Growing a Smart Pointer

PATRICE ROY (PATRICER AT GMAIL.COM)

C++ USERS GROUP, MUNICH, JAN. 2021

Who am I?

Father of five (four girls, one boy), ages 26 to 7 (might explain some background noises)

Feeds and cleans up after a varying number of animals (might explain some background noises too)
- Look for Paws of Britannia with your favorite search engine

Used to write military flight simulator code, among other things
- CAE Electronics Ltd, IREQ

Full-time teacher since 1998
- Collège Lionel-Groulx, Université de Sherbrooke
- Works a lot with game programmers

Incidentally, WG21 and WG23 member (although I've been really busy recently)
- Involved in SG14, among other study groups
- Occasional WG21 secretary

And so on…

Overview

What do we mean by « Smart Pointer »?

What do we mean by « Responsibility »?

Key Niches for Smart Pointers

Filling a Niche: Growing a Smart Pointer

- Selecting a Responsibility Handling Mechanism
- Putting it all Together

Reflecting on what we did

What do we mean by « Smart Pointer »?

What do we mean by « Smart Pointer »?

Pointers, even raw pointers, are not bad per se

- In fact, they are often quite useful!
- Contrary to popular belief (or FUD…), pointers do not leak memory by themselves

What do we mean by « Smart Pointer »?

Pointers, even raw pointers, are not bad per se

- In fact, they are often quite useful!
- Contrary to popular belief (or FUD…), pointers do not leak memory by themselves

```
void f() {
   const char *p = "I love my instructor";
   cout << p << endl; // prints important message
} // p dies here. No leak
```

What do we mean by « Smart Pointer »?

Pointers, even raw pointers, are not bad per se

- In fact, they are often quite useful!
- Contrary to popular belief (or FUD…), pointers do not leak memory by themselves

```
template <class T, size_t N>
void g(const T (&arr)[N]) {
   cout << arr[0];
   for(auto p = next(begin(arr)); p != end(arr); ++p)
      cout << " | " << *p;
} // p dies once the for loop concludes. No leak
```

What do we mean by « Smart Pointer »?

The main problem with pointers tends to be managing the lifetime of the *pointee*

What do we mean by « Smart Pointer »?

The main problem with pointers tends to be managing the lifetime of the *pointee*

```
class X { /* ... */ };
X* f();
void g(X*);
int main() {
   X *p = f(); // or auto p = f();
   g(p);
   delete p;
}
```

What do we mean by « Smart Pointer »?

The main problem with pointers tends to be managing the lifetime of the *pointee*

```
class X { /* ... */ };
X* f();
void g(X*);
int main() {
   X *p = f();
   g(p);
   delete p;
}
```

In how many ways can we go wrong here?

What do we mean by « Smart Pointer »?

The main problem with pointers tends to be managing the lifetime of the *pointee*

```
class X { /* ... */ };
X* f();
void g(X*);
int main() {
    X *p = f();
    g(p);
    delete p;
}
```

Where does `*p` come from? Some C++ code? Some C code? Some code written in another language? Something managed by the underlying platform?

What do we mean by « Smart Pointer »?

The main problem with pointers tends to be managing the lifetime of the *pointee*

```
class X { /* ... */ };
X* f();
void g(X*);
int main() {
   X *p = f();
   g(p);
   delete p;
}
```

Will `g(p)` call `delete` on `p`?

What do we mean by « Smart Pointer »?

The main problem with pointers tends to be managing the lifetime of the *pointee*

```
class X { /* ... */ };
X* f();
void g(X*);
int main() {
    X *p = f();
    g(p);
    delete p;
}
```

Will `g(p)` throw? If so, what will happen to `p`'s pointee?

What do we mean by « Smart Pointer »?

The main problem with pointers tends to be managing the lifetime of the *pointee*

```
class X { /* ... */ };
X* f();
void g(X*);
int main() {
   X *p = f();
   g(p);
   delete p;
}
```

Should the caller of `f()`, `main()` in this case, even be doing this?

What do we mean by « Smart Pointer »?

The main problem with pointers tends to be managing the lifetime of the *pointee*

```
class X { /* ... */ };
X* f();
void g(X*);
int main() {
   X *p = f();
   g(p);
   delete p;
}
```

Was the pointee allocated through `operator new`?

What do we mean by « Smart Pointer »?

The main problem with pointers tends to be managing the lifetime of the *pointee*

```
class X { /* ... */ };
X* f();
void g(X*);
int main() {
   X *p = f();
   g(p);
   delete p;
}
```

Was the pointee allocated through `operator new[]`?

What do we mean by « Smart Pointer »?

The main problem with pointers tends to be managing the lifetime of the *pointee*

```
class X { /* ... */ };
X* f();
void g(X*);
int main() {
   X *p = f();
   g(p);
   delete p;
}
```

Was the pointee allocated through `malloc()`?

What do we mean by « Smart Pointer »?

The main problem with pointers tends to be managing the lifetime of the *pointee*

```
class X { /* ... */ };
X* f();
void g(X*);
int main() {
   X *p = f();
   g(p);
   delete p;
}
```

…was the pointee allocated through some other mechanism?

What do we mean by « Smart Pointer »?

The main problem with pointers tends to be managing the lifetime of the *pointee*

```
class X { /* ... */ };
X* f();
void g(X*);
int main() {
   X *p = f();
   g(p);
   delete p;
}
```

…was the pointee dynamically allocated at all?

What do we mean by « Smart Pointer »?

The main problem with pointers tends to be managing the lifetime of the *pointee*

```
class X { /* ... */ };
X* f();
void g(X*);
int main() {
   X *p = f();
   g(p);
   delete p;
}
```

…was the pointee dynamically allocated at all?

```
// ... is f() written as:
X* f() {
   return new X;
}
// ...or
X* f() {
   static X object;
   return &object;
}
// ...or...
```

What do we mean by « Smart Pointer »?

The main problem with pointers tends to be managing the lifetime of the *pointee*

```
X* f() {
    return new X;
}
```

Bonus issue : what if the caller neglects to finalize the returned pointee?

With C++17, we can require that the return value is used, with `[[nodiscard]]`, but not that `operator delete` is applied to it…

What do we mean by « Smart Pointer »?

