art news

Kyle J. Knoepfel

1 February 2018

Value initialization (or removing `cet::zero_init`)

- Default-initialized, non-`static` fundamental types are set to indeterminate values.

```
// C++03
void f()
{
  int i;          // i = ? (dflt init)
  int j();        // function declaration
  int k = 0;      // k = 0 (value init)
  int l(0);       // l = 0 (direct init)
  int m = int(); // m = 0 (value init)

  // cetlib provides zero_init
  zero_init<int> n; // ”n" = 0
}
```


Fermilab

Value initialization (or removing `cet::zero_init`)

- Default-initialized, non-`static` fundamental types are set to indeterminate values.

```
// C++03
void f()
{
  int i;          // i = ? (dflt init)
  int j();        // function declaration
  int k = 0;      // k = 0 (value init)
  int l(0);       // l = 0 (direct init)
  int m = int(); // m = 0 (value init)

  // cetlib provides zero_init
  zero_init<int> n; // "n" = 0
}
```

```
// C++11 and newer
void g()
{
  // C++03 rules still apply, but also:
  auto p = int(); // p = 0 (value init)
  int q{};        // q = 0 (value init)
  int r{0};       // r = 0 (direct init)
}
```


Fermilab

Value initialization (or removing `cet::zero_init`)

- Default-initialized, non-`static` fundamental types are set to indeterminate values.

```
// C++03
void f()
{
  int i;          // i = ? (dflt init)
  int j();        // function declaration
  int k = 0;      // k = 0 (value init)
  int l(0);       // l = 0 (direct init)
  int m = int(); // m = 0 (value init)

  // cetlib provides zero_init
  zero_init<int> n; // "n" = 0
}
```

```
// C++11 and newer
void g()
{
  // C++03 rules still apply, but also:
  auto p = int(); // p = 0 (value init)
  int q{};        // q = 0 (value init)
  int r{0};       // r = 0 (direct init)
}
```

- Since straightforward, modern C++ methods exist for default initialization, we propose removing the `cet::zero_init` function template.

Fermilab

cet::map_vector<T> behaviors

- cet::map_vector<T> exists for representing a sparse vector
- There are several ways of inserting elements

Fermilab

cet::map_vector<T> behaviors

- cet::map_vector<T> exists for representing a sparse vector
- There are several ways of inserting elements

```
map_vector<string> mv;
mv[mv_key{3}] = "Alice";
mv[mv_key{7}] = "Erin";
mv[mv_key{5}] = "Charlie";
print(mv);
```


Fermilab

cet::map_vector<T> behaviors

- cet::map_vector<T> exists for representing a sparse vector
- There are several ways of inserting elements

```
map_vector<string> mv;
mv[mv_key{3}] = "Alice";
mv[mv_key{7}] = "Erin";
mv[mv_key{5}] = "Charlie";
print(mv);
```

```
3: Alice
5: Charlie
7: Erin
```


Fermilab

cet::map_vector<T> behaviors

- cet::map_vector<T> exists for representing a sparse vector
- There are several ways of inserting elements

```
map_vector<string> mv;
mv[mv_key{3}] = "Alice";
mv[mv_key{7}] = "Erin";
mv[mv_key{5}] = "Charlie";

auto const pairs = {make_pair(mv_key{6}, "David"),
                    make_pair(mv_key{4}, "Bob")};
mv.insert(cbegin(pairs), cend(pairs));
print(mv);
```


Fermilab

cet::map_vector<T> behaviors

- cet::map_vector<T> exists for representing a sparse vector
- There are several ways of inserting elements

```
map_vector<string> mv;
mv[mv_key{3}] = "Alice";
mv[mv_key{7}] = "Erin";
mv[mv_key{5}] = "Charlie";

auto const pairs = {make_pair(mv_key{6}, "David"),
                    make_pair(mv_key{4}, "Bob")};
mv.insert(cbegin(pairs), cend(pairs));
print(mv);
```

```
3: Alice
5: Charlie
7: Erin
14: David
12: Bob
```


Fermilab

cet::map_vector<T> behaviors

- cet::map_vector<T> exists for representing a sparse vector
- There are several ways of inserting elements

```
map_vector<string> mv;
mv[mv_key{3}] = "Alice";
mv[mv_key{7}] = "Erin";
mv[mv_key{5}] = "Charlie";

auto const pairs = {make_pair(mv_key{6}, "David"),
                    make_pair(mv_key{4}, "Bob")};
for (auto const& elem : pairs) {
  mv.push_back(elem);
}
print(mv);
```


