`Ptrs`, product-putting, `ProductPtrs`

Kyle J. Knoepfel

art stakeholder meeting

15 February 2022

What is an `art::Ptr`?

It is a persistable pointer to an element of a collection that is a product.

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

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

Otherwise, you should not use one:

1. It is bigger in memory than a bare pointer (32 bytes vs. 8 bytes).
2. It couples you to the art framework.
3. You sometimes need to deal with `Ptr`-remapping.

Up through 3.10, art has not supported a pointer to a product.

`ProductID` – an important `art::Ptr` ingredient

The framework stores an identifier of every product. They are used for internal bookkeeping purposes.

There are times when users have to interact with them ☹ (e.g.):

- Retrieving the parent collection of an `art::Ptr`
- Creating an `art::Ptr`
- Referring to a product within another persisted product.

One of the goals of art 3.11 is to make each of the above tasks easier.

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}
```


Fermilab

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));
}
```

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));
}
```

A caveat: We disabled CTAD for `cet::map_vector` due to ambiguity.

```
using mv_t = cet::map_vector<double>;
art::PutHandle<mv_t> mvH = ...;

art::Ptr<mv_t::value_type> p1{mvH, 1};  // Okay
art::Ptr<mv_t::mapped_type> p2{mvH, 1}; // Also okay
art::Ptr p3{mvH, 1};  // Compile-time error - no obvious choice for T
```

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

That's it!