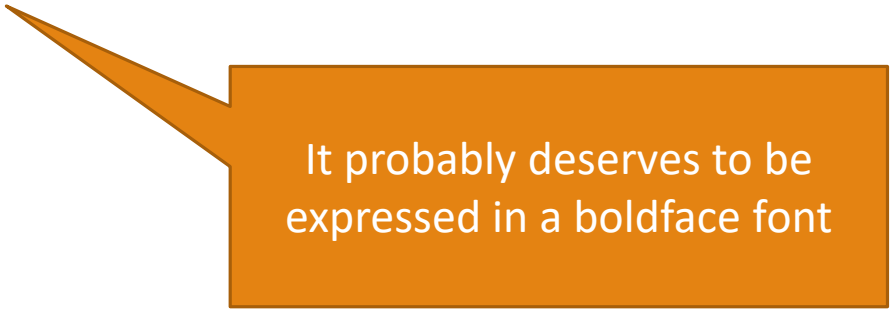
Using raw pointers on the boundaries of function interfaces is tricky, as the responsibility with respect to the pointee are unclear

- The key benefit of smart pointers is that they encode the responsibility with respect to the pointee into a type

What do we mean by « Smart Pointer »?

Using raw pointers on the boundaries of function interfaces is tricky, as the responsibility with respect to the pointee are unclear

- **The key benefit of smart pointers is that they encode the responsibility with respect to the pointee into a type**



What do we mean by « Responsibility »?

What do we mean by « Responsibility »?

Responsibility here can also be expressed as defining the rules or manner of ownership

What do we mean by « Responsibility »?

Responsibility here can also be expressed as defining the rules or manner of ownership

In C++ terms, ownership for a smart pointer determines what happens with the pointee when (or if) the object is:

- Copied
- Moved
- Destroyed

What do we mean by « Responsibility »?

Responsibility here can also be expressed as defining the rules or manner of ownership

In C++ terms, ownership for a smart pointer determines what happens with the pointee when (or if) the object is:

- Copied
- Moved
- Destroyed

Since smart pointers are objects, the fact that they have well-defined responsibility over the pointee can simplify significantly the code of objects handling pointees

What do we mean by « Responsibility »?

Compare:

- https://wandbox.org/permlink/2fV7sI3e2oZk9LEx
- https://wandbox.org/permlink/2vYrQfUri467X0S7

Simple, dynamically allocated arrays with fixed sizes

- Just the basics (special functions, `size()`, `begin()`, `end()`, some basic aliases)

Using a smart pointer…

- …reduces the number of source code lines by 24%
- …makes all explicit exception handling go away

Key Niches for Smart Pointer

Key Niches for Smart Pointer

With C++20, the standard library offers the following smart pointers

- unique_ptr<T>
- shared_ptr<T>
- weak_ptr<T>
- atomic<shared_ptr<T>>
- atomic<weak_ptr<T>>

Key Niches for Smart Pointer

With C++20, the standard library offers the following smart pointers

- unique_ptr<T>
  - There are variations such as `unique_ptr<T[]>` and versions that allow custom deleters
- shared_ptr<T>
  - Likewise, there is a `shared_ptr<T[]>` variation since C++17
- weak_ptr<T>
  - Collaborates with `shared_ptr<T>` in some corner cases
- atomic<shared_ptr<T>>
- atomic<weak_ptr<T>>
  - Useful for specialized, often lock-free applications

Key Niches for Smart Pointer

With C++20, the standard library offers the following smart pointers

- **unique_ptr<T>**
- **shared_ptr<T>**
- weak_ptr<T>
- atomic<shared_ptr<T>>
- atomic<weak_ptr<T>>

For most use cases, these two are the main standard smart pointers

Key Niches for Smart Pointer

With C++20, the standard library offers the following smart pointers

- **unique_ptr<T>**
- **shared_ptr<T>**
- weak_ptr<T>
- atomic<shared_ptr<T>>
- atomic<weak_ptr<T>>

In the majority of use cases, `unique_ptr<T>` is your friend; `shared_ptr<T>` is necessary, but more costly and its niche is more specialized

Key Niches for Smart Pointer

As a reminder:

- **The key benefit of smart pointers is that they encode the responsibility with respect to the pointee into a type**

Key Niches for Smart Pointer

Type	Niche
unique_ptr<T>	Single ownership. Non-copyable. Destroys the pointee when it is destroyed
shared_ptr<T>	Shared ownership. Copying, assigning and destroying all update a use count. When the last owner is destroyed, it destroys the pointee

Key Niches for Smart Pointer

Type	Niche
unique_ptr<T>	Single ownership. Non-copyable. Destroys the pointee when it is destroyed
shared_ptr<T>	Shared ownership. Copying, assigning and destroying all update a use count. When the last owner is destroyed, it destroys the pointee

It’s a restricted set of niches…

Key Niches for Smart Pointer

Type	Niche
unique_ptr<T>	Single ownership. Non-copyable. Destroys the pointee when it is destroyed
shared_ptr<T>	Shared ownership. Copying, assigning and destroying all update a use count. When the last owner is destroyed, it destroys the pointee

What additional rows could be interesting?

Key Niches for Smart Pointer

Type	Niche
unique_ptr<T>	Single ownership. Non-copyable. Destroys the pointee when it is destroyed
shared_ptr<T>	Shared ownership. Copying, assigning and destroying all update a use count. When the last owner is destroyed, it destroys the pointee
	No ownership. Behaves like a reference

Key Niches for Smart Pointer

Type	Niche
unique_ptr<T>	Single ownership. Non-copyable. Destroys the pointee when it is destroyed
shared_ptr<T>	Shared ownership. Copying, assigning and destroying all update a use count. When the last owner is destroyed, it destroys the pointee
T*	**No ownership. Behaves like a reference**

Key Niches for Smart Pointer

Type	Niche
unique_ptr<T>	Single ownership. Non-copyable. Destroys the pointee when it is destroyed
shared_ptr<T>	Shared ownership. Copying, assigning and destroying all update a use count. When the last owner is destroyed, it destroys the pointee
T*	**No ownership. Behaves like a reference**

Really. This is what raw pointers are useful for on function interfaces

Key Niches for Smart Pointer

Type	Niche
unique_ptr<T>	Single ownership. Non-copyable. Destroys the pointee when it is destroyed
shared_ptr<T>	Shared ownership. Copying, assigning and destroying all update a use count. When the last owner is destroyed, it destroys the pointee
observer_ptr<T>	**No ownership. Behaves like a reference**

This is another option: a « smart » pointer that is really dumb (quoth Walter E. Brown: « *The dumbest smart pointer* »)

Key Niches for Smart Pointer

Type	Niche
unique_ptr<T>	Single ownership. Non-copyable. Destroys the pointee when it is destroyed
shared_ptr<T>	Shared ownership. Copying, assigning and destroying all update a use count. When the last owner is destroyed, it destroys the pointee
observer_ptr<T>	**No ownership. Behaves like a reference**

