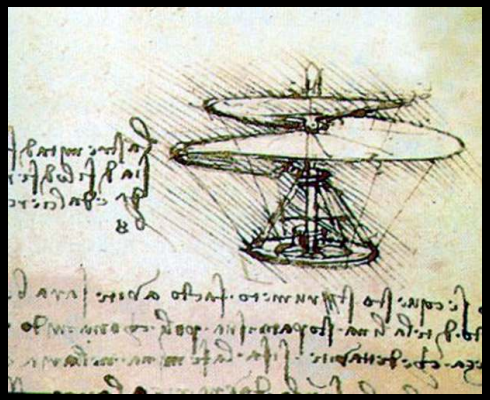
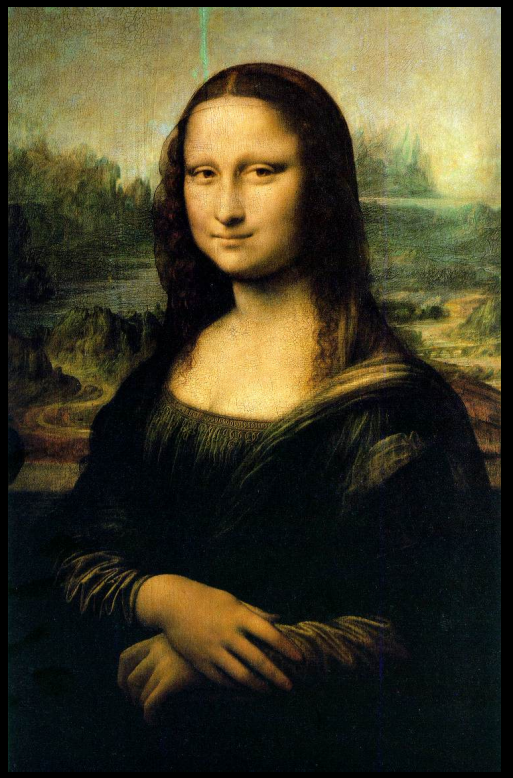




Using Assns and its smart query objects

Chris Green.

art/ **LArSoft** course.

August 6, 2015.

