Back To Basics
Smart Pointers

DAVID OLSEN

Cppcon
The C++ Conference
2022
September 12th-16th

```
void science(double* data, int N) {
  double* temp = new double[N*2];
  do_setup(data, temp, N);
  if (not needed(data, temp, N))
    return;
  calculate(data, temp, N);
  delete[] temp;
}
```

```
void science(double* data, int N) {
  double* temp = new double[N*2];
  do_setup(data, temp, N);
  if (not needed(data, temp, N))
    return;
  calculate(data, temp, N);
  delete[] temp;
}
```

Early return skips delete

```
void science(double* x, int N) {
  double* y = new double[N];
  double* z = new double[N];
  calculate(x, y, z, N);
  delete[] z;
  delete[] y;
}
```

```
void science(double* x, int N) {
  double* y = new double[N];
  double* z = new double[N];
  calculate(x, y, z, N);
  delete[] z;
  delete[] y;
}
```

If second new throws,
y is leaked

```
float* science(float* x, float* y, int N) {
  float* z = new float[N];
  saxpy(2.5, x, y, z, N);
  delete[] x;
  delete[] y;
  return z;
}
```

```
float* science(float* x, float* y, int N) {
  float* z = new float[N];
  saxpy(2.5, x, y, z, N);
  delete[] x;
  delete[] y;
  return z;
}
```

Can x and y be deleted?

Caller is expected to delete[] z

RAW POINTER

Too many uses

Single object vs. array

Owning vs. non-owning

Nullable vs. non-nullable

RAW POINTER

Too many uses

Single object vs. array

- Single: allocate with `new`, free with `delete`
- Array: allocate with `new[]`, free with `delete[]`
- Single: don’t use `++p`, `--p`, or `p[n]`
- Array: `++p`, `--p`, and `p[n]` are fine

Raw Pointer

Too many uses

Single object vs. array

Owning vs. non-owning

- Owner must free the memory when done
- Non-owner must never free the memory

RAW POINTER

Too many uses

Single object vs. array

Owning vs. non-owning

Nullable vs. non-nullable

- Some pointers can never be null
- It would be nice if the type system helped enforce that

RAW POINTER

Too many uses

Single object vs. array

Owning vs. non-owning

Nullable vs. non-nullable

The type system doesn't help

- `T*` can be used for all combinations of those characteristics

NVIDIA

Back to Basics: Smart Pointers

David Olsen, CppCon, 13 Sep 2022

Smart Pointer

Behaves like a pointer

- … at least one of the roles of a pointer
- Points to an object
- Can be dereferenced

Adds additional “smarts”

- Often limits behavior to certain of a pointer’s possible roles

SMART POINTER

What is it good for?

"Smart" can be almost anything

- Automatically release resources is most common
- Enforce restrictions, e.g. don't allow nullptr
- Extra safety checks

Sometimes the smarts are only in the name

- gsl::owner<T> is just a typedef of T*; it only has meaning for those reading the code

Raw Pointer

What is it good for?

Non-owning pointer to a single object

Use a smart pointer for all owning pointers

Use a span type in place of non-owning pointers to arrays

C++20 `std::span`, or `gsl::span`

UNIQUE_PTR

UNIQUE_PTR

Overview

Owns memory

Assumes it is the only owner

Automatically destroys the object and deletes the memory

Move-only type

UNIQUE_PTR

Overview

Defined in header <memory>

One required template parameter, which is the pointed-to type

```
template <typename T>
struct unique_ptr {
  // ...
  using element_type = T;
  using pointer = T*;
  // ...
};
```

UNIQUE_PTR

Basic usage - function

```
void calculate_more(HelperType&);

ResultType do_work(InputType inputs) {
  std::unique_ptr<HelperType> owner{new HelperType(inputs)};
  owner->calculate();
  calculate_more(*owner);
  return owner->important_result();
}
```

UNIQUE_PTR

Basic usage - function

```
void calculate_more(HelperType&);

ResultType do_work(InputType inputs) {
  std::unique_ptr<HelperType> owner{new HelperType(inputs)};
  owner->calculate();
  calculate_more(*owner);
  return owner->important_result();
}
```

Create unique_ptr with newly allocated memory

UNIQUE_PTR

Basic usage - function

```
void calculate_more(HelperType&);

ResultType do_work(InputType inputs) {
  std::unique_ptr<HelperType> owner{new HelperType(inputs)};
  owner->calculate();
  calculate_more(*owner);
  return owner->important_result();
}
```

Dereference the unique_ptr

UNIQUE_PTR

Basic usage - function

```
void calculate_more(HelperType&);

ResultType do_work(InputType inputs) {
  std::unique_ptr<HelperType> owner{new HelperType(inputs)};
  owner->calculate();
  calculate_more(*owner);
  return owner->important_result();
}
```

Delete happens automatically

UNIQUE_PTR

Basic usage - class

```
WidgetBase* create_widget(InputType);

class MyClass {
  std::unique_ptr<WidgetBase> owner;
public:
  MyClass(InputType inputs)
    : owner(create_widget(inputs)) { }
  ~MyClass() = default;
  // ... member functions that use owner-> ...
};
```

UNIQUE_PTR

Basic usage - class

```
WidgetBase* create_widget(InputType);

class MyClass {
  std::unique_ptr<WidgetBase> owner;
public:
  MyClass(InputType inputs)
    : owner(create_widget(inputs)) { }
  ~MyClass() = default;
  // ... member functions that use owner-> ...
};
```

UNIQUE_PTR

Basic usage - class

```
WidgetBase* create_widget(InputType);

class MyClass {
  std::unique_ptr<WidgetBase> owner;
public:
  MyClass(InputType inputs)
    : owner(create_widget(inputs)) { }
  ~MyClass() = default;
  // ... member functions that use owner-> ...
};
```

UNIQUE_PTR

Basic usage - class

```
WidgetBase* create_widget(InputType);

class MyClass {
  std::unique_ptr<WidgetBase> owner;
public:
  MyClass(InputType inputs)
    : owner(create_widget(inputs)) { }
  ~MyClass() = default;
  // ... member functions that use owner-> ...
};
```