See https://wandbox.org/permlink/wasX6P6GOrYLSpNZ for an example

Key Niches for Smart Pointer

Type	Niche
unique_ptr<T>	Single ownership. Non-copyable. Destroys the pointee when it is destroyed
shared_ptr<T>	Shared ownership. Copying, assigning and destroying update a use count. When the last owner is destroyed, it destroys the pointee
T* or observer_ptr<T>	No ownership. Behaves like a reference
	No ownership. Behaves like a reference. Offers added semantics with respect to the pointee

Key Niches for Smart Pointer

Type	Niche
unique_ptr<T>	Single ownership. Non-copyable. Destroys the pointee when it is destroyed
shared_ptr<T>	Shared ownership. Copying, assigning and destroying update a use count. When the last owner is destroyed, it destroys the pointee
T* or observer_ptr<T>	No ownership. Behaves like a reference
non_null_ptr<T> or other	**No ownership. Behaves like a reference. Offers added semantics with respect to the pointee**

This is quite useful, as it displaces validation semantics from client code to the type

Key Niches for Smart Po…

Type	Nich…
unique_ptr<T>	Singl… poin…
shared_ptr<T>	Shar… dest… own…
T* or observer_ptr<T>	No o…
non_null_ptr<T> or other	**No o… adde…**

```
// without
void f(X *p) {
   assert(p);
   // use p...
}
// with
void f(non_null_ptr<X> p) {
   // use p...
}
```

This is quite useful, as it displaces validation semantics from client code to the type

Key Niches for Smart Pointer

Type	Niche
unique_ptr<T>	Single ownership. Non-copyable. Destroys the pointee when it is destroyed
shared_ptr<T>	Shared ownership. Copying, assigning and destroying update a use count. When the last owner is destroyed, it destroys the pointee
T* or observer_ptr<T>	No ownership. Behaves like a reference
non_null_ptr<T> or other	**No ownership. Behaves like a reference. Offers added semantics with respect to the pointee**

See https://wandbox.org/permlink/rtn1KYSU2Cfyx9Tg for an example

Key Niches for Smart Pointer

Type	Niche
unique_ptr<T>	Single ownership. Non-copyable. Destroys the pointee when it is destroyed
shared_ptr<T>	Shared ownership. Copying, assigning and destroying update a use count. When the last owner is destroyed, it destroys the pointee
T* or observer_ptr<T>	No ownership. Behaves like a reference
non_null_ptr<T> or other…	No ownership. Behaves like a reference. Offers added semantics with respect to the pointee
	Exotic semantics

Key Niches for Smart Pointer

Type	Niche
unique_ptr<T>	Single ownership. Non-copyable. Destroys the pointee when it is destroyed
shared_ptr<T>	Shared ownership. Copying, assigning and destroying update a use count. When the last owner is destroyed, it destroys the pointee
T* or observer_ptr<T>	No ownership. Behaves like a reference
non_null_ptr<T> or other…	No ownership. Behaves like a reference. Offers added semantics with respect to the pointee
remote_ptr<T>	**Exotic semantics**

Idea : calling a function performs the required marshalling.
Non-trivial code (no time to do this today)…

Key Niches for Smart Pointer

Type	Niche
unique_ptr<T>	Single ownership. Non-copyable. Destroys the pointee when it is destroyed
shared_ptr<T>	Shared ownership. Copying, assigning and destroying update a use count. When the last owner is destroyed, it destroys the pointee
T* or observer_ptr<T>	No ownership. Behaves like a reference
non_null_ptr<T> or other…	No ownership. Behaves like a reference. Offers added semantics with respect to the pointee
remote_ptr<T>	Exotic semantics
	Single ownership. Duplicates the pointee when copied. Destroys the pointee when it is destroyed

Key Niches for Smart Pointer

Type	Niche
unique_ptr<T>	Single ownership. Non-copyable. Destroys the pointee when it is destroyed
shared_ptr<T>	Shared ownership. Copying, assigning and destroying update a use count. When the last owner is destroyed, it destroys the pointee
T* or observer_ptr<T>	No ownership. Behaves like a reference
non_null_ptr<T> or other…	No ownership. Behaves like a reference. Offers added semantics with respect to the pointee
remote_ptr<T>	Exotic semantics
dup_ptr<T>	**Single ownership. Duplicates the pointee when copied. Destroys the pointee when it is destroyed**

Let’s do this together :)

Filling a Niche: Growing a Smart Pointer

Filling a Niche: Growing a Smart Pointer

What are the use-cases for a `dup_ptr<T>`?

- We want pointer that takes ownership of the pointee, and duplicates the pointee when the pointer is duplicated
  - In this, `unique_ptr<T>` – being uncopyable – does not meet our needs
  - Neither does `shared_ptr<T>`, where copying the pointer shares the pointee

Filling a Niche: Growing a Smart Pointer

What are the use-cases for a `dup_ptr<T>`?

- We want pointer that takes ownership of the pointee, and duplicates the pointee when the pointer is duplicated
  - In this, `unique_ptr<T>` – being uncopyable – does not meet our needs
  - Neither does `shared_ptr<T>`, where copying the pointer shares the pointee
- We seek similar-to-value-semantics for objects that are copyable but that we want to allocate dynamically
  - There are objects like this
  - For example, any object with a private destructor can only be allocated dynamically by client code

Filling a Niche: Growing a Smart Pointer

Let’s do it one step at a time…

Filling a Niche: Growing a Smart Pointer

```
template <class T>
    class dup_ptr {
    };
```

Filling a Niche: Growing a Smart Pointer

```
template <class T>
    class dup_ptr {
    };
```

Well, that's a start :)

Filling a Niche: Growing a Smart Pointer

```
template <class T>
    class dup_ptr {
        T *p;
    };
```

Filling a Niche: Growing a Smart Pointer

```
template <class T>
    class dup_ptr {
        T *p;
    };
```

So, a `dup_ptr<T>` will host a `T*`. It's reasonable.

