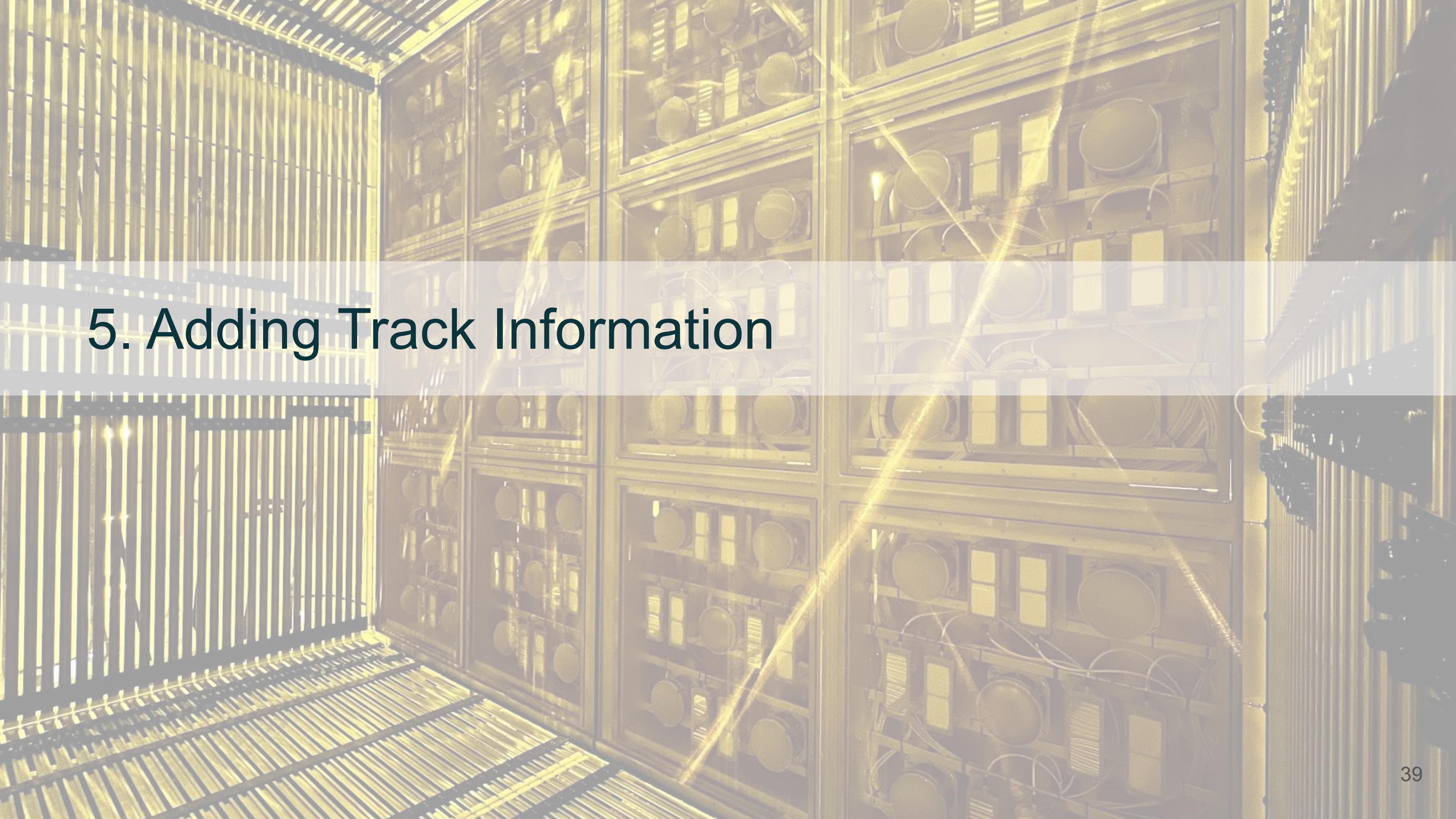
nPFParticles tells us how many particle we have reconstructed

nPrimaryChildren is the number of primary particles (children of the neutrino) we have reconstructed

- By viewing the tree, we can check that everything looks sensible…



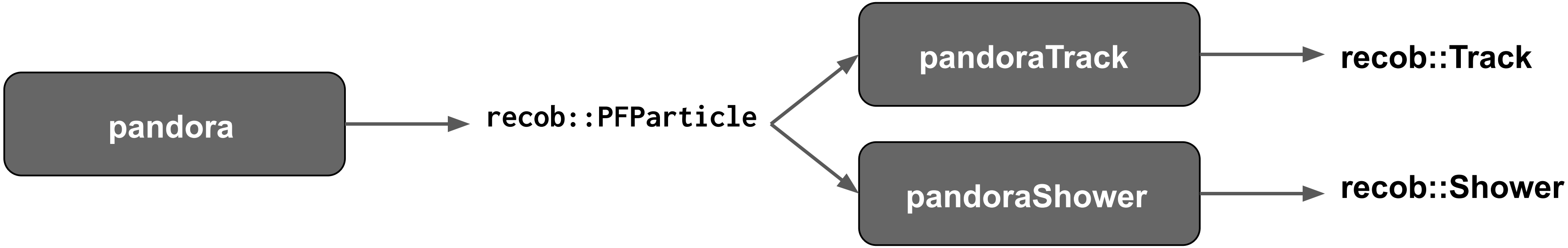
```
[Analysis > root -l analysisOutput.root
root [0]
Attaching file analysisOutput.root as _file0...
(TFile *) 0x30c2b70
[root [1] .ls
TFile**         analysisOutput.root
 TFile*         analysisOutput.root
  KEY: TDirectoryFile   ana;1   ana (AnalyzeEvents) folder
[root [2] ana->cd()
(bool) true
[root [3] .ls
TDirectoryFile*         ana     ana (AnalyzeEvents) folder
 KEY: TTree     tree;1  Output TTree
[root [4] tree->Scan()
************************************************
*    Row   * eventID.e * nPFPartic * nPrimaryC *
************************************************
*        0 *         1 *         3 *         2 *
*        1 *         2 *         6 *         5 *
*        2 *         3 *         4 *         3 *
*        3 *         4 *         3 *         2 *
*        4 *         5 *         4 *         3 *
*        5 *         6 *         3 *         2 *
*        6 *         7 *         5 *         3 *
*        7 *         8 *         4 *         2 *
*        8 *         9 *         4 *         2 *
*        9 *        10 *         4 *         2 *
************************************************
```



5. Adding Track Information

Let’s have a look at the length of our muon/proton tracks

In the SBND workflow, all PFParticles are fitted as both tracks and showers

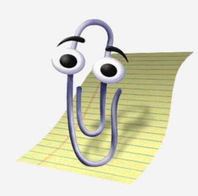


- The association we are after is:

`recob::PFParticle → recob::Track`

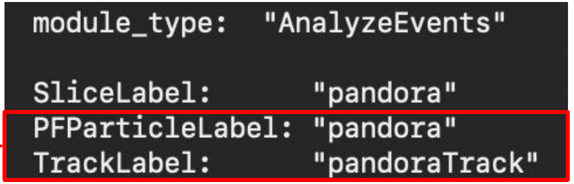
- But first, we’ll need to get the PFParticle handle so that we can initialise our **FindManyP** object

The details (bitty part)



In the configuration file add the label of the track producer, we’ll also need the PFParticle label too (because.. LArSoft)

In analysisConfig.fcl



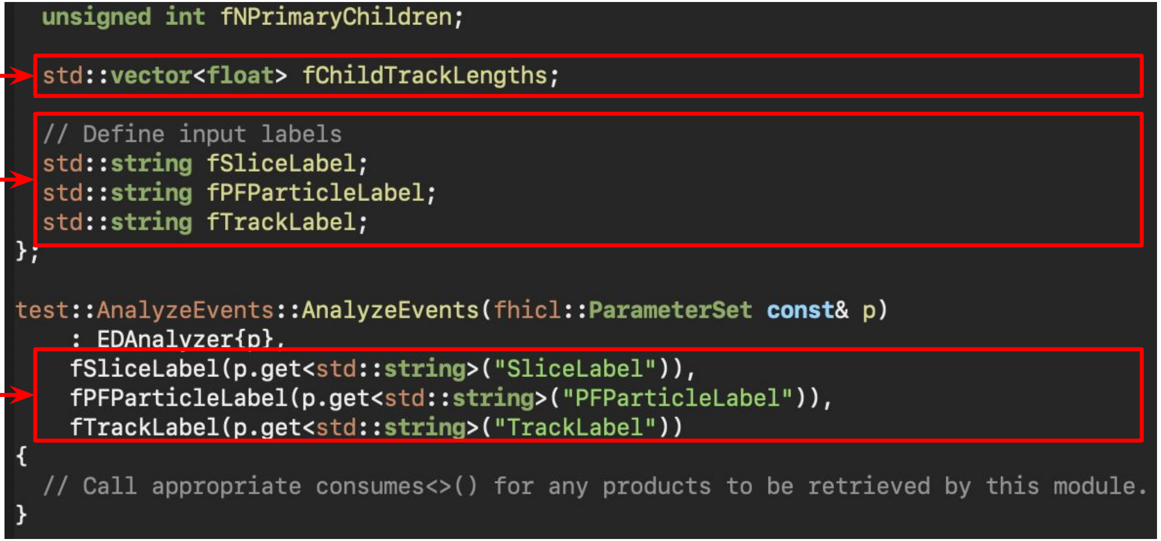
```
module_type:   "AnalyzeEvents"

SliceLabel:        "pandora"
PFParticleLabel: "pandora"
TrackLabel:        "pandoraTrack"
```

Add a new output to store the lengths of the reconstructed tracks

Add a new field to store the TrackLabel and PFParticleLabel that we set in the fcl above

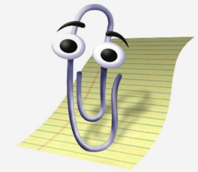
Initialise PFParticle/TrackLabel from the configuration

In analyzeEvents_module.cc

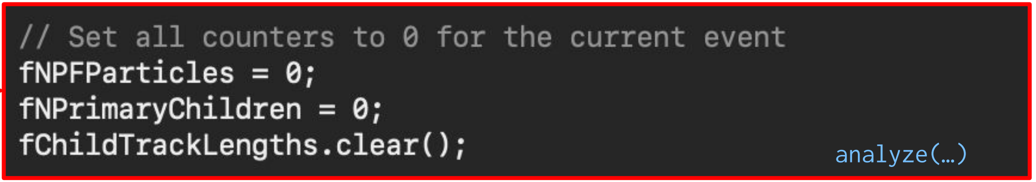


```
  unsigned int fNPrimaryChildren;

  std::vector<float> fChildTrackLengths;

  // Define input labels
  std::string fSliceLabel;
  std::string fPFParticleLabel;
  std::string fTrackLabel;
};

test::AnalyzeEvents::AnalyzeEvents(fhicl::ParameterSet const& p)
    : EDAnalyzer{p},
    fSliceLabel(p.get<std::string>("SliceLabel")),
    fPFParticleLabel(p.get<std::string>("PFParticleLabel")),
    fTrackLabel(p.get<std::string>("TrackLabel"))
{
  // Call appropriate consumes<>() for any products to be retrieved by this module.
}
```

