Expressive code with

C++

SMART POINTERS

Fluent {C++}

fluentcpp.com

Smart pointer basics

One thing that can rapidly clutter your C++ code and hinder its readability is memory management. Done badly, this can turn a simple logic into an inexpressive slalom of mess management, and make the code lose control over memory safety.

The programming task of ensuring that all objects are correctly deleted is very low in terms of levels of abstraction, and since writing good code essentially comes down to respecting levels of abstraction, you want to keep those tasks away from your business logic (or any sort of logic for that matter).

Smart pointers are made to deal with this effectively and relieve your code from the dirty work. This series of posts will show you how to take advantage of them to make your code both **more expressive and more correct**.

We're going to go deep into the subject and since **I want everyone to be able to follow** all of this series, there is no prerequisite and we start off here with the basics of smart pointers.

The stack and the heap

Like many other languages, C++ has several types of memories, that correspond to different parts of the physical memory. They are: the static, the stack, and the heap. The static is a topic rich enough to deserve its own moment of glory, so here we focus on the stack and the heap only.

The stack

Allocating on the stack is the default way to store objects in C++:

```
int f(int a)
{
    if (a > 0)
    {
        std::string s = "a positive number";
        std::cout << s << '\n';
    }
    return a;
}
```

Here `a` and `s` are stored on the stack. Technically this means that `a` and `s` are stored next to one another in memory because they have been pushed on a stacked maintained by the compiler. However these concerns are not so relevant for daily work.

There is one important, crucial, even fundamental thing to know about the stack though. It is at the basis of everything that follows in the rest of this series. And the good news is that it's very easy:

Objects allocated on the stack are automatically destroyed when they go out of scope.

You can re-read this a couple of times, maybe tatoo it on your forearm if needed, and print out a T-shirt to your spouse reading this statement so that you can be reminded of it regularly.

In C++ a scope is defined by a pair of brackets (`{` and `}`) except those used to initialize objects:

```
std::vector<int> v = {1, 2, 3}; // this is not a scope

if (v.size() > 0)
{ // this is the beginning of a scope
    ...
} // this is the end of a scope
```

And there are 3 ways for an object to go out of scope:

- encountering the next closing bracket (}),
- encountering a return statement,
- having an exception thrown inside the current scope that is not caught inside the current scope.

So in the first code example, `s` is destroyed at the closing bracket of the if statement, and `a` is destroyed at the return statement of the function.

The heap

The heap is where dynamically allocated objects are stored, that is to say **objects that are allocated with a call to new**, which returns a pointer:

```
int * pi = new int(42);
```

After the above statement, `pi` points to an int object allocated on the heap.

Ok strictly speaking, the memory allocated by new is called the free store. The heap is the memory allocated by `malloc`, `calloc` and `realloc` which are vestiges from C that are normally no longer used in new code, and which we are ignoring in this post (but we'll talk more about them later in the series). But the term 'heap' is so ubiquitous in developer jargon to talk about any dynamically allocated memory that I am using it here in that sense.

Anyway to destroy an object allocated by new, we have to do it manually by calling delete:

```
delete pi;
```

Contrary to the stack, objects allocated on the heap are **not destroyed automatically**. This offers the advantages of keeping them longer than the end of a scope, and without incurring any copy except those of pointers which are very cheap. Also, pointers allow to manipulate objects polymorphically: a pointer to a base class can in fact point to objects of any derived class.

But as a price to pay for this flexibility it puts you, the developer, in charge of their deletion.

And deleting `a` object on the heap is no trivial task: delete has to be called **once and only once** to deallocate a heap-based object. If it is not called the object is not deallocated, and its memory space is not reusable - this is called a memory leak. But on the other hand, a delete called more than once on the same address leads to undefined behaviour.

And this is where the code gets cluttered and loses expressiveness (and sometimes even correctness). Indeed, to make sure that all objects are correctly destroyed, the bookkeeping varies from a simple delete to a complex system of flags in the presence of early returns for example.

Also, some interfaces are ambiguous in terms of memory management. Consider the following example:

```
House* buildAHouse();
```

As a caller of this function, should I delete the pointer it returns? If I don't and no one does then it's a memory leak. But if I do and someone else does, then it's undefined behaviour. Between the devil and the deep blue sea.

I think all this has led to a bad reputation of C++ as being a complex language in terms of memory management.

But fortunately, smart pointers will take care of all of this for you.

RAII: the magic four letters

RAII is a very idiomatic concept in C++ that takes advantage of the essential property of the stack (look up on your arm, or at the upper body of your spouse) to simplify the memory management of objects on the heap. In fact RAII can even be used to make easy and safe the management of any kind of resource, and not only memory. Oh and I'm not going to write what these 4 letters mean because it is unimportant and confusing in my opinion. You can take them as the name of someone, like superhero of C++ for example.

The principle of RAII is simple: wrap a resource (a pointer for instance) into an object, and dispose of the resource in its destructor. And this is exactly what smart pointers do:

```
template <typename T>
class SmartPointer
{
public:
    explicit SmartPointer(T* p) : p_(p) {}
    ~SmartPointer() { delete p_; }

private:
    T* p_;
};
```

The point is that you can manipulate smart pointers as objects allocated on the stack. And the compiler will take care of automatically calling the destructor of the smart pointer because... **objects allocated on the stack are automatically destroyed when they go out of scope**. And this will therefore call delete on the wrapped pointer. Only once. In a nutshell, smart pointers behave like pointers, but when they are destroyed they delete the object they point to.

The above code example was only made to get a grasp of RAII. But by no means is it a complete interface of a realistic smart pointer.

First, a smart pointer syntactically behaves like a pointer in many way: it can be dereferenced with `operator*` or `operator->`, that is to say you can call `*sp` or `sp->member` on it. And it is also convertible to bool, so that it can be used in an if statement like a pointer:

```
if (sp)
{
    …
```

which tests the nullity of the underlying pointer. And finally, the underlying pointer itself is accessible with a `.get()` method.

Second, and maybe more importantly, there is a missing aspect from the above interface: it doesn't deal with copy! Indeed, as is, a `SmartPointer` copied also copies the underlying pointer, so the below code has a bug:

```
{
   SmartPointer<int> sp1(new int(42));
   SmartPointer<int> sp2 = sp1; // now both sp1 and sp2 point to
                                // the same object
} // sp1 and sp2 are both destroyed, the pointer is deleted twice!
```

Indeed, it deletes the underlying object twice, leading to undefined behaviour.

How to deal with copy then? This is a feature on which the various types of smart pointer differ. And it turns out that this lets you express your intentions in code quite precisely. Stay tuned, as this is what we see in the next part of this series.

`unique_ptr`,`shared_ptr`,`weak_ptr`, `scoped_ptr`, raw pointers - Knowing your smart pointers

Like we saw when discussing what smart pointers are about, some active decision has to be taken about how a smart pointer should be copied. Otherwise, a default copy constructor would likely lead to undefined behaviour.

It turns out that there are several valid ways to go about this, and this leads to a variety of smart pointers. And it is important to understand what these various smart pointers do because they are ways to **express a design** into your code, and therefore also to **understand a design** by reading code.

We see here the various types of pointers that exist out there, approximately sorted by decreasing order of usefulness (according to me):

- std::unique_ptr
- raw pointer
- std::shared_ptr
- std::weak_ptr
- boost::scoped_ptr
- std::auto_ptr

std::unique_ptr

As of this writing, this is the smart pointer to use by default. It came into the standard in C++11.

The semantics of `std::unique_ptr` is that it is the sole owner of a memory resource. A `std::unique_ptr` will hold a pointer and delete it in its destructor (unless you customize this, which is the topic of another part of the series).

This allows you to express your intentions in an interface. Consider the following function:

```
std::unique_ptr<House> buildAHouse();
```

It tells you that it gives you a pointer to a house, of which you are the owner. **No one else will delete this pointer** except the `unique_ptr` that is returned by the function. And since you get the ownership, this gives you confidence that you are free to modify the value of the pointed to object.

Note that `std::unique_ptr` is the preferred pointer to return from a factory function. Indeed, on the top of taking care of handling the memory, `std::unique_ptr` wraps a normal pointer and is therefore compatible with polymorphism.

But this works the other way around too, by passing an `std::unique_ptr` as a parameter:

```
class House
{
public:
    House(std::unique_ptr<PileOfWood> wood);
    …
```

In this case, the house takes ownership of the `PileOfWood`.

Note though that even when you receive a unique_ptr, you're not guaranteed that no one else has access to this pointer. Indeed, if another context keeps a copy of the pointer inside your `unique_ptr`, then modifying the pointed to object through the `unique_ptr` object will of course impact this other context. If you don't want this to happen, the way to express it is by using a **`unique_ptr` to `const`**:

```
std::unique_ptr<const House> buildAHouse();
      // for some reason, I don't want you
      // to modify the house you're being passed
```

To ensure that there is only one unique_ptr that owns a memory resource, std::unique_ptr cannot be copied. The ownership can however be transferred from one `unique_ptr` to another (which is how you can pass them or return them from a function) by moving a `unique_ptr` into another one.