If we can keep it that way, we'll share a cool side of `unique_ptr<T>` (without custom deleter) in that `sizeof(dup_ptr<T>)==sizeof(T*)`, which is an enviable property (no size overhead when compared to a raw pointer)

Filling a Niche: Growing a Smart Pointer

```
template <class T>
    class dup_ptr {
        T *p;
    };
```

A reasonable example of client code would be:

```
int main() {
    static_assert(
        sizeof(dup_ptr<int>{})==sizeof(int*)
    );
    dup_ptr<int> p;
    assert(p.empty()); // or assert(!p)
}
```

Filling a Niche: Growing a Smart Pointer

```
template <class T>
   class dup_ptr {
      T *p;
   public:
      constexpr dup_ptr() noexcept : p{} {
      }
      constexpr bool empty() const noexcept {
         return !p; // or p == nullptr;
      }
      constexpr operator bool() const noexcept { return !empty(); }
   };
```

Filling a Niche: Growing a Smart Pointer

```
template <class T>
   class dup_ptr {
      T *p;
   public:
      constexpr dup_ptr() noexcept
      }
      constexpr bool empty() const noexcept {
         return !p;
      }
      constexpr operator bool() const noexcept { return !empty(); }
   };
```

```
int main() {
   static_assert(
      sizeof(dup_ptr<int>{})==sizeof(int*)
   );
   dup_ptr<int> p;
   assert(p.empty());
}
```

Filling a Niche: Growing a Smart Pointer

```
template <class T>
   class dup_ptr {
      T *p{}; // or T p = nullptr;
   public:
      dup_ptr() = default;
      constexpr bool empty() const noexcept {
         return !p;
      }
      constexpr operator bool() const noexcept { return !empty(); }
   };
```

Filling a Niche: Growing a Smart Pointer

```
template <class T>
   class dup_ptr {
      T *p{};
   public:
      dup_ptr() = default;
      constexpr bool empty() co
         return !p;
      }
      constexpr operator bool() const noexcept { return !empty(); }
   };
```

```
int main() {
   static_assert(
      sizeof(dup_ptr<int>{})==sizeof(int*)
   );
   dup_ptr<int> p;
   assert(p.empty());
}
```

Filling a Niche: Growing a Smart Pointer

```
template <class T>
   class dup_ptr {
      T *p{};
   public:
      dup_ptr() = default;
      constexpr bool empty() const noexcept { return !p; }
      constexpr operator bool() const noexcept { return !empty(); }
      dup_ptr(T *p) noexcept : p{ p } {
      }
      ~dup_ptr() { delete p; }
   };
```

Filling a Niche: Growing a Smart Pointer

```
template <class T>
   class dup_ptr {
      T *p{};
   public:
      dup_ptr() = default;
      constexpr bool empty() cons
      constexpr operator bool() const noexcept { return !empty(); }
      dup_ptr(T *p) noexcept : p{ p } {
      }
      ~dup_ptr() { delete p; }
   };
```

`delete p;` is a no-op if `p` is `nullptr`

We could specialize `dup_ptr<T>` for the case `T[]` and do `delete[] p;` here, but I will not have the time to do this today (try it!)

Filling a Niche: Growing a Smart Pointer

```
template <class T>
   class dup_ptr {
      T *p{};
   public:
      // ...
      T& operator*() noexcept { return *p; }
      const T& operator*() const noexcept { return *p; }
      T* operator->() noexcept { return p; }
      const T* operator->() const noexcept { return p; }
   };
```

Filling a Niche: Growing a Smart Pointer

```
template <class T>
   class dup_ptr {
      T *p{};
   public:
      // ...
      T& operator*() noexcept { return *p; }
      const T& operator*() const noexcept { return *p; }
      T* operator->() noexcept { return p; }
      const T* operator->() const noexcept { return p; }
   };
```

`operator*()` lets one dereference a `dup_ptr<T>` as one would a `T*`. Note the `noexcept` specification : if `p` is `nullptr`, then dereferencing it is undefined behavior, so `noexcept` remains reasonable

Filling a Niche: Growing a Smart Pointer

```
template <class T>
   class dup_ptr {
      T *p{};
   public:
      // ...
      T& operator*() noexcept { return *p; }
      const T& operator*() const noexcept { return *p; }
      T* operator->() noexcept { return p; }
      const T* operator->() const noexcept { return p; }
   };
```

Filling a Niche: Growing a Smart Pointer

```
template <class T>
   class dup_ptr {
      T *p{};
   public:
      // ...
      T& operator*() noexcept { return *p; }
      const T& operator*() const noexcept { return *p; }
      T* operator->() noexcept { return p; }
      const T* operator->() const noexcept { return p; }
   };
```

`operator->()` is a bit of a magical beast : when invoked implicitly on an object, it re-invokes `operator->()` on the returned value, until the entity returned is a raw pointer

Filling a Niche: Growing a Smart Pointer

```
template <class T>
   class dup_ptr {
      T *p{};
   public:
      // ...default ctor, ctor accepting T*...
      // ...destructor, empty(), operator bool...
      // ...operator* and operator-> both const and non-const...
   };
```

We already have a usable, oversimplified pointer wrapper:
https://wandbox.org/permlink/Y8ORd1Qr8An5IpaL

Selecting a Responsibility Handling Mechanism

Selecting a Responsibility Handling Mechanism

Now, for the duplication semantics we want to implement…

Selecting a Responsibility Handling Mechanism

Now, for the duplication semantics we want to implement…

One interesting problem is that duplication of the pointee can be of (at least) two flavors:

- Copying
  - Usually applies to objects used directly
  - Often non-polymorphic,
  - Could often be `final`

Selecting a Responsibility Handling Mechanism

Now, for the duplication semantics we want to implement…

One interesting problem is that duplication of the pointee can be of (at least) two flavors:

- Copying
  - Usually applies to objects used directly
  - Often non-polymorphic,
  - Could often be `final`
- Cloning
  - Usually applies to polymorphic objects
  - Handled indirectly
  - Typically conceived for extension

Selecting a Responsibility Handling Mechanism

Now, for the duplication semantics we want to implement…

One interesting problem is that duplication of the pointee can be of (at least) two flavors:

- Copying
- Cloning

C++ has a standard way to duplicate objects through copying: the copy constructor

Selecting a Responsibility Handling Mechanism

Now, for the duplication semantics we want to implement…

One interesting problem is that duplication of the pointee can be of (at least) two flavors:

- Copying
- Cloning

C++ has a standard way to duplicate objects through copying: the copy constructor

C++ has… many ways to implement cloning

- Note that the situation is not much better in many other popular languages

Selecting a Responsibility Handling Mechanism

```
struct Int {
    int n = 0;
    Int() = default;
    Int(int n) : n{ n } {
    }
};
```

```
Int modify_locally(Int &x) {
    Int backup = x; // copy construction
    x.n++; // leaves backup intact
    cout << x.n;
    return backup;
}
int main() {
    Int x; // x.n == 0
    x = modify_locally(x); // displays 1
    cout << x.n; // displays 0
}
```