Fermilab

Fermi National Accelerator Laboratory

Office of Science / U.S. Department of Energy

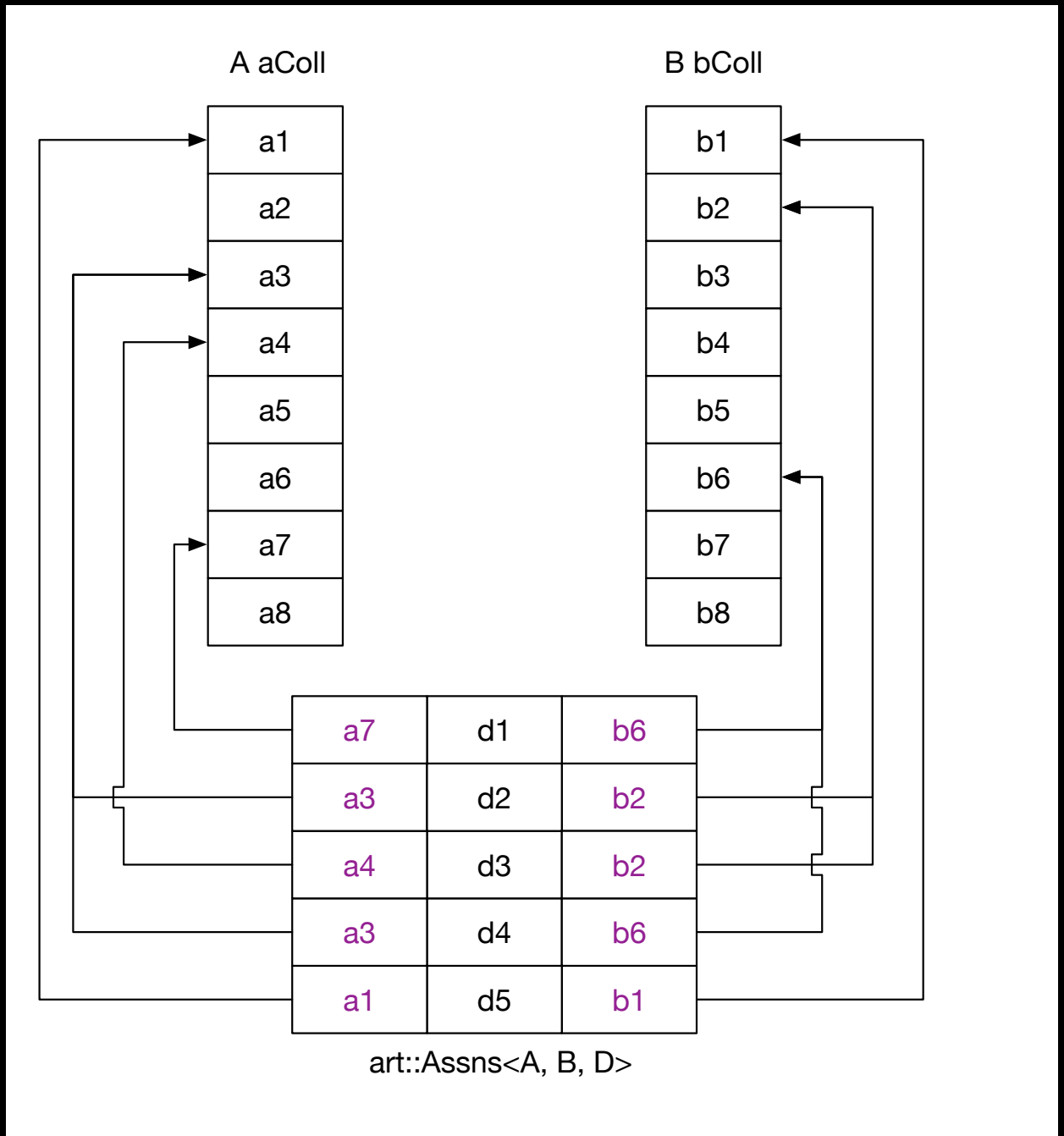
Managed by Fermi Research Alliance, LLC

What is an Assns and when should I use one?

What is an Assns?

- An `art::Assns` is a data product representing *bidirectional associations* between items in collections (in an `art::Event`) of objects of *different types*.
- In addition to recording the fact of an association between such items, it can also save an object recording information specific to the association, such as with `art::Assns<Track, Hit, ResidualInfo>`.

What is an Assns?



Assns and Ptrs

In order to avoid copying data, `art::Assns` stores the associations between items in different collections as pairs of `art::Ptr`, which leads us to ...

What is an `art::Ptr`*?*

What is an `art::Ptr`*?*

- A "persistent pointer."
- Refers to a particular item inside a persistent collection.
- Resolved, "on demand" when dereferenced (`*` or `->`).
- Construction:
  - From an `art::Handle` or `art::ValidHandle`:

    ```
    art::Ptr(handle, index);
    ```

  - From an `art::ProductID` (available from `produces<>()`, `H::id()` or `art::Event::put()`).
- Can check validity:
  1. `art::Ptr::isNonNull()`: is the index set?
  2. `art::Ptr::isAvailable()`: is the referred-to product specified and available in the event?
  3. In *Boolean* context: 1 `&&` 2 :

     ```
     art::Ptr p { handle, index };
     if (p) { ... }
     ```

When should I use an Assns?

- When bi- rather than monodirectional associations between items in collections is needed.
- When associations are made *after* collections have been put into the event.
- When there is important information relevant to the association itself rather than to one or other of the associated items.

Accessing the information in an Assns

Accessing the Information in an Assns

- Directly (*e.g.* `art::Event::getByLabel()`).
- Via a: *Smart Query Object* (*SQO*).

