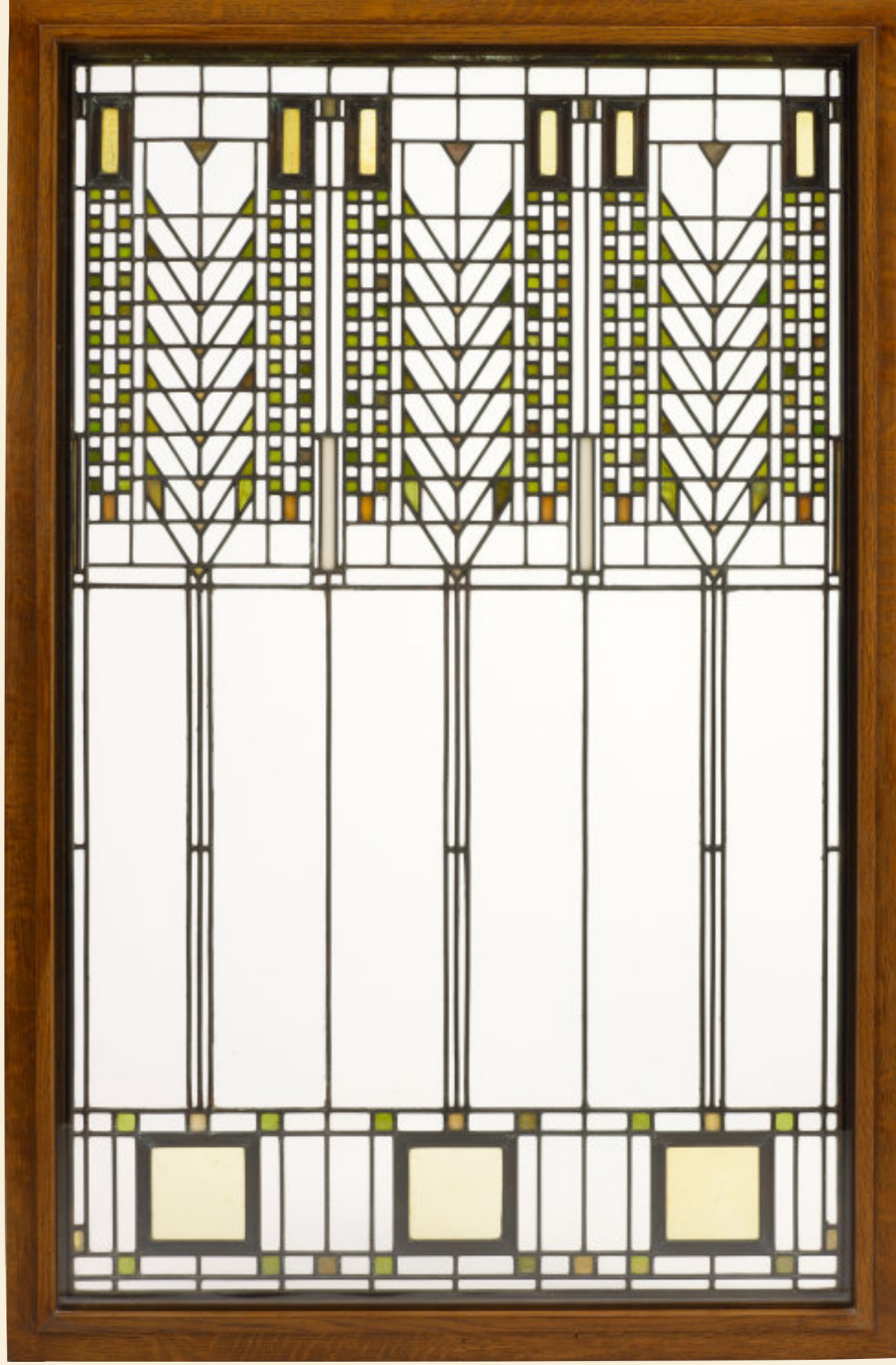
Session 11: Reading data and making Histograms

[art workbook sections 16, 17, and 18]

Adam Lyon
art/LArSoft School 2015



Frank Lloyd Wright, *Tree of Life*, Art Institute of Chicago

You will be performing exercises 6, 7, and 8 from the art workbook

How to get data products out of the event … and make a Root histogram … and loop over data