A move can be achieved by returning an std::unique_ptr by value from a function, or explicitly in code:

```
std::unique_ptr<int> p1 = std::make_unique(42);
std::unique_ptr<int> p2 = move(p1); // now p2 hold the resource
                                    // and p1 no longer hold anything
```

Raw pointers

"What?", you may be thinking. "We're talking about smart pointers, what are raw pointers doing here??"

Well, even if raw pointers are not smart pointers, they aren't 'dumb' pointers either. In fact there are legitimate reasons to use them although these reasons don't happen often. They share a lot with references, but the latter should be preferred except in some cases (but this is the topic of another article).

For now I only want to focus on what raw pointers and references express in code: **raw pointers and references represent access to an object, but not ownership**. In fact, this is the default way of passing objects to functions and methods:

```
void renderHouse(House const& house);
```

This is particularly relevant to note when you hold an object with a `unique_ptr` and want to pass it to an interface. You don't pass the `unique_ptr`, nor a reference to it, but rather a reference to the pointed to object:

```
std::unique_ptr<House> house = buildAHouse();
renderHouse(*house);
```

std::shared_ptr

`shared_ptr` entered the standard in C++11, but appeared in Boost well before that.

A single memory resource can be held by several `std::shared_ptr`s at the same time. The `shared_ptr`s internally maintain a count of how many of them there are holding the same resource, and when the last one is destroyed, it deletes the memory resource.

Therefore `std::shared_ptr` allows copies, but with a reference-counting mechanism to make sure that every resource is deleted once and only once.

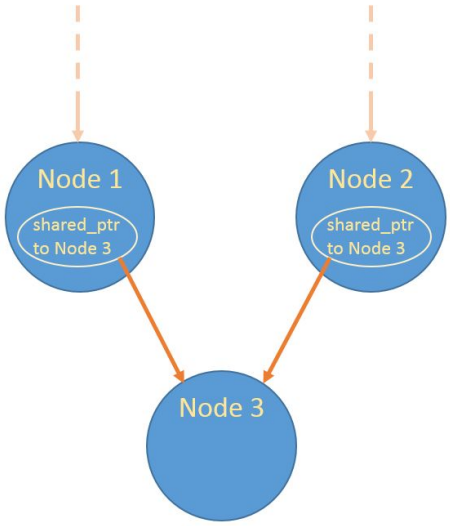
At first glance, `std::shared_ptr` looks like the panacea for memory management, as it can be passed around and still maintain memory safety.

But **`std::shared_ptr` should not be used by default**, for several reasons:

- Having several simultaneous holders of a resource makes for a **more complex** system than with one unique holder, like with `std::unique_ptr`.

- Having several simultaneous holders of a resource makes **thread-safety** harder,
- It makes the **code counter-intuitive** when an object is not shared in terms of the domain and still appears as "shared" in the code for a technical reason,
- It can incur a **performance** cost, both in time and memory, because of the bookkeeping related to the reference-counting.

One good case for using `std::shared_ptr` though is when objects are **shared in the domain**. Using shared pointers then reflects it in an expressive way. Typically, the nodes of a graphs are well represented as shared pointers, because several nodes can hold a reference to one other node.



std::weak_ptr

`weak_ptr` entered the language in C++11 but appeared in Boost well before that.

`std::weak_ptr`s can hold a reference to a shared object along with other `std::shared_ptr`s, but they don't increment the reference count. This means that if no more `std::shared_ptr` are holding an object, this object will be deleted even if some weak pointers still point to it.

For this reason, a weak pointer needs to check if the object it points to is still alive. To do this, it has to be copied into to a `std::shared_ptr`:

```
void useMyWeakPointer(std::weak_ptr<int> wp)
{
    if (std::shared_ptr<int> sp = wp.lock())
```

```
    {
        // the resource is still here and can be used
    }
    else
    {
        // the resource is no longer here
    }
}
```

A typical use case for this is about **breaking `shared_ptr` circular references**. Consider the following code:

```
struct House
{
    std::shared_ptr<House> neighbour;
};

std::shared_ptr<House> house1 = std::make_shared<House>();
std::shared_ptr<House> house2 = std::make_shared<House>();;
house1->neighbour = house2;
house2->neighbour = house1;
```

None of the houses ends up being destroyed at the end of this code, because the `shared_ptr`s points into one another. But if one is a `weak_ptr` instead, there is no longer a circular reference.

Another use case pointed out by this answer on Stack Overflow is that `weak_ptr` can be used to **maintain a cache**. The data may or may not have been cleared from the cache, and the `weak_ptr` references this data.

boost::scoped_ptr

`scoped_ptr` is present in Boost but was not included in the standard (at least as of C++17).

It simply disables the copy and even the move construction. So it is the sole owner of a resource, and its ownership cannot be transferred. Therefore, a `scoped_ptr` can only live inside... a scope. Or as a data member of an object. And of course, as a smart pointer, it keeps the advantage of deleting its underlying pointer in its destructor.

std::auto_ptr

`auto_ptr` was present in C++98, has been deprecated in C++11 and removed from the language in C++17.

It aimed at filling the same need as `unique_ptr`, but back when move semantics didn't exist in C++. It essentially does in its **copy constructor** what `unique_ptr` does in its move constructor. But `auto_ptr` is inferior to `unique_ptr` and you shouldn't use it if you have access to `unique_ptr`, because it can lead to erroneous code:

```
std::auto_ptr<int> p1(new int(42));
std::auto_ptr<int> p2 = p1; // it looks like p2 == p1, but no!
                                    // p1 is now empty and p2 uses the
resource
```

You know Andersen's The Ugly Duckling, where a poor little ducking is rejected by its siblings because it's not good-looking, and who turns out to grow into a beautiful swan? The story of `std::auto_ptr` is like this but going back in time: `std::auto_ptr` started by being the way to go to deal with ownership, and now it looks terrible in front of its siblings. It's like The Ugly Benjamin Button Duckling, if you will.

:)

In the next part of this series we will see how to simplify complex memory management by using the more advanced features of `std::unique_ptr`.

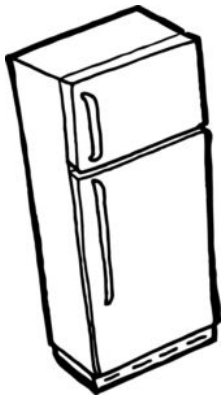
How to implement the pimpl idiom by using `unique_ptr`

The pimpl, standing for "pointer to implementation" is a widespread technique to cut compilation dependencies. We will see **how to implement the pimpl idiom with a smart pointer**.

Indeed, the pimpl idiom has an owning pointer in charge of managing a memory resource, so it sounds only logical to use a smart pointer, such as std::unique_ptr for example.

The pimpl

Just to have a common basis for discussion, let's quickly go over the pimpl idiom by putting together an example that uses it.



Say we have a class representing a fridge (yeah why not?), that works with an engine that it contains. Here is the header of this class:

```
#include "Engine.h"

class Fridge
{
public:
   void coolDown();
private:
   Engine engine_;
};
```

(the contents of the Engine class are not relevant here).

And here is its implementation file:

```
#include "Fridge.h"

void Fridge::coolDown()
{
   /* ... */
}
```

Now there is an issue with this design (that could be serious or not, depending on how many clients `Fridge` has). Since Fridge.h `#include`s Engine.h, any client of the `Fridge` class will indirectly `#include` the `Engine` class. So when the `Engine` class is modified, all the clients of `Fridge` have to recompile, even if they don't use `Engine` directly.

The pimpl idiom aims at solving this issue by adding a level of indirection, `FridgeImpl`, that takes on the `Engine`.

The header file becomes:

```
class Fridge
{
public:
   Fridge();
   ~Fridge();

   void coolDown();
private:
   class FridgeImpl;
   FridgeImpl* impl_;
};
```

Note that it no longer `#include` Engine.h.

And the implementation file becomes:

```
#include "Engine.h"
#include "Fridge.h"

class FridgeImpl
{
public:
   void coolDown()
   {
```

```
        /* ... */
    }
private:
    Engine engine_;
};

Fridge::Fridge() : impl_(new FridgeImpl) {}

Fridge::~Fridge()
{
    delete impl_;
}

void Fridge::coolDown()
{
    impl_->coolDown();
}
```

The class now delegates its functionalities and members to `FridgeImpl`, and Fridge only has to forward the calls and **manage the life cycle** of the `impl_` pointer.

What makes it work is that **pointers only need a forward declaration to compile**. For this reason, the header file of the `Fridge` class doesn't need to see the full definition of `FridgeImpl`, and therefore neither do `Fridge`'s clients.

Using std::unique_ptr to manage the life cycle

Today it's a bit unsettling to leave a raw pointer managing its own resource in C++. A natural thing to do would be to replace it with an `std::unique_ptr` (or with another smart pointer). This way the `Fridge` destructor no longer needs to do anything, and we can leave the compiler automatically generate it for us.

The header becomes:

```
#include <memory>

class Fridge
{
public:
    Fridge();
    void coolDown();
private:
```

```
    class FridgeImpl;
    std::unique_ptr<FridgeImpl> impl_;
};
```

And the implementation file becomes:

```
#include "Engine.h"
#include "Fridge.h"

class FridgeImpl
{
public:
    void coolDown()
    {
        /* ... */
    }
private:
    Engine engine_;
};

Fridge::Fridge() : impl_(new FridgeImpl) {}
```

Right? Let's build the program...