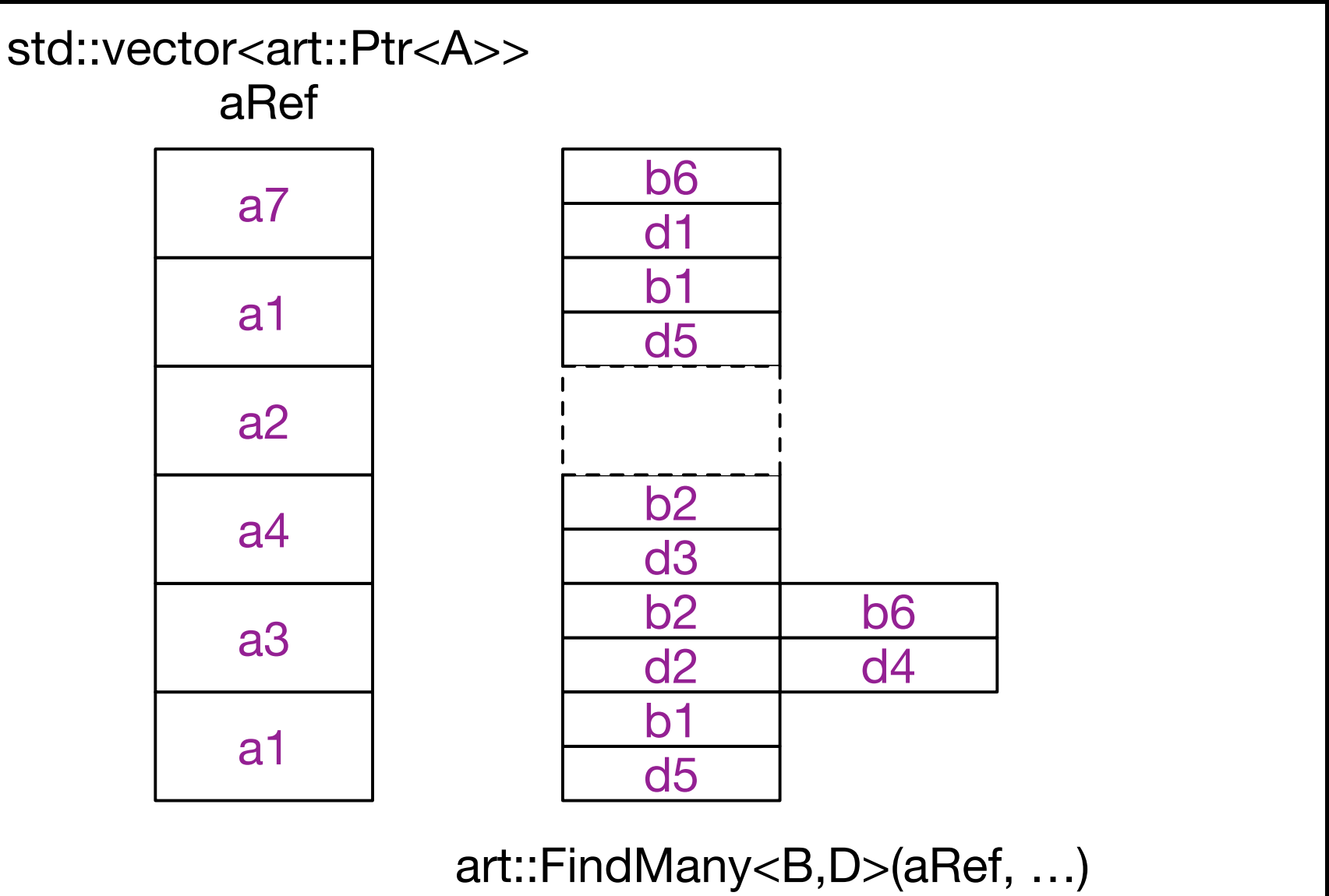
For an *Assns*, these *SQOs* organize a view of the specified association data based on a *reference collection* of items for which the user wishes to obtain associated items (and/or data):

art::FindOne{,P} For one-to-one or many-to-one associations.

art::FindMany{,P} For one-to-many or many-to-many associations.

N.B. Construction of these *SQOs* is expensive relative to use, so make one with as large a reference collection as you need for your task, and then access it as required.

