`std::unique_ptr`
Introduction

Who am I?

- 36 y/o CS-trained Open Source Software Developer
- Member of SDE group (since 2017) in CIT dep of GSI (since 2005)
- Projects: `Fair{`MQ`,`Root`,`Soft`}`, `asiofi`
- Interests:
  - Software Engineering Methods
  - System Programming (4+ years daily work experience with C++)
  - Distributed Computing
- Supporting Alice, CBM, Panda, and other FairRoot-based collaborations
- Contact: `d.klein@gsi.de`

Agenda

- Motivation
- Development of a naive unique pointer
  - Ingredients: RAII, pointer-like interfaces, move semantics
- Examples
- Take Aways

Goals:

- Introduce the unique_ptr concept, expose its simplicity and fundamental impact
- Build first intuition for the power of value semantics

Non-goals:

- Memory management in general
- Complete coverage of any of the above topics

Motivation

```
using MyData = std::array<ROOT::Math::XYZVector, 500>;
```

```
MyData data1;
```

Motivation

```
using MyData = std::array<ROOT::Math::XYZVector, 500>;
```

```
MyData data1;
```

- new object `data1` of type `MyData`
- placed on *stack* memory
- *value semantics* (assignment means copy)

Motivation

```cpp
using MyData = std::array<ROOT::Math::XYZVector, 500>;
```

```cpp
MyData* data2 = new MyData;
```

Motivation

```
using MyData = std::array<ROOT::Math::XYZVector, 500>;
```

- pointer `data2` to new object of type `MyData`
- stored on *heap* memory
- *pointer semantics* (assignment means reference)

```
MyData* data2 = new MyData;
```

Motivation

```
using MyData = std::array<ROOT::Math::XYZVector, 500>;

void workOn(const MyData&);



void f() {
  MyData data1;
  MyData* data2 = new MyData;
  workOn(data1); workOn(*data2);

}
```

Motivation

```
using MyData = std::array<ROOT::Math::XYZVector, 500>;

void workOn(const MyData&);



void f() {
  MyData data1;
  MyData* data2 = new MyData;
  workOn(data1); workOn(*data2);

} // data2 leaks!
```

Motivation

```cpp
using MyData = std::array<ROOT::Math::XYZVector, 500>;

void workOn(const MyData&);



void f() {
  MyData data1;
  MyData* data2 = new MyData;
  workOn(data1); workOn(*data2);
  delete data2; // Fixed!
}
```

Motivation

```cpp
using MyData = std::array<ROOT::Math::XYZVector, 500>;

void workOn(const MyData&);



void f() {
  MyData data1;
  MyData* data2 = new MyData;
  workOn(data1); workOn(*data2);
  delete data2; // Fixed! Sure?
}
```

Motivation

```cpp
using MyData = std::array<ROOT::Math::XYZVector, 500>;

void workOn(const MyData&) {
  throw std::runtime_error("smth went wrong");
}

void f() {
  MyData data1;
  MyData* data2 = new MyData;
  workOn(data1); workOn(*data2); // data2 still leaks!
  delete data2;
}
```

Motivation

```
usi
voi
  th
}
```

- **Leak-free** and **exception-safe** programming much easier with value semantics, because object lifetime is bound to the scope.
- How can we work as easy and safe with heap objects?

```
void f() {
  MyData data1;
  MyData* data2 = new MyData;
  workOn(data1); workOn(*data2); // data2 still leaks!
  delete data2;
}
```

Ingredient 1 of 4: RAII

```
class MyDataPtr {
 public:


 private:
  MyData* _p;
};
```

Resource Acquisition Is Initialization - https://en.cppreference.com/w/cpp/language/raii

Ingredient 1 of 4: RAII

```
class MyDataPtr {
 public:
  MyDataPtr() : _p(new MyData()) {}
  ~MyDataPtr() { delete _p; }
 private:
  MyData* _p;
};
```

Resource Acquisition Is Initialization - https://en.cppreference.com/w/cpp/language/raii