Creating the output



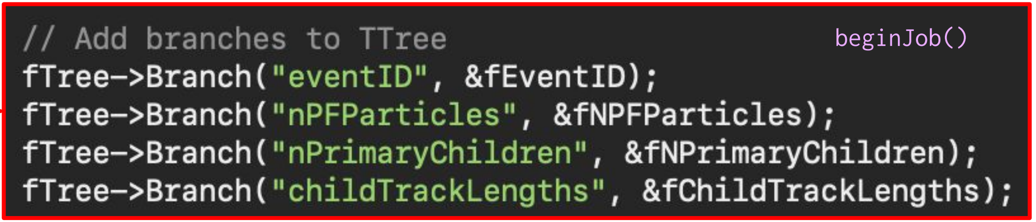
Reset the values stored in the vector for each event



```
// Set all counters to 0 for the current event
fNPFParticles = 0;
fNPrimaryChildren = 0;
fChildTrackLengths.clear();
```

analyze(…)

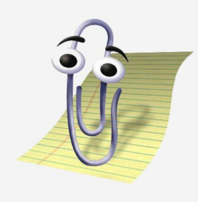
Add a new branch to the TTree using the vector defined on the previous slide



beginJob()

```
// Add branches to TTree
fTree->Branch("eventID", &fEventID);
fTree->Branch("nPFParticles", &fNPFParticles);
fTree->Branch("nPrimaryChildren", &fNPrimaryChildren);
fTree->Branch("childTrackLengths", &fChildTrackLengths);
```

The details, in **`analyze`**

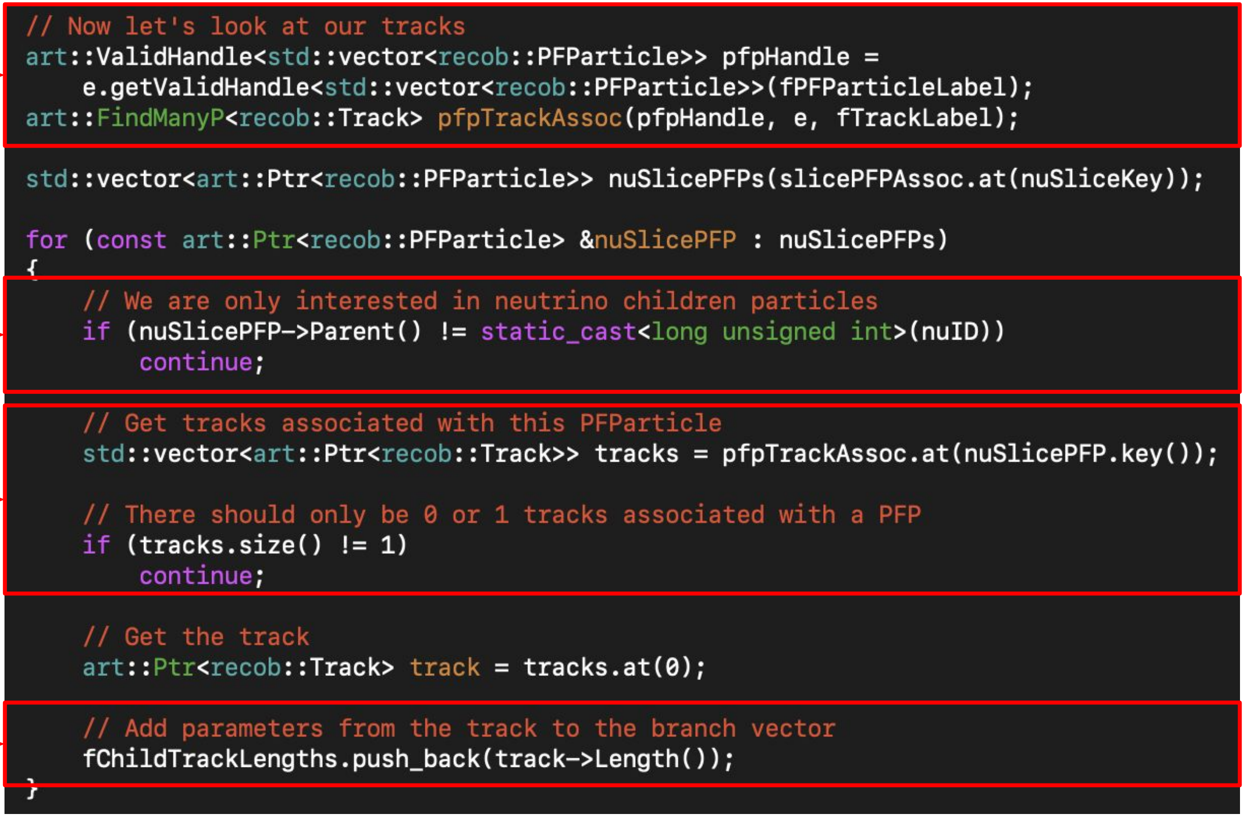


We need to get the handle to our PFParticles so that we can get the PFParticle -> Track associations

Checking that the parent of the current **`PFParticle`** is the neutrino

Get the vector of **`Track`** objects associated to the current **`PFParticle`** There should be only a single track associated with each **`PFParticle`**

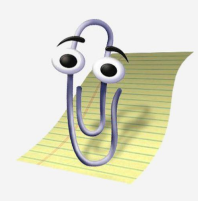
Now fill the vector of **`Track`** lengths we declared earlier



```
// Now let's look at our tracks
art::ValidHandle<std::vector<recob::PFParticle>> pfpHandle =
    e.getValidHandle<std::vector<recob::PFParticle>>(fPFParticleLabel);
art::FindManyP<recob::Track> pfpTrackAssoc(pfpHandle, e, fTrackLabel);

std::vector<art::Ptr<recob::PFParticle>> nuSlicePFPs(slicePFPAssoc.at(nuSliceKey));

for (const art::Ptr<recob::PFParticle> &nuSlicePFP : nuSlicePFPs)
{
    // We are only interested in neutrino children particles
    if (nuSlicePFP->Parent() != static_cast<long unsigned int>(nuID))
        continue;

    // Get tracks associated with this PFParticle
    std::vector<art::Ptr<recob::Track>> tracks = pfpTrackAssoc.at(nuSlicePFP.key());

    // There should only be 0 or 1 tracks associated with a PFP
    if (tracks.size() != 1)
        continue;

    // Get the track
    art::Ptr<recob::Track> track = tracks.at(0);

    // Add parameters from the track to the branch vector
    fChildTrackLengths.push_back(track->Length());
}
```

Another way to view our analysis results…

Let’s look at the track lengths



You can also use -n -1

Firstly, run over all your events by removing `-n 10` from the command like this:

```
lar -c run_analyzeEvents.fcl -s /path/to/input/file.root
```

Open the output file and draw the track lengths! (using treeName->Draw(“branch name”))

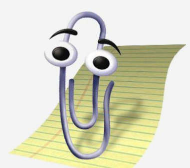
```
root -l analysisOutput.root
```

On the terminal

```
root[0] ana->cd()
root[1] tree->Draw(“childTrackLengths”)
```

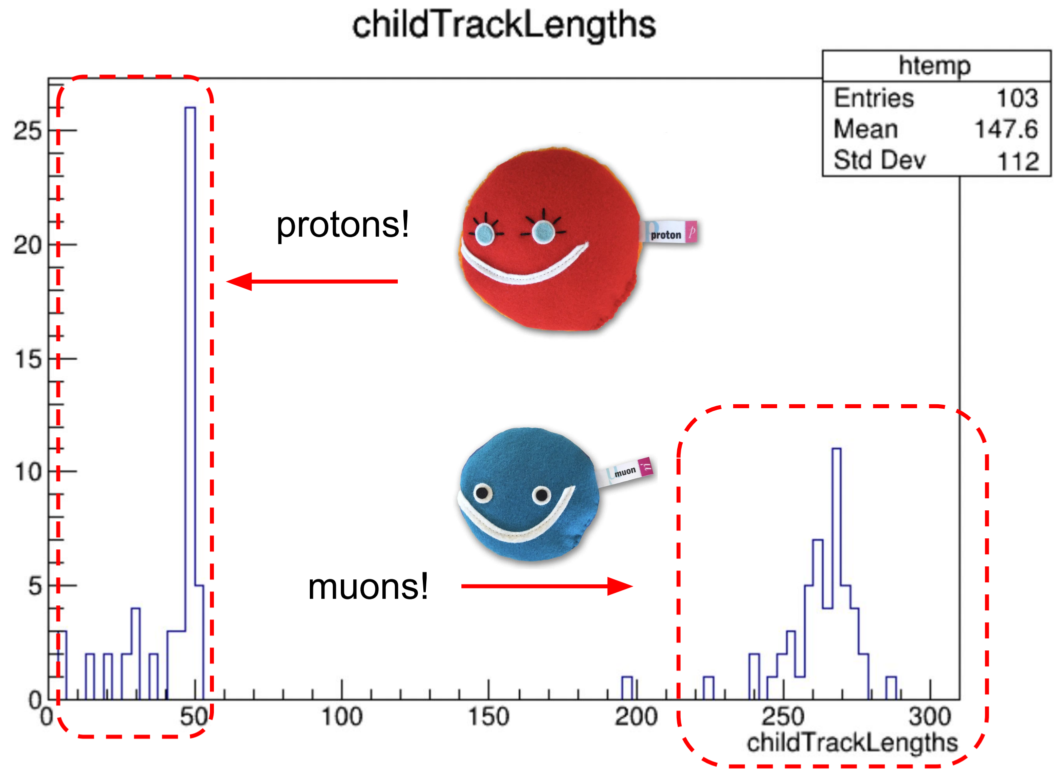
In the root terminal

What do you see?



You can almost make out what is likely to be separate muon and proton distributions!

Probably with some amount of contamination



protons!

muons!

6. Associations: Going a little deeper

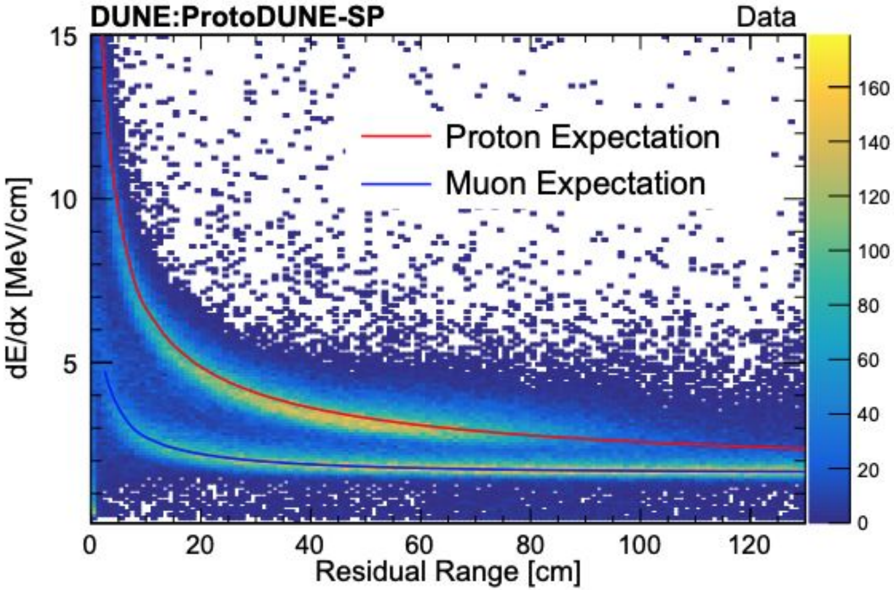
Particle Ionisation

A plot from ProtoDUNE-SP LArTPC showing the 2D dE/dx vs. residual range distributions for Muons and Protons produced in a test beam at CERN.