UNIQUE_PTR

Basic usage - class

```
WidgetBase* create_widget(InputType);

class MyClass {
  std::unique_ptr<WidgetBase> owner;
public:
  MyClass(InputType inputs)
    : owner(create_widget(inputs)) { }
  ~MyClass() = default;
  // ... member functions that use owner-> ...
};
```

UNIQUE_PTR

RAII

Very useful for implementing RAII

See “Back to Basics: RAII” by Andre Kostur, Tue 16:45-17:45, in Summit 2 & 3

UNIQUE_PTR

Move-only

Move only type

- No copy constructor or copy assignment operator

Unique ownership can't be copied

"Back to Basics: Move Semantics", David Olsen, CppCon 2020

UNIQUE_PTR

Sample implementation

```
template <typename T>
class unique_ptr {
  T* ptr;
public:
  unique_ptr() noexcept : ptr(nullptr) { }
  explicit unique_ptr(T* p) noexcept : ptr(p) { }
  ~unique_ptr() noexcept { delete ptr; }
  // ...
};
```

UNIQUE_PTR

Sample implementation

```
template <typename T>
class unique_ptr {
  T* ptr;
public:
  unique_ptr() noexcept : ptr(nullptr) { }
  explicit unique_ptr(T* p) noexcept : ptr(p) { }
  ~unique_ptr() noexcept { delete ptr; }
  // ...
};
```

UNIQUE_PTR

Sample implementation

```
template <typename T>
class unique_ptr {
  T* ptr;
public:
  unique_ptr() noexcept : ptr(nullptr) { }
  explicit unique_ptr(T* p) noexcept : ptr(p) { }
  ~unique_ptr() noexcept { delete ptr; }
  // ...
};
```

Points to the owned object

UNIQUE_PTR

Sample implementation

```
template <typename T>
class unique_ptr {
  T* ptr;
public:
  unique_ptr() noexcept : ptr(nullptr) { }
  explicit unique_ptr(T* p) noexcept : ptr(p) { }
  ~unique_ptr() noexcept { delete ptr; }
  // ...
};
```

UNIQUE_PTR

Sample implementation

```
template <typename T>
class unique_ptr {
  T* ptr;
public:
  unique_ptr() noexcept : ptr(nullptr) { }
  explicit unique_ptr(T* p) noexcept : ptr(p) { }
  ~unique_ptr() noexcept { delete ptr; }
  // ...
};
```

UNIQUE_PTR

Sample implementation

```
template <typename T>
class unique_ptr {
  T* ptr;
public:
  unique_ptr() noexcept : ptr(nullptr) { }
  explicit unique_ptr(T* p) noexcept : ptr(p) { }
  ~unique_ptr() noexcept { delete ptr; }
  // ...
};
```

UNIQUE_PTR

Sample implementation

```
template <typename T> struct unique_ptr {
  // ...
  unique_ptr(unique_ptr const&) = delete;
  unique_ptr(unique_ptr&& o) noexcept
    : ptr(std::exchange(o.ptr, nullptr)) { }
  unique_ptr& operator=(unique_ptr const&) = delete;
  unique_ptr& operator=(unique_ptr&& o) noexcept {
    delete ptr;
    ptr = o.ptr;
    o.ptr = nullptr;
    return *this;
  }
  // ...
};
```

UNIQUE_PTR

Sample implementation

```
template <typename T> struct unique_ptr {
  // ...
  unique_ptr(unique_ptr const&) = delete;
  unique_ptr(unique_ptr&& o) noexcept
    : ptr(std::exchange(o.ptr, nullptr)) { }
  unique_ptr& operator=(unique_ptr const&) = delete;
  unique_ptr& operator=(unique_ptr&& o) noexcept {
    delete ptr;
    ptr = o.ptr;
    o.ptr = nullptr;
    return *this;
  }
  // ...
};
```

Not copyable

UNIQUE_PTR

Sample implementation

```
template <typename T> struct unique_ptr {
  // ...
  unique_ptr(unique_ptr const&) = delete;
  unique_ptr(unique_ptr&& o) noexcept
    : ptr(std::exchange(o.ptr, nullptr)) { }
  unique_ptr& operator=(unique_ptr const&) = delete;
  unique_ptr& operator=(unique_ptr&& o) noexcept {
    delete ptr;
    ptr = o.ptr;
    o.ptr = nullptr;
    return *this;
  }
  // ...
};
```

Move constructor transfers ownership

UNIQUE_PTR

Sample implementation

```
template <typename T> struct unique_ptr {
  // ...
  unique_ptr(unique_ptr const&) = delete;
  unique_ptr(unique_ptr&& o) noexcept
    : ptr(std::exchange(o.ptr, nullptr)) { }
  unique_ptr& operator=(unique_ptr const&) = delete;
  unique_ptr& operator=(unique_ptr&& o) noexcept {
    delete ptr;
    ptr = o.ptr;
    o.ptr = nullptr;
    return *this;
  }
  // ...
};
```

Frees memory

UNIQUE_PTR

Sample implementation

```
template <typename T> struct unique_ptr {
  // ...
  unique_ptr(unique_ptr const&) = delete;
  unique_ptr(unique_ptr&& o) noexcept
    : ptr(std::exchange(o.ptr, nullptr)) { }
  unique_ptr& operator=(unique_ptr const&) = delete;
  unique_ptr& operator=(unique_ptr&& o) noexcept {
    delete ptr;
    ptr = o.ptr;
    o.ptr = nullptr;
    return *this;
  }
  // ...
};
```

Transfers ownership

UNIQUE_PTR

Sample implementation

```
template <typename T>
struct unique_ptr {
  // ...
  T& operator*() const noexcept {
    return *ptr;
  }
  T* operator->() const noexcept {
    return ptr;
  }
  // ...
};
```

UNIQUE_PTR

Sample implementation

```
template <typename T> struct unique_ptr {
  T* release() noexcept {
    T* old = ptr;
    ptr = nullptr;
    return old;
  }
  void reset(T* p = nullptr) noexcept {
    delete ptr;
    ptr = p;
  }
  T* get() const noexcept {
    return ptr;
  }
  explicit operator bool() const noexcept {
    return ptr != nullptr;
  }
};
```

