A NEW MEMORY MODEL FOR PODIO

Thomas Madlener | Nov 23, 2023

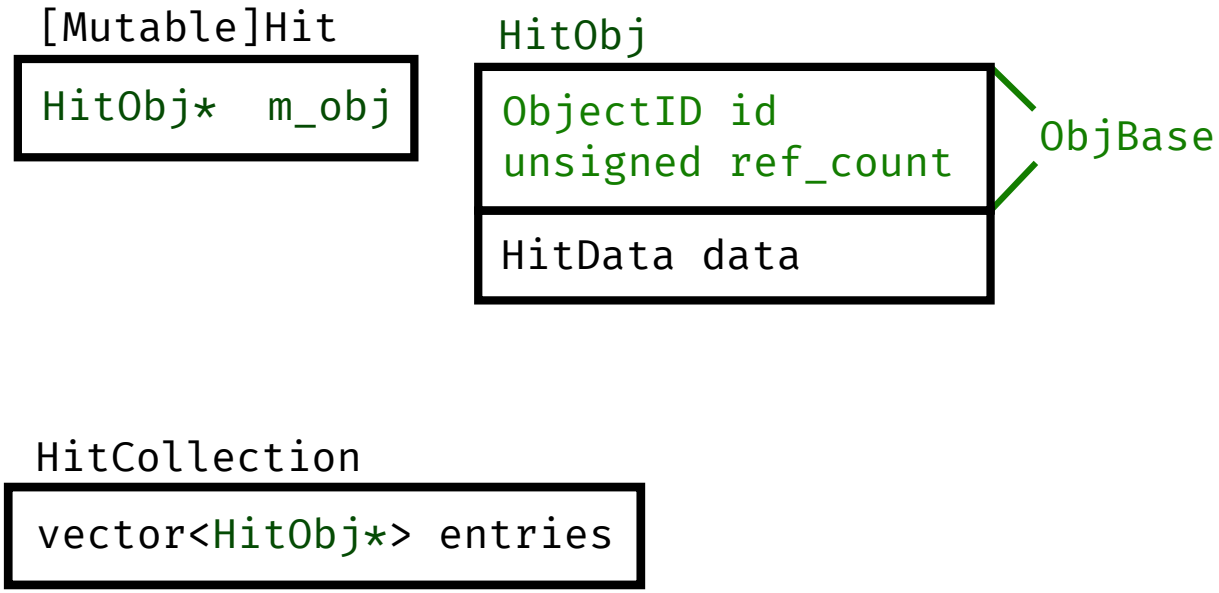
BASIC BUILDING BLOCKS

```
struct ObjectID {
  uint32_t collID{-1};
  int index{-1};
};

struct ObjBase {
  ObjectID id;
  atomic<unsigned> ref_count;

  // increase ref_count if untracked
  int acquire();
  // decrease ref_count if untracked and delete if 0
  int release();
  virtual ~ObjBase() = default;
};
```

GENERATED CODE LAYOUT

```
struct HitData { /* all the hit data members */ };

struct HitObj : public ObjBase {
  HitData data;
};

// Hit is exactly the same!
class MutableHit {
  HitObj* m_obj;
};

class HitCollection {
  vector<HitObj*> entries;
};
```

IN PICTURES


[Mutable]Hit
HitObj* m_obj
HitObj
ObjectID id
unsigned ref_count
ObjBase
HitData data
HitCollection
vector<HitObj*> entries

OBJBASE FUNCTIONS

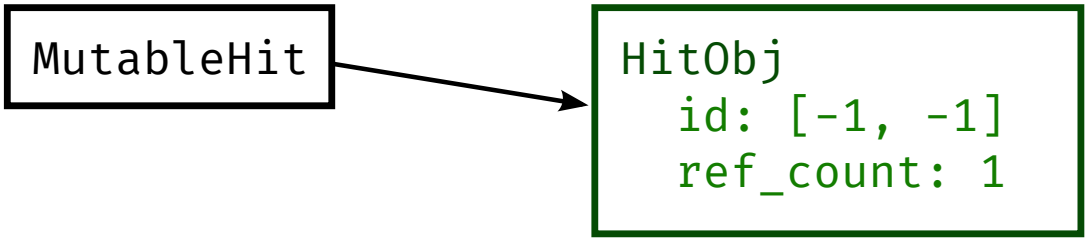
```
void ObjBase::acquire() {
  if (id.index == ObjectID::untracked)
    ++ref_count;
}

int ObjBase::release() {
  if (id.index == ObjectID::untracked)
    return 1;

  if (--ref_count == 0)
    delete this;

  return 0;
}
```

HANDLE CONSTRUCTORS / DESTRUCTORS

```
MutableHit::MutableHit() : m_obj(new HitObj()) {
  m_obj->acquire();
}

Hit::Hit(const Hit& other) : m_obj(other.m_obj) {
  m_obj && m_obj->acquire();
}

Hit::~Hit() {
  m_obj && m_obj->release();
}
```

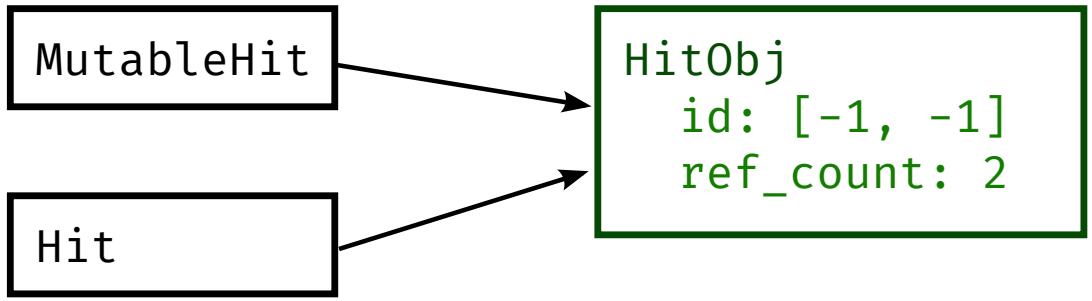
FREE STANDING FUNCTIONALITY

```
void foo(Hit hit);

MutableHit hit{};
```