The theoretical distributions for each particle type are given by the lines.

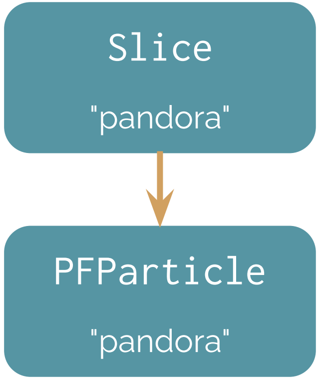
Good separation between Muons & Protons due the large difference in mass.

[2007.06722] First results on ProtoDUNE-SP...



More associations!

Earlier we looked at the association between `recob::Slice`s and `recob::PFParticle`s



`Slice`

"pandora"

`PFParticle`

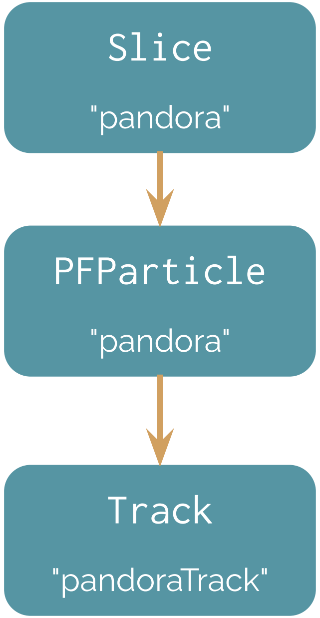
"pandora"

More details can be found in the doxygen entry.

More associations!

Earlier we looked at the association between `recob::Slice`s and `recob::PFParticle`s

…and then between `recob::PFParticle`s and `recob::Track`s.


`Slice`

"pandora"

`PFParticle`

"pandora"

`Track`

"pandoraTrack"

More details can be found in the doxygen entry.

More associations!

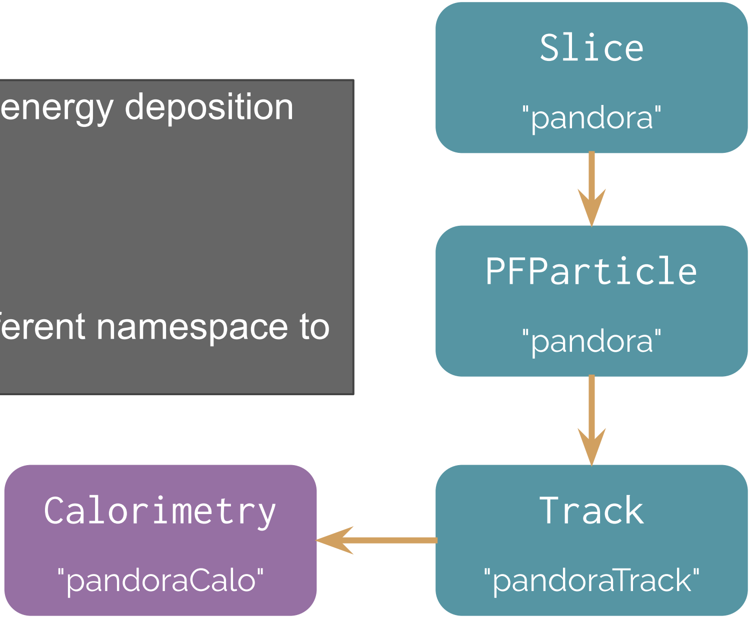
Earlier we looked at the association between `recob::Slice`s and `recob::PFParticle`s

…and then between `recob::PFParticle`s and `recob::Track`s.

…we can now make use of another association to get hold of the energy deposition information we need to to recreate that ProtoDUNE plot.

This time we need the anab::Calorimetry object…

Notice I have drawn in a different colour to indicate it lives in a different namespace to the other objects we've been looking at so far (anab not recob)



`Slice`

"pandora"

`PFParticle`

"pandora"

`Track`

"pandoraTrack"

`Calorimetry`

"pandoraCalo"

More details can be found in the doxygen entry.

More associations!

Earlier we looked at the association between `recob::Slice`s and `recob::PFParticle`s

…and then between `recob::PFParticle`s and `recob::Track`s.

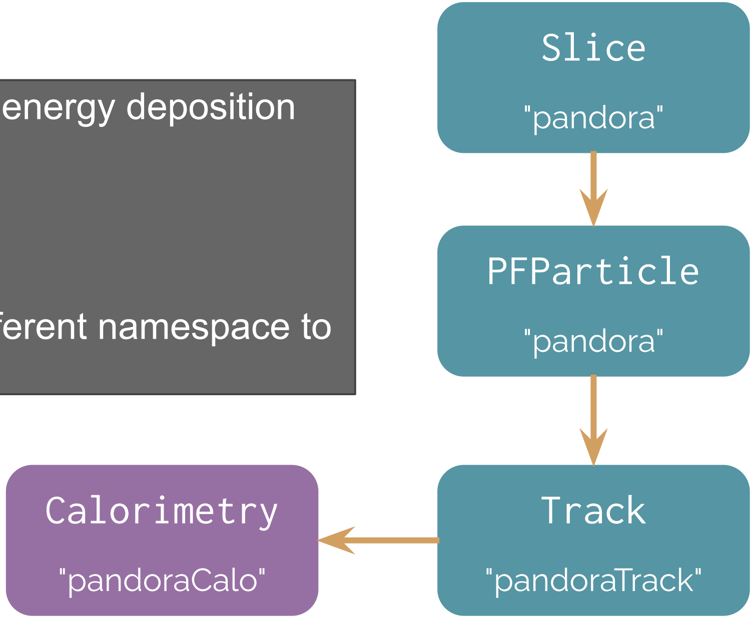
…we can now make use of another association to get hold of the energy deposition information we need to to recreate that ProtoDUNE plot.

This time we need the `anab::Calorimetry` object…

Notice I have drawn in a different colour to indicate it lives in a different namespace to the other objects we've been looking at so far (`anab` not `recob`)

We have at least one separate calorimetry object for each of the three planes

The object contains vectors of dQ/dx, dE/dx, Residual Range etc values. Each entry corresponds to a trajectory point.



More details can be found in the doxygen entry.

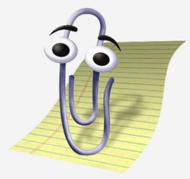
Accessing Calorimetry

These steps should feel familiar:

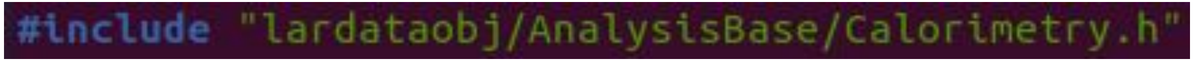
1. Add the relevant header for the `anab::Calorimetry` object
2. Add the module label to your configuration file and access it in the constructor
3. Add any declarations & branches for new variables you want to push to your tree
4. Access the list of `anab::Calorimetry` objects from a list of `recob::Track` objects using `art::FindManyP`
5. Fill your tree variables with information from your `anab::Calorimetry` object.

Try making a start on this and we'll go through it in more detail in a few minutes…

Accessing Calorimetry (1)

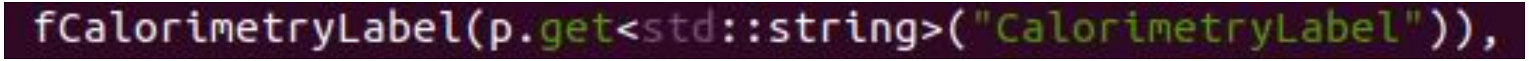


1. Add the relevant header for the `anab::Calorimetry` object

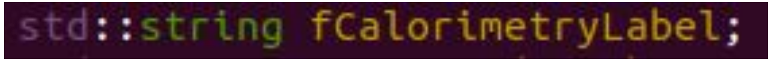


```
#include "lardataobj/AnalysisBase/Calorimetry.h"
```

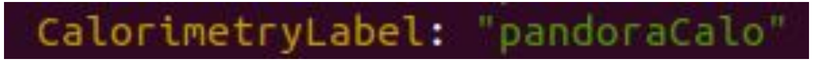
2. Add the module label to your configuration file and access it in the constructor



```
fCalorimetryLabel(p.get<std::string>("CalorimetryLabel")),
```

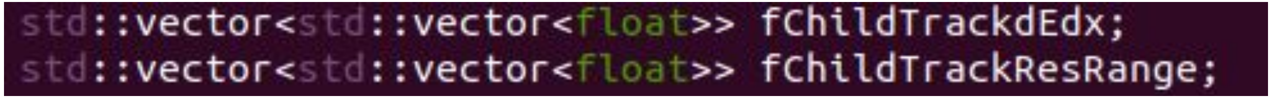


```
std::string fCalorimetryLabel;
```



```
CalorimetryLabel: "pandoraCalo"
```

3. Add any declarations & branches for new variables you want to push to your tree

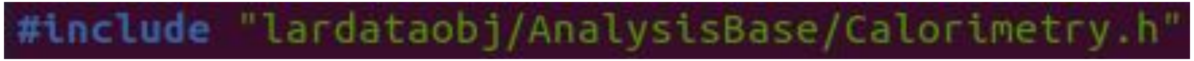


```
std::vector<std::vector<float>> fChildTrackdEdx;
std::vector<std::vector<float>> fChildTrackResRange;
```

```
fTree->Branch("childTrackdEdx", &fChildTrackdEdx);
fTree->Branch("childTrackResRange", &fChildTrackResRange);
```

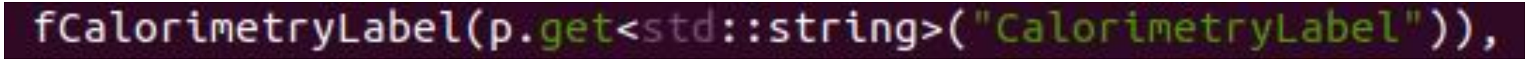
Accessing Calorimetry (1)

1. Add the relevant header for the `anab::Calorimetry` object

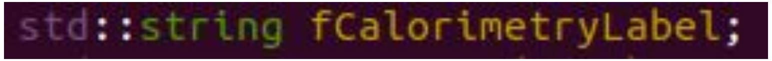


```
#include "lardataobj/AnalysisBase/Calorimetry.h"
```

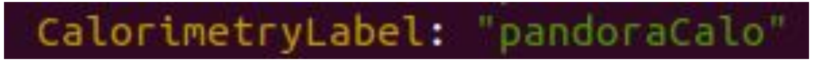
2. Add the module label to your configuration file and access it in the constructor



```
fCalorimetryLabel(p.get<std::string>("CalorimetryLabel")),
```

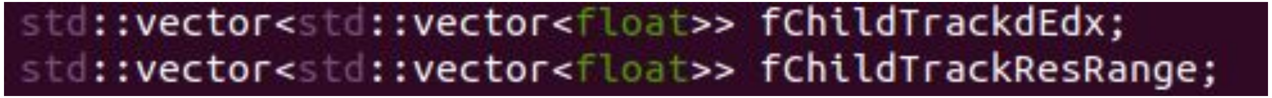


```
std::string fCalorimetryLabel;
```



```
CalorimetryLabel: "pandoraCalo"
```