Selecting a Responsibility Handling Mechanism

```
struct Int {
    int n = 0;
    Int() = default;
    Int(int n) : n{ n } {
    }
};
```

Sign that `Int` is nicely copy-constructible: it has no `virtual` member functions (deriving publicly from `Int` would be… questionable)

```
Int modify_locally(Int &x) {
    Int backup = x; // copy construction
    x.n++; // leaves backup intact
    cout << x.n;
    return backup;
}
int main() {
    Int x; // x.n == 0
    x = modify_locally(x); // displays 1
    cout << x.n; // displays 0
}
```

Selecting a Responsibility Handling Mechanism

```
class Image {
   // ...
public:
   void modify();
   virtual void display() const = 0;
   virtual ~Image() = default;
};
class Png : public Image {
   // ...
   void display() const override;
};
```

```
Image* modify_locally(Image *x) {
   // illegal! Image is abstract
   auto backup = new Image{ *x };
   x->modify();
   x->display();
   return backup;
}
int main() {
   Image *p = new Png;
   p = modify_locally(p);
   p->display();
}
```

Selecting a Responsibility Handling Mechanism

```
class Image {
   // ...
public:
   void modify();
   virtual void display() const = 0;
   virtual ~Image() = default;
};
class Png : public Image {
   // ...
   void display() con
};
```

Abstract (pure virtual) member function…

```
Image* modify_locally(Image *x) {
   // illegal! Image is abstract
   auto backup = new Image{ *x };
   x->modify();
   x->display();
   return backup;
}
int main() {
   Image *p = new Png;
   p = modify_locally(p);
   p->display();
}
```

Expression `new Image` is illegal since `Image` is abstract

Selecting a Responsibility Handling Mechanism

```
class Image {
   // ...
public:
   void modify();
   virtual void display() const {}
   virtual ~Image() = default;
};
class Png : public Image {
   // ...
   void display() const override;
};
```

```
Image* modify_locally(Image *x) {
   // legal, but incorrect
   auto backup = new Image{ *x };
   x->modify();
   x->display();
   return backup;
}
int main() {
   Image *p = new Png;
   p = modify_locally(p);
   p->display();
}
```

Selecting a Responsibility Handling Mechanism

```
class Image {
   // ...
public:
   void modify();
   virtual void display() const {}
   virtual ~Image() = default;
};
class Png : public Image {
   // ...
   void display()
};
```

Not abstract anymore… but it's not helpful (abstract was better in this case)

```
Image* modify_locally(Image *x) {
   // legal, but incorrect
   auto backup = new Image{ *x };
   x->modify();
   x->display();
   return backup;
}
int main() {
   Image *p = new Png;
   p = modify_locally(p);
   p->display();
}
```

This is legal as `Image` is not abstract, but we have slicing: we only copy the `Image` part of the object, losing all that is specific to `Png`

Selecting a Responsibility Handling Mechanism

```
class Image {
   // ...
public:
   void modify();
   virtual void display() const {}
   virtual ~Image() = default;
};
class Png : public Image {
   // ...
   void display() const override;
};
```

```
Image* modify_locally(Image *x) {
   // legal, but incorrect
   auto backup = new Image{ *x };
   x->modify();
   x->display();
   return backup;
}
int main() {
   Image *p = new Png;
   p = modify_locally(p);
   p->display();
}
```

The idea here is that only `*x` knows what `*x` is. We need subjective duplication… Cloning!

Selecting a Responsibility Handling Mechanism

Cloning is subjective duplication. Typically, this involves:

- A virtual `clone()` member function that duplicates the most derived object
- A `protected` copy constructor that performs the duplication
  - We want `protected` as client code cannot usually determine if it's safe to call

There are other ways to achieve this, but this will suffice for our discussion

Selecting a Responsibility Handling Mechanism

```
class Image {
protected:
  Image(const Image&) = default;
public:
  virtual Image *clone() const = 0;
  virtual ~Image() = default; // etc.
};
class Png : public Image {
  Png* clone() const override
    { return new Png{ *this }; }
protected:
  Png(const Png&);
};
```

```
Image* modify_locally(Image *x) {
   auto backup = x->clone(); // Ok
   x->modify();
   x->display();
   return backup;
}
int main() {
   Image *p = new Png;
   p = modify_locally(p);
   p->display();
}
```

Selecting a Responsibility Handling Mechanism

```
class Image {
protected:
  Image(const Image&) = default;
public:
  virtual Image *clone() const = 0;
  virtual ~Image() = default; // etc.
};
class Png : public Image {
  Png* clone() const override
    { return new Png{ *this }; }
protected:
  Png(const Png&);
};
```

```
Image* modify_locally(Image *x) {
   auto backup = x->clone(); // Ok
   x->modify();
   x->display();
   return backup;
}
int main() {
   Image *p = new Png;
   p = modify_locally(p);
   p->display();
}
```

Selecting a Responsibility Handling Mechanism

```
class Image {
protected:
  Image(const Image&) = default;
public:
  virtual Image *clone() const = 0;
  virtual ~Image() = default; // etc.
};
class Png : public Image {
  Png* clone() const override
    { return new Png{ *this }; }
protected:
  Png(const Png&);
};
```

```
Image* modify_locally(Image *x) {
   auto backup = x->clone(); // Ok
   x->modify();
   x->display();
   return backup;
}
int main() {
   Image *p = new Png;
   p = modify_locally(p);
   p->display();
}
```

This compiles and works… and leaks (do you see it?). Our simple smart pointer might visibly have its use…

Selecting a Responsibility Handling Mechanism

What's the problem space?

- Some objects can usually be duplicated through copy construction
- Some objects can usually be duplicated through cloning
- There might be more exotic cases that have escaped us… so we need an escape hatch

Selecting a Responsibility Handling Mechanism

What's the problem space?