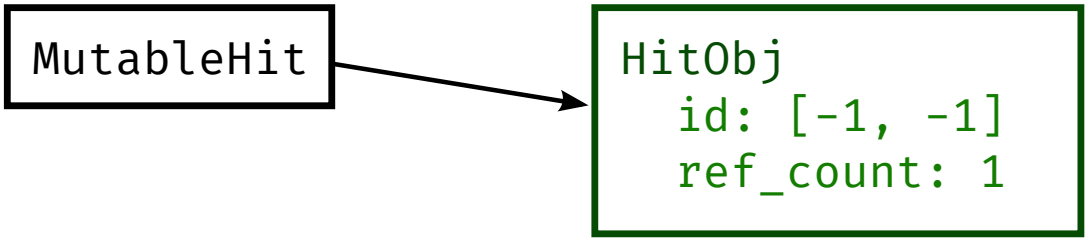
FREE STANDING FUNCTIONALITY

```
void foo(Hit hit);

MutableHit hit{};
foo(hit); // implicit conversion
```



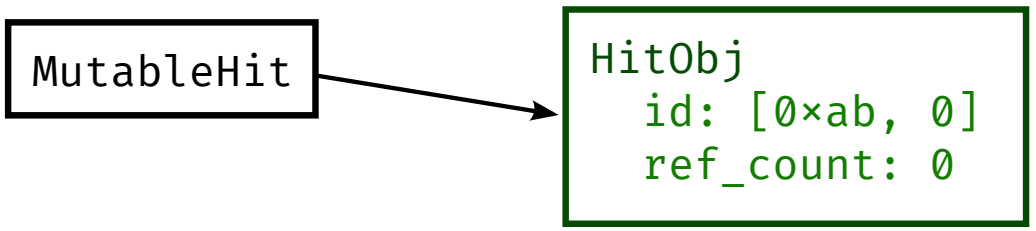
FREE STANDING FUNCTIONALITY

```
void foo(Hit hit);

MutableHit hit{};
foo(hit);
hit.setEnergy(3.14f);
```



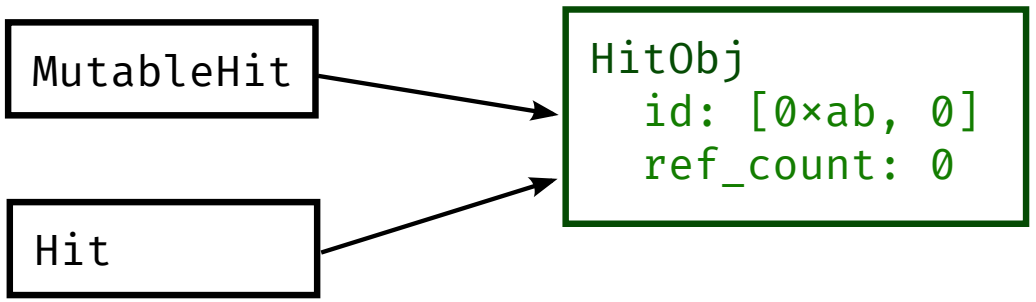
COLLECTION FUNCTIONALITY

```
void foo(Hit hit);

HitCollection hitColl{};
auto hit = hitColl.create();
```



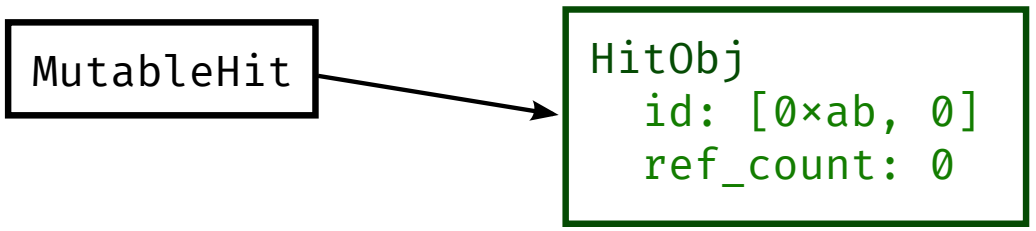
COLLECTION FUNCTIONALITY

```
void foo(Hit hit);

HitCollection hitColl{};
auto hit = hitColl.create();
foo(hit); // implicit conversion
```



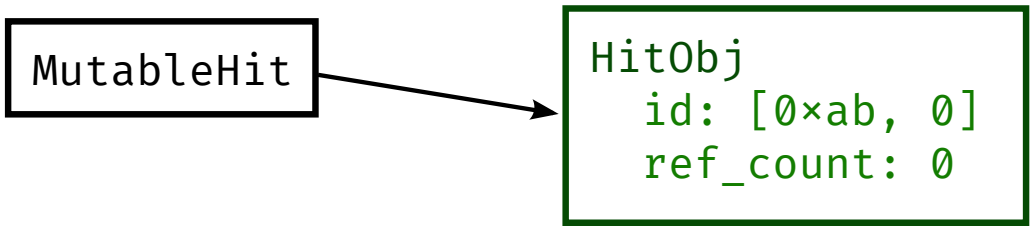
COLLECTION FUNCTIONALITY

```
void foo(Hit hit);

HitCollection hitColl{};
auto hit = hitColl.create();
foo(hit);
hit.setEnergy(3.14f);
```



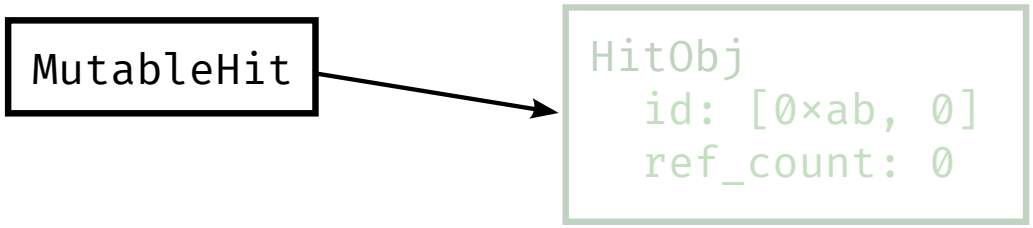
THE CURRENT ISSUE

```
{
  HitCollection hitColl{};
  auto hit = hitColl.create();

}
```