Ingredient 1 of 4: RAII

```
class MyDataPtr {
 public:
  MyDataPtr() : _p(new MyData()) {}
  ~MyDataPtr() { delete _p; }
 private:
  MyData* _p;
};

{
  MyDataPtr data;
}
```

Resource Acquisition Is Initialization - https://en.cppreference.com/w/cpp/language/raii

Ingredient 1 of 4: RAII

```
class MyDataPtr {
 public:
  MyDataPtr() : _p(new MyData()) {}
  ~MyDataPtr() { delete _p; }
 private:
  MyData* _p;
};

{
  MyDataPtr data;
}
```

✅ Lifetime of heap object bound to scope

Resource Acquisition Is Initialization - https://en.cppreference.com/w/cpp/language/raii

Ingredient 2 of 4: Data Accessors

```
class MyDataPtr {
 public:


 private:
  MyData* _p;
};

{
  MyDataPtr data;
  data->size();
  (*data)[42] = {3.,1.,4.};
}
```

Ingredient 2 of 4: Data Accessors

```
class MyDataPtr {
 public:
  MyData* operator->() const { return _p; }
  MyData& operator*() const { return *_p; }
 private:
  MyData* _p;
};

{
  MyDataPtr data;
  data->size();
  (*data)[42] = {3.,1.,4.};
}
```

member of pointer (or member access) operator

indirection (or dereference) operator

Member access operators - https://en.cppreference.com/w/cpp/language/operator_member_access

Ingredient 2 of 4: Data Accessors

```
class MyDataPtr {
 public:
  MyData* operator->() const { return _p; }
  MyData& operator*() const { return *_p; }
 private:
  MyData* _p;
};

{
  MyDataPtr data;
  data->size();
  (*data)[42] = {3.,1.,4.};
}
```

✅ Familiar data access interface!

Member access operators - https://en.cppreference.com/w/cpp/language/operator_member_access

Ingredient 3 of 4: Pointer Interface

```
class MyDataPtr {
 public:



 private:
  MyData* _p;
};

{
  MyDataPtr data;
  MyData* data2 = data.get();
  data.reset(nullptr);
  if (data) // ...
}
```

Ingredient 3 of 4: Pointer Interface

```
class MyDataPtr {
 public:
  MyData* get() const { return _p; }
  void reset(MyData* np) { delete _p; _p = np; }
  explicit operator bool() const { return _p != nullptr; }
 private:
  MyData* _p;
};

{
  MyDataPtr data;
  MyData* data2 = data.get();
  data.reset(nullptr);
  if (data) // ...
}
```

Ingredient 3 of 4: Pointer Interface

```
class MyDataPtr {
 public:
  MyData* get() const { return _p; }
  void reset(MyData* np) { delete _p; _p = np; }
  explicit operator bool() const { return _p != nullptr; }
 private:
  MyData* _p;
};

{
  MyDataPtr data;
  MyData* data2 = data.get();
  data.reset(nullptr);
  if (data) // ...
}
```

✅ Read/Write pointer address

Ingredient 4 of 4: Ownership (I)

```
class MyDataPtr {
 public:
  ~MyDataPtr() { delete _p; }
  MyData* get() const { return _p; }

 private:
  MyData* _p;
};

{
  MyDataPtr data;
  MyData* data2 = data.get(); delete data2;

}
```

Ingredient 4 of 4: Ownership (I)

```
class MyDataPtr {
 public:
  ~MyDataPtr() { delete _p; }
  MyData* get() const { return _p; }

 private:
  MyData* _p;
};

{
  MyDataPtr data;
  MyData* data2 = data.get(); delete data2;

} // double delete
```

Ingredient 4 of 4: Ownership (I)

```
class MyDataPtr {
 public:
  ~MyDataPtr() { delete _p; }
  MyData* get() const { return _p; }
  MyData* release() { MyData* r = _p; _p = nullptr; return r; }
 private:
  MyData* _p;
};

{
  MyDataPtr data;
  MyData* data2 = data.get(); delete data2;
  MyData* data3 = data.release(); delete data3;
} // OK
```

Returns **non-owning** raw pointer!