Oops, we get the following compilation errors!

```
use of undefined type 'FridgeImpl'
can't delete an incomplete type
```

Can you see what's going on here?

Destructor visibility

There is a rule in C++ that says that deleting a pointer leads to undefined behaviour if:

- this pointer has type `void*`, or
- the type pointed to is incomplete, that is to say is only forward declared, like `FridgeImpl` in our header file.

`std::unique_ptr` happens to check in its destructor if the definition of the type is visible before calling delete. So it refuses to compile and to call delete if the type is only forward declared.

In fact, `std::unique_ptr` is not the only component to provide this check: Boost also proposes the `checked_delete` function and its siblings to make sure that a call to delete is well-formed.

Since we removed the declaration of the destructor in the Fridge class, the compiler took over and defined it for us. But compiler-generated methods are declared **inline**, so they are implemented in the header file directly. And there, the type of `FridgeImpl` is incomplete. Hence the error.

The fix would then be to declare the destructor and thus prevent the compiler from doing it for us. So the header file becomes:

```
#include <memory>

class Fridge
{
public:
   Fridge();
   ~Fridge();
   void coolDown();
private:
   class FridgeImpl;
   std::unique_ptr<FridgeImpl> impl_;
};
```

And we can still use the default implementation for the destructor that the compiler would have generated. But we need to put it in the implementation file, **after the definition of `FridgeImpl`**:

```
#include "Engine.h"
#include "Fridge.h"

class FridgeImpl
{
public:
   void coolDown()
   {
      /* ... */
   }
private:
   Engine engine_;
};
```

```
Fridge::Fridge() : impl_(new FridgeImpl) {}

Fridge::~Fridge() = default;
```

This program compiles, run and works.

There are plenty other important aspects to consider when implementing a pimpl in C++. For this I advise you to have a look at the dedicated section in Herb Sutter's Exceptional C++.

How to Transfer `unique_ptrs` From a Set to Another Set

Transferring a `std::unique_ptr` to another `std::unique_ptr` is an easy thing to do:

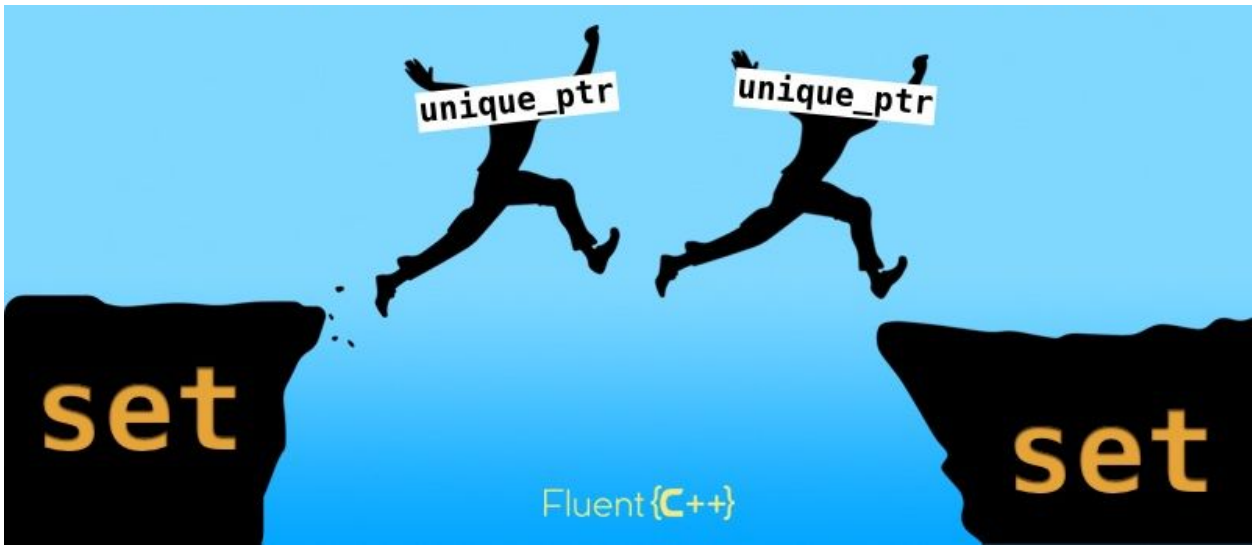
```
std::unique_ptr<int> p1 = std::make_unique<int>(42);
std::unique_ptr<int> p2;

p2 = std::move(p1); // the contents of p1 have been transferred to p2
```

Easy peasy, lemon squeezy.

Now what if those `unique_ptrs` are living inside of two sets? It should be just as easy to transfer those in the first set over to the second set, right?

It turns out that it's not easy, neither peasy, and even less lemon squeezy. Unless you have C++17, in which case it's a breeze. But before C++17, it's not. Here are various alternatives you can use to approach this.

Let's see the motivating problem first.



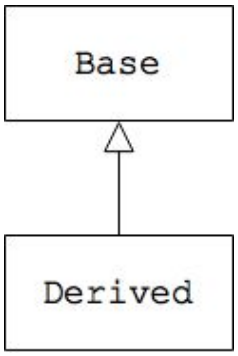
The case: transfering sets of unique_ptrs

We start by seeing what a `std::set` of `std::unique_ptr` would represent, and then we see what problem happens when trying to transfer the contents of one set to another.

Sets of unique_ptrs: unique and polymorphic

To begin with, you may have wondered why do a `unique_ptr` on an `int` as in the above example. Except for showing a simple example, well, it has no use at all.

A more realistic case would be one of runtime polymorphism via inheritance, with a Base class that can have Derived classes:



And we would use the base class polymorphically by holding it with some sort of handle (pointer or reference). To encapsulate the memory management, we would use a `std::unique_ptr<Base>`.

Now if we want a collection of several objects implementing Base, but that could be of any derived classes, we can use **a collection of `unique_ptr<Base>`s**.

Finally, we may want to prevent our collection to have duplicates. This is what `std::set` does. Note that to implement this constraint, `std::set` needs a way to compare its objects together.

Indeed, by declaring a set this way:

```
std::set<std::unique_ptr<Base>>
```

the comparison between elements of the set will call the `operator<` of `std::unique_ptr`, which compares the memory addresses of the pointers inside them.

In most cases, this is not what you want. When we think "no duplicates", it generally means "no logical duplicates" as in: no two elements have the same value. And not "no two elements are located at the same address in memory".

To implement no logical duplicates, we need to call the `operator<` on Base (provided that it exists, maybe using an id provided by `Base` for instance) to compare elements and determines whether they are duplicates. And to make the set use this operator, we need to customize the comparator of the set:

```
struct ComparePointee
{
    template<typename T>
    bool operator()(std::unique_ptr<T> const& up1, std::unique_ptr<T>
const& up2)
    {
        return *up1 < *up2;
    }
};

std::set<std::unique_ptr<int>, ComparePointee> mySet;
```

To avoid writing this type every time we instantiate such a set in code, we can hide its technical aspects behind an alias:

```
template<typename T>
using UniquePointerSet = std::set<std::unique_ptr<T>,ComparePointee>;
```

Transferring unique_ptrs between two sets

Ok. We're all set (ha-ha) and ready to transfer the elements of a set to another one. Here are our two sets:

```
UniquePointerSet<Base> source;
source.insert(std::make_unique<Derived>());

UniquePointerSet<Base> destination;
```

To transfer elements efficiently, we use the insert method:

```
destination.insert(begin(source), end(source));
```

But this leads to a compilation error!

```
error: use of deleted function 'std::unique_ptr<_Tp,
_Dp>::unique_ptr(const std::unique_ptr<_Tp, _Dp>&) [with _Tp =
Base; _Dp = std::default_delete<Base>]'
```

Indeed, the insert methods attemps to make a copy of the `unique_ptr` elements.

What to do then?

C++17's new method on set: merge

sets and maps in C++ are internally implemented as trees. This lets them ensure the algorithmic complexities guaranteed by the methods of their interface. Before C++17, it didn't show in the interface.

C++17 adds the merge method to sets:

```
destination.merge(source);
```

This makes destination **take over** the nodes of the tree inside of source. It's like performing a splicing on lists. So after executing this line, destination has the elements that source had, and source is empty.

And since it's only the nodes that get modified, and not what's inside them, the `unique_ptr`s don't feel a thing. They are not even moved.

`destination` now has the `unique_ptr`s, end of story.

Now if you don't have C++17 in production, which is the case of a lot of people at the time I'm writing these lines, what can you do?

We can't move from a set

The standard algorithm to move elements from a collection to another collection is `std::move`. Here is how it works with `std::vector`:

```
std::vector<std::unique_ptr<Base>> source;
source.push_back(std::make_unique<Derived>());

std::vector<std::unique_ptr<Base>> destination;

std::move(begin(source),end(source),std::back_inserter(destination));
```

after the execution of this line, `destination` has the elements that source had and source is not empty, but has empty `unique_ptr`s.