A little review first…

Templates

The things with Something<somethingElse>

Templates allow C++ functions or classes to operate on many different types without being rewritten for each one
[Wikipedia]

```
// Vectors
std::vector<int>   anIntCollection;
std::vector<float> aFloatCollection;
std::vector<tex::GenParticle> aGenParticleCollection;

// Maps
std::map<std::string, float> myDictionary;

// If you have a parameter called pset
auto calorimeterHalfLength = pset.get<double>("calHalfLength");
auto zaxis = pset.get<CLHEP::Hep3Vector>("zaxis");
```

Typedefs

Typedefs are type aliases. Used to simplify code. Also useful for future-proofing

If a `typedef` is for a collection (e.g. `vector`), then say so!

Not the same as `auto` [why?]

```
// The below is very typical
// If we wanted to change std::vector to std::list, rest of code stays same
typedef std::vector<GenParticle> GenParticleCollection;

// Use:  (type must be specified since you are making a new one)
GenParticleCollection myGenParticleCollection;

// typedef != auto
auto myGenParticleCollection;  // Compiler error - can’t tell what you want

// Ok - myGenParticleCollection will be the return type of the function
auto const& myGenParticleCollection = SomeFunctionThatGetsGenParticleCollection();
```

Data product names

To pull a data product out of the event, you must know its name

The data product type (e.g. `GenParticleCollection`) may not be enough

- **More than one producer may add a `GenParticleCollection` to the event** [explain a scenario where this happens]
- **A producer may add more than one `GenParticleCollection` to the event** [explain a scenario where this happens]
- **A `GenParticleCollection` may have been produced by a previous job**

Which `GenParticleCollection` do you want?

Data product names

To pull a data product out of the event, you must know its name

The data product type (e.g. `GenParticleCollection`) may not be enough

- **More than one producer may add a `GenParticleCollection` to the event — disambiguate with module label**
- **A producer may add more than one `GenParticleCollection` to the event — disambiguate with instance name**
- **A `GenParticleCollection` may have been produced by a previous job — disambiguate with process name**

Which `GenParticleCollection` do you want?

Data product names

You specify a data product with its `art::InputTag`

```
#include "art/Utilities/InputTag.h"

art::InputTag tag("ModuleLabel:InstanceName:ProcessName");
```

You may not need all of the input tag parts (e.g. ProcessName is rarely used). An exception will be thrown if art cannot determine which data product to return

In practice, you need the module label and often need the instance name

```
art::InputTag tag("evtgen"); // Just the Module label
art::InputTag tag("evtgen:alg1"); // Module label and instance name
art::InputTag tag("evtgen::myRunFrom2015"); // Module label and process name (rare)
```

Data product name

Important note:

The data product type is not included in the inputTag. You'll see why …

Note added after talk:

Leaving off the process name is ok if you want the current process. Leaving off module label or instance name will mean product won't be found (e.g. products are never aggregated, either they're found or they're not).

Storage and Restrictions

Data are stored as Root branches. Note the names and double _

```
TypeName_moduleLabel_InstanceName_ProcessName
```

So, your data product names should only have A-Za-z0-9
No _.,;-: allowed (:: for namespace is ok)