UNIQUE_PTR

Sample implementation

```
template <typename T> struct unique_ptr {
  T* release() noexcept {
    T* old = ptr;
    ptr = nullptr;
    return old;
  }
  void reset(T* p = nullptr) noexcept {
    delete ptr;
    ptr = p;
  }
  T* get() const noexcept {
    return ptr;
  }
  explicit operator bool() const noexcept {
    return ptr != nullptr;
  }
};
```

Gives up ownership

UNIQUE_PTR

Sample implementation

```
template <typename T> struct unique_ptr {
  T* release() noexcept {
    T* old = ptr;
    ptr = nullptr;
    return old;
  }
  void reset(T* p = nullptr) noexcept {
    delete ptr;
    ptr = p;
  }
  T* get() const noexcept {
    return ptr;
  }
  explicit operator bool() const noexcept {
    return ptr != nullptr;
  }
};
```

Cleans up
Takes ownership

UNIQUE_PTR

Sample implementation

```
template <typename T> struct unique_ptr {
  T* release() noexcept {
    T* old = ptr;
    ptr = nullptr;
    return old;
  }
  void reset(T* p = nullptr) noexcept {
    delete ptr;
    ptr = p;
  }
  T* get() const noexcept {
    return ptr;
  }
  explicit operator bool() const noexcept {
    return ptr != nullptr;
  }
};
```

UNIQUE_PTR

Sample implementation

```
template <typename T> struct unique_ptr {
  T* release() noexcept {
    T* old = ptr;
    ptr = nullptr;
    return old;
  }
  void reset(T* p = nullptr) noexcept {
    delete ptr;
    ptr = p;
  }
  T* get() const noexcept {
    return ptr;
  }
  explicit operator bool() const noexcept {
    return ptr != nullptr;
  }
};
```

Test for non-empty

MAKE_UNIQUE

```
template <typename T, typename... Args>
unique_ptr<T> make_unique(Args&&... args);
```

Combines together:

- Allocates memory
- Constructs a `T` with the given arguments
- Wraps it in a `std::unique_ptr<T>`

Prefer using make_unique to creating a unique_ptr explicitly

MAKE_UNIQUE

```
template <typename T, typename... Args>
unique_ptr<T> make_unique(Args&&... args);
```

Can't be deduced
Must be explicit

Combines together:

- Allocates memory
- Constructs a `T` with the given arguments
- Wraps it in a `std::unique_ptr<T>`

Prefer using make_unique to creating a unique_ptr explicitly

MAKE_UNIQUE

Example

```
std::unique_ptr<HelperType> owner{new HelperType(inputs)};
```

is better written as

```
auto owner = std::make_unique<HelperType>(inputs);
```

MAKE_UNIQUE

Non-example

```
std::unique_ptr<WidgetBase> owner;
MyClass(InputType inputs)
  : owner(create_widget(inputs)) { }
```

`make_unique` doesn’t help here

because allocation/construction happens within `create_widget`

UNIQUE_PTR

Array types

unique_ptr is specialized for array types

- Calls delete[] instead of delete
- Provides operator[]

make_unique is specialized for array types

- Argument is number of elements, not constructor arguments

UNIQUE_PTR

Array types

Fixing the first example from the beginning of the talk:

```
void science(double* data, int N) {
  auto temp = std::make_unique<double[]>(N*2);
  do_setup(data, temp.get(), N);
  if (not needed(data, temp.get(), N))
    return;
  calculate(data, temp.get(), N);
}
```

UNIQUE_PTR

Array types

Fixing the first example from the beginning of the talk:

```
void science(double* data, int N) {
  auto temp = std::make_unique<double[]>(N*2);
  do_setup(data, temp.get(), N);
  if (not needed(data, temp.get(), N))
    return;
  calculate(data, temp.get(), N);
}
```

UNIQUE_PTR

Array types

Fixing the first example from the beginning of the talk:

```
void science(double* data, int N) {
  auto temp = std::make_unique<double[]>(N*2);
  do_setup(data, temp.get(), N);
  if (not needed(data, temp.get(), N))
    return;
  calculate(data, temp.get(), N);
}
```

`std::unique_ptr<double[]>`

UNIQUE_PTR

Array types

Fixing the first example from the beginning of the talk:

```
void science(double* data, int N) {
  auto temp = std::make_unique<double[]>(N*2);
  do_setup(data, temp.get(), N);
  if (not needed(data, temp.get(), N))
    return;
  calculate(data, temp.get(), N);
}
```

unique_ptr destructor calls `delete[]`

UNIQUE_PTR

Transfer ownership

Use move constructor/assignment to transfer ownership

```
auto a = std::make_unique<T>();
// ...
std::unique_ptr<T> b{ a.release() };
// ...
a.reset(b.release());
```

Don't do that!

UNIQUE_PTR

Transfer ownership

Use move constructor/assignment to transfer ownership

```
auto a = std::make_unique<T>();
// ...
std::unique_ptr<T> b{ std::move(a) };
// ...
a = std::move(b);
```

Let unique_ptr handle the details

UNIQUE_PTR

Transfer ownership

To transfer ownership to a function, pass std::unique_ptr by value

To return ownership from a function, return std::unique_ptr by value

UNIQUE_PTR

Transfer ownership

To transfer ownership to a function, pass std::unique_ptr by value

To return ownership from a function, return std::unique_ptr by value

```
float* science(
        float* x,
        float* y, int N) {
  float* z = new float[N];
  saxpy(2.5, x, y, z, N);
  delete[] x;
  delete[] y;
  return z;
}
```

UNIQUE_PTR

Transfer ownership

To transfer ownership to a function, pass std::unique_ptr by value

To return ownership from a function, return std::unique_ptr by value

```
std::unique_ptr<float[]> science(
        std::unique_ptr<float[]> x,
        std::unique_ptr<float[]> y, int N) {
  auto z = std::make_unique<float[]>(N);
  saxpy(2.5, x.get(), y.get(), z.get(), N);


  return z;
}
```

UNIQUE_PTR

Transfer ownership

To transfer ownership to a function, pass std::unique_ptr by value

To return ownership from a function, return std::unique_ptr by value

```
std::unique_ptr<float[]> science(
        std::unique_ptr<float[]> x,
        std::unique_ptr<float[]> y, int N) {
  auto z = std::make_unique<float[]>(N);
  saxpy(2.5, x.get(), y.get(), z.get(), N);


  return z;
}
```