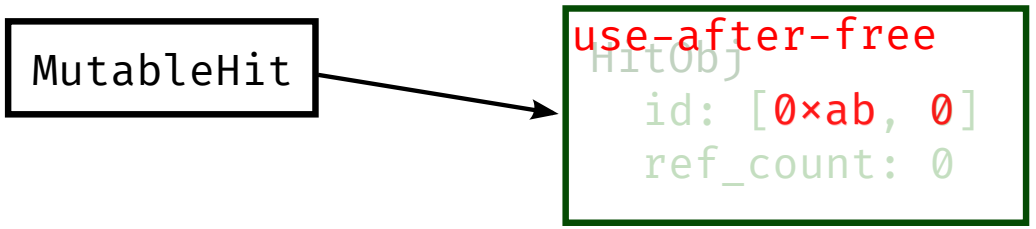
THE CURRENT ISSUE

```
{
  HitCollection hitColl{};
  auto hit = hitColl.create();
  hitColl.clear(); // deletes all Obj*
}
```



THE CURRENT ISSUE

```
{
  HitCollection hitColl{};
  auto hit = hitColl.create();
  hitColl.clear();
} // <--- ~MutableHit is called
```



THE CURRENT ISSUE

- Managed object and “reference count” live in the same object
  - Even if destructor has nothing to do it needs to check first
- Hard to trigger consistently at runtime - *Heisenbug*
  - Nevertheless happens in a path that is **always taken**
- Tools like *AddressSanitizer* immediately spot this
  - podio#492
  - podio#174

MAYBESHAREDPTR

```
template <typename T>
class MaybeSharedPtr {
  // constructors, etc.

  struct ControlBlock {
    atomic<unsigned> count;
    atomic<bool> owned;
  };

  T* m_ptr{nullptr};
  ControlBlock* m_ctrlBlk{nullptr};
};
```

- **Decouple lifetime of control block and managed object**
- May or may not own its resource (“smart” pointer)

MAYBESHAREDPTR (IMPLEMENTATION)

```
MaybeSharedPtr::MaybeSharedPtr(const MaybeSharedPtr& other) :
  m_ptr(other.m_ptr), m_ctrlBlk(other.m_ctrlBlk) {
  m_ctrlBlk && m_ctrlBlock->count++;
}

MaybeSharedPtr::~MaybeSharedPtr() {
  // No ctrl block -> no work
  if (m_ctrlBlk && --m_ctrlBlk->count == 0) {
    // We have to clean up the control block anyway
    // Do we need to clean up resource as well?
    if (m_ctrlBlk->owned) {
      delete m_ptr;
    }
    delete m_ctrlBlk;
  }
}
```

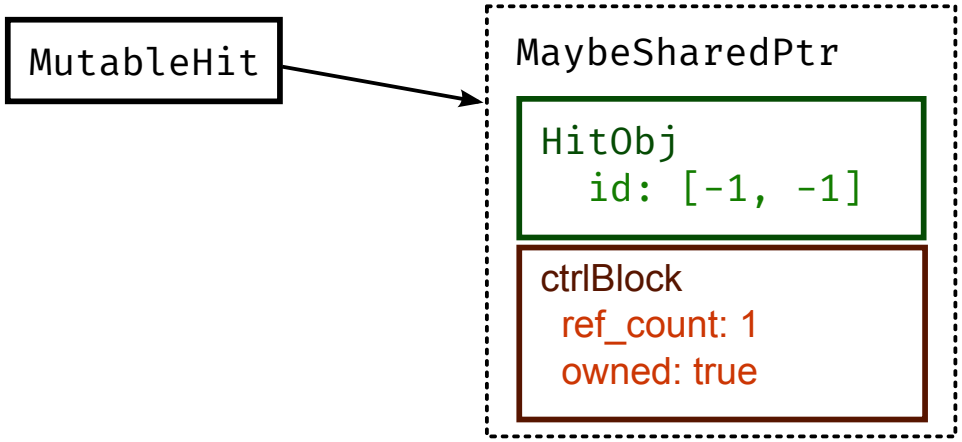
NEW MEMORY MODEL (ATTEMPT #1)

```
class MutableHit {
  ~MutableHit() = default;
  MaybeSharedPtr<HitObj> m_obj;
};

class Hit {
  ~MutableHit() = default;
  HitObj* m_obj;
};
```

- Mutable handles their `Obj` via the `MaybeSharedPtr`
- Default (immutable) handles don't manage their `Obj*` at all
  - Assume that the collection does this for us

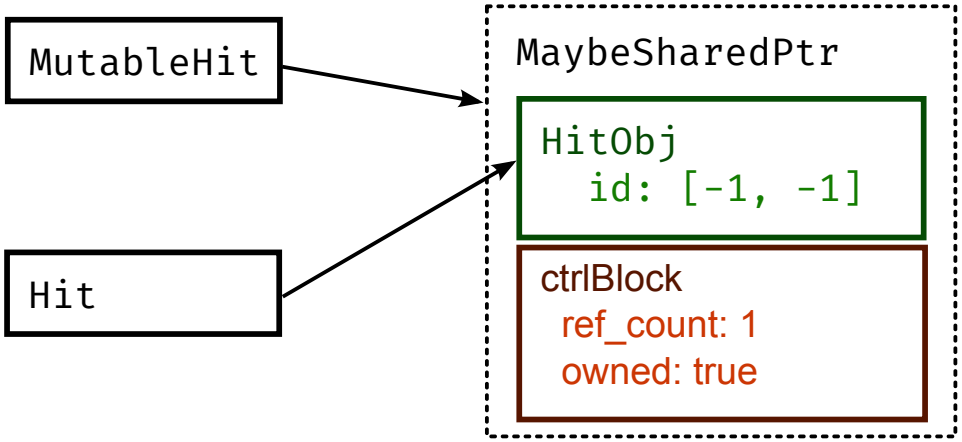
FREE STANDING FUNCTIONALITY

```
void foo(Hit hit);

MutableHit hit{};
```