List the data products in a file…

```
$ art -c ~/art-workbook/art-workbook/FileDumper/fileDumper.fcl \
                $TOYEXPERIMENT_DIR/inputFiles/input04.art
...
PROCESS NAME | MODULE LABEL.. | PRODUCT INSTANCE NAME | DATA PRODUCT TYPE............................ | SIZE
exampleInput | detsim........ | inner................ | std::vector<tex::Intersection>............... | ..15
exampleInput | randomsaver... | ..................... | std::vector<art::RNGsnapshot>................ | ...3
exampleInput | TriggerResults | ..................... | art::TriggerResults.......................... | ...-
exampleInput | makeHits...... | ..................... | art::Assns<tex::TrkHit,tex::Intersection,void> | ..35
exampleInput | makeHits...... | ..................... | std::vector<tex::TrkHit>..................... | ..35
exampleInput | fitter........ | ..................... | std::vector<tex::FittedHelixData>............ | ...3
exampleInput | evtgen........ | ..................... | std::vector<tex::GenParticle>................ | ...5
exampleInput | detsim........ | outer................ | std::vector<tex::Intersection>............... | ..24
```

```
$ product_sizes_dumper $TOYEXPERIMENT_DIR/inputFiles/input04.art    # $ART_FQ_DIR/bin/product_sizes_dumper

... (watch for hard to see double underscores) ...

Details for branch: Events

     Size in bytes   Fraction Data Product Name
            692608      0.246 art::RNGsnapshots_randomsaver__exampleInput.
            594770      0.211 tex::TrkHits_makeHits__exampleInput.
            538314      0.191 tex::Intersections_detsim_outer_exampleInput.
            397242      0.141 tex::Intersections_detsim_inner_exampleInput.
            296311      0.105 tex::FittedHelixDatas_fitter__exampleInput.
            181940      0.065 tex::GenParticles_evtgen__exampleInput.
            108689      0.039 tex::Intersectiontex::TrkHitvoidart::Assns_makeHits__exampleInput.
              5645      0.002 EventAuxiliary
              4271      0.002 art::TriggerResults_TriggerResults__exampleInput.
```

Making the InputTag a parameter

Module labels depend on the generating FCL, and you typically don't know that when you write YOUR module

So make the InputTag a parameter to your module so it can be configured at runtime

You will have to look at the FCL that produced the data or look at the data in the input file to determine the input tag

Remember module LABEL, not module TYPE

```
physics : {

  producers : {
    pizza : {
      module_type : makeHits
    }
  }
```

```
physics : {

  analyzers : {
    myanalyzer : {
      module_type : myana
      inputTag : pizza
    }
  }
```

Requesting data

You ask for a *Handle* to the data product

The *Handle* is an object that can point to the actual data and other things

You'll see this in the exercise…

```
tex::ReadGens1::ReadGens1(fhicl::ParameterSet const& pset):
  art::EDAnalyzer(pset),
  gensTag_(pset.get<std::string>("genParticlesInputTag")){
}

void tex::ReadGens1::analyze(art::Event const& event ){
 art::Handle<GenParticleCollection> gens;
 event.getByLabel(gensTag_, gens);
 std::cout << "ReadGens1::analyze event: "  // Should use message logger
           << event.id().event()
          << " GenParticles: "
          << gens->size()
          << std::endl;
}

DEFINE_ART_MODULE(tex::ReadGens1)
```

Requesting data

```
tex::ReadGens1::ReadGens1(fhicl::ParameterSet const& pset):
  art::EDAnalyzer(pset),
  gensTag_(pset.get<std::string>("genParticlesInputTag")){
}

void tex::ReadGens1::analyze(art::Event const& event ){
  art::Handle<GenParticleCollection> gens;
  event.getByLabel(gensTag_, gens);
  std::cout << "ReadGens1::analyze event: "  // Should use message logger
            << event.id().event()
            << " GenParticles: "
            << gens->size()
            << std::endl;
}

DEFINE_ART_MODULE(tex::ReadGens1)
```

`getByLabel` tries to find the data product; one of three outcomes:

1) Data product is found in event and handle points to it

2) Data product is not in event; dereferencing Handle will throw exception

3) [Note added after talk]: Multiple modules will never match. e.g. if a data product has an instance name, it must be in the input tag to match it.

Checking for validity

```
tex::ReadGens1::ReadGens1(fhicl::ParameterSet const& pset):
  art::EDAnalyzer(pset),
  gensTag_(pset.get<std::string>("genParticlesInputTag")){
}

void tex::ReadGens1::analyze(art::Event const& event ){
  art::Handle<GenParticleCollection> gens;
  event.getByLabel(gensTag_, gens);

  if ( gens.isValid() ) {
    std::cout << "ReadGens1::analyze event: "  // Should use message logger
              << event.id().event()
              << " GenParticles: "
              << gens->size()
              << std::endl;
  }
  else {
    std::cout << "Aaaack - Where’s my GenParticleCollection?" << std::endl;
  }
}
```