- Some objects can usually be duplicated through copy construction
- Some objects can usually be duplicated through cloning
- There might be more exotic cases that have escaped us… so we need an escape hatch

There is no such thing as a std::clonable interface

- We can make suppositions, but we'll need to keep our options open

Selecting a Responsibility Handling Mechanism

Basic idea: let's design duplicating function objects

- To keep the architecture efficient, let's suppose them to be stateless
- For fun, you can tweak our `dup_ptr` to accept stateful versions, but then you will need to store them and will lose the `sizeof(dup_ptr<T>)==sizeof(T*)` property

Selecting a Responsibility Handling Mechanism

```
struct Copier {
   template <class T> T* operator()(const T *p) const {
      return new T{ *p };
   }
};
struct Cloner {
   template <class T> T* operator()(const T *p) const {
      return p->clone();
   }
};
```

Selecting a Responsibility Handling Mechanism

```
struct Copier {
    template <class T> T* operator()(const T *p) const {
        return new T{ *p };
    }
};
struct Cloner {
    template <class T> T* operator()(const T *p) const {
        return p->clone();
    }
};
```

We will cover the basic cases we know of implicitly, using these two duplication mechanisms

Selecting a Responsibility Handling Mechanism

```
struct Copier {
    template <class T> T* operator()(const T *p) const {
        return new T{ *p };
    }
};
struct Cloner {
    template <class T> T* operator()(const T *p) const {
        return p->clone();
    }
};
```

The idea is that one can add more duplicating object types as long as one respects the interface

Selecting a Responsibility Handling Mechanism

```
struct Copier {
    template <class T> T* operator()(const T *p) const {
        return new T{ *p };
    }
};
struct Cloner {
    template <class T> T* operator()(const T *p) const {
        return p->clone();
    }
};
```

To keep things simple, we will consider `p!=nullptr` to be a precondition of these duplication mechanisms

Selecting a Responsibility Handling Mechanism

Let’s pick an appropriate-by-default duplicating function object

- … but let’s make sure client code can pick one if it « knows better » than we do

Selecting a Responsibility Handling Mechanism

```
template <class T, class Dup = ...>
   class dup_ptr {
      T *p{};
   public:
      // ... use a Dup object whenever we want to duplicate *p
   };
```

Selecting a Responsibility Handling Mechanism

```
template <class T, class Dup = ...>
   class dup_ptr {
      T *p{};
   public:
      // ... use a Dup object whenever we want to duplicate *p
   };
```

Type `Dup` would be what is often called a Policy class

Selecting a Responsibility Handling Mechanism

How do we pick the right type for `Dup`?

- One option is to say « Suppose a `clonable` interface of our design; if `T` inherits from `clonable`, then we clone, otherwise we copy »

Selecting a Responsibility Handling Mechanism

How do we pick the right type for `Dup`?

- One option is to say « Suppose a `clonable` interface of our design; if `T` inherits from `clonable`, then we clone, otherwise we copy »

```
// hypothetical « clonable » interface
struct clonable {
   virtual clonable *clone() const = 0;
   virtual ~clonable() = default;
protected:
   clonable() = default;
   clonable(const clonable&) = default;
};
```

Selecting a Responsibility Handling Mechanism

```
#include <type_traits>
template <class T, class Dup = std::conditional_t<
   std::is_base_of_v<clonable, T>, Cloner, Copier
>>
   class dup_ptr {
      T *p{};
   public:
      // ... use a Dup object whenever we want to duplicate *p
   };
```

Selecting a Responsibility Handling Mechanism

```
#include <type_traits>
template <class T, class Dup = std::conditional_t<
   std::is_base_of_v<clonable, T>, Cloner, Copier
>>
   class dup_ptr {
      T *p{};
   public:
      // ... use a Dup object whenever we want to duplicate *p
   };
```

`conditional_t<cond,T,F>` is equivalent to type `T` if `cond` is `true`, and equivalent to type `F` if `cond` is `false`. It's a compile-time « `if` » for types

Selecting a Responsibility Handling Mechanism

```
#include <type_traits>
template
    std::
>>
    class
        T
    publi
        //
    };
```

```
// easy to roll out your own:
template <bool, class, class> struct conditional_;
template <class T, class F>
    struct conditional_<true, T, F> {
        using type = T;
    };
template <class T, class F>
    struct conditional_<false, T, F> {
        using type = F;
    };
template <bool cond, class T, class F>
    using conditional_t_ = typename conditional_<cond,T,F>::type;
```

Selecting a Responsibility Handling Mechanism

```
#include <type_traits>
template <class T, class Dup = std::conditional_t<
   std::is_base_of_v<clonable, T>, Cloner, Copier
>>
   class dup_ptr {
      T *p{};
   public:
      // ... use a Dup object whenever we want to duplicate *p
   };
```

What we are doing here is picking a default duplication strategy for `T` based on the properties of `T`

If this default strategy is inappropriate, client code can explicitly use one that suits its needs

Selecting a Responsibility Handling Mechanism

How do we pick the right type for `Dup`?

- One option is to say « Suppose a `clonable` interface of our design; if `T` inherits from `clonable`, then we clone, otherwise we copy »
  - This works but is un-idiomatic for C++
  - Imposing a specific interface to types is intrusive coupling. Some languages encourage this, but not us

Selecting a Responsibility Handling Mechanism

How do we pick the right type for `Dup`?

- One option is to say « Suppose a `clonable` interface of our design; if `T` inherits from `clonable`, then we clone, otherwise we copy »
  - This works but is un-idiomatic for C++
  - Imposing a specific interface to types is intrusive coupling. Some languages encourage this, but not us
- Another option is to say « if `T` has a `const` member function named `clone`, then we clone, otherwise we copy »

Selecting a Responsibility Handling Mechanism

```
#include <type_traits>
template <class, class = void> struct has_clone : std::false_type { };
template <class T>
   struct has_clone <T, std::void_t< decltype(std::declval<const T*>()->clone()) >>
      : std::true_type {
   };
template <class T> constexpr bool has_clone_v = has_clone<T>::value;
template <class T, class Dup = std::conditional_t<has_clone_v<T>, Cloner, Copier>>
   class dup_ptr {
      T *p{};
   public:
      // ... use a Dup object whenever we want to duplicate *p
   };
```

Selecting a Responsibility Handling Mechanism

```
#include <type_traits>

template <class, class = void> struct has_clone : std::false_type { };

template <class T>

   struct has_clone <T, std::void_t< decltype(std::declval<const T*>()->clone()) >>

      : std::true_type {

   };

template <class T> constexpr bool has_clo

template <class T, class Dup = std::condi

   class dup_ptr {

      T *p{};

   public:

      // ... use a Dup object whenever we

   };
```

`void_t` was invented by Walter E. Brown. It's a brilliant trick that lets us detect the validity of expressions at compile time and make choices based on the result

https://en.cppreference.com/w/cpp/types/void_t

Selecting a Responsibility Handling Mechanism

```
#include <type_traits>

template <class, class = void> struct has_clone : std::false_type { };

template <class T>

   struct has_clone <T, std::void_t< decltype(std::declval<const T*>()->clone()) >>

      : std::true_type {

   };

template <class T

template <class T

   class dup_ptr

      T *p{};

   public:

      // ... use

   };
```