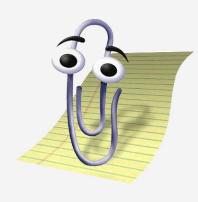
3. Add any declarations & branches for new variables you want to push to your tree



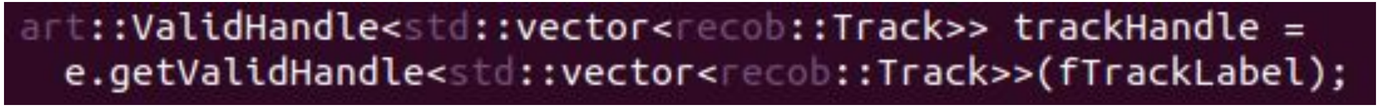
```
std::vector<std::vector<float>> fChildTrackdEdx;
std::vector<std::vector<float>> fChildTrackResRange;
```

```
fTree->Branch("childTrackdEdx", &fChildTrackdEdx);
fTree->Branch("childTrackResRange", &fChildTrackResRange);
```

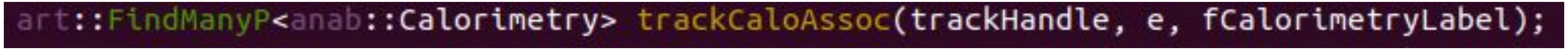
Accessing Calorimetry (2)



4. Access the list of `anab::Calorimetry` objects from a list of `recob::Track` objects using `art::FindManyP`

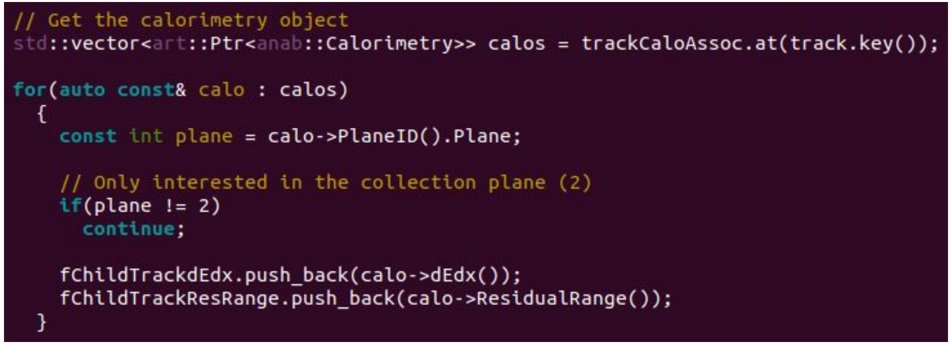


```
art::ValidHandle<std::vector<recob::Track>> trackHandle =
   e.getValidHandle<std::vector<recob::Track>>(fTrackLabel);
```



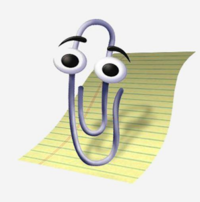
```
art::FindManyP<anab::Calorimetry> trackCaloAssoc(trackHandle, e, fCalorimetryLabel);
```

5. Fill your tree variables with information from your `anab::Calorimetry` object.

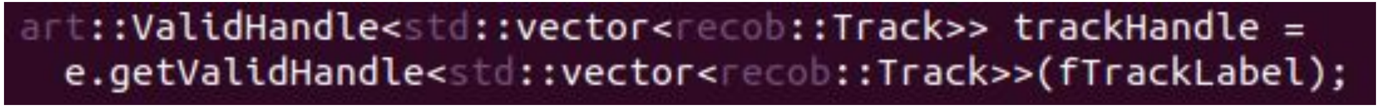


```
// Get the calorimetry object
std::vector<art::Ptr<anab::Calorimetry>> calos = trackCaloAssoc.at(track.key());

for(auto const& calo : calos)
  {
    const int plane = calo->PlaneID().Plane;

    // Only interested in the collection plane (2)
    if(plane != 2)
      continue;

    fChildTrackdEdx.push_back(calo->dEdx());
    fChildTrackResRange.push_back(calo->ResidualRange());
  }
```

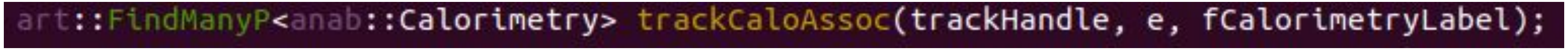
Accessing Calorimetry (2)



4. Access the list of `anab::Calorimetry` objects from a list of `recob::Track` objects using `art::FindManyP`

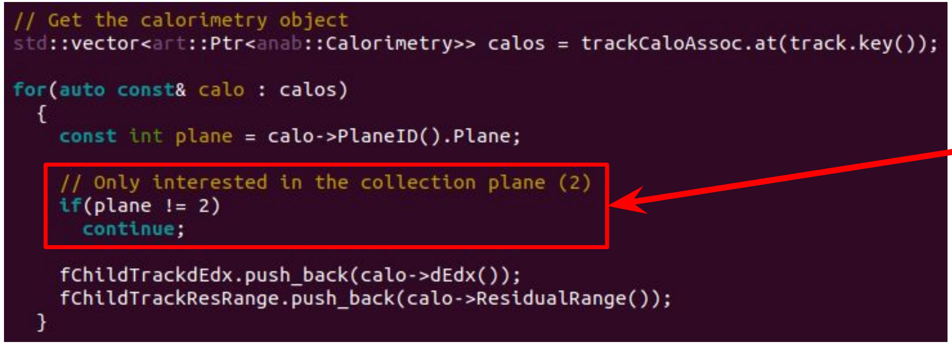


```
art::ValidHandle<std::vector<recob::Track>> trackHandle =
  e.getValidHandle<std::vector<recob::Track>>(fTrackLabel);
```



```
art::FindManyP<anab::Calorimetry> trackCaloAssoc(trackHandle, e, fCalorimetryLabel);
```

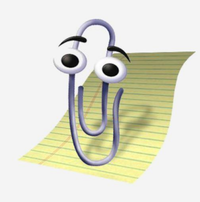
5. Fill your tree variables with information from your `anab::Calorimetry` object.



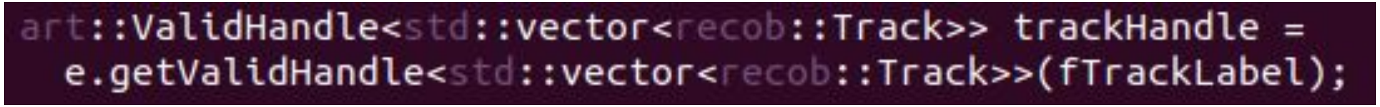
```
// Get the calorimetry object
std::vector<art::Ptr<anab::Calorimetry>> calos = trackCaloAssoc.at(track.key());

for(auto const& calo : calos)
  {
    const int plane = calo->PlaneID().Plane;

    // Only interested in the collection plane (2)
    if(plane != 2)
      continue;

    fChildTrackdEdx.push_back(calo->dEdx());
    fChildTrackResRange.push_back(calo->ResidualRange());
  }
```

Remember, there are separate calorimetry objects for each plane, let's only consider the collection plane.

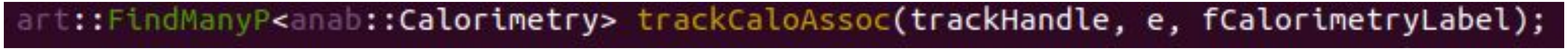
Accessing Calorimetry (2)



4. Access the list of `anab::Calorimetry` objects from a list of `recob::Track` objects using `art::FindManyP`

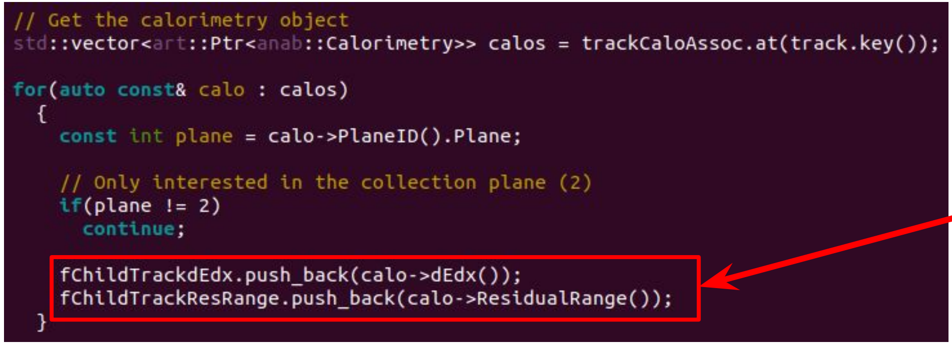


```
art::ValidHandle<std::vector<recob::Track>> trackHandle =
  e.getValidHandle<std::vector<recob::Track>>(fTrackLabel);
```



```
art::FindManyP<anab::Calorimetry> trackCaloAssoc(trackHandle, e, fCalorimetryLabel);
```

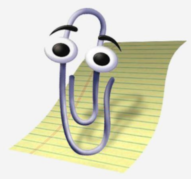
5. Fill your tree variables with information from your `anab::Calorimetry` object.



```
// Get the calorimetry object
std::vector<art::Ptr<anab::Calorimetry>> calos = trackCaloAssoc.at(track.key());

for(auto const& calo : calos)
  {
    const int plane = calo->PlaneID().Plane;

    // Only interested in the collection plane (2)
    if(plane != 2)
      continue;

    fChildTrackdEdx.push_back(calo->dEdx());
    fChildTrackResRange.push_back(calo->ResidualRange());
  }
```

We can insert the whole vectors in one go!

Histogram time!



You should be pretty familiar with rebuilding & running your analyzer now…

You can now use your calorimetry branches to make a 2D histogram in ROOT.

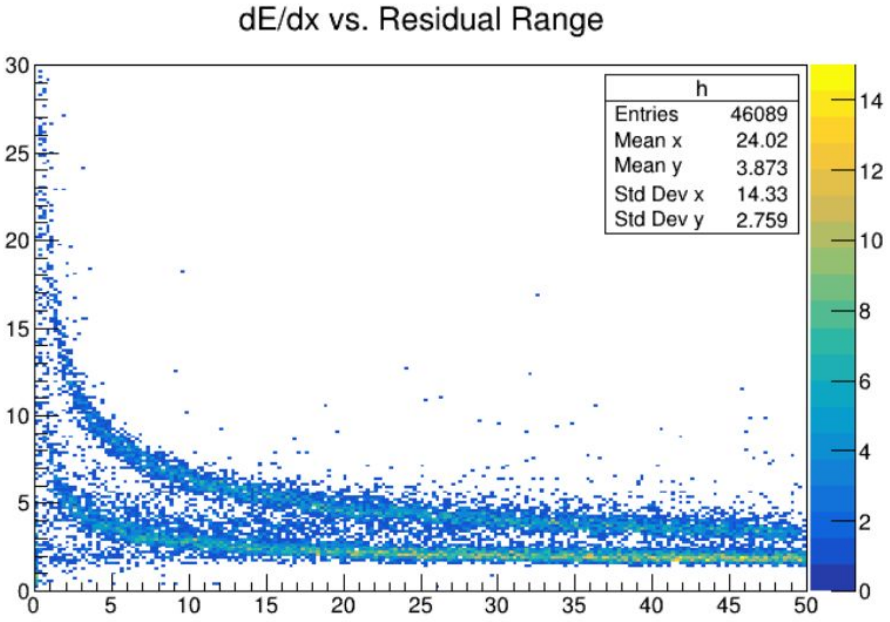
```
root[0] ana->cd()
```

```
root[1] TH2D *h = new TH2D("h","dE/dx vs. Residual Range", 200, 0, 50, 200, 0, 30)
```

```
root[2] tree->Draw("childTrackdEdx:childTrackResRange>>h", "", "colz")
```

You should see something like this!