Fermilab

cet::map_vector<T> behaviors

- cet::map_vector<T> exists for representing a sparse vector
- There are several ways of inserting elements

```
map_vector<string> mv;
mv[mv_key{3}] = "Alice";
mv[mv_key{7}] = "Erin";
mv[mv_key{5}] = "Charlie";

auto const pairs = {make_pair(mv_key{6}, "David"),
                    make_pair(mv_key{4}, "Bob")};
for (auto const& elem : pairs) {
  mv.push_back(elem);
}
print(mv);
```

```
3: Alice
5: Charlie
7: Erin
14: David
19: Bob
```


Fermilab

cet::map_vector<T> behaviors

- Where this unintuitive behavior bites you:

```
void MyProducer::produce(art::Event& evt) override
{
  auto const pid = getProductID<map_vector<string>>();
  auto mv = std::make_unique<map_vector<string>>();
  auto pv = std::make_unique<art::PtrVector<string>>();
  auto const pairs = {make_pair(mv_key{6}, "David"),
                      make_pair(mv_key{4}, "Bob")};
  for (auto const& e : pairs) {
    mv->push_back(e);
    pv->push_back(art::Ptr<string>{pid, e.first, e.productGetter(pid)});
  }
  evt.put(move(mv));
  evt.put(move(pv));
}
```


Fermilab

cet::map_vector<T> and art::Ptr<T>

- Where this unintuitive behavior bites you:

```
void MyProducer::produce(art::Event& evt) override
{
  auto const pid = getProductID<map_vector<string>>();
  auto mv = std::make_unique<map_vector<string>>();
  auto pv = std::make_unique<art::PtrVector<string>>();
  auto const pairs = {make_pair(mv_key{6}, "David"),
                      make_pair(mv_key{4}, "Bob")};
  for (auto const& e : pairs) {
    mv->push_back(e);
    pv->push_back(art::Ptr<string>{pid, e.first, e.productGetter(pid)});
  }
  evt.put(move(mv));
  evt.put(move(pv));
}
```

Specified `art::Ptr` keys do not point to the correct `map_vector` key.

Fermilab

cet::map_vector<T> and art::Ptr<T>

- A downstream module then wants to look at the art::Ptrs.

```
void MyAnalyzer::analyze(art::Event const& e) override
{
  auto const& ptrs = e.getValidHandle<art::PtrVector<string>>(...);
  for (auto const& ptr : ptrs) {
    if (ptr) {
      std::cout << *ptr << '\n';
    }
  }
}
```

- The validity check on ‘ptr’ returns true because the container into which it points exists.
- A segmentation violation occurs on the dereference.

Fermilab

Proposed changes to `art::Ptr<T>` behavior

- Current behavior is that if you construct an art::Ptr that points to a non-existent element of a collection, dereferencing that Ptr results in undefined behavior.
- We can add a check on the art::Ptr key (index) while dereferencing that would be inexpensive.
- The constraint: *art::Ptrs can be created only to elements in collections that have a `const`-qualified `size()` accessor*.
  - This is probably always the case, but not necessarily.

Fermilab

`cet::map_vector<T>` insertion behavior

1. Using `operator[]` has the same semantics as `std::map<T>::operator[]`

2. Using `insert(b,e)` requires that the user is responsible for ensuring sorted elements in `[b,e)`.
   - *If the `map_vector` object already has entries, the inserted elements have the key value of the last entry, incremented by 1, and then added to the key value of all elements in `[b,e)`.*

3. By using `push_back(pr)`, if the `map_vector` object already has entries, the key value of the inserted element has the key value of `pr` plus the key value of the last entry, incremented by 1.

Only option 1 is intuitive.

Fermilab

Possible changes to `cet::map_vector<T>` insertion behavior

- At a minimum, the art team can document the difference between `push_back`, `insert`, and `operator[]` and leave the behavior as is
- Change the `push_back/insert` behavior so that users are required to insert values in the correct order (i.e. *art* does no automatic incrementing of key values)—this is faster than calling `operator[]` for those who can guarantee such ordering
- Remove the `push_back/insert` function altogether and just keep `operator[]`

Thoughts?

Fermilab