Making EMShower thread-safe and related issues

Kyle J. Knoepfel

10 September 2019

LArSoft coordination meeting

Recommended development pattern making things thread-safe

- Start with a module that people use (e.g. EMShower)

- Make all data members ‘const’. This guarantees that, without const-casting:
  - Objects with fundamental types cannot be changed.
  - Only ‘const’-qualified member functions may be called for user-defined types.

- All data members must be initialized in the initialization list of the constructor.
  - i.e. no ‘reconfigure’ calls

- Make sure that all functions that are called are thread-safe.
  - https://indico.fnal.gov/event/20453/session/8/contribution/10/material/slides/0.pdf

Lessons learned from EMShower

- The EMShower module itself was not difficult to make “all const”.

- It’s all of the function calls under the covers, where things are tricky.

Lessons learned from EMShower

- The EMShower module itself was not difficult to make “all const”.
- It’s all of the function calls under the covers, where things are tricky.
- Requires services
  - Geometry
  - DetectorPropertiesService
- Requires algorithms
  - EMShowerAlg, which depends on
    - ShowerEnergyAlg
    - CalorimetryAlg
    - ProjectionMatchingAlg

Lessons learned from EMShower

- The EMShower module itself was not difficult to make “all const”.
- It’s all of the function calls under the covers, where things are tricky.
- Requires services
  - Geometry (thread-safe within a run)
  - DetectorPropertiesService (probably thread-safe within an event)
- Requires algorithms
  - EMShowerAlg, which depends on
    - ShowerEnergyAlg (thread-safe within a run)
    - CalorimetryAlg (thread-safe within a run, unless exp. + C lifetime correction is chosen)
    - ProjectionMatchingAlg (thread-unsafe)

Lessons learned from EMShower

- The EMShower module itself was not difficult to make “all const”.

- It’s all of the function calls under the covers, where things are tricky.

- Req
  - G
  - D

- Req
  - E
    - ShowerEnergyAlg (thread-safe within a run)
    - CalorimetryAlg (thread-safe within a run, unless exp. + C lifetime correction is chosen)
    - ProjectionMatchingAlg (thread-unsafe)

You cannot be assured of thread-safety if you interact with the object only through the (abstract) base class.

Avoid hierarchies:

- Interact with concrete types
- In LArSoft, most services/providers need not be polymorphic

Upgrade example: Projection Matching Algorithm

- Located in `larreco/RecoAlg/PMAlg`
- Several global, mutable variables that must be removed for PMAlg to used in a multi-threaded environment
- Much of it appears to not be used, but it's difficult to know!
- In one week's time, I removed **one** function-local, static, mutable variable.
- Why does it take so long?
  - The code has not been maintained (according to the git history)
  - Mutable global variables make behavior hard to predict
  - There is not a rigorous test suite (only have "product sizes" to go off)
- I would like to address the other static variables this week or next.
- Also working on `DetectorPropertiesService`.

Will give more updates as I go.

Related issues

- These are issues I've encountered while upgrading `EMShower` to be thread-safe

A bug in EMShowerAlg

```
// Look at each shower
std::vector<int>
shower::EMShowerAlg::CheckShowerPlanes(...) {
...
for (auto it = initialShowers.cbegin(); it != initialShowers.cend(); ++it) {

  if (std::distance(initialShowers.cbegin(), it) > 0)
    continue;
  ...
```

- Only first element is ever looked at. Oops.
- Original author acknowledges this is a bug.
- I have not yet removed it because I am not certain what the effect will be.