What do you find most interesting about the distribution?



Try playing around with the axis labels/style options using the GUI.

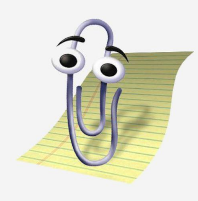
You can save the plot at the end too!

7. A **very** simple PID

Finding the longest track

- Since we have generated a single muon and proton with defined momenta, we can be reasonably confident that they will be very different lengths in each event.
- We can harness this as a very simple particle identification technique for our sample.
- Let's loop through our neutrino children to find which track was the longest track in each neutrino hierarchy. We should do this in a separate loop before the main analysis loop.

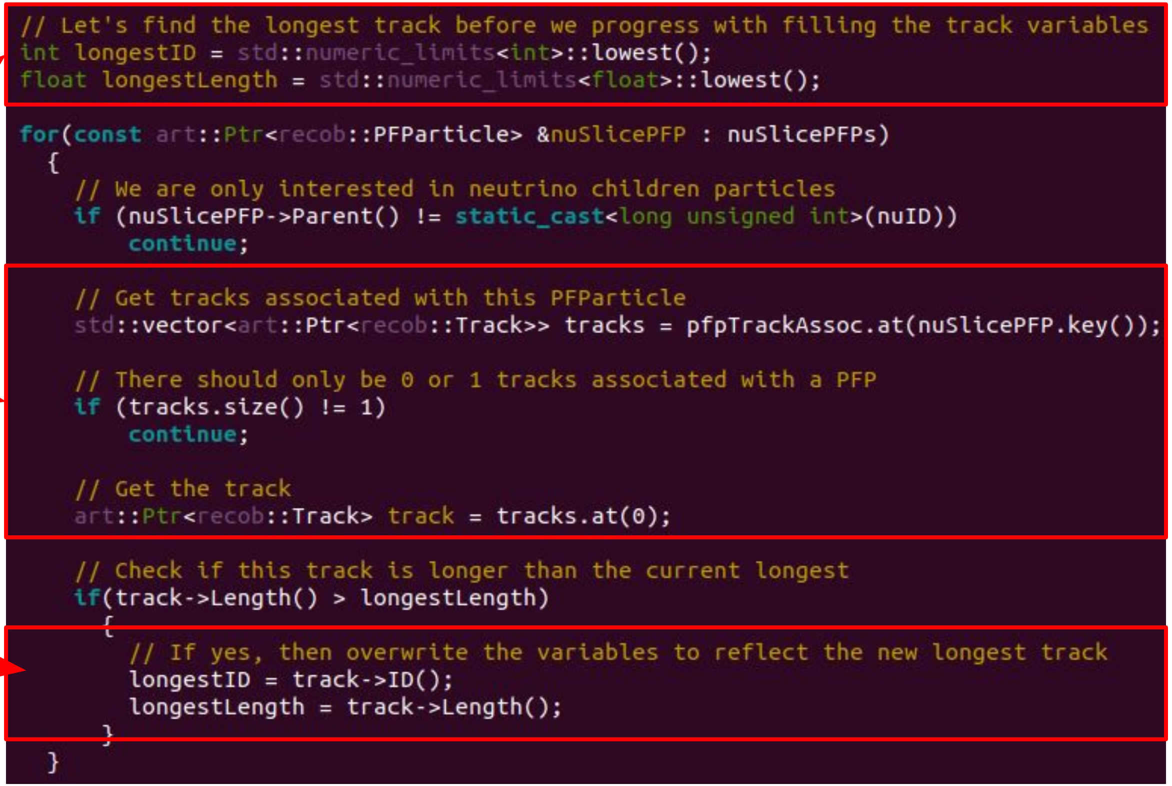
Finding the longest track (1)



We make some variables to track which track was longest and what that length was.

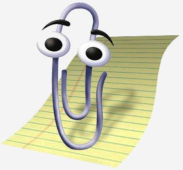
Then we loop through the PFPs and get their associated tracks, just like we do in the main analysis loop.

Within the loop we check whether this track replaces our current longest.

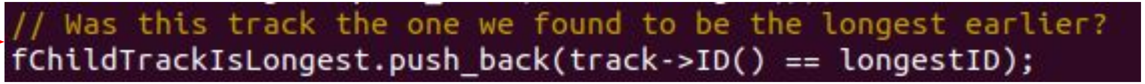


```
// Let's find the longest track before we progress with filling the track variables
int longestID = std::numeric_limits<int>::lowest();
float longestLength = std::numeric_limits<float>::lowest();

for(const art::Ptr<recob::PFParticle> &nuSlicePFP : nuSlicePFPs)
 {
  // We are only interested in neutrino children particles
  if (nuSlicePFP->Parent() != static_cast<long unsigned int>(nuID))
    continue;

  // Get tracks associated with this PFParticle
  std::vector<art::Ptr<recob::Track>> tracks = pfpTrackAssoc.at(nuSlicePFP.key());

  // There should only be 0 or 1 tracks associated with a PFP
  if (tracks.size() != 1)
    continue;

  // Get the track
  art::Ptr<recob::Track> track = tracks.at(0);

  // Check if this track is longer than the current longest
  if(track->Length() > longestLength)
   {
    // If yes, then overwrite the variables to reflect the new longest track
    longestID = track->ID();
    longestLength = track->Length();
   }
 }
```

Finding the longest track (2)



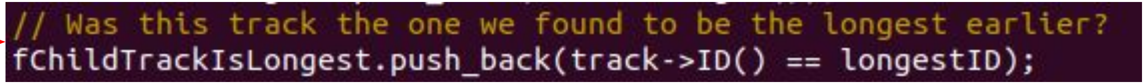
In our main loop we can then add a variable which is a boolean (true/false) describing whether this track is the longest or not.



```
// Was this track the one we found to be the longest earlier?
fChildTrackIsLongest.push_back(track->ID() == longestID);
```

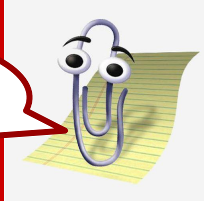
Finding the longest track (2)

In our main loop we can then add a variable which is a boolean (true/false) describing whether this track is the longest or not.



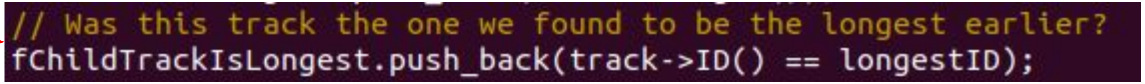
```
// Was this track the one we found to be the longest earlier?
fChildTrackIsLongest.push_back(track->ID() == longestID);
```

Finding the longest track (2)



What else do we need to add? I've left some stuff out!

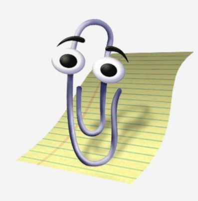
In our main loop we can then add a variable which is a boolean (true/false) describing whether this track is the longest or not.

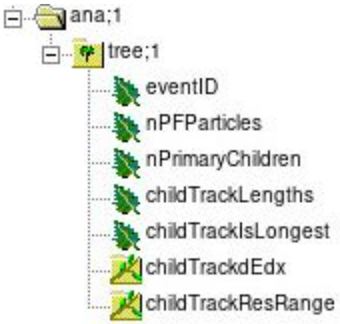


```
// Was this track the one we found to be the longest earlier?
fChildTrackIsLongest.push_back(track->ID() == longestID);
```

Once you think you have included all the necessary additions you will, as usual, need to recompile your analyzer and run it over your reconstruction file again…

More plots, YAY!





- ana;1
  - tree;1
    - eventID
    - nPFParticles
    - nPrimaryChildren
    - childTrackLengths
    - childTrackIsLongest
    - childTrackdEdx
    - childTrackResRange

Now we know which tracks are the longest, and which tracks are just common garden tracks. We can use this to split our plots up…

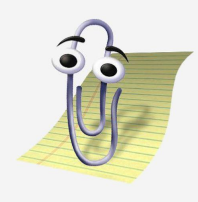
Let’s open our file again, this time making two versions of our dE/dx vs. Residual Range histogram.

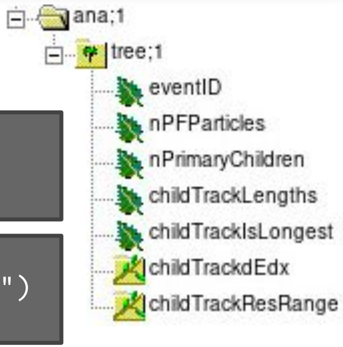
```
root[0] ana->cd()
```

```
root[1] TH2D *hLong = new TH2D(“hLong”,“dE/dx vs. Residual Range”, 200, 0, 50, 200, 0, 30)
```

```
root[2] TH2D *hShort = new TH2D(“hShort”,“dE/dx vs. Residual Range”, 200, 0, 50, 200, 0, 30)
```

More plots, YAY!





ana;1
tree;1
eventID
nPFParticles
nPrimaryChildren
childTrackLengths
childTrackIsLongest
childTrackdEdx
childTrackResRange

This time we need to include our condition on the draw command…

```
root[3] tree->Draw("childTrackdEdx:childTrackResRange>>hLong", "childTrackIsLongest", "")
```

```
root[4] tree->Draw("childTrackdEdx:childTrackResRange>>hShort", "!childTrackIsLongest", "same")
```

We need to tell the two apart… Let’s draw them in different colours!

```
root[5] hLong->SetMarkerColor(kMagenta+2)
```

Alternative colour options are here: https://root.cern.ch/doc/master/classTColor.html

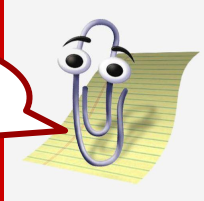
```
root[6] hShort->SetMarkerColor(kOrange+2)
```

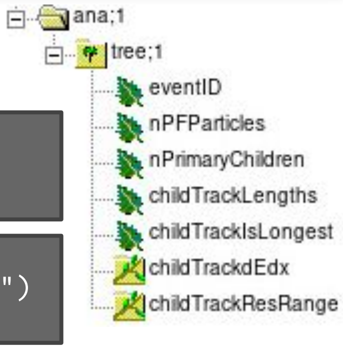
```
root[6] c1->Modified()
```

Tell the canvas (default c1) to implement these changes and redraw the canvas

More plots, YAY!