`declval<T>()` is an hypothetical (no joke!) function that lets one reason at compile-time as if we had a `T`, regardless of how that `T` would have been constructed

The whole « implementation » is:

```
template <class T> T&& declval();
```

Selecting a Responsibility Handling Mechanism

```
#include <type_traits>

template <class, class = void> struct has_clone : std::false_type { };

template <class T>

   struct has_clone <T, std::void_t< decltype(std::declval<const T*>()->clone()) >>

      : std::true_type {

   };

template <class T> constexpr bool has_clone_v = h                    value;

template <class T, class Dup = st

   class dup_ptr {

      T *p{};

   public:

      // ... use a Dup object whenever we want to duplicate *p

   };
```

Here, concretely, we are validating that it would be possible to call a `clone()` member function on a `const T*`

Selecting a Responsibility Handling Mechanism

```
#include <type_traits>
template <class, class = void> struct has_clone : std::false_type { };
template <class T>
   struct has_clone <T, std::void_t< decltype(std::declval<const T*>()->clone()) >>
      : std::true_type {
   };
template <class T> constexpr bool has_clone_v = has_clone<T>::value;
template <class T, class Dup = std::conditional_t<has_clone_v<T>, Cloner, Copier>>
   class dup_ptr {
      T *p{};
   public:
      // ... use a Dup object whenever we w
   };
```

Thus, instead of imposing a common base class to all clonable types, we are looking for a member function with a specific name and signature

Selecting a Responsibility Handling Mechanism

How do we pick the right type for `Dup`?

- One option is to say « Suppose a `clonable` interface of our design; if `T` inherits from `clonable`, then we clone, otherwise we copy »
  - This works but is un-idiomatic for C++
  - Imposing a specific interface to types is intrusive coupling. Some languages encourage this, but not us
- Another option is to say « if `T` has a `const` member function named `clone`, then we clone, otherwise we copy »
- If we have a C++20 compiler, we can simply define a `clonable` concept

Selecting a Responsibility Handling Mechanism

```
#include <concepts>
template <class T>
   concept clonable = requires(const T *p) {
      { p->clone() } -> std::convertible_to<T*>;
   };
template <class T, class Dup = std::conditional_t<clonable<T>, Cloner, Copier>>
   class dup_ptr {
      T *p{};
   public:
      // ... use a Dup object whenever we want to duplicate *p
   };
```

Selecting a Responsibility Handling Mechanism

```
#include <concepts>
template <class T>
   concept clonable = requires(const T *p) {
      { p->clone() } -> std::convertible_to<T*>;
   };
template <class T, class Dup = std::conditional_t<clonable<T>, Cloner, Copier>>
   class dup_ptr {
      T *p{};
   public:
      // ... use a Dup object whenever we want to duplicate *p
   };
```

Here, we are stating `T` is `clonable` if, given a `const T *`, we can invoke `clone()` on it and get as a result something convertible to `T*`

Selecting a Responsibility Handling Mechanism

```
#include <concepts>
template <class T>
   concept clonable = requires(const T *p) {
      { p->clone() } -> std::convertible_to<T*>;
   };
template <class T> requires clonable<T> class dup_ptr {
      // ... Clone whenever we want to duplicate *p
   };
template <class T> requires !clonable<T> class dup_ptr {
      // ... Copy whenever we want to duplicate *p
   };
```

Selecting a Responsibility Handling Mechanism

```
#include <concepts>
template <class T>
   concept clonable = requires(const T *p) {
      { p->clone() } -> std::convertible_to<T*>;
   };
template <class T> requires clonable<T> class dup_ptr {
      // ... Clone whenever we want to duplicate *p
   };
template <class T> requires !clonable<T> class dup_ptr {
      // ... Copy whenever we want to duplicate *p
   };
```

We could to it this way if we were convinced there would never be other options. In our case, keeping a client-approved escape hatch is probably safer

Selecting a Responsibility Handling Mechanism

How do we pick the right type for `Dup`?

- One option is to say « Suppose a `clonable` interface of our design; if `T` inherits from `clonable`, then we clone, otherwise we copy »
  - This works but is un-idiomatic for C++
  - Imposing a specific interface to types is intrusive coupling. Some languages encourage this, but not us
- Another option is to say « if `T` has a `const` member function named `clone`, then we clone, otherwise we copy »
- If we have a C++20 compiler, we can simply define a `clonable` concept

What if client code has more exotic duplication mechanisms?

- Then, client code will supply its own `Dup` policy instead of relying on our defaults

Selecting a Responsibility Handling Mechanism

```
template <class T, class Dup = ...>
   class dup_ptr {
      T *p{};
      // ...
   };
class Exo { // exotic :)
   // private
   Exo(const Exo&);
public:
   Exo() = default;
   Exo* duplicate() const;
};
```

```
template <class T, class D>
   void f(dup_ptr<T, D> p) { /* use *p */ }
int main() {
   struct exo_dup {
      Exo* operator()(const Exo *p) const {
         return p->duplicate();
      }
   };
   dup_ptr<Exo, exo_dup>
      p{ new Exo };
   f(p);
}
```

Selecting a Responsibility Handling Mechanism

```
template <class T, class Dup = ...>
   class dup_ptr {
      T *p{};
      // ...
   };
class Exo { // exotic :)
   // private
   Exo(const Exo&);
public:
   Exo() = default;
   Exo* duplicate() const;
};
```

```
template <class T, class D>
   void f(dup_ptr<T, D> p) { /* use *p */ }
int main() {
   struct exo_dup {
      Exo* operator()(const Exo *p) const {
         return p->duplicate();
      }
   };
   dup_ptr<Exo, exo_dup>
      p{ new Exo };
   f(p);
}
```

Selecting a Responsibility Handling Mechanism

```
template <class T, class Dup = ...>
   class dup_ptr {
      T *p{};
      // ...
   };
class Exo { // exotic :)
   // private
   Exo(const Exo&);
public:
   Exo() = default;
   Exo* duplicate() const;
};
```

```
template <class T, class D>
   void f(dup_ptr<T, D> p) { /* use *p */  }
auto duplicator() {
   return [](auto p) {
     return p->duplicate();
   };
}
int main() {
   dup_ptr<Exo, decltype(duplicator())>
      p{ new Exo };
   f(p);
}
```

Putting it all Together

Putting it all Together

A smart pointer defines responsibily over the pointee,

Thus, the selected duplication mechanism will intervene in special member functions