Let's try to do the same thing with our sets now:

```
UniquePointerSet<Base> source;
source.insert(std::make_unique<Derived>());

UniquePointerSet<Base> destination;

std::move(begin(source), end(source), std::inserter(destination,
end(destination)));
```

We get the same compilation error as in the beginning, some unique_ptrs are getting copied:

```
error: use of deleted function 'std::unique_ptr<_Tp,
_Dp>::unique_ptr(const std::unique_ptr<_Tp, _Dp>&)
```

This may look surprising. The purpose of the `std::move` algorithm is to avoid making copies on the `unique_ptr` elements and move them instead, so why are they being copied??

The answer lies in how the set provides access to its elements. When dereferenced, a set's iterator does not return a `unique_ptr&`, but rather a **`const`** `unique_ptr&`. It is to make sure that the values inside of the set don't get modified without the set being aware of it. Indeed, it could break its invariant of being sorted.

So here is what happens:

- `std::move` dereferences the iterator on set and gets a `const unique_ptr&`,
- it calls `std::move` on that references, thus getting a `const unique_ptr&&`,
- it calls the `insert` method on the insert output iterator and passes it this const `unique_ptr&&`,
- the `insert` method has two overloads: one that takes a `const unique_ptr&`, and one that takes a `unique_ptr&&`. Because of the const in the type we're passing, the compiler cannot resolve this call to the second method, and calls the first one instead.

Then the insert output iterator calls calls the insert overload on the set that takes a const `unique_ptr&` and in turn calls the copy constructor of `unique_ptr` with that lvalue reference, and that leads to the compilation error.

Making a sacrifice

So before C++17, moving elements from a set doesn't seem to be possible. Something has to give: either moving, or the sets. This leads us to two possible aspects to give up on.

Keeping the set but paying up for the copies

To give up on the move and accepting to copy the elements from a set to another, we need to make a copy of the contents pointed by the `unique_ptr`s.

For this, let's assume that `Base` has is a polymorphic `clone` method overridden in `Derived`:

```
class Base
{
public:
    virtual std::unique_ptr<Base> cloneBase() const = 0;

    // rest of Base...
};

class Derived : public Base
{
public:
    std::unique_ptr<Base> cloneBase() const override
    {
        return std::make_unique<Derived>(*this);
    }

    // rest of Derived...
};
```

At call site, we can make copies of the unique_ptrs from a set over to the other one, for instance this way:

```
auto clone = [](std::unique_ptr<Base> const& pointer){ return
pointer->cloneBase(); };
std::transform(begin(source), end(source), std::inserter(destination,
end(destination)), clone);
```

Or, with a for loop:

```
for (auto const& pointer : source)
{
    destination.insert(pointer->cloneBase());
}
```

Keeping the move and throwing away the set

The set that doesn't let the move happen is the source set. If you only need the destination to have unique elements, you can replace the source set by a `std::vector`.

Indeed, `std::vector` does not add a `const` to the value returned by its iterator. We can therefore move its elements from it with the `std::move` algorithm:

```
std::vector<std::unique_ptr<Base>> source;
source.push_back(std::make_unique<Derived>(42));

std::set<std::unique_ptr<Base>> destination;

std::move(begin(source), end(source), std::inserter(destination,
end(destination)));
```

Then the destination set contains a `unique_ptr` that has the contents that used to be in the one of the source, and the source vector now contains an empty `unique_ptr`.

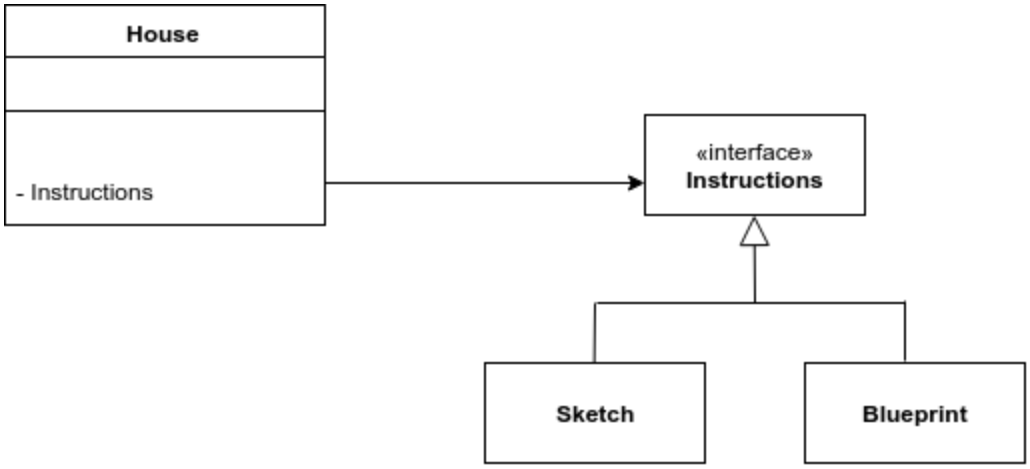
Live at head

You can see that there are ways around the problem of transferring `unique_ptr`s from a set to another one. But the real solution is the `merge` method of `std::set` in C++17.

The standard library is getter better and better as the language evolves. Let's do what we can to move (ha-ha) to the latest version of C++, and never look back.

Custom deleters

Let's take the example of a `House` class, that carries its building Instructions with it, which are polymorphic and can be either a `Sketch` or a full-fledged `Blueprint`:



One way to deal with the life cycle of the `Instructions` is to store them as a `unique_ptr` in the `House`. And say that a copy of the house makes a deep copy of the instructions:

```
class House
{
public:
    explicit House(std::unique_ptr<Instructions> instructions)
        : instructions_(std::move(instructions)) {}
    House(House const& other)
        : instructions_(other.instructions_->clone()) {}

private:
    std::unique_ptr<Instructions> instructions_;
};
```

Indeed, `Instructions` has a polymorphic clone, which is implemented by the derived classes:

```
class Instructions
{
public:
    virtual std::unique_ptr<Instructions> clone() const = 0;
    virtual ~Instructions(){};
```

```
};

class Sketch : public Instructions
{
public:
    std::unique_ptr<Instructions> clone() const
    {
        return std::unique_ptr<Instructions>(new Sketch(*this));
    }
};


class Blueprint : public Instructions
{
public:
    std::unique_ptr<Instructions> clone() const
    {
        return std::unique_ptr<Instructions>(new Blueprint(*this));
    }
};
```

Here is a way to construct a house:

```
enum class BuildingMethod
{
    fromSketch,
    fromBlueprint
};

House buildAHouse(BuildingMethod method)
{
    if (method == BuildingMethod::fromSketch)
        return House(std::unique_ptr<Instructions>(new Sketch));
    if (method == BuildingMethod::fromBlueprint)
        return House(std::unique_ptr<Instructions>(new Blueprint));
    throw InvalidBuildMethod();
}
```

where the building method may come from user input.

The situations gets more challenging when objects can come from another memory source, like the stack for example:

```
Blueprint blueprint;
House house(???); // how do I pass the blueprint to the house?
```

Indeed, we can't bind a `unique_ptr` to a stack-allocated object, because calling `delete` on it would cause undefined behaviour.

One solution would be to make a copy of the blueprint and allocating it on the heap. This may be OK, or it may be costly (I've come across a similar situation once where it was the bottleneck of the program).

But passing objects allocated on the stack is a legitimate need. We then don't want the `House` to destroy the `Instructions` in its destructor when the object comes from the stack.

How can `std::unique_ptr` help here?

Seeing the real face of `std::unique_ptr`

Most of the time, the C++ unique pointer is used as `std::unique_ptr<T>`. But its complete type has a second template parameter, its deleter:

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

`std::default_delete<T>` is a function object that calls delete when invoked. But it is only the default type for Deleter, and it can be changed for a custom deleter.

This opens the possibility to use unique pointers for types that have a specific code for disposing of their resources. This happens in legacy code coming from C where a function typically takes care of deallocating an object along with its contents:

```
struct GizmoDeleter
{
    void operator()(Gizmo* p)
    {
        oldFunctionThatDeallocatesAGizmo(p);
    }
};

using GizmoUniquePtr = std::unique_ptr<Gizmo, GizmoDeleter>;
```

(By the way, this technique is quite helpful as a step to simplify legacy code, in order to make it compatible with `std::unique_ptr`.)

Now armed with this feature, let's go back to our motivating scenario.

Using several deleters

Our initial problem was that we wanted the unique_ptr to delete the Instructions, except when they came from the stack in which case we wanted it to leave them alone.

The deleter can be customized to delete or not delete, given the situation. For this we can use **several deleting functions, all of the same function type** (being `void(*)(Instructions*)`):

```
using InstructionsUniquePtr = std::unique_ptr<Instructions,
void(*)(Instructions*)>;
```

The deleting functions are then:

```
void deleteInstructions(Instructions* instructions){ delete
instructions;}
void doNotDeleteInstructions(Instructions* instructions){}
```

One deletes the object, and the other doesn't do anything.

To use them, the occurrences of `std::unique_ptr<Instructions>` needs to be replaced with `InstructionUniquePtr`, and the unique pointers can be constructed this way:

```
if (method == BuildingMethod::fromSketch)
  return House(InstructionsUniquePtr(new Sketch,
deleteInstructions));
if (method == BuildingMethod::fromBlueprint)
  return House(InstructionsUniquePtr(new Blueprint,
deleteInstructions));
```

Except when the parameter comes from the stack, in which case the no-op deleter can be used:

```
Blueprint blueprint;
House house(InstructionsUniquePtr(&blueprint,
doNotDeleteInstructions));
```

As iaanus pointed out on Reddit, we should note that if the `unique_ptr` is moved out of the scope of the stack object, it would point to a resource that doesn't exist any more. Using the `unique_ptr` after this point causes a memory corruption.