Arguments are now unique_ptr

UNIQUE_PTR

Transfer ownership

To transfer ownership to a function, pass std::unique_ptr by value

To return ownership from a function, return std::unique_ptr by value

```
std::unique_ptr<float[]> science(
        std::unique_ptr<float[]> x,
        std::unique_ptr<float[]> y, int N) {
  auto z = std::make_unique<float[]>(N);
  saxpy(2.5, x.get(), y.get(), z.get(), N);


  return z;
}
```

Return type is also unique_ptr

UNIQUE_PTR

Transfer ownership

To transfer ownership to a function, pass std::unique_ptr by value

To return ownership from a function, return std::unique_ptr by value

```
std::unique_ptr<float[]> science(
        std::unique_ptr<float[]> x,
        std::unique_ptr<float[]> y, int N) {
  auto z = std::make_unique<float[]>(N);
  saxpy(2.5, x.get(), y.get(), z.get(), N);


  return z;
}
```

No need to delete
unique_ptr does that

UNIQUE_PTR

Transfer ownership

```
WidgetBase* create_widget(InputType);
```

better communicates its intent if changed to

```
std::unique_ptr<WidgetBase> create_widget(InputType);
```

UNIQUE_PTR

Gotchas

Make sure only one unique_ptr for a block of memory

```
T* p = ...;
std::unique_ptr<T> a{p};
std::unique_ptr<T> b{p};
// crash due to double free
```

```
auto c = std::make_unique<T>();
std::unique_ptr<T> d{c.get()};
// crash due to double free
```

Don’t create a unique_ptr from a pointer unless you know where the pointer came from and that it needs an owner

UNIQUE_PTR

Gotchas

unique_ptr doesn't solve the dangling pointer problem

```
T* p = nullptr;
{
  auto u = std::make_unique<T>();
  p = u.get();
}
// p is now dangling and invalid
auto bad = *p; // undefined behavior
```

UNIQUE_PTR

Collection

`std::vector<std::unique_ptr<T>>` just works

```
{
  std::vector<std::unique_ptr<T>> v;
  v.push_back(std::make_unique<T>());
  std::unique_ptr<T> a;
  v.push_back(std::move(a));
  v[0] = std::make_unique<T>();
  auto it = v.begin();
  v.erase(it);
}
```

SHARED_PTR

SHARED_PTR

Overview

Owns memory

Shared ownership

- Many `std::shared_ptr` objects work together to manage one object

Automatically destroys the object and deletes the memory

Copyable

SHARED_PTR

Overview

Defined in header <memory>

One required template parameter, which is the pointed-to type

```
template <typename T>
struct shared_ptr {
  // ...
  using element_type = T;
  // ...
};
```

Shared Ownership

Ownership is shared equally

- No way to force a shared_ptr to give up its ownership

Cleanup happens when the last shared_ptr gives up ownership

Shared Ownership

Examples

Real world

- Community garden
- Open source project
- Shared responsibility for maintenance
- They survive as long as one person is willing to do the work

Shared Ownership

Examples

In code

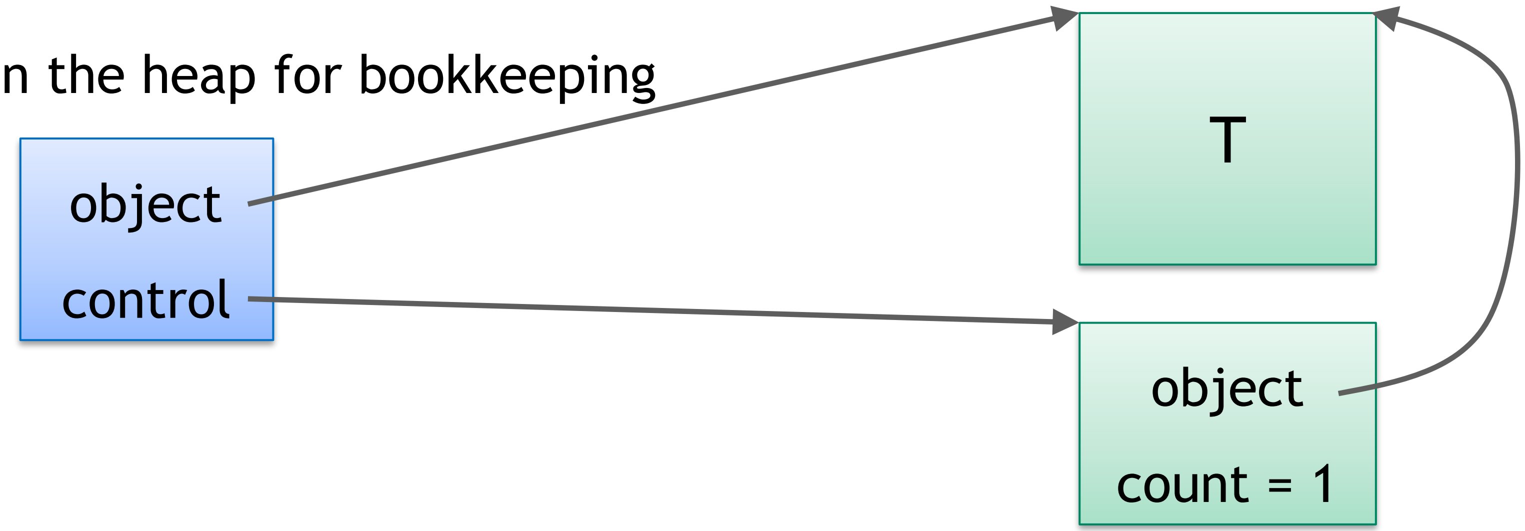
- UI widgets
- Promise/future
- Often implemented with reference counting or garbage collection