Returns **owning** raw pointer!

Ingredient 4 of 4: Ownership (I)

```
class MyDataPtr {
 public:
  ~MyDataPtr() { delete _p; }
  MyData* get() const { return _p; }
  MyData* release() { MyData* r = _p; _p = nullptr; return r; }
 private:
  MyData* _p;
};

{
  MyDataPtr data;
  MyData* data2 = data.get(); delete data2;
  MyData* data3 = data.release(); delete data3;
}
```

Returns **owning** raw pointer!

Modern convention (I.11):
Raw pointer **non-owning by default**!

Ingredient 4 of 4: Ownership (II)

```
class MyDataPtr {
 public:
  MyDataPtr(const MyDataPtr&) = default;
  MyDataPtr& operator=(const MyDataPtr&) = default;
 private:
  MyData* _p;
};

{
  MyDataPtr data;
  MyDataPtr data2 = data;
}
```

copy ctor

copy assignment operator

Ingredient 4 of 4: Ownership (II)

```
class MyDataPtr {
 public:
  MyDataPtr(const MyDataPtr&) = default;
  MyDataPtr& operator=(const MyDataPtr&) = default;
 private:
  MyData* _p;
};

{
  MyDataPtr data;
  MyDataPtr data2 = data;
} // double delete
```

Ingredient 4 of 4: Ownership (II)

```
class MyDataPtr {
 public:
  MyDataPtr(const MyDataPtr&) = default;
  MyDataPtr& operator=(const MyDataPtr&) = default;
 private:
  MyData* _p;
};

{
  MyDataPtr data;
  MyDataPtr data2 = data;
} // double delete
```

❌ Copy violates intended ownership semantics

We need ownership **transfer**! But how?

Crash Course: Move Semantics

```
class MyDataPtr {
 public:
  MyDataPtr() : _p(new MyData()) {}
  ~MyDataPtr() { delete _p; }









};

MyDataPtr data;
MyDataPtr data2 = data; // not what we want
```

Crash Course: Move Semantics

```
class MyDataPtr {
 public:
  MyDataPtr() : _p(new MyData()) {}
  ~MyDataPtr() { delete _p; }
  MyDataPtr(const MyDataPtr&) = delete;
  MyDataPtr& operator=(const MyDataPtr&) = delete;



};

MyDataPtr data;
MyDataPtr data2 = data; // does not compile!
```

(deleted) copy ctor

(deleted) copy assignment operator

Crash Course: Move Semantics

```
class MyDataPtr {
 public:
  MyDataPtr() : _p(new MyData()) {}
  ~MyDataPtr() { delete _p; }
  MyDataPtr(const MyDataPtr&) = delete;
  MyDataPtr& operator=(const MyDataPtr&) = delete;
  MyDataPtr(MyDataPtr&& rhs) : _p(rhs.release()) {}
  MyDataPtr& operator=(MyDataPtr&&)
    { reset(rhs.release()); return this; }
};

MyDataPtr data;
MyDataPtr data2 = data; // does not compile!
MyDataPtr data3 = std::move(data); // move construction!
// data.get() == nullptr !
```

move ctor

move assignment operator

Crash Course: Move Semantics

```
class MyDataPtr {
 public:
  MyDataPtr() : _p(new MyData()) {}
  ~MyDataPtr() { delete _p; }
  MyDataPtr(const MyDataPtr&) = delete;
  MyDataPtr& operator=(const MyDataPtr&) = delete;
  MyDataPtr(MyDataPtr&& rhs) : _p(rhs.release()) {}
  MyDataPtr& operator=(MyDataPtr&&)
    { reset(rhs.release()); return this; }
};

MyDataPtr data;
MyDataPtr data2 = data; // does not compile!
MyDataPtr data3 = std::move(data); // move construction!
// data.get() == nullptr !
```

rvalue reference

Crash Course: Move Semantics

```
class MyDataPtr {
 public:
  MyDataPtr() : _p(new MyData()) {}
  ~MyDataPtr() { delete _p; }
  MyDataPtr(const MyDataPtr&) = delete;
  MyDataPtr& operator=(const MyDataPtr&) = delete;
  MyDataPtr(MyDataPtr&& rhs) : _p(rhs.release()) {}
  MyDataPtr& operator=(MyDataPtr&&)
    { reset(rhs.release()); return this; }
};

MyDataPtr data;
MyDataPtr data2 = data; // does not compile!
MyDataPtr data3 = std::move(data); // move construction!
// data.get() == nullptr !
```