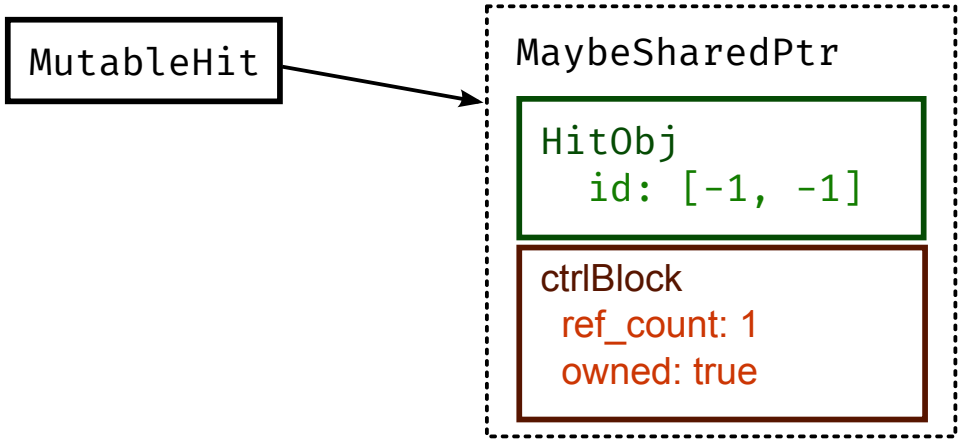
FREE STANDING FUNCTIONALITY

```
void foo(Hit hit);

MutableHit hit{};
foo(hit); // implicit conversion
```



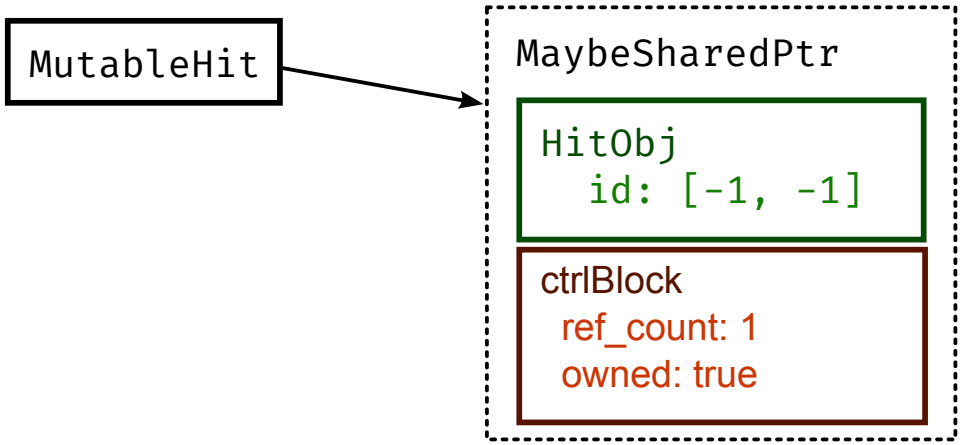
FREE STANDING FUNCTIONALITY

```
void foo(Hit hit);

MutableHit hit{};
foo(hit);
hit.setEnergy(3.14)
```



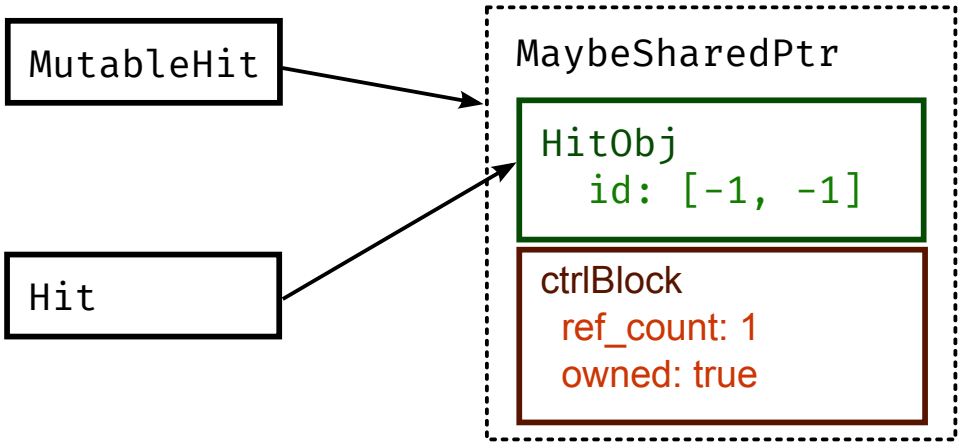
COLLECTION FUNCTIONALITY

```
void foo(Hit hit);

HitCollection hitColl{};
MutableHit hit{};
```



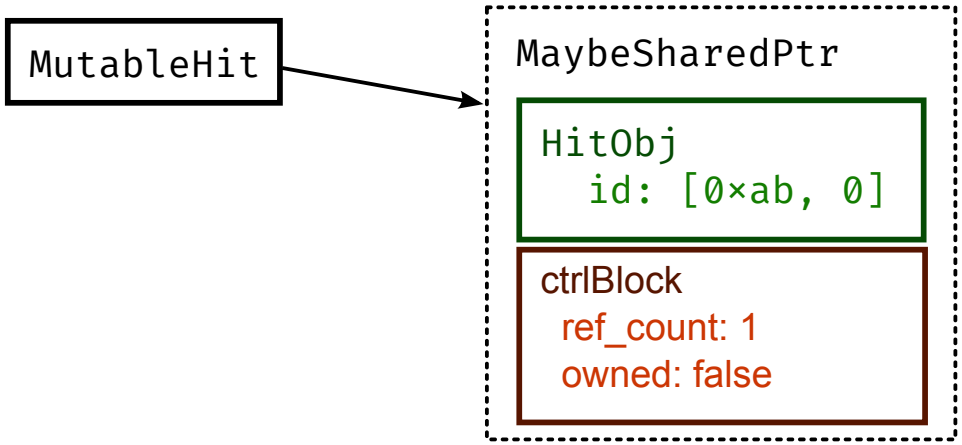
COLLECTION FUNCTIONALITY

```
void foo(Hit hit);

HitCollection hitColl{};
MutableHit hit{};
foo(hit); // implicit conversion
```