Why don’t you try this for the track length plot too?





```
ana;1
  tree;1
    eventID
    nPFParticles
    nPrimaryChildren
    childTrackLengths
    childTrackIsLongest
    childTrackdEdx
    childTrackResRange
```

This time we need to include our condition on the draw command…

```
root[3] tree->Draw("childTrackdEdx:childTrackResRange>>hLong", "childTrackIsLongest", "")
```

```
root[4] tree->Draw("childTrackdEdx:childTrackResRange>>hShort", "!childTrackIsLongest", "same")
```

We need to tell the two apart… Let’s draw them in different colours!

```
root[5] hLong->SetMarkerColor(kMagenta+2)
```

Alternative colour options are here: https://root.cern.ch/doc/master/classTColor.html

```
root[6] hShort->SetMarkerColor(kOrange+2)
```

```
root[6] c1->Modified()
```

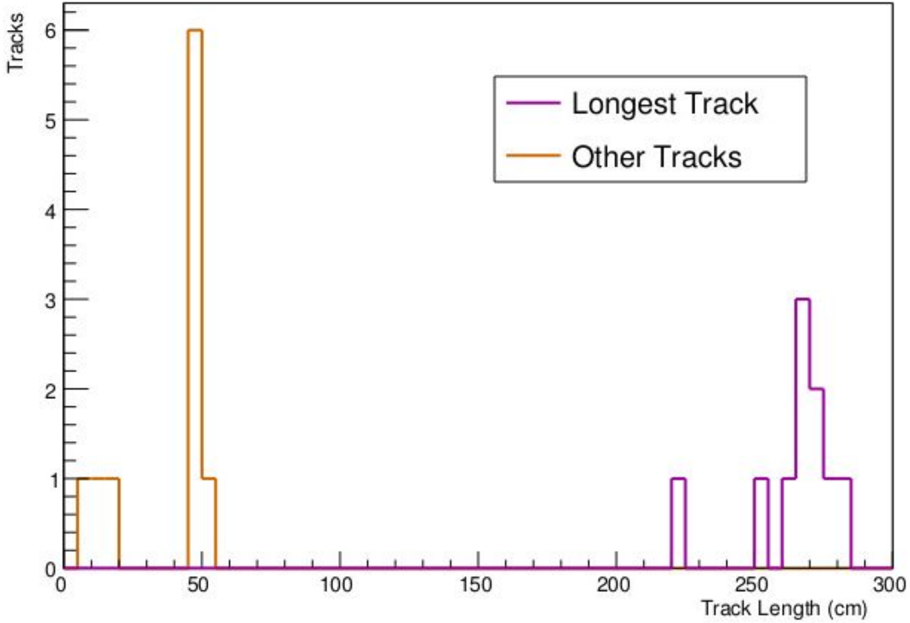
Tell the canvas (default c1) to implement these changes and redraw the canvas

Some final plots…

Track lengths

You should've seen that there were two clearly separated distributions for the longest track compared to the other tracks.

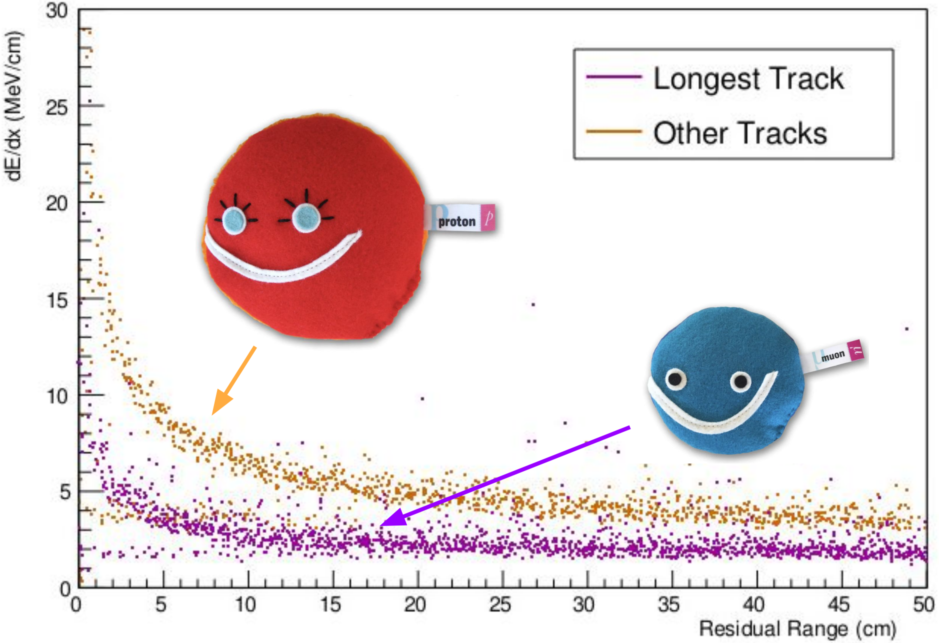
Why is this?



Energy deposition

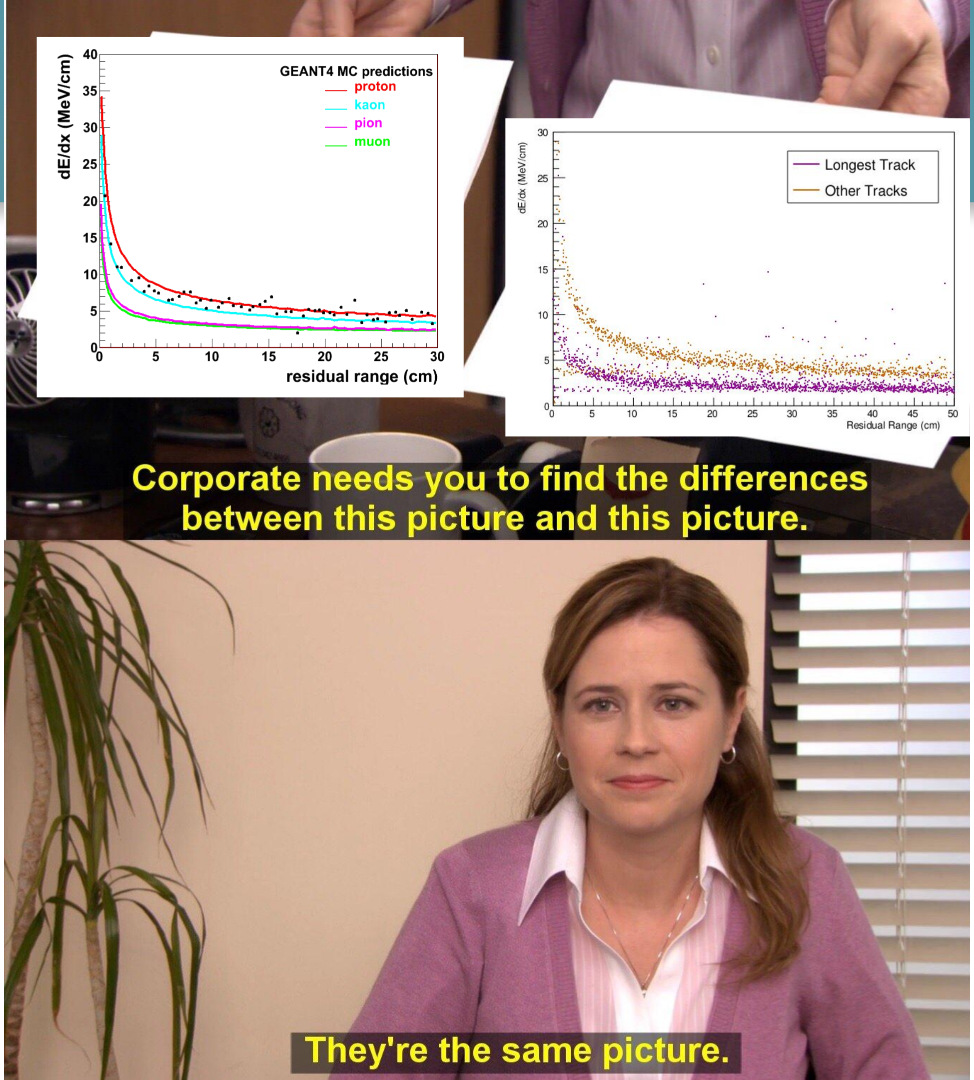
By plotting our dE/dx vs. Residual Range separately curve based on which track was longer we see a clear difference between the distributions.

This results from the fact that the proton is more highly ionising than the muon as it moves through the argon.



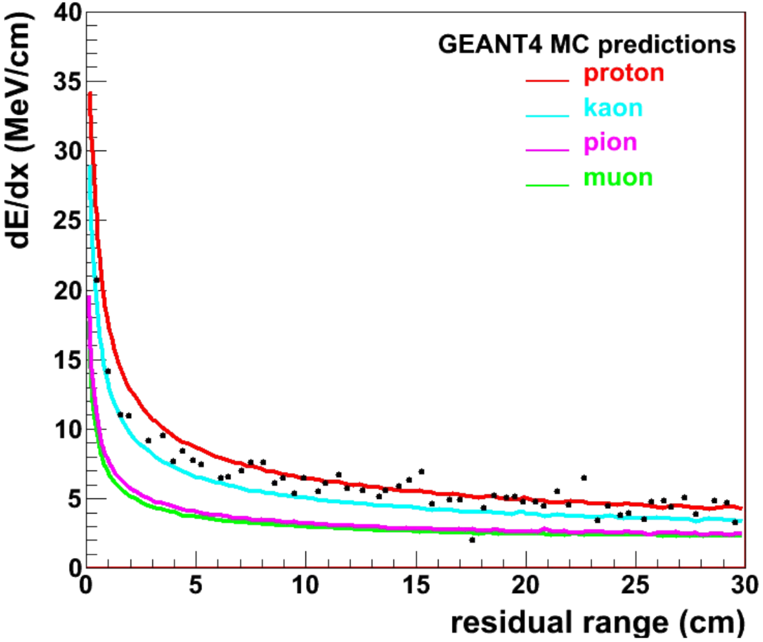
arXiv:1205.6747v2
[physics.ins-det] 5 Jun 2012

This ArgoNeuT plot shows the theoretical separating power of the average dE/dx vs. residual range distributions. The overlaid black data points show a single stopping track in the ArgoNeuT detector.

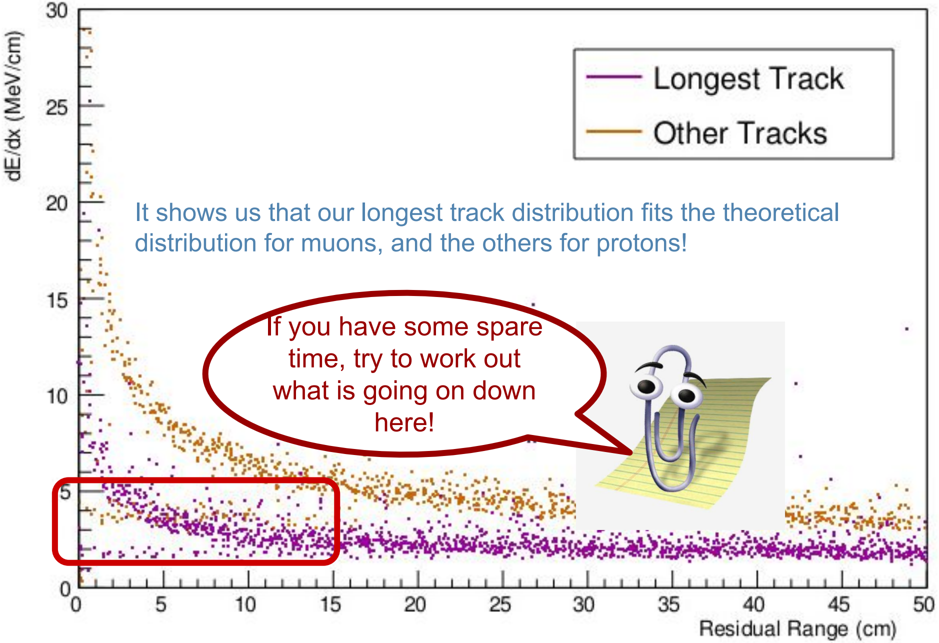


Energy distributions

arXiv:1205.6747v2 [physics.ins-det] 5 Jun 2012



This ArgoNeuT plot shows the theoretical separating power of the average dE/dx vs. residual range distributions. The overlaid black data points show a single stopping track in the ArgoNeuT detector.