And, like Bart noted in a comment, we should note that if the constructor of `House` were to take more than one argument then we should declare the construction of the `unique_ptr` in a separate statement, like this:

```
InstructionsUniquePtr instructions(new Sketch, deleteInstructions);
return House(move(instructions), getHouseNumber());
```

Indeed there could be a memory leak if an exception was thrown (cf Item 17 of Effective C++).

And also that when we don't use custom deleters, we shouldn't use new directly, but prefer `std::make_unique` that lets you pass the arguments for the construction of the pointed-to object.

Safety belt

If you're careful to avoid memory corruptions, using a custom deleter solves the initial problem but it induces a little change in the semantics of the passed argument, that can be at the source of many bugs.

In general, holding an `std::unique_ptr` means being its owner. And this means that it is OK to modify the pointed-to object. But in the case where the object comes from the stack (or from wherever else when it is passed with the no-op deleter), the unique pointer is just **holding a reference to an externally owned object**. In this case, you don't want the unique pointer to modify the object, because it would have side effects on the caller, and this would makes the code much more harder to reason about.

Therefore, when using this technique make sure to work on **pointer to const objects**:

```
using InstructionsUniquePtr =
       std::unique_ptr<const Instructions, void(*)(const
Instructions*)>;
```

and the deleters become:

```
void deleteInstructions(const Instructions* instructions){ delete
instructions;}
void doNotDeleteInstructions(const Instructions* instructions){}
```

This way, the unique pointer can't cause trouble outside of the class. This will save you a sizable amount of debugging.

Custom deleters are ugly

When you look at the expression to define a `unique_ptr` with a custom deleter:

```
std::unique_ptr<const Computer, void(*)(const Computer*)>;
```

This is dense enough that looking at it too long is dangerous for your eyes. We shouldn't spread such an expression all over the production code. So the natural way to go about this is to write an alias:

```
using ComputerConstPtr = std::unique_ptr<const Computer,
void(*)(const Computer*)>;
```

which fares better in an interface:

```
void plugIn(ComputerConstPtr computer);
```

But the ugliness is still there when we create new instances of the `unique_ptr` because we have to pass a deleter each time:

```
ComputerConstPtr myComputer(new Computer, deleteComputer);
```

Where we defined deleters:

```
void deleteComputer(const Computer* computer){ delete computer;}
void doNotDeleteComputer(const Computer* computer){}
```

This poses three issues. The first one is that we shouldn't have to specify anything in the case where we want the smart pointer to delete its resource. It's what smart pointers are made for in the first place.

Granted, this one it particular because it could have to not delete its resource for some occurrences. But why would the nominal case of deleting it be burdened because of the special case?

The second issue appears with namespaces, and comes down to verbosity. Imagine that our Computer type was inside a nested namespace, like often in production code:

```
namespace store
{
```

```
    namespace electronics
    {
        namespace gaming
        {
            class Computer
            {
                // ...
            };

         using ComputerConstPtr = std::unique_ptr<const Computer,
void(*)(const Computer*)>;
            void deleteComputer(const Computer* computer);
            void doNotDeleteComputer(const Computer* computer);
        }
    }
}
```

And then at call site:

```
store::electronics::gaming::ComputerConstPtr myComputer(new
store::electronics::gaming::Computer,
store::electronics::gaming::deleteComputer);
```

This is a tough line of code. And for saying little.

The last issue is that we define a delete and a `doNotDelete` function for **each type on which we want to custom deleters**. And even if their implementation has nothing specific to the type `Computer`, or any other type. However, note that even templating the deleters this way:

```
template<typename T>
void doDelete(const T* p)
{
    delete p;
}

template<typename T>
void doNotDeleteComputer(const T* x)
{
}
```

...doesn't make the code lighter. Indeed, we still need to specify the template type when instantiating the pointer:

```
store::electronics::gaming::ComputerConstPtr myComputer(new
store::electronics::gaming::Computer,
doDelete<store::electronics::gaming::Computer>);
```

A unique interface

Here is what Fluent C++ reader Sergio Adán suggested, and that can fix the two above issues: **use the same interface for all custom deleters on all types**.

This can be defined in another namespace, a technical one. Let's call this namespace `util` for the example.

Then in this namespace, we write all the common code that creates the custom `unique_ptr`. Let's call this helper `MakeConstUnique` for instance. Here is its code:

```
namespace util
{
    template<typename T>
    void doDelete(const T* p)
    {
        delete p;
    }

    template<typename T>
    void doNotDelete(const T* x)
    {
    }

    template<typename T>
    using CustomUniquePtr = std::unique_ptr<const T, void(*)(const
T*)>;

    template<typename T>
    auto MakeConstUnique(T* pointer)
    {
        return CustomUniquePtr<T>(pointer, doDelete<T>);
    }

    template<typename T>
    auto MakeConstUniqueNoDelete(T* pointer)
    {
        return CustomUniquePtr<T>(pointer, doNotDelete<T>);
```

```
    }
}
```

With this code, no need to define anything else to start using a `unique_ptr` on a particular type with custom deleters. For instance, to create an instance of a `unique_ptr` that does a delete of its resource when it gets out of scope, we write:

```
auto myComputer = util::MakeConstUnique(new
store::electronics::gaming::Computer);
```

And to create one that **does not** delete its resource:

```
auto myComputer = util::MakeConstUniqueNoDelete(new
store::electronics::gaming::Computer);
```

What is interesting about this interface is that:

- there is no longer any mention of delete in the common case,
- we can now use `auto`, thanks to the return type of `MakeConstUnique`.

Note that all this made us go down to **one occurrence of the namespace** of `Computer`, instead of three.

Specific deleters

Now what if, for some reason, we didn't want to call delete on the class Computer, but a particular dedicated function? This can happen in types coming from C for example:

```
void deleteComputer(const Computer* computer)
{
    specificFunctionThatFreesAComputer(computer);
}
```

To keep using `MakeConstUnique` with this type, we can make a total specialization of this template function for the type `Computer`. We could do this in the module defining Computer, by reopening the util namespace:

```
namespace util
{
    template<>
```

```
    auto MakeConstUnique(store::electronics::gaming::Computer*
pointer)
    {
    return
CustomUniquePtr<store::electronics::gaming::Computer>(pointer,
specificFunctionThatFreesAComputer);
    }
}
```

In this case, the client code probably doesn't allocate its pointer with new either.

Whichever way, a resource must be disposed of

Let's now test our interface, by adding a bit of logging in the Computer class:

```
class Computer
{
public:
    explicit Computer(std::string&& id) : id_(std::move(id)){}
    ~Computer(){std::cout << id_ << " destroyed\n";}
private:
    std::string id_;
};
```

And let's pass both a resource on the heap and a resource on the stack to our interface:

```
store::electronics::gaming::Computer c("stack-based computer");

auto myHeapBasedComputer = util::MakeConstUnique(new
store::electronics::gaming::Computer("heap-based computer"));

auto myStackBasedComputer = util::MakeConstUniqueNoDelete(&c);
```

When run this code outputs:

```
Heap-based computer destroyed
Stack-based computer destroyed
```

Changes of deleter during the life of a `unique_ptr`

Let's use a `unique_ptr` with a custom deleter and see **when that deleter can change** during the life of the `unique_ptr`.

Here is a toy example we use an `unique_ptr` on `int`, with a customisable deleter:

```
using IntDeleter = void(*)(int*);
using IntUniquePtr = std::unique_ptr<int, IntDeleter>;
```

One deleter is to be used for even numbers, and another one for odd numbers:

```
void deleteEvenNumber(int* pi)
{
    std::cout << "Delete even number " << *pi << '\n';
    delete pi;
}

void deleteOddNumber(int* pi)
{
    std::cout << "Delete odd number " << *pi << '\n';
    delete pi;
}
```

Assigning from another std::unique_ptr

Consider the following code:

```
IntUniquePtr p1(new int(42), deleteEvenNumber);
IntUniquePtr p2(new int(43), deleteOddNumber);
p1 = move(p2);
```

`p1`, that contains an even number with the appropriate deleter, is taking over the ownership of the resource in `p2`. The question is: how will it destroy this resource? Will it use the deleter it was built with, or rather bring over the deleter of `p2` along with the ownership of its resource?

Here is what this program outputs (the deleters are printing out the info - look at their code at the top of the article):

```
Delete even number 42
Delete odd number 43
```

Each resource is deleted with the correct deleter, which means that the assignment did bring over the deleter. This makes sense because the resources would not be disposed of with the correct deleter otherwise.

Resetting the pointer

Another way to change the resource contained in an `std::unique_ptr` is to call its reset method, like in the following simple example:

```
std::unique_ptr<int> p1(new int(42));
p1.reset(new int(43));
```

The reset method calls the deleter on the current resource (42), and then takes on the new one (43).