✅ **Move-only** semantics

Last Step: Generalize on the pointer type

```
class MyDataPtr {
  ...
 private:
  MyData* _p;
};
```

≅std::unique_ptr

```
template <class T>
class unique_ptr {
 ...
 private:
  T* _p;
};
```

https://en.cppreference.com/w/cpp/memory/unique_ptr

Examples (I)

```cpp
// create heap object, store owning pointer
std::unique_ptr<MyData> data{new MyData}; // or
std::unique_ptr<MyData> data = std::make_unique<MyData>(); // or
auto data = std::make_unique<MyData>();
```

https://en.cppreference.com/w/cpp/memory/unique_ptr/make_unique
https://en.cppreference.com/w/cpp/utility/move

Examples (I)

```
// create heap object, store owning pointer
std::unique_ptr<MyData> data{new MyData}; // or
std::unique_ptr<MyData> data = std::make_unique<MyData>(); // or
auto data = std::make_unique<MyData>();

// transfer ownership (explicit!)
std::unique_ptr<MyData> data2 = std::move(data);

void modern(std::unique_ptr<MyData>);
modern(std::move(data2));

void cstyle(MyData*);
cstyle(data2.release());
```

https://en.cppreference.com/w/cpp/memory/unique_ptr/make_unique
https://en.cppreference.com/w/cpp/utility/move

Examples (II)

```
// pass to function without ownership transfer
auto data = std::make_unique<MyData>();

void observe(const MyData&);
observe(*data);

void modify(MyData&);
modify(*data);

void cstyle(MyData*);
cstyle(data->get());
```

Examples (III)

```cpp
// return owning pointer
std::unique_ptr<MyData> make_data() {
  auto ret = std::make_unique<MyData>();
  return ret; // copy elision / return value optimization
}
std::unique_ptr<MyData> data = make_data();
```

https://stackoverflow.com/questions/12953127/what-are-copy-elision-and-return-value-optimization

Examples (III)

```
// return owning pointer (return by value)
std::unique_ptr<MyData> make_data() {
  auto ret = std::make_unique<MyData>();
  return ret; // copy elision / return value optimization
}
std::unique_ptr<MyData> data = make_data();

// legacy owning raw pointer interface / implicit cast to base pointer
// FairModule* cave = new FairCave("CAVE");
// cave->SetGeometryFileName("cave_vacuum.geo");
// run->AddModule(cave);
std::unique_ptr<FairModule> cave = std::make_unique<FairCave>("CAVE");
cave->SetGeometryFileName("cave_vacuum.geo");
run->AddModule(cave.release()); // transfers ownership!
```

Taken from [FairRoot/examples/simulation/Tutorial1/macros/run_tutorial1.C]

Take aways

- Prefer value semantics over pointer semantics
  - RAII almost universally applicable today thanks to move semantics (C++11), adopt it!
  - Add `std::unique_ptr` to your toolbox
  - Avoid explicit use of `new` and `delete` (ES.60)
- Modern convention: Raw pointers are non-owning by default! (I.11)
  - Use comments to explicitly mark owning raw pointers, if you have to use them

Further Material

- CppCon 2016: Herb Sutter “Leak-Freedom in C++... By Default.”
- CppCon 2019: Arthur O'Dwyer “Back to Basics: Smart Pointers”
- CppCon 2019: Matthew Fleming “The Smart Pointers I Wish I Had”
- CppCon 2020: Rainer Grimm “Back to Basics: Smart Pointers”