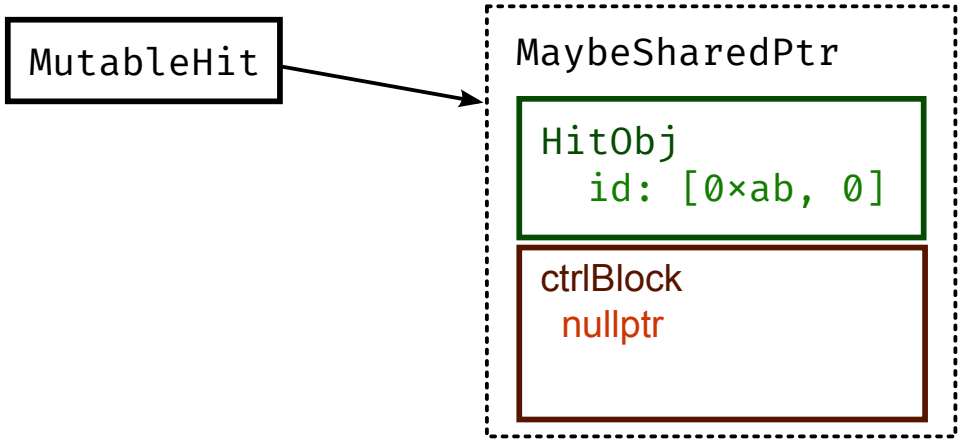
COLLECTION FUNCTIONALITY

```
void foo(Hit hit);

HitCollection hitColl{};
MutableHit hit{};
foo(hit);
hitColl.push_back(hit); // ownership goes to collection
```



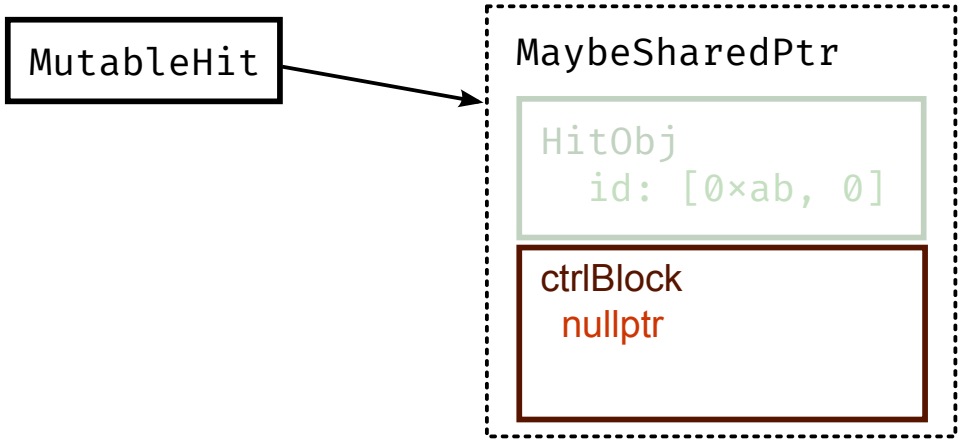
THE ORIGINAL ISSUE

```
{
  HitCollection hitColl{};
  auto hit = hitColl.create();

}
```



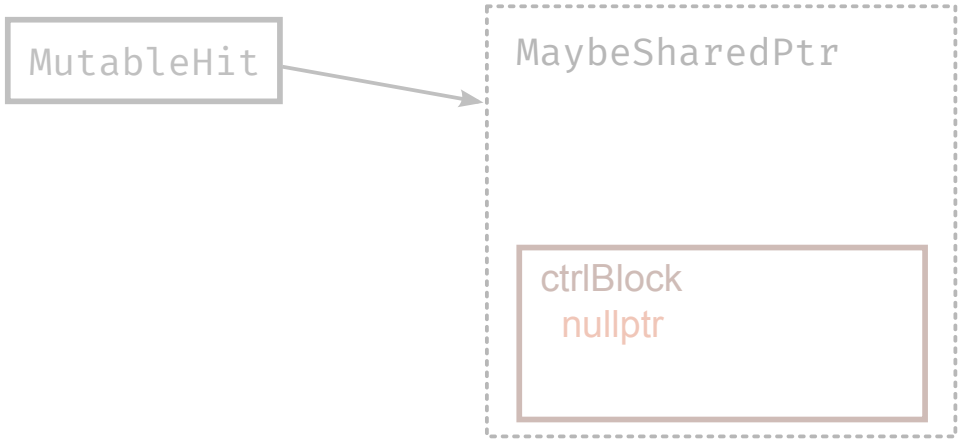
THE ORIGINAL ISSUE

```
{
  HitCollection hitColl{};
  auto hit = hitColl.create();
  hitColl.clear(); // deletes all Obj*
}
```



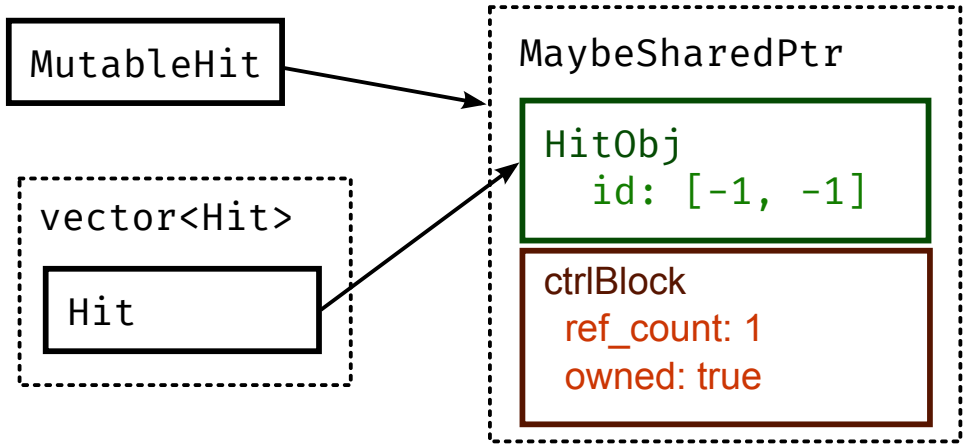
THE ORIGINAL ISSUE

```
{
  HitCollection hitColl{};
  auto hit = hitColl.create();
  hitColl.clear();
} // <--- ~MutableHit is called
```