SHARED_PTR

Reference counting

Shared ownership implemented with reference counting

Control block on the heap for bookkeeping

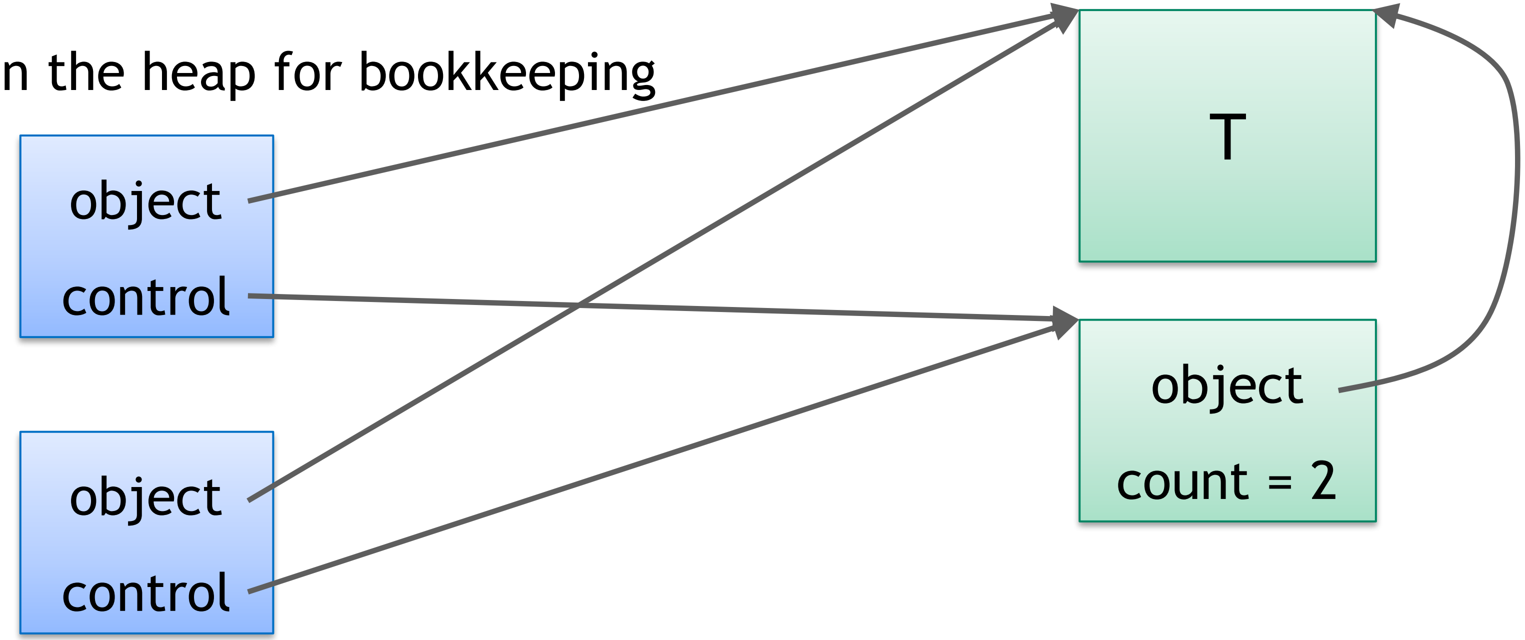


SHARED_PTR

Reference counting

Shared ownership implemented with reference counting

Control block on the heap for bookkeeping



SHARED_PTR

API

```
template <typename T>
struct shared_ptr {
  // ...
  shared_ptr() noexcept;
  explicit shared_ptr(T*);
  ~shared_ptr() noexcept;
  // ...
};
```

SHARED_PTR

API

```
template <typename T>
struct shared_ptr {
  // ...
  shared_ptr() noexcept;
  explicit shared_ptr(T*);
  ~shared_ptr() noexcept;
  // ...
};
```

Creates empty shared_ptr

SHARED_PTR

API

```
template <typename T>
struct shared_ptr {
  // ...
  shared_ptr() noexcept;
  explicit shared_ptr(T*);
  ~shared_ptr() noexcept;
  // ...
};
```

Starts managing an object

NVIDIA

SHARED_PTR

API

```
template <typename T>
struct shared_ptr {
  // ...
  shared_ptr() noexcept;
  explicit shared_ptr(T*);
  ~shared_ptr() noexcept;
  // ...
};
```

Decrements count
Cleanup if count == 0

SHARED_PTR

API

```
template <typename T>
struct shared_ptr {
  // ...
  shared_ptr(shared_ptr const&) noexcept;
  shared_ptr(shared_ptr&&) noexcept;
  shared_ptr(unique_ptr<T>&&);

  shared_ptr& operator=(shared_ptr const&) noexcept;
  shared_ptr& operator=(shared_ptr&&) noexcept;
  shared_ptr& operator=(unique_ptr<T>&&);
  // ...
};
```

SHARED_PTR

API

Copies object and control block pointers
Increments count

```
template <typename T>
struct shared_ptr {
  // ...
  shared_ptr(shared_ptr const&) noexcept;
  shared_ptr(shared_ptr&&) noexcept;
  shared_ptr(unique_ptr<T>&&);

  shared_ptr& operator=(shared_ptr const&) noexcept;
  shared_ptr& operator=(shared_ptr&&) noexcept;
  shared_ptr& operator=(unique_ptr<T>&&);
  // ...
};
```

SHARED_PTR

API

```
template <typename T>
struct shared_ptr {
  // ...
  shared_ptr(shared_ptr const&) noexcept;
  shared_ptr(shared_ptr&&) noexcept;
  shared_ptr(unique_ptr<T>&&);

  shared_ptr& operator=(shared_ptr const&) noexcept;
  shared_ptr& operator=(shared_ptr&&) noexcept;
  shared_ptr& operator=(unique_ptr<T>&&);
  // ...
};
```

Transfers ownership

SHARED_PTR

API

```
template <typename T>
struct shared_ptr {
  // ...
  shared_ptr(shared_ptr const&) noexcept;
  shared_ptr(shared_ptr&&) noexcept;
  shared_ptr(unique_ptr<T>&&);

  shared_ptr& operator=(shared_ptr const&) noexcept;
  shared_ptr& operator=(shared_ptr&&) noexcept;
  shared_ptr& operator=(unique_ptr<T>&&);
  // ...
};
```

Transfers ownership

SHARED_PTR

API

```
template <typename T>
struct shared_ptr {
  // ...
  shared_ptr(shared_ptr const&) noexcept;
  shared_ptr(shared_ptr&&) noexcept;
  shared_ptr(unique_ptr<T>&&);

  shared_ptr& operator=(shared_ptr const&) noexcept;
  shared_ptr& operator=(shared_ptr&&) noexcept;
  shared_ptr& operator=(unique_ptr<T>&&);
  // ...
};
```

