Fermilab | U.S. DEPARTMENT OF ENERGY | Office of Science

art 3.10, 3.11, and 3.12

Kyle J. Knoepfel

LArSoft coordination meeting

1 November 2022

Preliminaries

- LArSoft has been using art 3.09 since July 2021
- Some large-ish changes in art since then

  Only some of them are breaking
- https://github.com/art-framework-suite/art/releases
- Today, I'll cover (most of) those changes

art 3.10

Bug fixes

- #113: Correct handling of exceptions thrown from non-main thread.
- #114: Respect `-n` flag for custom input sources.
- #117: Restored major version number when typing `art --version`
- art-framework-suite/fhicl-cpp#8: Optimized handling of long validated configuration sequences

Full Changelog: https://github.com/art-framework-suite/art/commits/v3_10_00

Possibly breaking: The fhicl-cpp bug fix could break user code that includes `fhiclcpp/types/Sequence.h` in any header files used during dictionary generation.

art 3.11 (big)

Usability improvements

- Users no longer need to include the `art/Framework/Core/ModuleMacros.h` header in their modules; it is provided automatically.
- Plugins (including modules, services, tools, etc.) no longer need to explicitly link against Boost's filesystem library.
- Calling `Event::put(...)` now returns an `art::PutHandle<T>` object, which can be used to immediately access the framework-owned product, or more easily make `art::Ptr<T>` objects.
- Users will be able to access an `art::Ptr<T>` 's parent collection by invoking the `art::Ptr<T>::parentAs<...>()` function template.
- Due to some code rearrangements, compiled user module `.so` files should be slightly smaller than in previous art versions.

New features

- #86: Users can now create `art::ProductPtr<T>` objects to point to other products.

art 3.11 (big)

Breaking changes

- Users are now required to provide either the `art::fullRun()` or `art::runFragment()` product semantics when inserting products into the `art::Run`; the similar products semantics must be provided for `art::SubRun` products.
- The `Event::removeCachedProduct(Handle<T>&)` function has been replaced with directly invoking `Handle<T>::removeProduct()`.
- All `rewind` API has been removed from the input source and empty-event timestamp plugins.
- Several adjustments to the output module API--only implementers of an output module need to be aware of these.

art 3.11 (big)

Breaking changes

- Users are now required to provide either the `art::fullRun()` or `art::runFragment()` product semantics when inserting products into the `art::Run`; the similar products semantics must be provided for `art::SubRun` products.

You will need to specify a *product semantic* when calling `(Sub)Run::put`.

Specifying…	…means the product corresponds to
`art::fullRun()`	**The full run**
`art::runFragment()`	**All events processed for that run fragment**
`art::runFragment(rs)`	*The custom set of event ranges rs*
`art::fullSubRun()`	**The full subrun**
`art::subRunFragment()`	**All events processed for that subrun fragment**
`art::subRunFragment(rs)`	*The custom set of event ranges rs*

Getting parent collection of `art::Ptr`

Getting parent collection of `art::Ptr`

Up through art 3.10

```
art::Ptr<T> ptr = ...;
*ptr; // Ensure parent product is in memory

art::Handle<std::vector<U>> h;
e.get(ptr.id(), h);
auto const& ts = *h;
```

Getting parent collection of `art::Ptr`

Up through art 3.10

```
art::Ptr<T> ptr = ...;
*ptr; // Ensure parent product is in memory

art::Handle<std::vector<U>> h;
e.get(ptr.id(), h);
auto const& ts = *h;
```

art 3.11

```
art::Ptr<T> ptr = ...;
auto const& ts = ptr.parentAs<std::vector>(); // or ...
auto const& ts = ptr.parentAs<std::vector<U>>(); // if T != U
```

How do you create an `art::Ptr`?

How do you create an `art::Ptr`?

Up through art 3.10

Collection and Ptrs created in **different** modules

```
auto tsH = e.getValidHandle<std::vector<T>>(tag);

auto ptrs = std::make_unique<std::vector<art::Ptr<T>>>();
for (size_t i = 0; i != tsh->size(); ++i) {
  ptrs->push_back(art::Ptr<T>{tsH, i});
}
e.put(move(ptrs));
```

How do you create an `art::Ptr`?