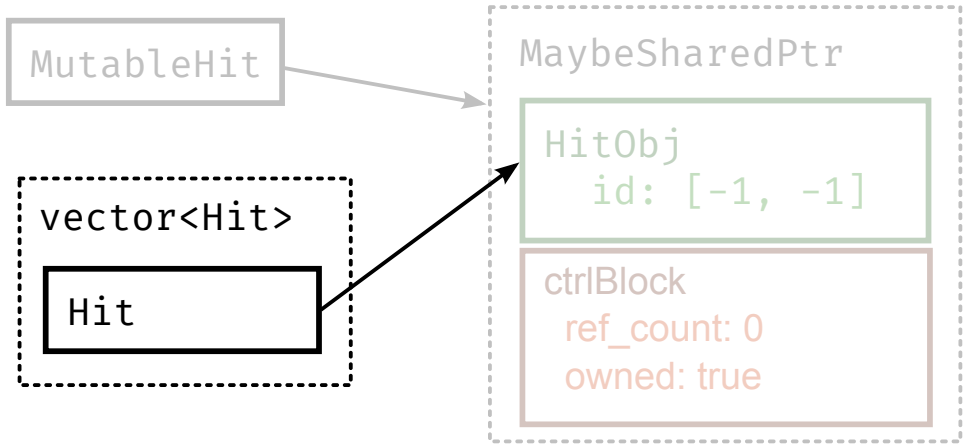
A NEW ISSUE: DANGLING REFERENCES

```
std::vector<Hit> hits;
{
  auto hit = MutableHit{};
  hits.push_back(hit); // implicit conversion
}
```



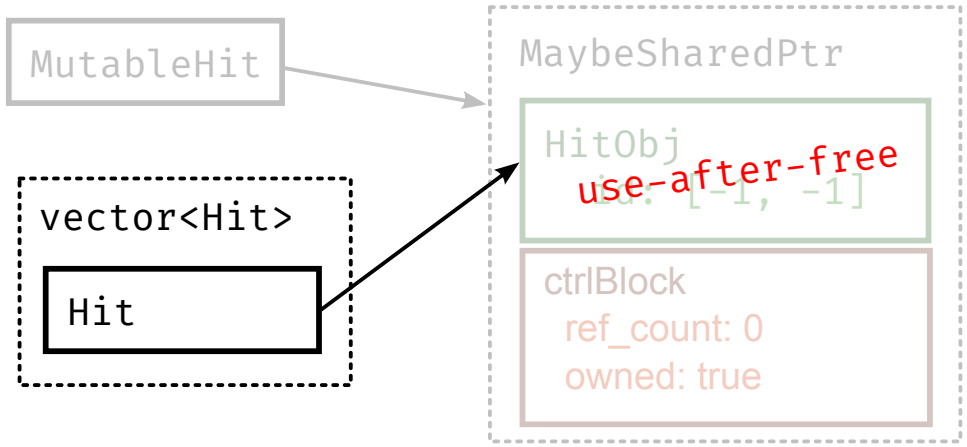
A NEW ISSUE: DANGLING REFERENCES

```
std::vector<Hit> hits;
{
  auto hit = MutableHit{};
  hits.push_back(hit);
} // <--- ~MutableHit() is called
```



A NEW ISSUE: DANGLING REFERENCES

```
std::vector<Hit> hits;
{
  auto hit = MutableHit{};
  hits.push_back(hit);
}
auto e = hits[0].getEnergy();
```



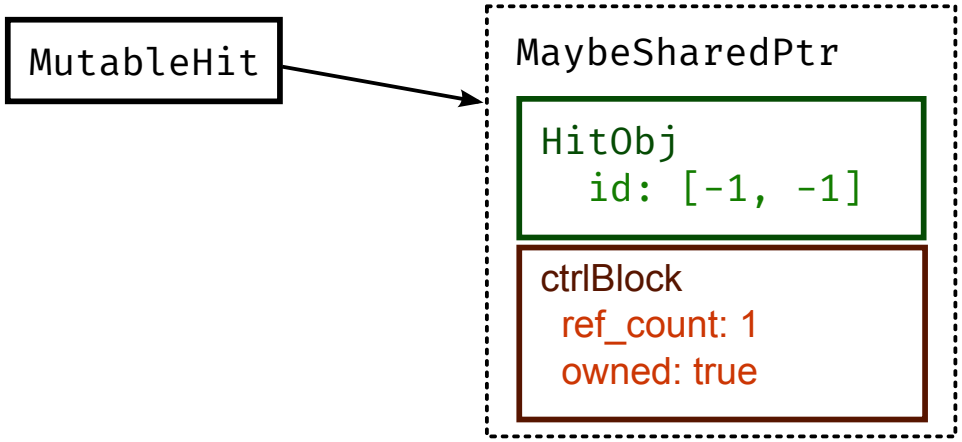
NEW MEMORY MODEL (ATTEMPT #2)

```
class MutableHit {
  ~MutableHit() = default;
  MaybeSharedPtr<HitObj> m_obj;
};

class Hit {
  ~MutableHit() = default;
  MaybeSharedPtr<HitObj> m_obj;
};
```

- All handles are managed by `MaybeSharedPtr`
- All handles double in size (now 2 pointers)
- *Observable behavior* unchanged to current podio