SHARED_PTR

API

```
template <typename T>
struct shared_ptr {
  // ...
  T& operator*() const noexcept;
  T* operator->() const noexcept;
  // ...
};
```

SHARED_PTR

API

```
template <typename T>
struct shared_ptr {
  // ...
  void reset(T*);
  T* get() const noexcept;
  long use_count() const noexcept;
  explicit operator bool() const noexcept;
};
```

SHARED_PTR

API

```
template <typename T>
struct shared_ptr {
  // ...
  void reset(T*);
  T* get() const noexcept;
  long use_count() const noexcept;
  explicit operator bool() const noexcept;
};
```

SHARED_PTR

API

```
template <typename T>
struct shared_ptr {
  // ...
  void reset(T*);
  T* get() const noexcept;
  long use_count() const noexcept;
  explicit operator bool() const noexcept;
};
```

SHARED_PTR

API

```
template <typename T>
struct shared_ptr {
  // ...
  void reset(T*);
  T* get() const noexcept;
  long use_count() const noexcept;
  explicit operator bool() const noexcept;
};
```

SHARED_PTR

API

```
template <typename T>
struct shared_ptr {
  // ...
  void reset(T*);
  T* get() const noexcept;
  long use_count() const noexcept;
  explicit operator bool() const noexcept;
};
```

MAKE_SHARED

```
template <typename T, typename... Args>
shared_ptr<T> make_shared(Args&&... args);
```

Combines together:

- One memory allocation for both the object and the control block
- Constructs a T with the given arguments
- Initializes the control block
- Wraps them in a `std::shared_ptr<T>` object

Prefer using make_shared to creating a shared_ptr directly

SHARED_PTR

Shared ownership

To share ownership, additional shared_ptr objects must be created or assigned from an existing shared_ptr, not from the raw pointer

```
{
  T* p = ...;
  std::shared_ptr<T> a(p);
  std::shared_ptr<T> b(p);
} // runtime error: double free
```

SHARED_PTR

Shared ownership

To share ownership, additional shared_ptr objects must be created or assigned from an existing shared_ptr, not from the raw pointer

```
{
  auto a = std::make_shared<T>();
  std::shared_ptr<T> b(a.get());
} // runtime error: double free
```

SHARED_PTR

Shared ownership

To share ownership, additional shared_ptr objects must be created or assigned from an existing shared_ptr, not from the raw pointer

```
{
  auto a = std::make_shared<T>();
  std::shared_ptr<T> b(a);
  std::shared_ptr<T> c;
  c = b;
}
```

SHARED_PTR

Thread safety

Updating the same control block from different threads is thread safe

```
auto a = std::make_shared<int>(42);
std::thread t([](std::shared_ptr<int> b) {
  std::shared_ptr<int> c = b;
  work(*c);
}, a);
{
  std::shared_ptr<int> d = a;
  a.reset((int*)nullptr);
  more_work(*d);
}
t.join();
```

SHARED_PTR

Thread safety

Updating the same control block from different threads is thread safe

```
auto a = std::make_shared<int>(42);
std::thread t([](std::shared_ptr<int> b) {
  std::shared_ptr<int> c = b;
  work(*c);
}, a);
{
  std::shared_ptr<int> d = a;
  a.reset((int*)nullptr);
  more_work(*d);
}
t.join();
```

Increment count

Decrement count

SHARED_PTR

Thread safety

Updating the same control block from different threads is thread safe

```
auto a = std::make_shared<int>(42);
std::thread t([](std::shared_ptr<int> b) {
  std::shared_ptr<int> c = b;
  work(*c);
}, a);
{
  std::shared_ptr<int> d = a;
  a.reset((int*)nullptr);
  more_work(*d);
}
t.join();
```

Read object

SHARED_PTR

Thread safety

Updating the managed object from different threads is not thread safe

```
auto a = std::make_shared<int>(42);
std::thread t([](std::shared_ptr<int> b) {
  std::shared_ptr<int> c = b;
  *c = 100;
}, a);
{
  std::shared_ptr<int> d = a;
  a.reset((int*)nullptr);
  *d = 200;
}
t.join();
```

SHARED_PTR

Thread safety

Updating the managed object from different threads is not thread safe

```
auto a = std::make_shared<int>(42);
std::thread t([](std::shared_ptr<int> b) {
  std::shared_ptr<int> c = b;
  *c = 100;
}, a);
{
  std::shared_ptr<int> d = a;
  a.reset((int*)nullptr);
  *d = 200;
}
t.join();
```

Write to object
Data race!

SHARED_PTR

Thread safety

Updating the same shared_ptr object from different threads is not thread safe

```
auto a = std::make_shared<int>(42);
std::thread t([&]() {
  work(*a);
});
a = std::make_shared<int>(100);
t.join();
```

SHARED_PTR

Thread safety

Updating the same shared_ptr object from different threads is not thread safe

```
auto a = std::make_shared<int>(42);
std::thread t([&]() {
  work(*a);
});
a = std::make_shared<int>(100);
t.join();
```

Capture ‘a’ by reference

SHARED_PTR

Thread safety

Updating the same shared_ptr object from different threads is not thread safe

```
auto a = std::make_shared<int>(42);
std::thread t([&]() {
  work(*a);
});
a = std::make_shared<int>(100);
t.join();
```

Read and write
Data race!

SHARED_PTR

Arrays

shared_ptr added support for array types in C++17

make_shared added support for array types in C++20

Use array types with shared_ptr with caution

- Make sure your standard library is new enough

UNIQUE_PTR VS SHARED_PTR

Single owner: use unique_ptr

Multiple owners: use shared_ptr

Non-owning reference: use something else entirely

When in doubt, prefer unique_ptr

- Easier to switch from unique_ptr to shared_ptr than the other way around

Advanced Stuff

WEAK_PTR

WEAK_PTR

Overview

A non-owning reference to a shared_ptr-managed object

Knows when the lifetime of the managed object ends

```
std::weak_ptr<int> w;
{
  auto s = std::make_shared<int>(42);
  w = s;
  std::shared_ptr<int> t = w.lock();
  if (t) printf("%d\n", *t);
}
std::shared_ptr<int> u = w.lock();
if (!u) printf("empty\n");
```