- Default constructor
- Copy constructor
- Move constructor
- Copy assignment
- Move assignment
- Destructor

Putting it all Together

A smart pointer defines responsibily over the pointee,

Thus, the selected duplication mechanism will intervene in special member functions

- Default constructor
- **Copy constructor**
- Move constructor
- **Copy assignment**
- Move assignment
- Destructor

Putting it all Together

A smart pointer defines responsibily over the pointee,

Thus, the selected duplication mechanism will intervene in special member functions

- Default constructor
- **Copy constructor**
- Move constructor
- **Copy assignment**
- Move assignment
- Destructor

We will look at the others too, but duplication of the pointee will occur in copying functions

Putting it all Together

A smart pointer defines responsibily over the pointee,

Thus, the selected duplication mechanism will intervene in special member functions

- Default constructor
- **Copy constructor**
- Move constructor
- **Copy assignment**
- Move assignment
- Destructor

Some of these functions will use the copy-and-swap idiom for assignment. We will only define the `swap()` member function between two `dup_ptr<T,D>` with the same `D` type:

```
void swap(dup_ptr &other) {
    using std::swap;
    swap(p, other.p);
}
```

Putting it all Together

A smart pointer defines responsibily over the pointee,

Thus, the selected duplication mechanism will intervene in special member functions

- **Default constructor**
- Copy constructor
- Move constructor
- Copy assignment
- Move assignment
- **Destructor**

Putting it all Together

A smart pointer defines responsibily over the pointee,

Thus, the selected duplication mechanism will intervene in special member functions

- **Default constructor**
- Copy constructor
- Move constructor
- Copy assignment
- Move assignment
- **Destructor**

```
template <class T, class Dup = ...>
   class dup_ptr {
      T *p{};
   public:
      dup_ptr() = default;
      ~dup_ptr() {
        delete p;
      }
      // ...
   };
```

Putting it all Together

A smart pointer defines responsibily over the pointee,

Thus, the selected duplication mechanism will intervene in special member functions

- Default constructor
- Copy constructor
- **Move constructor**
- Copy assignment
- **Move assignment**
- Destructor

Move operations can be implemented simply between two `dup_ptr<T,D>` instances if we assume `D` to be stateless (our situation)

If we decide that the `D` object needs to be stored, then we need to reflect on whether it should be movable, and what it should mean in that case

Putting it all Together

A smart pointer defines responsibily over the pointee,

Thus, the selected duplication mechanism will intervene in special member functions

- Default constructor
- Copy constructor
- **Move constructor**
- Copy assignment
- **Move assignment**
- Destructor

```
// ...
dup_ptr(dup_ptr &&other)
   : p{ std::exchange(other.p, nullptr) } {
}
dup_ptr& operator=(dup_ptr &&other) {
   dup_ptr{ std::move(other) }.swap(*this);
   return *this;
}
// ...
```

Putting it all Together

A smart pointer defines responsibily over the pointee,

Thus, the selected duplication mechanism will intervene in special member functions

- Default constructor
- **Copy constructor**
- Move constructor
- **Copy assignment**
- Move assignment
- Destructor

The copy operations will duplicate the pointee according to the duplication policy associated with the type (the `D` in `dup_ptr<T,D>`).

We will not try do duplicate a null pointer, thus respecting the duplication policies' precondition

Putting it all Together

A smart pointer defines responsibily over the pointee,

Thus, the selected duplication mechanism will intervene in special member functions

- Default constructor
- **Copy constructor**
- Move constructor
- **Copy assignment**
- Move assignment
- Destructor

```
// ...
dup_ptr(const dup_ptr &other)
   : p{ other.empty()? nullptr : Dup{}(other.p) } {
}
dup_ptr& operator=(const dup_ptr &other) {
   dup_ptr{ other }.swap(*this);
   return *this;
}
// ...
```

Putting it all Together

A smart pointer defines responsibily over the pointee,

Thus, the selected duplication mechanism will intervene in special member functions

- Default constructor
- Copy constructor
- Move constructor
- Copy assignment
- Move assignment
- Destructor

Full source code is available on https://wandbox.org/permlink/zuXLKc8tUKpdKaYu

Putting it all Together

A smart pointer defines responsibily over the pointee,

Thus, the selected duplication mechanism will intervene in special member functions

- Default constructor
- Copy constructor
- Move constructor
- Copy assignment
- Move assignment
- Destructor

There's a lot more we could do with time: specialize for the `T[]` case, add relational operators, implement covariant constructors and assignment, etc.

Full source code is available on https://wandbox.org/permlink/zuXLKc8tUKpdKaYu

Reflecting on what we did

Reflecting on what we did

In C++, we don’t need to use pointers all that much, but when we do…

Raw pointers are fine and useful sometimes

However, their responsibility (if any) towards the pointee is often unclear

- Ideally, either encapsulate them in « handle types » such as `std::vector<T>` or `std::string`, or use them as non-owning observers of their pointee

The key benefit of smart pointers is that they encode the responsibility with respect to the pointee into a type

- Their purpose is clearer

Reflecting on what we did

Standard C++ smart pointers are small in number but very good at what they do

- `std::unique_ptr` is a beautiful idea
  - Simplifies code and solves many problems at little-to-no cost depending on the use-case
- `std::shared_ptr` has a more narrow niche, but occupies it well
  - It's tricky to write, so prefer using standard, well-tested ones to rolling out your own

Reflecting on what we did

There are niches not (yet) covered by standard smart pointers

- We covered one with `dup_ptr`
- We wrote code to have fun, obviously, but we had a use-case in mind

Just because a niche is left uncovered does not mean we should standardize a solution for it

- Sometimes, the niche is too narrow
- At other times, the solutions we know are not of `std::`-level quality
  - I would not dare suggesting a strategy that introduces a mandatory dependency on a std::clonable interface!

Reflecting on what we did

Our `dup_ptr<T,D>` had interesting properties:

- We implicitly supplied a `D` type for the most common cases
- We allowed client code to supply its own `D` type for atypical cases
- Provided `D` remains stateless and we do not need to store it, we can provide a `dup_ptr<T>` which incurs no size overhead when compared with `T*`

Reflecting on what we did

Deducing a `D` policy class for `dup_ptr<T,D>` can be done in various ways

- Imposing an intrusive type relationship
  - if `T` derives from `clonable`…
- Detecting the validity of relied-upon expressions
  - if I can call `clone()` from an hypothetical `const T*`…
- Detecting conformity to a concept
  - if `T` satisfies `clonable`…
- etc.

Reflecting on what we did

I had fun, hope you had fun too!

Questions?