- Will remove conditional if others think it's okay.

art::Ptrs again

- LArSoft uses many containers of art::Ptr<T> objects.
- Usual danger is dereferencing the art::Ptr an unnecessary number of times.

```
art::PtrVector<recob::Hit> const hitPtrs = get_hits(...);
for (art::Ptr<recob::Hit> const& hitPtr : hitPtrs) {
  auto const view = hitPtr->View();          // Expensive
  auto const peak = hitPtr->PeakTime();      // Expensive
  auto const integral = hitPtr->Integral(); // Expensive
  ...
}
```

art::Ptrs again

- LArSoft uses many containers of `art::Ptr<T>` objects.
- Usual danger is dereferencing the `art::Ptr` an unnecessary number of times.

```
art::PtrVector<recob::Hit> const hitPtrs = get_hits(...);
for (art::Ptr<recob::Hit> const& hitPtr : hitPtrs) {
  recob::Hit const& hit = *hitPtr; // Expensive
  auto const view = hit.View();
  auto const peak = hit.PeakTime();
  auto const integral = hit.Integral();
  ...
}
```

art::Ptrs again

- LArSoft uses many containers of `art::Ptr<T>` objects.
- Usual danger is dereferencing the `art::Ptr` an unnecessary number of times.

```
art::PtrVector<recob::Hit> const hitPtrs = get_hits(...);
for (art::Ptr<recob::Hit> const& hitPtr : hitPtrs) {
  recob::Hit const& hit = *hitPtr; // Expensive
  auto const view = hit.View();
  auto const peak = hit.PeakTime();
  auto const integral = hit.Integral();
  ...
}
```

- This requires explicit dereferencing by the user.
- A better solution is to present an already dereferenced `art::Ptr` to the user.

lar::to_element

- LArSoft uses many containers of `art::Ptr<T>` objects
- This requires explicit dereferencing by the user.

```
#include "lardata/ArtDataHelpers/ToElement.h"
#include "range/v3/view.hpp"

using namespace ranges::view;
using lar::to_element;

art::PtrVector<recob::Hit> const hitPtrs = get_hits(...);
for (recob::Hit const& hit : hitPtrs | transform(to_element)) {
  auto const view = hit.View();
  auto const peak = hit.PeakTime();
  auto const integral = hit.Integral();
  ...
}
```

- A better solution is to present an already dereferenced `art::Ptr` to the user.

lar::to_element

- LArSoft uses many containers of `art::Ptr<T>` objects
- This requires explicit dereferencing by the user.

```
#include "lardata/ArtDataHelpers/ToElement.h"
#include "range/v3/view.hpp"

using namespace ranges::view;
using lar::to_element;

art::PtrVector<recob::Hit> const hitPtrs = get_hits(...);
for (recob::Hit const& hit : hitPtrs | transform(to_element)) {
  auto const view = hit.View();
  auto const peak = hit.PeakTime();
  auto const integral = hit.Integral();
  ...
}
```

Expensive dereference is localized *and* hidden

- A better solution is to present an already dereferenced `art::Ptr` to the user.

This is a range-based approached to algorithms

- We often write *procedural* code (the “how”) instead of *declarative* code (the “what”)

This is a range-based approached to algorithms

- We often write *procedural* code (the “how”) instead of *declarative* code (the “what”)

```
double sum = 0.;
art::PtrVector<recob::Hit> const hitPtrs = get_hits(...);
for (art::Ptr<recob::Hit> const& hitPtr : hitPtrs) {
  if (hitPtr->View() == geo::kU) {
    sum += hitPtr->Integral();
  }
}
```

This is a range-based approached to algorithms

- We often write *procedural* code (the “how”) instead of *declarative* code (the “what”)

```
using namespace ranges::view;

double sum = 0.;
art::PtrVector<recob::Hit> const hitPtrs = get_hits(...);
for (recob::Hit const& hit : hitPtrs | transform(to_element)) {
  if (hit.View() == geo::kU) {
    sum += hitPtr.Integral();
  }
}
```