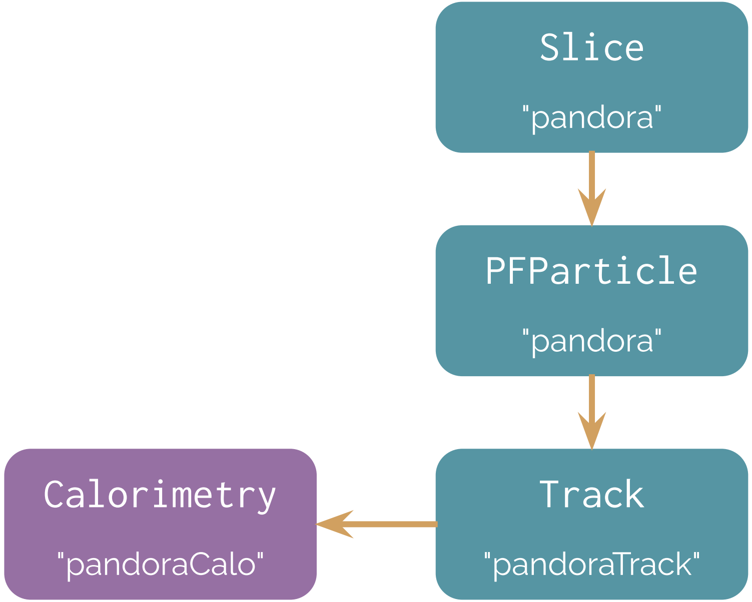
It shows us that our longest track distribution fits the theoretical distribution for muons, and the others for protons!

If you have some spare time, try to work out what is going on down here!

8. Recovering t_0

Detector system associations

We have previously looked at associations between reconstructed quantities for the purpose of accessing geometry and calorimetry information about the particles in our events.



Slice

"pandora"

PFParticle

"pandora"

Track

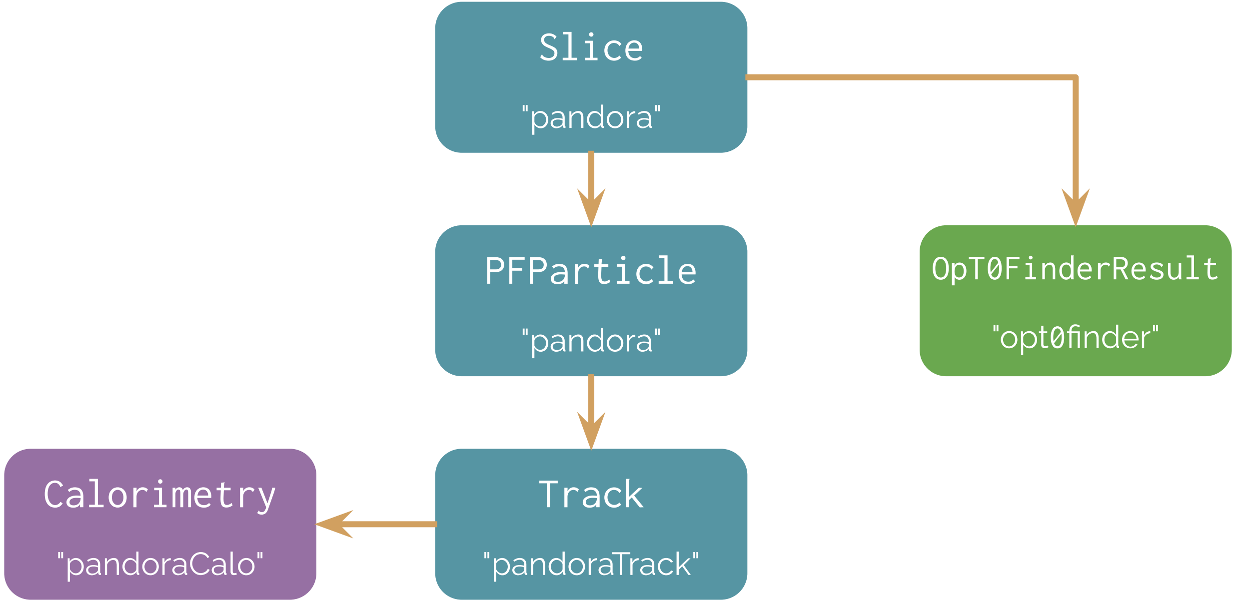
"pandoraTrack"

Calorimetry

"pandoraCalo"

Detector system associations

We have previously looked at associations between reconstructed quantities for the purpose of accessing geometry and calorimetry information about the particles in our events.



We can also look at associations between the different detector systems:

TPC, PDS & CRT

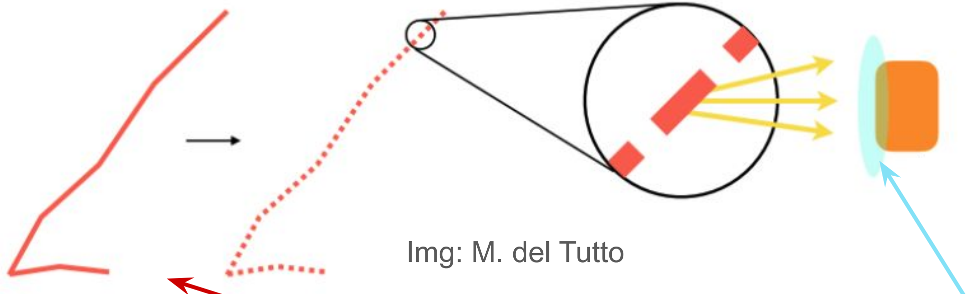
In this scenario we are going to use the precision timing of the PDS to set the t0 of the TPC reconstruction and thus the relative x-position.

Flash Matching

We have previously looked at associations between reconstructed quantities for the purpose of accessi

articles

in our events.



Img: M. del Tutto

We try and match the charge image we saw in the TPC to the light image we saw with the PDS, if they agree we can use the PDS' much more precise timing to adjust the timing (x-position) of our TPC slice.

k at

ween the

r systems:

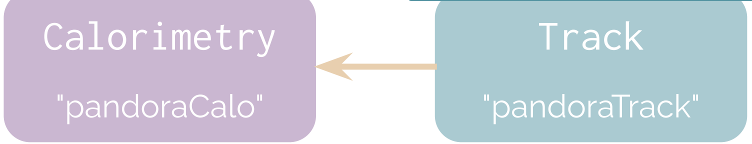
RT.

we are going

sion timing of

he t0 of the

TPC reconstruction and thus the relative x-position.



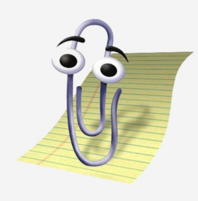
Adding Flash Matching Information

We're going to leave you to try and add this one on your own. The object is called `sbn::OpT0Finder` and lives here. You will need to:

- Add the relevant header
- Add the module label to the fcl file and access it in the analyzer
- Use the association to access the object
- Sometimes there are multiple OpT0Finder results per slice, you should pick the one with the largest score variable.
- Save the time variable from the object to your tree.

We will go through all of this in a moment so don't worry if you get stuck, this is hard!

Adding OpT0Finder

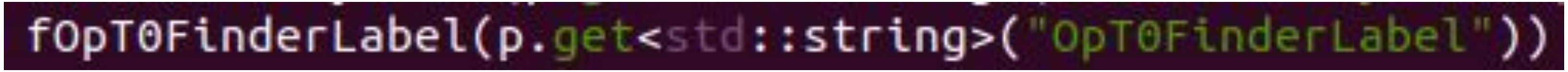


Add the relevant header

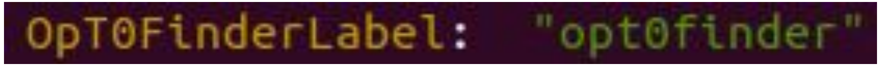
```
// SBN(D) includes
#include "sbnobj/Common/Reco/OpT0FinderResult.h"
```

Add the module label to the fcl file and access it in the analyzer

```
std::string fOpT0FinderLabel;
```



```
fOpT0FinderLabel(p.get<std::string>("OpT0FinderLabel"))
```

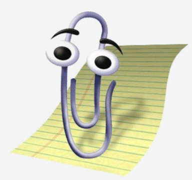


```
OpT0FinderLabel:  "opt0finder"
```

Use the association to access the object

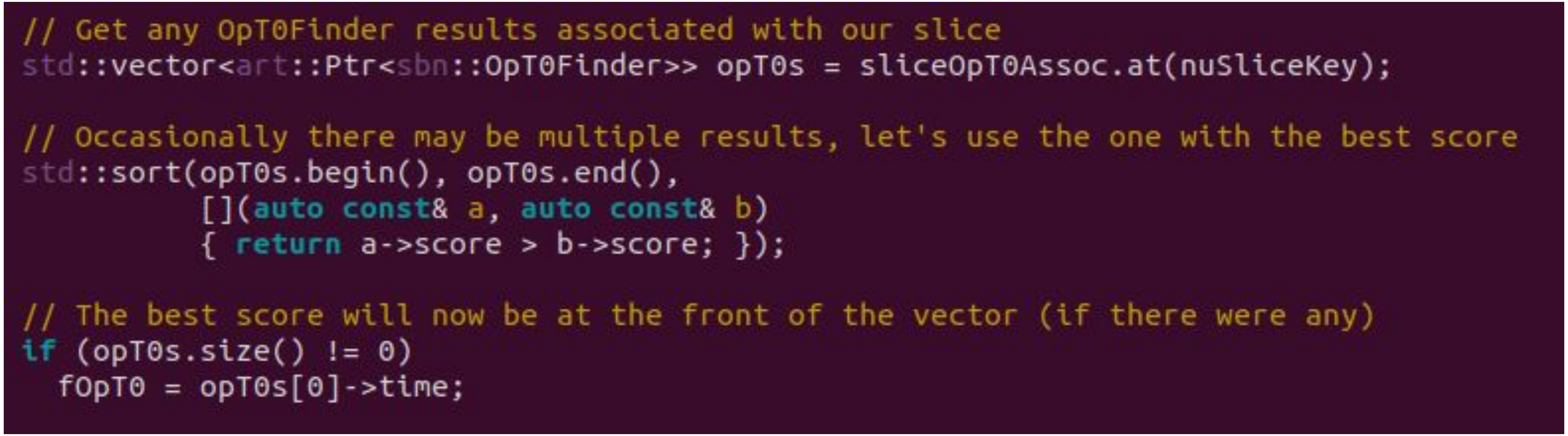
```
art::FindManyP<sbn::OpT0Finder> sliceOpT0Assoc(sliceHandle, e, fOpT0FinderLabel);
```

Accessing OpT0Finder



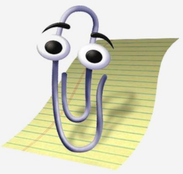
- Sometimes there are multiple OpT0Finder results per slice, you should pick the one with the largest score variable.