Up through art 3.10

Collection and Ptrs created in **different** modules

```
auto tsH = e.getValidHandle<std::vector<T>>(tag);

auto ptrs = std::make_unique<std::vector<art::Ptr<T>>>();
for (size_t i = 0; i != tsh->size(); ++i) {
  ptrs->push_back(art::Ptr<T>{tsH, i});
}
e.put(move(ptrs));
```

Collection and Ptrs created in **same** module

```
auto ts = std::make_unique<std::vector<T>>();
// fill ts

auto ts_pid = e.getProductID<std::vector<T>>();
auto getter = e.productGetter(ts_pid);
auto ptrs = std::make_unique<std::vector<art::Ptr<T>>>();
for (size_t i = 0; i != ts->size(); ++i) {
  ptrs->push_back(art::Ptr<T>{ts_pid, i, getter});
}
e.put(move(ts));
e.put(move(ptrs));
```

How do you create an `art::Ptr`?

art 3.11

Collection and Ptrs created in **different** modules

```
auto tsH = e.getValidHandle<std::vector<T>>(tag);

auto ptrs = std::make_unique<std::vector<art::Ptr<T>>>();
for (size_t i = 0; i != tsh->size(); ++i) {
  ptrs->push_back(art::Ptr<T>{tsH, i});
}
e.put(move(ptrs));
```

Collection and Ptrs created in **same** module

```
auto ts = std::make_unique<std::vector<T>>();
// fill ts
auto tsH = e.put(move(ts));

auto ptrs = std::make_unique<std::vector<art::Ptr<T>>>();
for (size_t i = 0; i != tsH->size(); ++i) {
  ptrs->push_back(art::Ptr<T>{tsH, i});
}
e.put(move(ptrs));
```

How do you create an `art::Ptr`?

art 3.11

Collection and Ptrs created in **different** modules

```
auto tsH = e.getValidHandle<std::vector<T>>(tag);

auto ptrs = std::make_unique<std::vector<art::Ptr<T>>>();
for (size_t i = 0; i != tsh->size(); ++i) {
  ptrs->push_back(art::Ptr<T>{tsH, i});
}
e.put(move(ptrs));
```

Collection and Ptrs created in **same** module

```
auto ts = std::make_unique<std::vector<T>>();
// fill ts
auto tsH = e.put(move(ts));

auto ptrs = std::make_unique<std::vector<art::Ptr<T>>>();
for (size_t i = 0; i != tsH->size(); ++i) {
  ptrs->push_back(art::Ptr<T>{tsH, i});
}
e.put(move(ptrs));
```

How do you create an `art::Ptr`?

art 3.11

Collection and Ptrs created in **different** modules

```
auto tsH = e.getValidHandle<std::vector<T>>(tag);

auto ptrs = std::make_unique<std::vector<art::Ptr<T>>>();
for (size_t i = 0; i != tsh->size(); ++i) {
  ptrs->push_back(art::Ptr<T>{tsH, i});
}
e.put(move(ptrs));
```

Collection and Ptrs created in **same** module

```
auto ts = std::make_unique<std::vector<T>>();
// fill ts
auto tsH = e.put(move(ts));

auto ptrs = std::make_unique<std::vector<art::Ptr<T>>>();
for (size_t i = 0; i != tsH->size(); ++i) {
  ptrs->push_back(art::Ptr<T>{tsH, i});
}
e.put(move(ptrs));
```

What is this?

Introducing `art::PutHandle<T>` with art 3.11

- Lightweight handle that allows immediate (and immutable) access to the product put onto the event (and subrun, etc.)
- You do not have to use it—i.e. you can discard the return value of `Event::put`.

Introducing `art::PutHandle<T>` with art 3.11

- Lightweight handle that allows immediate (and immutable) access to the product put onto the event (and subrun, etc.)
- You do not have to use it—i.e. you can discard the return value of `Event::put`.
- A `PutHandle` is always valid.

  Cannot default-construct or invalidate a `PutHandle`.
- You *cannot* access provenance information through a `PutHandle`.

Introducing `art::PutHandle<T>` with art 3.11

- Lightweight handle that allows immediate (and immutable) access to the product put onto the event (and subrun, etc.)
- You do not have to use it—i.e. you can discard the return value of `Event::put`.
- A `PutHandle` is always valid.

  Cannot default-construct or invalidate a `PutHandle`.
- You *cannot* access provenance information through a `PutHandle`.
- Implicitly convertible to a `ProductID` to support backwards compatibility for now.

```
art::ProductID id = e.put(move(ts)); // Old form -- allowed in art 3.11

art::PutHandle<std::vector<T>> tsH = e.put(move(ts)); // New form

auto tsh = e.put(move(ts)); // Encouraged form
```