Avoids the exception

Requesting data

A shorter way

```
// Replace
art::Handle<GenParticleCollection> gens;
event.getByLabel(gensTag_, gens);

// with
auto gens = event.getValidHandle<GenParticleCollection>(gensTag_);
```

This returns an `art::ValidHandle<T>`

If the data cannot be retrieved, then a ProductNotFound exception is thrown

Typically, art will simply skip processing the event and continue to the next (see exercise for different behavior)

Dereferencing

You can dereference many times or once

```
auto gensH = event.getValidHandle<GenParticleCollection>(gensTag_);
std::cout << "There are " << gensH->size() << " gen particles" << std::endl;
for ( const auto aGen : *gensH ) {
 // do stuff with aGen
}

// OR

auto gensH = event.getValidHandle<GenParticleCollection>(gensTag_);
auto const& gens = *gensH;
std::cout << "There are " << gens.size() << " gen particles" << std::endl;
for ( const auto aGen : gens ) {
 // do stuff with aGen
}
```

Do what is more readable (speed difference is tiny)

art & Root

art links with Root!

You can call anything in Root from art

(This is irrespective of the fact that art serializes with Root i/o)

There are some helpers that make things easier from within art

art::TFileService

TFileService gives you access to Root i/o that won't interfere with art's use of Root i/o

Uh, what's a Service?

An art service is a globally accessible object that maintains its state from access to access (ignoring multi-threading behavior)

You can retrieve a particular service with

```
art::ServiceHandle<ServiceType> theService;
```

Preferred over singletons (singletons are EVIL, as we're learning in g-2)

Do not abuse services

Since services are accessible from multiple modules, you may be tempted to put data in a service to be passed to a later module

RESIST!

Data to be passed to later modules should go through the event, never a service.

Passing data via a service may ruin multi-threading capabilities and data may change without provenance.

Histograms (getting to TFileService)

You can have `TH1D*` objects in your modules (histogram to fill)

Your module constructor should set the pointer to null

```
// Module class declaration has private member datum TH1D* hNGens_;

tex::FirstHist1::FirstHist1(fhicl::ParameterSet const& pset):
  art::EDAnalyzer(pset),
  gensTag_(pset.get<std::string>("genParticlesInputTag")),
  hNGens_(nullptr)
{}
```

Note the use of `nullptr`

Booking the histogram

Do this in your module's beginJob method

```
void tex::FirstHist1::beginJob() {
  art::ServiceHandle<art::TFileService> tfs;
  hNGens_ = tfs->make<TH1D>( "hNGens",
                             "Number of generated particles per event",
                             20, 0., 20.);
}
```

Note the use of `tfs->make<TH1D>( TH1D C'tor args )`

Filling the Histogram and saving

Do this in your module's analyze method

```
void tex::FirstHist1::analyze(art::Event const& event ) {
   auto gens = event.getValidHandle<GenParticleCollection>(gensTag_);
   hNGens_->Fill(gens->size());
}
```

You specify the output file in your FHICL

```
services : {
   TFileService  : { fileName : "output/firstHist1.root" }
}
```

You can override file name on the art command line

```
$ art -c myFhicl.fcl -T myAwesomeHistograms.root
```

Now get to work!

You should know enough to perform exercises 6 and 7 from the art workbook (sections 16 and 17)

See https://web.fnal.gov/project/ArtDoc/Shared%20Documents/art-documentation.pdf

Looping over the data

Plotting the size of stuff is nice, but you probably want to see some data too!

We'll look more at `GenParticleCollection` and plot particle type, creation position and creation momentum