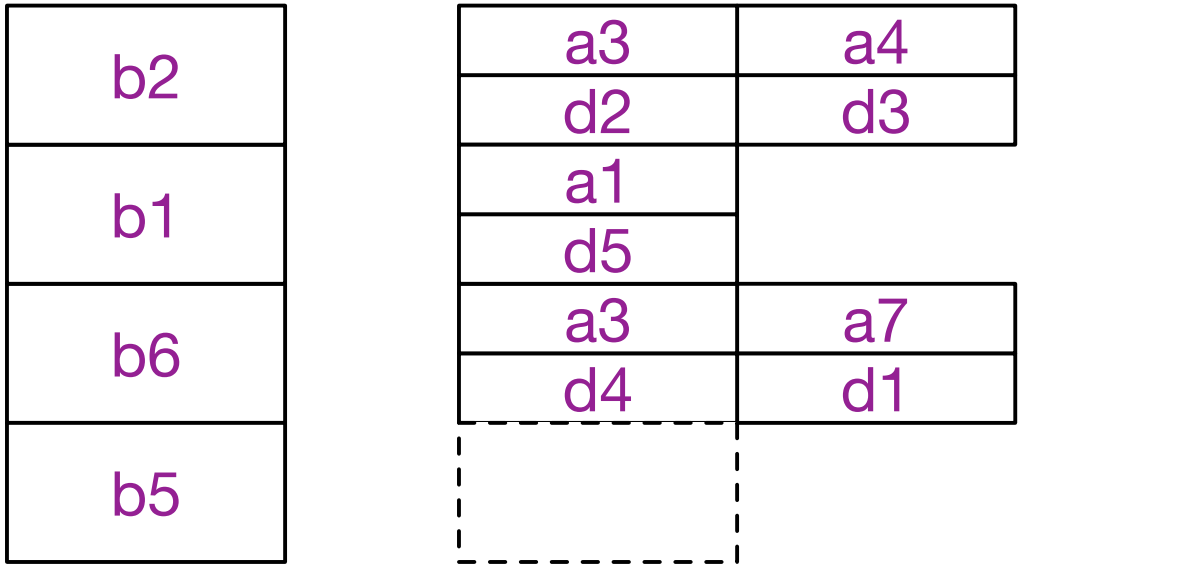
Data organization with an SQO for Assns



Data organization with an SQO for Assns

std::vector<art::Ptr<B>>

bRef



art::FindMany<A,D>(bRef, …)

Constructing an SQO for Assns

```
SQO<B[, D]>(X x, art::Event const & e,
            art::InputTag const & tag);
```

where:

- *SQO* is *art::Find{One,Many}{,P}*
- *B* is the type of the associated items you wish to access.
- *D* (optional) is the data type that provides information about each association.
- *x* is a reference collection of items to which the *B* objects are related.
- *e* is the event provided to you as an argument to *e.g.* your module's `analyze()` or `produce()` function.
- *tag* refers to the `art::Assns<A, B, D>` containing the associations you wish to access.

Specifying the reference collection for an Assns SQO

The reference collection may be specified as any of:

- An `art::Handle<A>` or `art::ValidHandle<A>`.
- An arbitrary sequence of `A const *`, including `art::View<A>`.
- An arbitrary sequence of `art::Ptr<A>`, including `art::PtrVector<A>` and `std::vector<art::Ptr<A>>`.
- A *brace-enclosed-initializer-list* of `art::Ptr<A>`.

Aside: what is a `cet::maybe_ref`?

`cet::maybe_ref` is a "*smart reference*":

- Re-seatable.
- Treat `cet::maybe_ref<A>` as `A &`.
- Treat `cet::maybe_ref<A const>` as `A const &`.
- Check validity with `isValid()`.
- Throws when attempting to dereference an invalid (unset) referent.

Using a FindOne

```
bool isValid() const;
size_type size() const;
cet::maybe_ref<B const>
  at(size_type) const;
// FindOne<B>
void get(size_type,
         cet::maybe_ref<B const> &) const;
// FindOne<B, D>
void get(size_type,
         cet::maybe_ref<B const> &,
         cet::maybe_ref<D const> &) const;
cet::maybe_ref<D const> data(size_type) const;
```

Using a FindOneP

```
bool isValid() const;
size_type size() const;
art::Ptr<B> const &
  at(size_type) const;
// FindOneP<B>
void get(size_type,
         art::Ptr<B> &) const;
// FindOneP<B, D>
void get(size_type,
         art::Ptr<B> &,
         cet::maybe_ref<D const> &) const;
cet::maybe_ref<D const> data(size_type) const;
```

Using a FindMany

```
bool isValid() const;
size_type size() const;
std::vector<B const *>
  at(size_type) const;
// FindMany<B>
void get(size_type,
         std::vector<B const *> &) const;
// FindMany<B, D>
void get(size_type,
         std::vector<B const *> &,
         std::vector<D const *> &) const;
std::vector<D const *> data(size_type) const;
```

Using a FindManyP

```
bool isValid() const;
size_type size() const;
std::vector<art::Ptr<B>> const &
  at(size_type) const;
// FindManyP<B>
void get(size_type,
         std::vector<art::Ptr<B>> &) const;
// FindManyP<B, D>
void get(size_type,
         std::vector<art::Ptr<B>> &,
         std::vector<D const *> &) const;
std::vector<D const *> data(size_type) const;
```