Creating collections and Assns in the same module

art 3.10 example

```
using uints = vector<size_t>;
using strings = vector<string>;

void produce(art::Event& e)
{
  // Collections among which we will make associations
  auto vui = std::make_unique<uints>(uints{2, 0, 1});
  auto vs = std::make_unique<strings>(strings{"one", "two", "zero"});

  // Product IDs and getters
  art::ProductID vui_pid = e.getProductID<uints>();
  art::ProductID vs_pid = e.getProductID<strings>();

  art::EDProductGetter const* vui_getter = e.productGetter(vui_pid);
  art::EDProductGetter const* vs_getter = e.productGetter(vs_pid);

  // Now create the Assns
  auto assns = std::make_unique<art::Assns<size_t, string>>();
  assns->addSingle(art::Ptr<size_t>{vui_pid, 1, vui_getter}, art::Ptr<string>{vs_pid, 2, vs_getter});
  assns->addSingle(art::Ptr<size_t>{vui_pid, 2, vui_getter}, art::Ptr<string>{vs_pid, 0, vs_getter});
  assns->addSingle(art::Ptr<size_t>{vui_pid, 0, vui_getter}, art::Ptr<string>{vs_pid, 1, vs_getter});

  e.put(move(vui));
  e.put(move(vs));
  e.put(move(av));
}
```

Creating collections and Assns in the same module

art 3.11 example

```
using uints = vector<size_t>;
using strings = vector<string>;

void produce(art::Event& e)
{
  // Collections among which we will make associations
  auto vuiH = e.put(std::make_unique<uints>(uints{2, 0, 1}));
  auto vsH = e.put(std::make_unique<strings>(strings{"one", "two", "zero"}));

  // Now create the Assns
  auto assns = std::make_unique<art::Assns<size_t, string>>();
  assns->addSingle(art::Ptr<size_t>{vuiH, 1}, art::Ptr<string>{vsH, 2});
  assns->addSingle(art::Ptr<size_t>{vuiH, 2}, art::Ptr<string>{vsH, 0});
  assns->addSingle(art::Ptr<size_t>{vuiH, 0}, art::Ptr<string>{vsH, 1});
  e.put(move(assns));
}
```

Creating collections and Assns in the same module

art 3.11 example

```
using uints = vector<size_t>;
using strings = vector<string>;

void produce(art::Event& e)
{
  // Collections among which we will make associations
  auto vuiH = e.put(std::make_unique<uints>(uints{2, 0, 1}));
  auto vsH = e.put(std::make_unique<strings>(strings{"one", "two", "zero"}));

  // Now create the Assns
  auto assns = std::make_unique<art::Assns<size_t, string>>();
  assns->addSingle(art::Ptr<size_t>{vuiH, 1}, art::Ptr<string>{vsH, 2});
  assns->addSingle(art::Ptr<size_t>{vuiH, 2}, art::Ptr<string>{vsH, 0});
  assns->addSingle(art::Ptr<size_t>{vuiH, 0}, art::Ptr<string>{vsH, 1});
  e.put(move(assns));
}
```

Enough type information in handles that you shouldn't need to specify the `art::Ptr` type.

Creating collections and Assns in the same module

art 3.11 example

```
using uints = vector<size_t>;
using strings = vector<string>;

void produce(art::Event& e)
{
  // Collections among which we will make associations
  auto vuiH = e.put(std::make_unique<uints>(uints{2, 0, 1}));
  auto vsH = e.put(std::make_unique<strings>(strings{"one", "two", "zero"}));

  // Now create the Assns
  auto assns = std::make_unique<art::Assns<size_t, string>>();
  assns->addSingle(art::Ptr<size_t>{vuiH, 1}, art::Ptr<string>{vsH, 2});
  assns->addSingle(art::Ptr<size_t>{vuiH, 2}, art::Ptr<string>{vsH, 0});
  assns->addSingle(art::Ptr<size_t>{vuiH, 0}, art::Ptr<string>{vsH, 1});
  e.put(move(assns));
}
```

Enough type information in handles that you shouldn't need to specify the `art::Ptr` type.

*For art 3.11, **we have added type-deduction guides** (CTAD):*

```
art::Ptr{vuiH, 1} inferred as art::Ptr<size_t>{vuiH, 1}
art::Ptr{vsH, 1}  inferred as art::Ptr<string>{vsH, 1}
```

Creating collections and Assns in the same module

art 3.11 example

```
using uints = vector<size_t>;
using strings = vector<string>;

void produce(art::Event& e)
{
  // Collections among which we will make associations
  auto vuiH = e.put(std::make_unique<uints>(uints{2, 0, 1}));
  auto vsH = e.put(std::make_unique<strings>(strings{"one", "two", "zero"}));

  // Now create the Assns
  auto assns = std::make_unique<art::Assns<size_t, string>>();
  assns->addSingle(art::Ptr{vuiH, 1}, art::Ptr{vsH, 2});
  assns->addSingle(art::Ptr{vuiH, 2}, art::Ptr{vsH, 0});
  assns->addSingle(art::Ptr{vuiH, 0}, art::Ptr{vsH, 1});
  e.put(move(assns));
}
```