But the reset method only takes **one argument**, which is the new resource. It cannot be passed a deleter along with this new resource. For that reason, it can no longer be used directly in our example with even and odd numbers. Indeed, the following code:

```
IntUniquePtr p1(new int(42), deleteEvenNumber);
p1.reset(new int(43)); // can't pass deleteOddNumber
```

naturally outputs:

```
Delete even number 42
Delete even number 43
```

which is incorrect in our case.

In fact we could manually change the deleter in a separate statement, by exploiting the fact that the `get_deleter` method of `unique_ptr` returns the deleter by non-const reference (thanks to Marco Arena for pointing this out):

```
p1.get_deleter() = deleteOddNumber;
```

But why doesn't reset have a deleter argument? And how to hand over a new resource to an `std::unique_ptr` along with its appropriate deleter in a single statement?

Howard Hinnant, who is amongst many other things lead designer and author of the `std::unique_ptr` component, answers this question on Stack Overflow:

Here is how to use his answer in our initial example:

```
IntUniquePtr p1(new int(42), deleteEvenNumber);
p1 = IntUniquePtr(new int(43), deleteOddNumber);
```

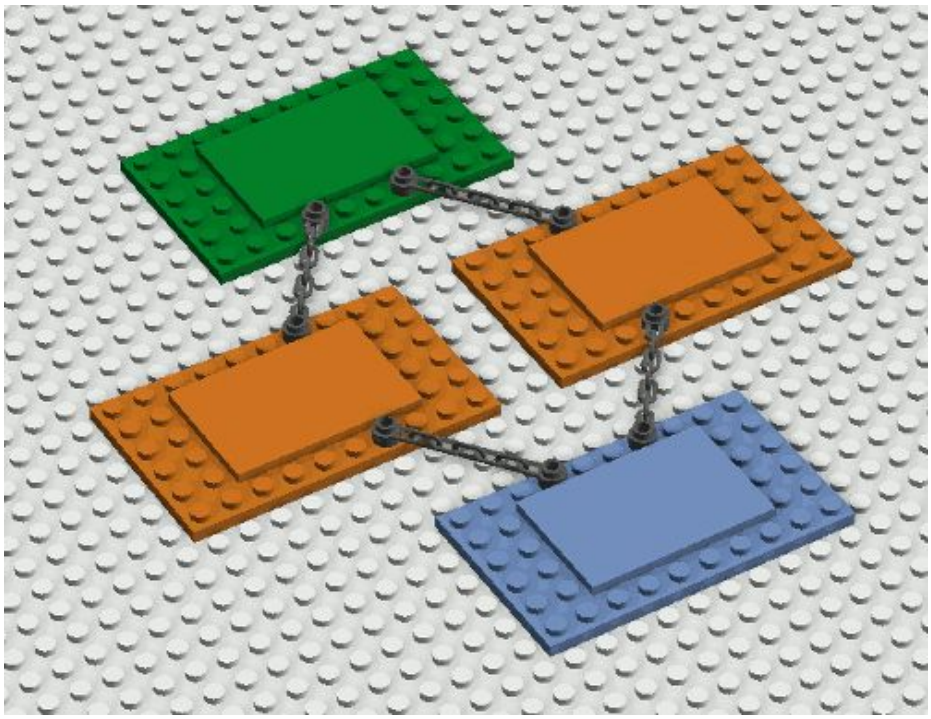
which gives the following desired output:

```
Delete even number 42
Delete odd number 43
```

How to Return a Smart Pointer AND Use Covariance

By Raoul Borges

in C++, it seems that we can have covariant return, or smart pointer return, but not both. Or can we?



Note: In this discussion, we will avoid type erasure as it generates a lot of boiler plate, which goes against our objective here. We will assume instead a fully generic OO solution. Also, this is not a discovery: Partial implementations of the techniques shown below can easily be found on the Internet. We are standing on the shoulders of the giants out there, and just compiling it all together in one post.

The problem: Covariant return type vs. smart pointers

C++ has support for covariant return type. That is, you can have the following code:

```
struct Base {};
struct Derived : Base {};
```

```
struct Parent
{
   virtual Base * foo();
} ;

struct Child : Parent
{
   virtual Derived * foo() override ;
} ;
```

Here, we expect the `foo` method from `Child` to return `Base *` for a successful overriding (and compilation!). With the covariant return type, we can actually replace `Base *` by any of its derived types. For example, `Derived *`.

This works for pointers, and for references... But the moment you try to use smart pointers:

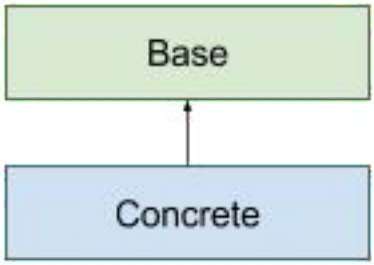
```
#include <memory>

struct Base {};
struct Derived : Base {};

struct Parent
{
   virtual std::unique_ptr<Base> foo();
} ;

struct Child : Parent
{
   virtual std::unique_ptr<Derived> foo() override ;
} ;
```

... the compiler generates an error.

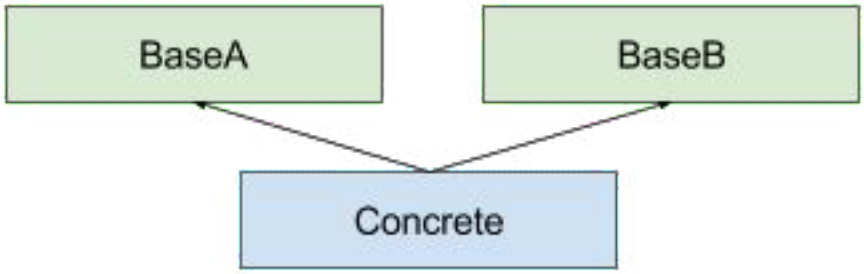
Use cases

Since the problem is general, let's take a wide panel of use cases with increasing complexity:

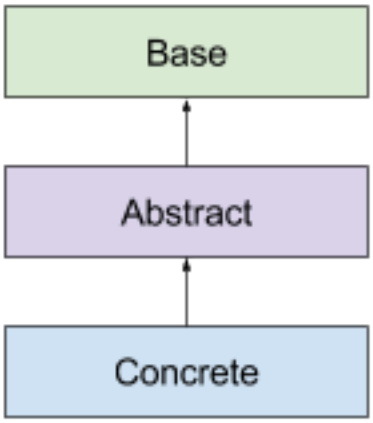
- Simple hierarchy:



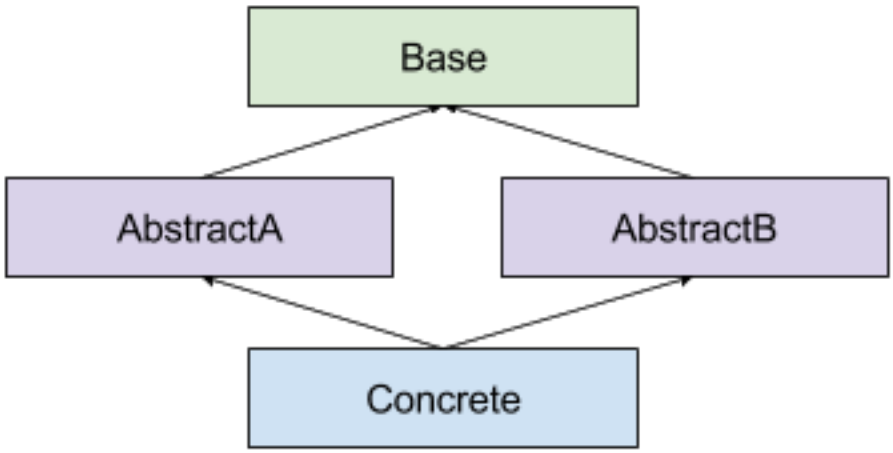
- Multiple inheritance:



- Deep hierarchy:



- Diamond inheritance:



By handling all those cases in a natural way, the solution should be usable for most production problems.

Preamble: Separation of concerns + private virtual function

Instead of having one clone member function handling everything, we will separate it into two member functions. In the following piece of code:

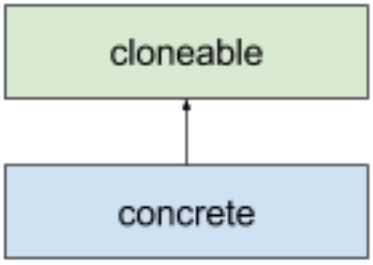
```
class some_class
{
public:
   std::unique_ptr<some_class> clone() const
   {
      return std::unique_ptr<some_class>(this->clone_impl());
   }

private:
   virtual some_class * clone_impl() const
   {
      return new some_class(*this) ;
   }
};
```

The first function, `clone_impl()`, does the actual work of cloning using the copy-constructor. It offers a strong guarantee (as long as the copy-constructor offers it), and transfers the ownership of the pointer to the newly created object. While this is usually unsafe, we assume that in this case no one can call this function except the `clone()` function, which is enforced by the private access of `clone_impl()`.

The second function, `clone()`, retrieves the pointer, and gives its ownership to a unique_ptr. This function cannot fail by itself, so it offers the same strong guarantee as `clone_impl()`.