WEAK_PTR

Overview

A non-owning reference to a shared_ptr-managed object

Knows when the lifetime of the managed object ends

```
std::weak_ptr<int> w;
{
  auto s = std::make_shared<int>(42);
  w = s;
  std::shared_ptr<int> t = w.lock();
  if (t) printf("%d\n", *t);
}
std::shared_ptr<int> u = w.lock();
if (!u) printf("empty\n");
```

WEAK_PTR

Overview

A non-owning reference to a shared_ptr-managed object

Knows when the lifetime of the managed object ends

```
std::weak_ptr<int> w;
{
  auto s = std::make_shared<int>(42);
  w = s;
  std::shared_ptr<int> t = w.lock();
  if (t) printf("%d\n", *t);
}
std::shared_ptr<int> u = w.lock();
if (!u) printf("empty\n");
```

WEAK_PTR

Overview

A non-owning reference to a shared_ptr-managed object

Knows when the lifetime of the managed object ends

```
std::weak_ptr<int> w;
{
  auto s = std::make_shared<int>(42);
  w = s;
  std::shared_ptr<int> t = w.lock();
  if (t) printf("%d\n", *t);
}
std::shared_ptr<int> u = w.lock();
if (!u) printf("empty\n");
```

WEAK_PTR

Overview

A non-owning reference to a shared_ptr-managed object

Knows when the lifetime of the managed object ends

```
std::weak_ptr<int> w;
{
  auto s = std::make_shared<int>(42);
  w = s;
  std::shared_ptr<int> t = w.lock();
  if (t) printf("%d\n", *t);
}
std::shared_ptr<int> u = w.lock();
if (!u) printf("empty\n");
```

WEAK_PTR

Overview

A non-owning reference to a shared_ptr-managed object

Knows when the lifetime of the managed object ends

```
std::weak_ptr<int> w;
{
  auto s = std::make_shared<int>(42);
  w = s;
  std::shared_ptr<int> t = w.lock();
  if (t) printf("%d\n", *t);
}
std::shared_ptr<int> u = w.lock();
if (!u) printf("empty\n");
```

WEAK_PTR

Overview

A non-owning reference to a shared_ptr-managed object

Knows when the lifetime of the managed object ends

```
std::weak_ptr<int> w;
{
  auto s = std::make_shared<int>(42);
  w = s;
  std::shared_ptr<int> t = w.lock();
  if (t) printf("%d\n", *t);
}
std::shared_ptr<int> u = w.lock();
if (!u) printf("empty\n");
```

WEAK_PTR

What is it good for?

Only useful when object is managed by shared_ptr

Caching

- Keep a reference to an object for faster access
- Don’t want that reference to keep the object alive

Dangling references

Custom Deleters

Custom Deleters

What if cleanup action is something other than calling `delete` ?

```
FILE* fp = fopen("readme.txt", "r");
fread(buffer, 1, N, fp);
fclose(fp);
```

Custom Deleters

What if cleanup action is something other than calling `delete` ?

```
FILE* fp = fopen("readme.txt", "r");
fread(buffer, 1, N, fp);
fclose(fp);
```

Might be forgotten or skipped

Custom Deleters

unique_ptr

unique_ptr has an extra defaulted template parameter for the delete

```
template <typename T,
          typename Deleter = std::default_delete<T>>
class unique_ptr;
```

Type `Deleter` must have an `operator()(T*)`

make_unique doesn't support custom deleters

- unique_ptr with custom deleter must be constructed directly

Custom Deleters

unique_ptr

```
struct fclose_deleter {
  void operator()(FILE* fp) const { fclose(fp); }
};
using unique_FILE = std::unique_ptr<FILE, fclose_deleter>;

{
  unique_FILE fp(fopen("readme.txt", "r"));
  fread(buffer, 1, N, fp.get());
}
```

CUSTOM DELETERS

unique_ptr

```
struct fclose_deleter {
  void operator()(FILE* fp) const { fclose(fp); }
};
using unique_FILE = std::unique_ptr<FILE, fclose_deleter>;

{
  unique_FILE fp(fopen("readme.txt", "r"));
  fread(buffer, 1, N, fp.get());
}
```

Custom Deleters

unique_ptr

```
struct fclose_deleter {
  void operator()(FILE* fp) const { fclose(fp); }
};
using unique_FILE = std::unique_ptr<FILE, fclose_deleter>;

{
  unique_FILE fp(fopen("readme.txt", "r"));
  fread(buffer, 1, N, fp.get());
}
```

Custom Deleters

unique_ptr

```
struct fclose_deleter {
  void operator()(FILE* fp) const { fclose(fp); }
};
using unique_FILE = std::unique_ptr<FILE, fclose_deleter>;

{
  unique_FILE fp(fopen("readme.txt", "r"));
  fread(buffer, 1, N, fp.get());
}
```

CUSTOM DELETERS

unique_ptr

```
struct fclose_deleter {
  void operator()(FILE* fp) const { fclose(fp); }
};
using unique_FILE = std::unique_ptr<FILE, fclose_deleter>;

{
  unique_FILE fp(fopen("readme.txt", "r"));
  fread(buffer, 1, N, fp.get());
}
```

fclose called automatically

CUSTOM DELETERS

shared_ptr

Custom deleter for shared_ptr is passed to constructor, where it is type erased

```
struct fclose_deleter {
  void operator()(FILE* fp) const { fclose(fp); }
};

{
  std::shared_ptr<FILE> fp(fopen("readme.txt", "r"),
                              fclose_deleter{});
  fread(buffer, 1, N, fp.get());
  std::shared_ptr<FILE> fp2(fp);
}
```

Custom Deleters

shared_ptr

Custom deleter for shared_ptr is passed to constructor, where it is type erased

```
struct fclose_deleter {
  void operator()(FILE* fp) const { fclose(fp); }
};

{
  std::shared_ptr<FILE> fp(fopen("readme.txt", "r"),
                           fclose_deleter{});
  fread(buffer, 1, N, fp.get());
  std::shared_ptr<FILE> fp2(fp);
}
```

Custom Deleters

shared_ptr

Custom deleter for shared_ptr is passed to constructor, where it is type erased

```
struct fclose_deleter {
  void operator()(FILE* fp) const { fclose(fp); }
};

{
  std::shared_ptr<FILE> fp(fopen("readme.txt", "r"),
                           fclose_deleter{});
  fread(buffer, 1, N, fp.get());
  std::shared_ptr<FILE> fp2(fp);
}
```