This is a range-based approached to algorithms

- We often write *procedural* code (the “how”) instead of *declarative* code (the “what”)

```
using namespace ranges::view;
auto hits_on_u_plane = [](auto const& hit) { return hit.View() == geo::kU; };

double sum = 0.;
art::PtrVector<recob::Hit> const hitPtrs = get_hits(...);
for (recob::Hit const& hit : hitPtrs |
                             transform(to_element) |
                             filter(hits_on_u_plane)) {
  sum += hit.Integral();
}
```

This is a range-based approached to algorithms

- We often write *procedural* code (the “how”) instead of *declarative* code (the “what”)

```
using namespace ranges::view;
auto hits_on_u_plane = [](auto const& hit) { return hit.View() == geo::kU; };
auto to_integral = [](auto const& hit) { return hit.Integral(); };

double sum = 0.;
art::PtrVector<recob::Hit> const hitPtrs = get_hits(...);
for (double const integral : hitPtrs |
                             transform(to_element) |
                             filter(hits_on_u_plane) |
                             transform(to_integral)) {
  sum += integral;
}
```

This is a range-based approached to algorithms

- We often write *procedural* code (the “how”) instead of *declarative* code (the “what”)

```
using namespace ranges::view;
auto hits_on_u_plane = [](auto const& hit) { return hit.View() == geo::kU; };
auto to_integral = [](auto const& hit) { return hit.Integral(); };

art::PtrVector<recob::Hit> const hitPtrs = get_hits(...);
double sum = accumulate(hitPtrs |
                        transform(to_element) |
                        filter(hits_on_u_plane) |
                        transform(to_integral),
                        0.);
```

This is a range-based approached to algorithms

- We often write *procedural* code (the “how”) instead of *declarative* code (the “what”)

```
using namespace ranges::view;
auto hits_on_u_plane = [](auto const& hit) { return hit.View() == geo::kU; };
auto to_integral = [](auto const& hit) { return hit.Integral(); };

double sum = accumulate(get_hits(...) |
                        transform(to_element) |
                        filter(hits_on_u_plane) |
                        transform(to_integral),
                        0.);
```

This is a range-based approached to algorithms

- We often write *procedural* code (the “how”) instead of *declarative* code (the “what”)

```
using namespace ranges::view;
auto hits_on_u_plane = [](auto const& hit) { return hit.View() == geo::kU; };
auto to_integral = [](auto const& hit) { return hit.Integral(); };

double const sum = accumulate(get_hits(...) |
                              transform(to_element) |
                              filter(hits_on_u_plane) |
                              transform(to_integral),
                              0.);
```

This is a range-based approached to algorithms

- We often write *procedural* code (the “how”) instead of *declarative* code (the “what”)

```
using namespace ranges::view;
auto hits_on_u_plane = [](auto const& hit) { return hit.View() == geo::kU; };
auto to_integral = [](auto const& hit) { return hit.Integral(); };

double const sum = accumulate(get_hits(...) |
                              transform(to_element) |
                              filter(hits_on_u_plane) |
                              transform(to_integral),
                              0.);
```

`lar::to_element` is available on the **lardata:feature/team_for_mt** branch

`util::CreateAssn` *almost* unnecessary

- The `util::CreateAssn` overloads encapsulate Assns-creation idioms, which would otherwise be more verbose.
  - There are many overloads—hard to understand
  - Due to older interface, there is a lot of coupling to the framework

- The overloads are *almost* unnecessary with the `art::PtrMaker` template.
  - They do provide one-to-many Assns support, which is not supported by Assns directly.

- *art* 3.04 will support `Assns::addMany`

- **Proposal:** once LArSoft has upgraded to *art* 3.04, `util::CreateAssn` should be first deprecated, then removed.

Services for *art* 3.04 (ish)

- Experiments have requested the ability to remove unused services from their configuration.
  - It has been difficult for the framework to know how to do this.

- The *art* project has been granted permission to implement a system that allows for service-configuration pruning.

- In addition, the system will:
  - Enable ‘art --print-description’ to tell you what services are required for a given plugin
  - Restrict where service handles can be created.

- Stay tuned.