Simple Hierarchy: Covariance + Name hiding

Using the technique above, we can now produce a simple OO hierarchy:




```
class cloneable
{
public:
   virtual ~cloneable() {}

   std::unique_ptr<cloneable> clone() const
   {
      return std::unique_ptr<cloneable>(this->clone_impl());
   }

private:
   virtual cloneable * clone_impl() const = 0;
};

////////////////////////////////////////////////////////////////////

class concrete : public cloneable
{
public:
   std::unique_ptr<concrete> clone() const
   {
      return std::unique_ptr<concrete>(this->clone_impl());
   }

private:
   virtual concrete * clone_impl() const override
   {
      return new concrete(*this);
```

```
    }
};

int main()
{
    std::unique_ptr<concrete> c = std::make_unique<concrete>();
    std::unique_ptr<concrete> cc = c->clone();

    cloneable * p = c.get();
    std::unique_ptr<clonable> pp = p->clone();
}
```

Do you see what we did, here?

By separating the concerns, we were able to use covariance at each level of the hierarchy to produce a `clone_impl` member function returning the exact type of pointer we wanted.

And using a little (usually) annoying feature in C++, name hiding (i.e. when declaring a name in a derived class, this name hides all the symbols with the same name in the base class), we hide (not override) the `clone()` member function to return a smart pointer of the exact type we wanted.

When cloning from a concrete, we obtain a `unique_ptr<concrete>`, and when cloning from a cloneable, we obtain a `unique_ptr<cloneable>`.

One could get uneasy at the idea of having a clone_impl member function using a RAII-unsafe transfer of ownership, but the problem is mitigated as the member function is private, and is called only by clone. This limits the risk as the user of the class can't call it by mistake.

This solves the problem but adds some amount of boilerplate code.

Simple Hierarchy, v2: Enter the CRTP

The CRTP is a C++ idiom that enables the injection of the derived class name into its templated base. You can learn about it in the series on CRTP on Fluent C++.

We will use it to declare methods with the correct derived prototypes in the CRTP base class, methods that will then be injected through inheritance into the derived class itself:

```
template <typename Derived, typename Base>
class clone_inherit<Derived, Base> : public Base
```

```
{
public:
   std::unique_ptr<Derived> clone() const
   {
  return
std::unique_ptr<Derived>(static_cast<Derived*>(this->clone_impl()));
   }

private:
   virtual clone_inherit * clone_impl() const override
   {
      return new Derived(*this);
   }
};
```

`clone_inherit` is a CRTP that knows its derived class, but also all its direct base class. It implements the covariant `clone_impl()` and hiding `clone()` member functions as usual, but they use casts to move through the hierarchy of types.

This enables us to change the concrete class defined above into:

```
class concrete
   : public clone_inherit<concrete, cloneable>
{
};

int main()
{
   std::unique_ptr<concrete> c = std::make_unique<concrete>();
   std::unique_ptr<concrete> cc = b->clone();

   cloneable * p = c.get();
   std::unique_ptr<clonable> pp = p->clone();
}
```

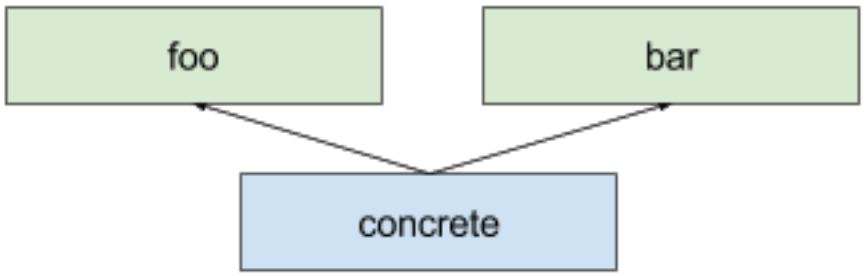
As you can see, the concrete class is now free of clutter.

This effectively adds a polymorphic and covariant `clone()` to a hierarchy of class.

This CRTP is the foundation of our general solution: Every next step will build upon it.

Multiple Inheritance: Variadic templates to the rescue

One complication of OO hierarchies is multiple inheritance.



In our case, how can we extend our solution to support the case where the concrete class inherits from two bases classes that both provide the same clone feature?

The solution first needs the two base classes, `foo` and `bar`, to offer the `clone`/`clone_impl` member functions:

```
class foo
{
public:
   virtual ~foo() = default;

   std::unique_ptr<foo> clone() const
   {
      return std::unique_ptr<foo>(this->clone_impl());
   }

private:
   virtual foo * clone_impl() const = 0;
};

////////////////////////////////////////////////////////////////////////

class bar
{
public:
   virtual ~bar() = default;

   std::unique_ptr<bar> clone() const
   {
      return std::unique_ptr<bar>(this->clone_impl());
   }
```

```
private:
   virtual bar * clone_impl() const = 0;
};
```

There's a bit of boilerplate, here, but we'll address it later. For now, we must solve the inheritance issue, and C++11 provides us with an easy solution: Variadic templates.

We only need to modify the `clone_inherit` CRTP to support it:

```
template <typename Derived, typename ... Bases>
class clone_inherit : public Bases...
{
public:
   std::unique_ptr<Derived> clone() const
   {
      return
std::unique_ptr<Derived>(static_cast<Derived*>(this->clone_impl()));
   }

private:
   virtual clone_inherit * clone_impl() const override
   {
      return new Derived(static_cast<const Derived & >(*this));
   }
};
```

We can now write our concrete class using it:

```
class concrete
   : public clone_inherit<concrete, foo, bar>
{
};
```

Last, but not least, we can use our classes with both covariance and smart pointers:

```
int main()
{
   std::unique_ptr<concrete> c = std::make_unique<concrete>();

   std::unique_ptr<concrete> cc = c->clone();

   foo * f = c.get();
```

```
   std::unique_ptr<foo> ff = f->clone();

   bar * b = c.get();
   std::unique_ptr<bar> bb = b->clone();
}
```

Multiple Inheritance v2: Specialization to the rescue

Now, let's address the clutter: Both `foo` and `bar` offer the same "cloneable" feature. And in our case, both should be virtually destructible.

The solution is to specialize `clone_inherit` to handle the case when no base class is desired, provide the virtual destructors, and have foo and bar inherit from it:

```
template <typename Derived, typename ... Bases>
class clone_inherit : public Bases...
{
public:
   virtual ~clone_inherit() = default;

   std::unique_ptr<Derived> clone() const
   {
      return
std::unique_ptr<Derived>(static_cast<Derived*>(this->clone_impl()));
   }

private:
   virtual clone_inherit * clone_impl() const override
   {
      return new Derived(static_cast<const Derived & >(*this));
   }
};

////////////////////////////////////////////////////////////////////////

template <typename Derived>
class clone_inherit<Derived>
{
public:
   virtual ~clone_inherit() = default;
```

```
   std::unique_ptr<Derived> clone() const
   {
      return
std::unique_ptr<Derived>(static_cast<Derived*>(this->clone_impl()));
   }

private:
   virtual clone_inherit * clone_impl() const = 0;
};
This way, we can now write:
class foo
   : public clone_inherit<foo>
{
};

//////////////////////////////////////////////////////////////////////

class bar
   : public clone_inherit<bar>
{
};

//////////////////////////////////////////////////////////////////////

class concrete
   : public clone_inherit<concrete, foo, bar>
{
};
```

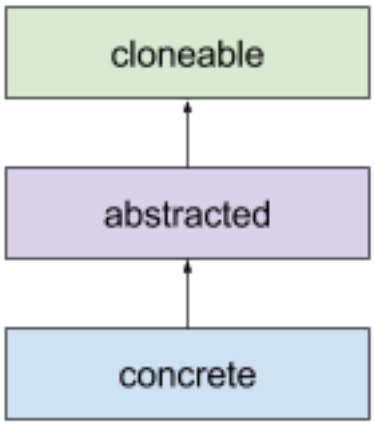
Last, but not least, we can use our classes with both covariance and smart pointers:

```
int main()
{
   std::unique_ptr<concrete> c = std::make_unique<concrete>();

   std::unique_ptr<concrete> cc = c->clone();

   foo * f = c.get();
   std::unique_ptr<foo> ff = f->clone();

   bar * b = c.get();
   std::unique_ptr<bar> bb = b->clone();
```

```
}
```

Deep Hierarchy: Abstracting

Another complication of OO Hierarchies is that they can go deeper than two levels:



The thing is, as Scott Meyers advised us, non-leaf classes are not supposed to be instantiable by themselves (More Effective C++, item 33).

In our case, the `clone_impl` method in the non-leaf class must then be pure virtual.

Our solution must thus support the choice of declaring `clone_impl` pure virtual, or implemented.