CUSTOM DELETERS

shared_ptr

Custom deleter for shared_ptr is passed to constructor, where it is type erased

```
struct fclose_deleter {
  void operator()(FILE* fp) const { fclose(fp); }
};

{
  std::shared_ptr<FILE> fp(fopen("readme.txt", "r"),
                           fclose_deleter{});
  fread(buffer, 1, N, fp.get());
  std::shared_ptr<FILE> fp2(fp);
}
```

fclose called automatically

smart_ptr Extras

CASTS

To have share_ptrs of different types that manage the same object

dynamic_pointer_cast, static_pointer_cast, const_pointer_cast, reinterpret_pointer_cast

```
std::shared_ptr<WidgetBase> p = create_widget(inputs);
std::shared_ptr<BlueWidget> b =
    std::dynamic_pointer_cast<BlueWidget>(p);
if (b)
  b->do_something_blue();
```

Aliasing Constructor

Two shared_ptrs use same control block, but have unrelated object pointers

Useful for pointers to subobjects of managed objects

```
struct Outer {
  int a;
  Inner inner;
};

void f(std::shared_ptr<Outer> op) {
  std::shared_ptr<Inner> ip(op, &op->inner);
  // ...
}
```

SHARED_FROM_THIS

To convert `this` into a shared_ptr

- Class derives from `enable_shared_from_this`
- Object is already managed by a shared_ptr
- `return this->shared_from_this();`

CONCLUSIONS

Raw Pointers vs Smart Pointers

Raw pointers can fulfill lots of roles

- Can't fully communicate the programmer's intent

Smart pointers can be very powerful

- Automatic tasks, especially cleanup
- Extra checking
- Limited API, to better express programmer's intent

STANDARD VS CUSTOM SMART POINTERS

Standard C++ has two commonly used smart pointers

- unique_ptr and shared_ptr
- Use them whenever they fit your needs

Don't limit yourself to standard smart pointers

- If your framework has smart pointers, use them
- Write your own if necessary
- "The Smart Pointers I Wish I Had," Matthew Fleming, CppCon 2019

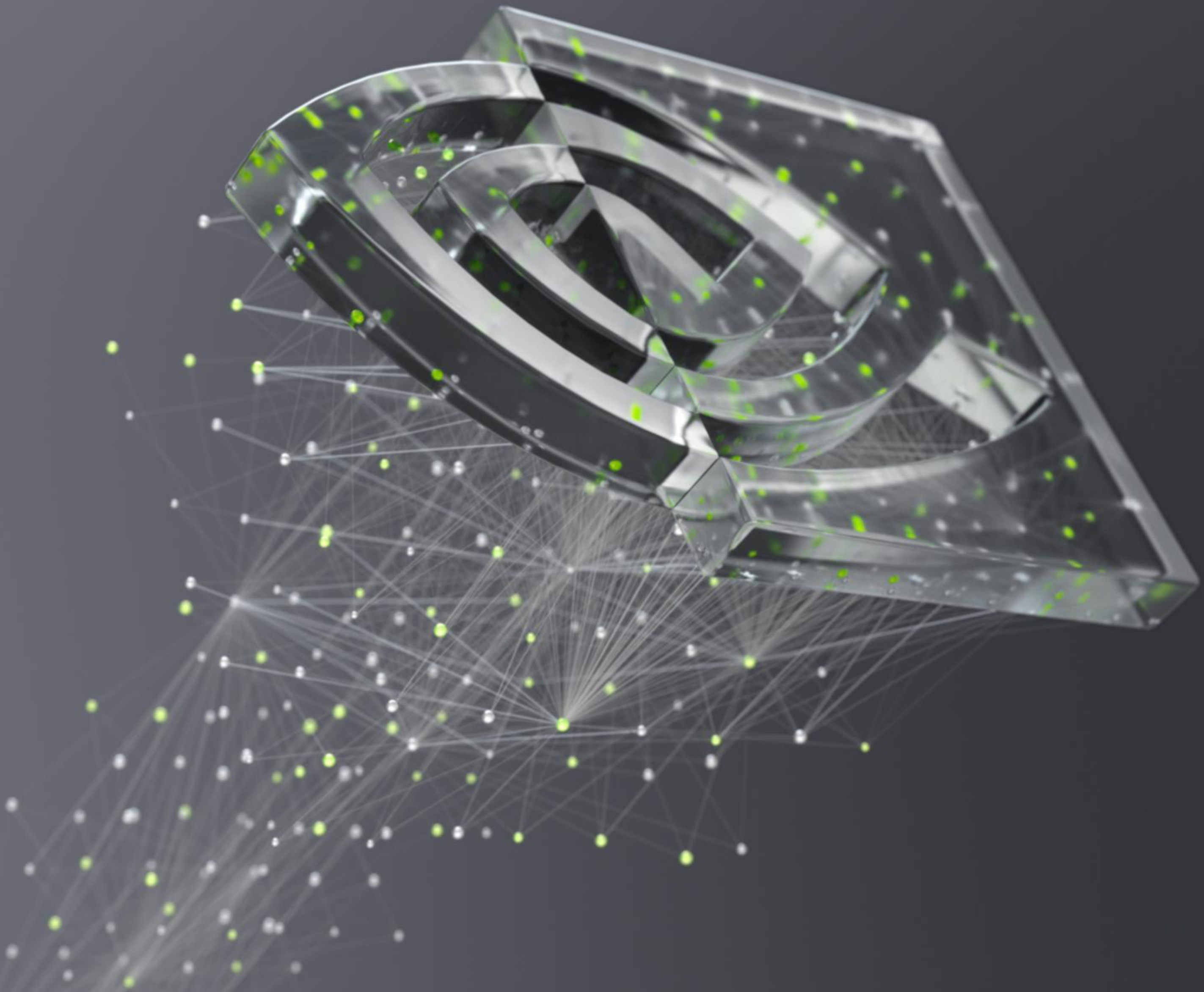
GUIDELINES

Use smart pointers to represent ownership

Prefer unique_ptr over shared_ptr

Use make_unique and make_shared

Pass/return unique_ptr to transfer ownership between functions



NVIDIA