Creating collections and Assns in the same module

art 3.11 example

```
using uints = vector<size_t>;
using strings = vector<string>;

void produce(art::Event& e)
{
  // Collections among which we will make associations
  auto vuiH = e.put(std::make_unique<uints>(uints{2, 0, 1}));
  auto vsH = e.put(std::make_unique<strings>(strings{"one", "two", "zero"}));

  // Now create the Assns
  auto assns = std::make_unique<art::Assns<size_t, string>>();
  assns->addSingle(art::Ptr{vuiH, 1}, art::Ptr{vsH, 2});
  assns->addSingle(art::Ptr{vuiH, 2}, art::Ptr{vsH, 0});
  assns->addSingle(art::Ptr{vuiH, 0}, art::Ptr{vsH, 1});
  e.put(move(assns));
}
```

We also allow implicit construction of an `art::Ptr`…

Creating collections and Assns in the same module

art 3.11 example

```
using uints = vector<size_t>;
using strings = vector<string>;

void produce(art::Event& e)
{
  // Collections among which we will make associations
  auto vuiH = e.put(std::make_unique<uints>(uints{2, 0, 1}));
  auto vsH = e.put(std::make_unique<strings>(strings{"one", "two", "zero"}));

  // Now create the Assns
  auto assns = std::make_unique<art::Assns<size_t, string>>();
  assns->addSingle({vuiH, 1}, {vsH, 2});
  assns->addSingle({vuiH, 2}, {vsH, 0});
  assns->addSingle({vuiH, 0}, {vsH, 1});
  e.put(move(assns));
}
```

Introducing `art::ProductPtr<P>` with art 3.11

It is a persistable pointer to a product.

You should use an `art::ProductPtr` **only if**:

1. You need a pointer to a data product, and
2. You need that pointer to be preserved in an output file for other framework jobs

Introducing `art::ProductPtr<P>` with art 3.11

It is a persistable pointer to a product.

You should use an `art::ProductPtr` **only if**:

1. You need a pointer to a data product, and
2. You need that pointer to be preserved in an output file for other framework jobs

They are easy to create `art::ProductPtr<P> ptr{h};` where h is of type `art::Handle<P>`, `art::ValidHandle<P>`, or `art::PutHandle<P>`.

Introducing `art::ProductPtr<P>` with art 3.11

It is a persistable pointer to a product.

You should use an `art::ProductPtr` **only if**:

1. You need a pointer to a data product, and
2. You need that pointer to be preserved in an output file for other framework jobs

They are easy to create `art::ProductPtr<P> ptr{h};` where h is of type `art::Handle<P>`, `art::ValidHandle<P>`, or `art::PutHandle<P>`.

A `ProductPtr` can be:

- Its own data product
- A data member of a data product
- An element of a data-product collection

art 3.12

art 3.12

New features

- GCC 12.1 and Clang 14.0.6 support (C++17)
- Improvements to `cet::exempt_ptr`, such as easier-to-use equality comparisons, and a C++17 type-deduction guide
- The `end_paths` and `trigger_paths` parameters are automatically calculated and added to the final post-processed configuration (issue #126)
- Improved reporting whenever fast-cloning is disabled (see art-root-io)

Bug fixes

- Fast-cloning is now disabled whenever multi-scheduled execution has been enabled (see art-root-io)
- Restored the user-configurable `SkipEvent` functionality whenever an exception is thrown (issue #127)
- Fixed `const`-qualification of `TFileDirectory::mkdir()` (issue art-framework-suite/art-root-io#12)

Full Changelog: https://github.com/art-framework-suite/art/commits/v3_12_00

Fermilab

art 3.12

New features

- GCC 12.1 and Clang 14.0.6 support (C++17)
- Improvements to `cet::exempt_ptr`, such as easier-to-use equality comparisons, and a C++17 type-deduction guide
- The `end_paths` and `trigger_paths` parameters are automatically calculated and added to the final post-processed configuration (issue #126)
- Improved reporting whenever fast-cloning is disabled (see art-root-io)

```
%MSG-w FastCloning:  PostProcessEvent 01-Aug-2022 16:40:40 CDT  run: 1 subRun: 0 event: 4
Fast cloning has been deactivated for the following reasons:
  - Event-selection has been specified in the RootOutput configuration.
  - More than one schedule (4) is being used.
  - Events are not in entry order.
%MSG
```

Full Changelog: https://github.com/art-framework-suite/art/commits/v3_12_00

Plans for art

- Tentative plan is that art 3.13 will support C++20 (assuming external libraries can support it)

  Timeline is undetermined.

- Sometime down the road, we would like to poll the experiments for their framework needs.