First, we add a dedicated type who will be used to "mark" a type:

```
template <typename T>
class abstract_method
{
};
```

Then, we partially specialize the clone_inherit class again to use that type, which means (because of the previous specialization), 4 different clone_inherit implementations:

```
// general: inheritance + clone_impl implemented
template <typename Derived, typename ... Bases>
class clone_inherit : public Bases...
{
public:
   virtual ~clone_inherit() = default;
```

```
   std::unique_ptr<Derived> clone() const
   {
      return
std::unique_ptr<Derived>(static_cast<Derived*>(this->clone_impl()));
   }

private:
   virtual clone_inherit * clone_impl() const override
   {
      return new Derived(static_cast<const Derived & >(*this));
   }
};

////////////////////////////////////////////////////////////////////

// specialization: inheritance + clone_impl NOT implemented
template <typename Derived, typename ... Bases>
class clone_inherit<abstract_method<Derived>, Bases...> : public
Bases...
{
public:
   virtual ~clone_inherit() = default;

   std::unique_ptr<Derived> clone() const
   {
      return
std::unique_ptr<Derived>(static_cast<Derived*>(this->clone_impl()));
   }

private:
   virtual clone_inherit * clone_impl() const = 0;
};

////////////////////////////////////////////////////////////////////

// specialization: NO inheritance + clone_impl implemented
template <typename Derived>
class clone_inherit<Derived>
{
public:
   virtual ~clone_inherit() = default;
```

```
   std::unique_ptr<Derived> clone() const
   {
      return
std::unique_ptr<Derived>(static_cast<Derived*>(this->clone_impl()));
   }

private:
   virtual clone_inherit * clone_impl() const override
   {
      return new Derived(static_cast<const Derived & >(*this));
   }
};

////////////////////////////////////////////////////////////////////////

// specialization: NO inheritance + clone_impl NOT implemented
template <typename Derived>
class clone_inherit<abstract_method<Derived>>
{
public:
   virtual ~clone_inherit() = default;

   std::unique_ptr<Derived> clone() const
   {
      return
std::unique_ptr<Derived>(static_cast<Derived*>(this->clone_impl()));
   }

private:
   virtual clone_inherit * clone_impl() const = 0;
};
```

It starts to be is a lot of code, but this will enable the user to actually use the feature with no boilerplate at all, as demonstrated by the following code:

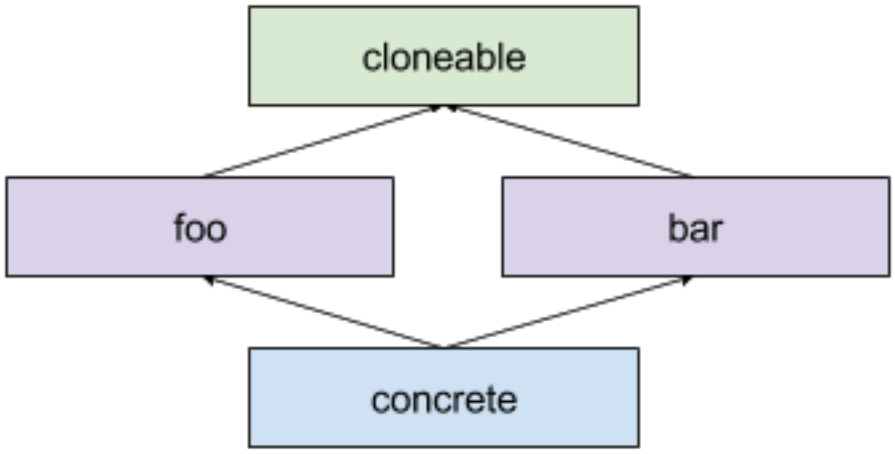
```
class cloneable
   : public clone_inherit<abstract_method<cloneable>>
{
};

////////////////////////////////////////////////////////////////////////

class abstracted
```

```
    : public clone_inherit<abstract_method<abstracted>, cloneable>
{
};

//////////////////////////////////////////////////////////////////////

class concrete
    : public clone_inherit<concrete, abstracted>
{
};

int main()
{
    std::unique_ptr<concrete> c = std::make_unique<concrete>();
    std::unique_ptr<concrete> cc = c->clone();

    abstracted * a = c.get();
    std::unique_ptr<abstracted> aa = a->clone();

    cloneable * p = c.get();
    std::unique_ptr<clonable> pp = p->clone();
}
```

Again, we succeeded in not cluttering too much the user code, and make this pattern scalable.

Diamond Inheritance: Virtual-ing

Yet another complication of OO Hierarchies is that we can have a diamond inheritance:



In C++, this means we have a choice to do: Is the base class inherited virtually, or not?

This choice must thus be provided by clone_inherit. The thing is, declaring a virtual inheritance is much more tricky because of the template parameter pack... Or is it?

Let's write a class that will do the indirection:

```
template <typename T>
class virtual_inherit_from : virtual public T
{
   using T::T;
};
```

This class actually applies the virtual inheritance to its base class `T`, which is exactly what we wanted. Now, all we need is to use this class to explicit our virtual inheritance need:

```
class foo
   : public clone_inherit<abstract_method<foo>,
virtual_inherit_from<cloneable>>
{
};

class bar
   : public clone_inherit<abstract_method<bar>,
virtual_inherit_from<cloneable>>
{
};

////////////////////////////////////////////////////////////////////

class concrete
   : public clone_inherit<concrete, foo, bar>
{
};

int main()
{
   std::unique_ptr<concrete> c = std::make_unique<concrete>();
   std::unique_ptr<concrete> cc = c->clone();

   foo * f = c.get();
   std::unique_ptr<foo> ff = c->clone();


   bar * b = c.get();
```

```
    std::unique_ptr<bar> bb = c->clone();

    cloneable * p = c.get();
    std::unique_ptr<cloneable> pp = p->clone();
}
```

Again, we succeeded in not cluttering too much the user code, and make this pattern scalable.
... *Et voilà!*

The whole package

The whole clone-ing code is:

```
//////////////////////////////////////////////////////////////////////

template <typename T>
class abstract_method
{
};

//////////////////////////////////////////////////////////////////////

template <typename T>
class virtual_inherit_from : virtual public T
{
    using T::T;
};

//////////////////////////////////////////////////////////////////////

template <typename Derived, typename ... Bases>
class clone_inherit : public Bases...
{
public:
    virtual ~clone_inherit() = default;

    std::unique_ptr<Derived> clone() const
    {
        return
std::unique_ptr<Derived>(static_cast<Derived*>(this->clone_impl()));
    }
```

```
protected:
   //         desirable, but impossible in C++17
   //         see: http://cplusplus.github.io/EWG/ewg-active.html#102
   // using typename... Bases::Bases;

private:
   virtual clone_inherit * clone_impl() const override
   {
      return new Derived(static_cast<const Derived & >(*this));
   }
};

////////////////////////////////////////////////////////////////////////

template <typename Derived, typename ... Bases>
class clone_inherit<abstract_method<Derived>, Bases...> : public
Bases...
{
public:
   virtual ~clone_inherit() = default;

   std::unique_ptr<Derived> clone() const
   {
      return std::unique_ptr<Derived>(static_cast<Derived
*>(this->clone_impl()));
   }

protected:
   //         desirable, but impossible in C++17
   //         see: http://cplusplus.github.io/EWG/ewg-active.html#102
   // using typename... Bases::Bases;

private:
   virtual clone_inherit * clone_impl() const = 0;
};

////////////////////////////////////////////////////////////////////////

template <typename Derived>
class clone_inherit<Derived>
{
public:
   virtual ~clone_inherit() = default;
```

```
   std::unique_ptr<Derived> clone() const
   {
      return
std::unique_ptr<Derived>(static_cast<Derived*>(this->clone_impl()));
   }

private:
   virtual clone_inherit * clone_impl() const override
   {
      return new Derived(static_cast<const Derived & >(*this));
   }
};

////////////////////////////////////////////////////////////////////////

template <typename Derived>
class clone_inherit<abstract_method<Derived>>
{
public:
   virtual ~clone_inherit() = default;

   std::unique_ptr<Derived> clone() const
   {
      return
std::unique_ptr<Derived>(static_cast<Derived*>(this->clone_impl()));
   }

private:
   virtual clone_inherit * clone_impl() const = 0;
};

////////////////////////////////////////////////////////////////////////
```

... and the user code is:

```
////////////////////////////////////////////////////////////////////////

class cloneable
   : public clone_inherit<abstract_method<cloneable>>
{
};
```

```
////////////////////////////////////////////////////////////////////////

class foo
   : public clone_inherit<abstract_method<foo>,
virtual_inherit_from<cloneable>>
{
};

////////////////////////////////////////////////////////////////////////

class bar
   : public clone_inherit<abstract_method<bar>,
virtual_inherit_from<cloneable>>
{
};

////////////////////////////////////////////////////////////////////////

class concrete
   : public clone_inherit<concrete, foo, bar>
{
};

////////////////////////////////////////////////////////////////////////
```

... which is not bad, all in all.

Would we use it in production code? While this set of techniques is interesting, it doesn't compile on Visual Studio 2017 (virtual inheritance, diamond and covariance don't mix well in Visual Studio), which is in our case, a showstopper. But it compiles at least with GCC 5.4.0+, as well as Clang 3.8.0+.

This set of techniques shows how, by using a clever but all-in-all simple combo of two orthogonal C++ paradigms, object oriented and generic (templates), we can factor out code to produce results with a concision that would have been difficult or impossible to get in other C-like languages.

It also shows a list of techniques (simulated covariance, inheritance indirection providing features) that can be applied elsewhere, each relying on C++ features assembled like lego pieces to produce the desired result.

Which is pretty cool IMHO.
:-)