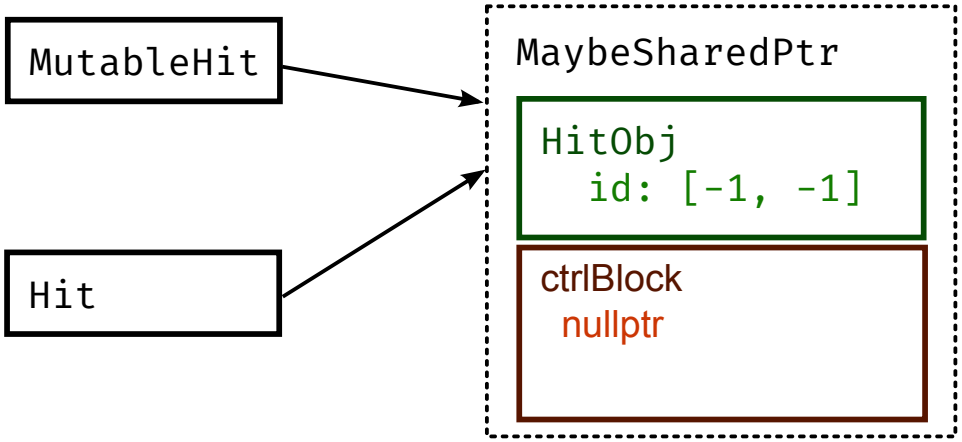
FREE STANDING FUNCTIONALITY

```
void foo(Hit hit);

MutableHit hit{};
```



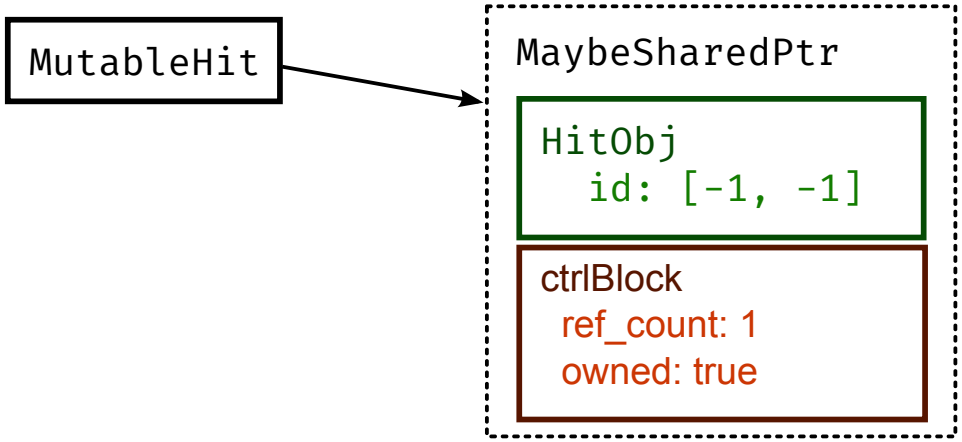
FREE STANDING FUNCTIONALITY

```
void foo(Hit hit);

MutableHit hit{};
foo(hit); // implicit conversion
```



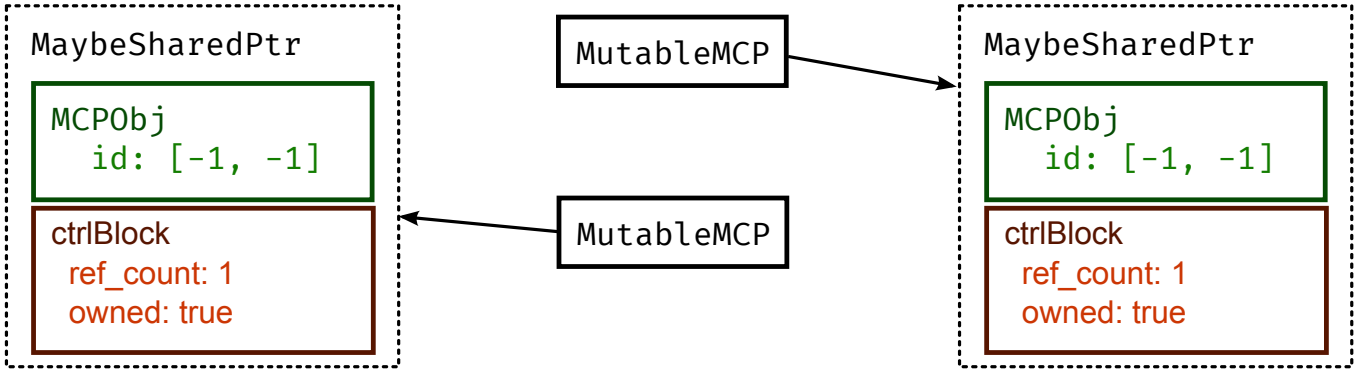
FREE STANDING FUNCTIONALITY

```
void foo(Hit hit);

MutableHit hit{};
foo(hit);
hit.setEnergy(3.14)
```



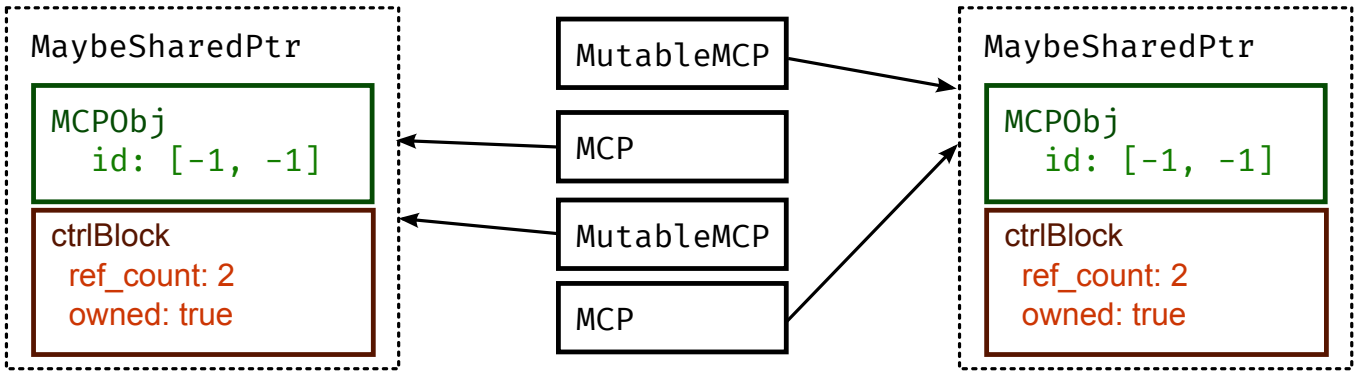
NEW ISSUE: CYCLIC REFERENCES

```
MutableMCP mc1{};
MutableMCP mc2{};
```