We'll look at the recommended way to loop over items in the collection

We'll look at other ways to do the loop that you may encounter

And we'll look at `CLHEP::Hep3Vector` and `CLHEP::HepLorentzVector`

Remember…

```
// $TOYEXPERIMENT_INC/MCDataProducts/GenParticle.h

#include "toyExperiment/DataProducts/PDGCode.h"
#include "CLHEP/Vector/LorentzVector.h"
#include "CLHEP/Vector/ThreeVector.h"

namespace tex {
 class GenParticle {
  public:
   GenParticle();

   GenParticle( PDGCode::type                    pdgId,
                art::Ptr<GenParticle> const&      parent,
                CLHEP::Hep3Vector const&          position,
                CLHEP::HepLorentzVector const& momentum );
   PDGCode::type                    pdgId()           const { return  _pdgId;              }
   bool                             hasParent()       const { return  _parent.isNonnull(); }
   bool                             hasChildren()     const { return !_children.empty();   }
   art::Ptr<GenParticle> const&     parent()          const { return  _parent;             }
   children_type const&             children()        const { return  _children;           }
   art::Ptr<GenParticle> const&     child( size_t i ) const { return  _children.at(i);     }
   CLHEP::Hep3Vector const&         position()        const { return  _position;           }
   CLHEP::HepLorentzVector const& momentum()          const { return  _momentum;           }
```

Let's look at the header file together [go there]

CLHEP

Convenience classes for physics-y things (Geant uses them too)

See CLHEP Documentation (Google it)

CLHEP::Hep3Vector

CLHEP::HepLorentzVector

Looping over data

You have a `GenParticleCollection`, which is a `std::vector<tex::GenParticle>`. You want to loop over the contents of the vector.

There are lots of ways to loop over the contents of a vector…

<u>Really old school way:</u>

```
// Have gensH, a handle to GenParticleCollection
auto const& gens = *gensH;   // gens is the stl::vector of GenParticle's
for ( unsigned int i=0; i < gens.size(); ++i ) {
  doSomethingWith( gens[i] );  // DANGEROUS - NO RANGE CHECKING
  doSomethingWith( gens.at(i) );  // Better - Does range checking
}
```

A note

I prefer to make handles obvious by adding H to the end

```
// Have gensH, a handle to GenParticleCollection
auto const& gens = *gensH;   // gens is the stl::vector of GenParticle's
```

The workbook doesn't do this, and so my `gensH` **is the same as the workbook** `gens`

Keep this in mind as you go between these slides and the workbook (and I'm not consistent in the slides either!)

You will have to deal with similar issues when you read code, so get used to it :-)

A note about []

In practice, use of [] in a for loop will be fine, so long as the bounds of the for loop are correct.

BUT, say after a long night of partying you inadvertently introduce

```
// Have gensH, a handle to GenParticleCollection
auto const& gens = *gensH;   // gens is the stl::vector of GenParticle's
for ( unsigned int i=0; i < someOtherVector.size(); ++i ) {
  doSomethingWith( gens[i] );  // DANGEROUS - NO RANGE CHECKING
}
```

This is actually a typical mistake I see (and have made myself)
If `someOtherVector.size() > gens.size()`, you will exceed the bounds of gens. Uh oh!

You may get a segmentation fault. You may get bogus data.

This is going to be a hard bug to find! Use `at` or a safer loop…

Repeat after me

I will employ features of C++ to help users (and me) avoid mistakes

A little extra typing is worth the trouble!

Every mistake you can render nonexistent saves time, money, frustration, …

A note about ++i and i++

```
unsigned int i =5;
doSomethingWith( gens[i++] ); // passes gens[5], i is now 6
i = 5;
doSomethingWith( gens[++i] ); // i is now 6, passes gens[6]
```

i++ may involve creating a temporary. For a for loop, behavior is identical

```
for ( unsigned int i=0; i < gens.size(); ++i ) {
 doSomethingWith( gens.at(i) );  // Better - Does range checking
}

// Same behavior as

for ( unsigned int i=0; i < gens.size(); i++ ) {
 doSomethingWith( gens.at(i) );  // Better - Does range checking
}
```

For such for loops, `++i` is preferred, but a modern compiler will optimize `i++`. But, better to avoid this for style…

Using STL iterators

You will see this, and variants, in older but STL-aware code,

```
GenParticleCollection::const_iterator i=gens.begin();
GenParticleCollection::const_iterator end=gens.end();

for ( ; i != end; ++i ){

  CLHEP::Hep3Vector const& p = i->momentum();

  hP4_->Fill( p.mag() );
}
```

auto can be your friend

```
for ( auto j=gens.cbegin() ; j != gens.cend() ; ++j ){

  CLHEP::Hep3Vector const& p = j->momentum();

  hP4_->Fill( p.mag() );
}
```

By the way, what's wrong with this?

```
for ( auto const j=gens.begin() ; j != gens.end() ; ++j ){
  CLHEP::Hep3Vector const& p = j->momentum();
  hP4_->Fill( p.mag() );
}
```

Iterators are fun

Iterators are like special pointers for STL containers (e.g. vector)

Advancing an iterator (++`iter`) goes to the next element

Dereference (* or ->) gets the element the iterator points to

You are PAST THE END of the container when the iterator is equal to `container.end()` - also equivalent to "not found"

Range for - the preferred way!