- Save the time variable from the object to your tree.



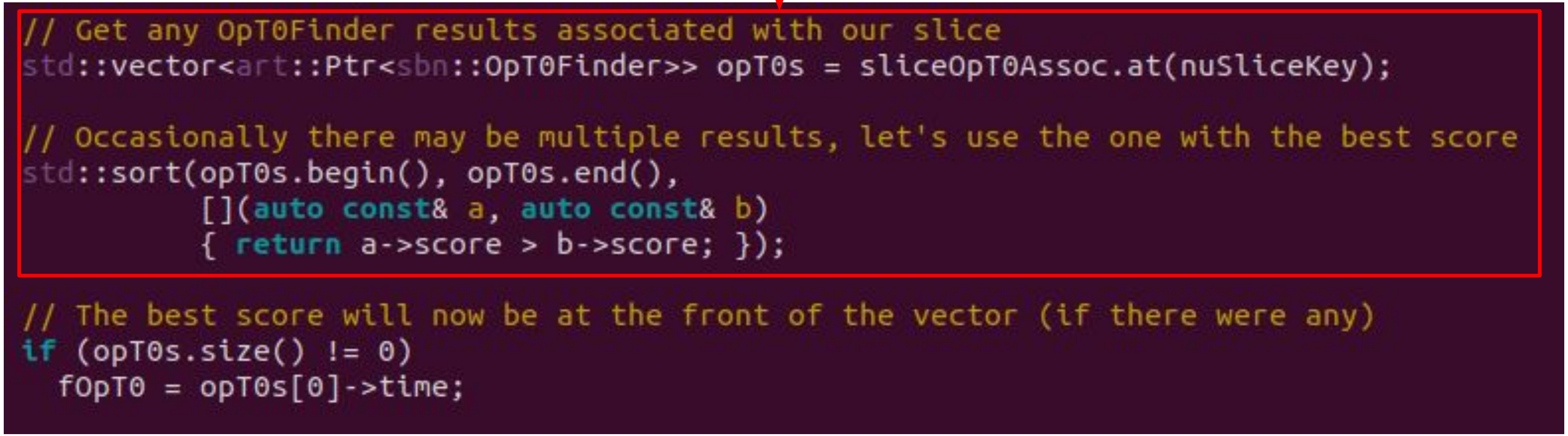
```
// Get any OpT0Finder results associated with our slice
std::vector<art::Ptr<sbn::OpT0Finder>> opT0s = sliceOpT0Assoc.at(nuSliceKey);

// Occasionally there may be multiple results, let's use the one with the best score
std::sort(opT0s.begin(), opT0s.end(),
          [](auto const& a, auto const& b)
          { return a->score > b->score; });

// The best score will now be at the front of the vector (if there were any)
if (opT0s.size() != 0)
  fOpT0 = opT0s[0]->time;
```

A few noteworthy points…



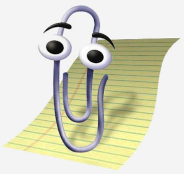
1. This uses our slice object so needs to happen in the slice loop.

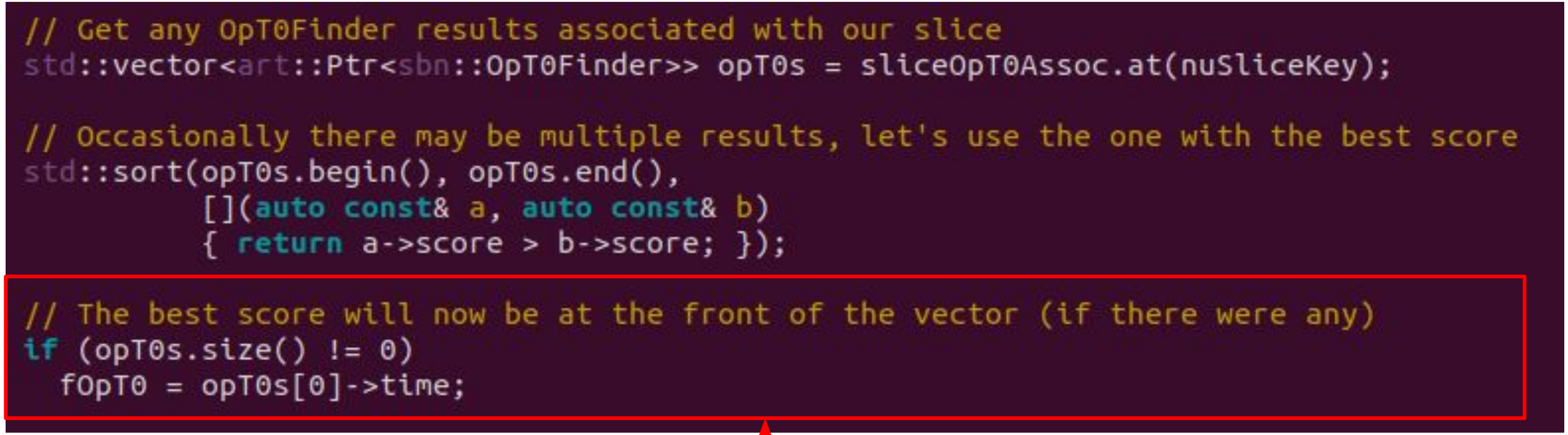
2. You may well have found the top scoring object in a different way. Many approaches are legitimate.



```
// Get any OpT0Finder results associated with our slice
std::vector<art::Ptr<sbn::OpT0Finder>> opT0s = sliceOpT0Assoc.at(nuSliceKey);

// Occasionally there may be multiple results, let's use the one with the best score
std::sort(opT0s.begin(), opT0s.end(),
          [](auto const& a, auto const& b)
          { return a->score > b->score; });

// The best score will now be at the front of the vector (if there were any)
if (opT0s.size() != 0)
  fOpT0 = opT0s[0]->time;
```

A few noteworthy points…



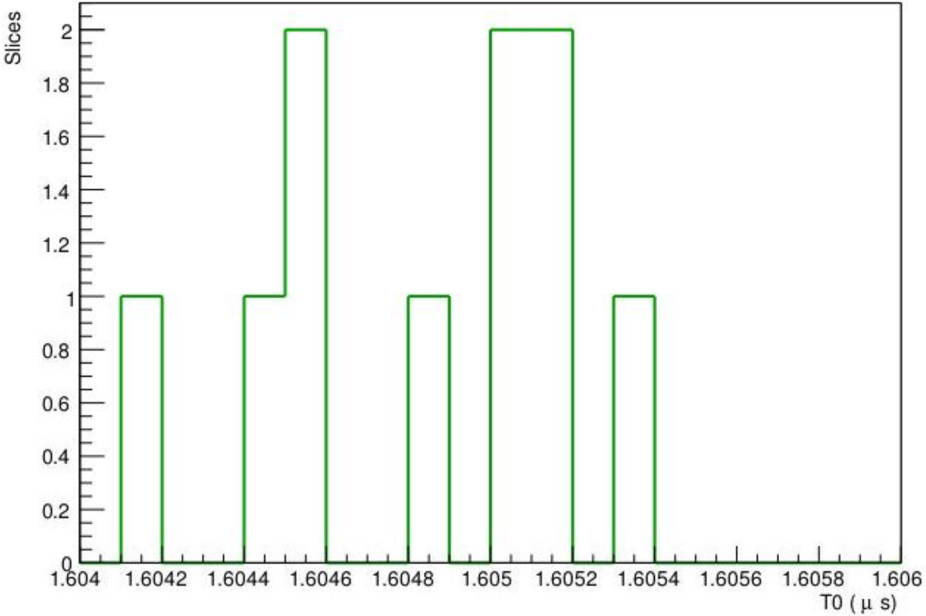


```
// Get any OpT0Finder results associated with our slice
std::vector<art::Ptr<sbn::OpT0Finder>> opT0s = sliceOpT0Assoc.at(nuSliceKey);

// Occasionally there may be multiple results, let's use the one with the best score
std::sort(opT0s.begin(), opT0s.end(),
          [](auto const& a, auto const& b)
          { return a->score > b->score; });

// The best score will now be at the front of the vector (if there were any)
if (opT0s.size() != 0)
  fOpT0 = opT0s[0]->time;
```

3. We need to have defined fOpT0 and added it as a branch too.

T0 Results

Remember way back in the simulation tutorial? You defined t0 to be 1600ns.



- Your OpT0 results should give you values close to that original simulated time.
- Last year we discovered this number to be off and it took us a long time and asking other experts to understand why.
- Worth remembering that all of us still have to ask questions all the time, so never worry about reaching out with questions!

Final notes

ROOT Workflows

- These tutorials focus on using ROOT via a VNC connection
- Trying to open root files (or any visualisation) via a standard ssh connection will result in bad times
- You can often set up a VNC over an ssh connection (e.g. to the Fermilab GPVMs)
- You can also copy root files to your local machine and run root macros locally (the TTree files are much smaller than the art files and root can be compiled on a laptop fairly easily with minimal dependencies)

Documentation and additional information

The documentation for each art object/tool we have looked at lives here:

- `recob::PFParticle` - https://nusoft.fnal.gov/larsoft/doxsvn/html/classrecob_1_1PFParticle.html
- `art::FindManyP` - https://nusoft.fnal.gov/larsoft/doxsvn/html/classart_1_1FindManyP.html
- `recob::Track` - https://nusoft.fnal.gov/larsoft/doxsvn/html/classrecob_1_1Track.html
- `anab::Calorimetry` - https://nusoft.fnal.gov/larsoft/doxsvn/html/classanab_1_1Calorimetry.html

Remember you can look at all of the objects and their corresponding producers in any reco file by looking at an event dump:

```
lar -c eventdump.fcl -s /path/to/reco/file.root -n 1
```

Some important file locations

Our version of the code lives here:

```
$MRB_SOURCE/sbndcode/sbndcode/Workshop/Analysis/.FinishedModule/AnalyzeEvents_module.cc
```

```
$MRB_SOURCE/sbndcode/sbndcode/Workshop/Analysis/.FinishedModule/analysisConfig.fcl
```

```
$MRB_SOURCE/sbndcode/sbndcode/Workshop/Analysis/.FinishedModule/run_analyzeEvents.fcl
```

Type `ls -a` *in the directories to see hidden files and directories*

Previous tutorials (SBND-based)

Ed Tyley & Rhiannon Jones' tutorial from 2022 is here:
https://indico.ph.ed.ac.uk/event/130/contributions/1747/

Ed Tyley & Rhiannon Jones' tutorial from 2021 is here:
https://indico.ph.ed.ac.uk/event/91/contributions/1417/

Owen Goodwin's tutorial from 2020 is here:
https://indico.hep.manchester.ac.uk/getFile.py/access?contribId=12&sessionId=4&resId=0&materialId=slides&confId=5856

Rhiannon Jones' tutorial from 2019 is here:
https://indico.hep.manchester.ac.uk/getFile.py/access?contribId=13&sessionId=4&resId=0&materialId=slides&confId=5544

Leigh Whitehead's tutorial from 2018 is here:
https://indico.hep.manchester.ac.uk/getFile.py/access?contribId=13&sessionId=2&resId=0&materialId=slides&confId=5372