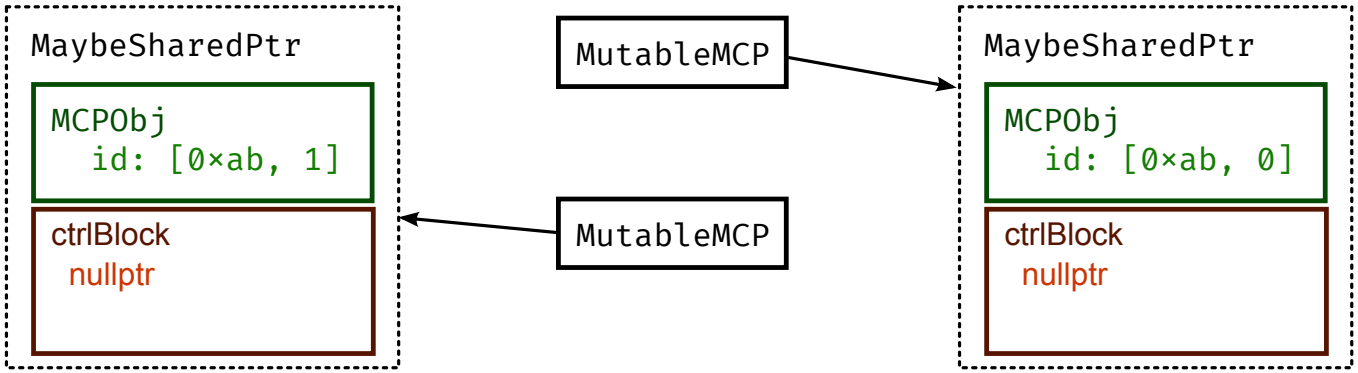
NEW ISSUE: CYCLIC REFERENCES

```
MutableMCP mc1{};
MutableMCP mc2{};

mc1.setParent(mc2); // <--- implicit conversion
mc2.setDaughter(mc1); // <--- implicit conversion
```



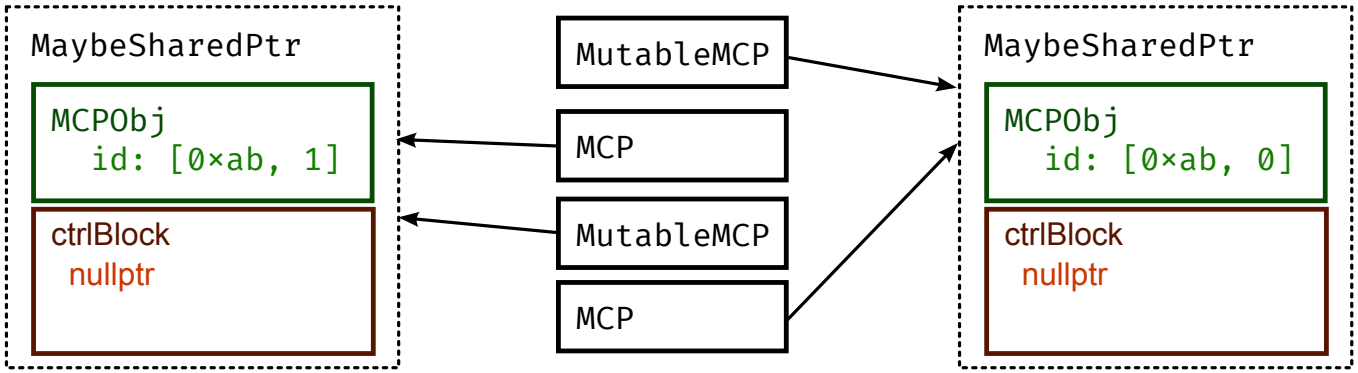
SOLUTION: COLLECTOINS AS FACTORIES

```
ExampleMCPCollection coll{};
auto mc1 = coll.create();
auto mc2 = coll.create();
```



SOLUTION: COLLECTOINS AS FACTORIES

```
ExampleMCPCollection coll{};
auto mc1 = coll.create();
auto mc2 = coll.create();

mc1.setParent(mc2); // <--- implicit conversion
mc2.setDaughter(mc1); // <--- implicit conversion
```



SOLUTION: COLLECTOIN AS FACTORIES

- Like all shared pointer types, cyclic references keep objects alive infinitely
- Can break the cycle in podio by not initializing the control block
  - Data managed by collection, no control block necessary
- **Remaining issue: dangling references**

```
Hit hit{};
{
  ExampleHitCollection coll{};
  hit = coll.create();
} // <--- Data of Hit destroyed here
hit.getEnergy(); // <--- use-after-free
```

THE REAL ISSUE

- podio promises *value semantics*
- What it delievers are effectively *reference semantics* that appear to be value semantics
  - Work as expected in almost all cases
  - Where they don’t meet expectations things get tricky quickly

POTENTIAL SOLUTIONS

- Manage expectations, i.e. document behavior to avoid surprises
- Make Mutable handles behave like values with move semantics
  - Very hard to abuse in multithreaded context
  - Collections can no longer serve as factories
- **Properly solving it breaks existing code!**

SUMMARY & NEXT STEPS

- Subtle lifetime issue in all podio generated EDMs
  - Hard to trigger, but **impossible to avoid currently**
- Fixing it requires a change in the podio memory model (podio#514)
  - Can be made transparent with `MaybeSharedPtr`
  - Keeps all subtle lifetime issues that currently exist
- Redesign API for a proper fix to make these issues impossible
  - Very breaking change. **After podio v1.0**