Introduced in C++11

```
// Have gensH, a handle to GenParticleCollection
auto const& gens = *gensH;   // gens is the stl::vector of GenParticle's
for ( GenParticle const& aGen : gens ) {
  doSomethingWith( aGen );
}
```

```
// Have gensH, a handle to GenParticleCollection
for ( GenParticle const& aGen : *gensH ) {  // Use the handle directly
  doSomethingWith( aGen );
}
```

```
// Have gensH, a handle to GenParticleCollection
for ( auto const& aGen : *gensH ) {   // Use auto, but harder to read
  doSomethingWith( aGen );
}
```

What's with the const& ?

```
// Have gensH, a handle to GenParticleCollection
for ( GenParticle const& aGen : *gensH ) {  // Use the handle directly
  doSomethingWith( aGen );
}
```

We need the `const` because `art::Handle::operator*()` returns a const thing

We want to use the & to avoid having an unnecessary copy to a temporary

```
// Have gensH, a handle to GenParticleCollection
for ( GenParticle const aGen : *gensH ) {  // Use the handle directly
  doSomethingWith( aGen ); // aGen is an unnecessary copy - but is legal
                           //     and slower
}
```

One last thing - lambdas

Lambdas are "anonymous functions" (functions without a name) - can be handy

They are cool, but advanced C++

You'll see them looking like

```
[](type name){doSomething;};
```

Here's an example loop…

```
std::for_each(gens.begin(), gens.end(),
              [](auto const& i){ std::cout << i.momentum().mag() << std::endl; });
```

Lambdas

STL algorithms are clever and useful and

See cppreference; They follow a general form…

```
std::some_algorithm( begin_iterator, end_iterator,
                     doSomething( *iterator ) );
```

For example,

```
auto fillIt = [myHist](auto const& elem){ myHist->Fill(elem.position().x(); };
std::for_each( gens.cbegin(), gens.cend(), fillIt);
cet::for_all( gens, fillIt );   // Shortcut for std::for_each
```

Better than a range for? Up to you

```
for ( auto const& elem : gens ) { myHist->Fill(elem.position().x(); }
```

Lambdas

Note added after talk:

```
// This won't work (lambda has no access to myHist)
auto fillIt = [](auto const& elem){ myHist->Fill(elem.position().x(); };

// Put myHist in the [] to transmit that symbol to the lambda
auto fillIt = [myHist](auto const& elem){ myHist->Fill(elem.position().x(); };
std::for_each( gens.cbegin(), gens.cend(), fillIt);
cet::for_all( gens, fillIt );   // Shortcut for std::for_each
```

The body of the lambda is in an isolated scope — symbols in the scope where the lambda is defined are not available to the lambda body `{ lambda body is the code within the braces }`

BUT - Symbol names in the [ ] (e.g. `[myHist]` **above), WILL be made available to the lambda body and the lambda can change them. This is like a python "closure"**

See http://en.cppreference.com/w/cpp/language/lambda

Better example of stl::algorithm

```
auto const firstBigMomentum = std::find_if( gens.cbegin(), gens.cend(),
          [](auto const& elem){ return elem.momentum().mag() > 50.0; }; );

if ( firstBigMomentum != gens.end() ) {
     std::cout << “Woohoo! Found a particle with big momentum "
               << firstBigMomentum.momentum().mag() << “ GeV” << std::endl;
}
else {
     std::cout << "No big momentum particles - I’m sad" << std::endl;
}
```

Now get to work!

You should know enough to perform exercises 6, 7, and 8 from the art workbook (sections 16, 17 and 18)

See https://web.fnal.gov/project/ArtDoc/Shared%20Documents/art-documentation.pdf