# THE CRTP

- **The Curiously Recurring Template Pattern (CRTP)**
- **What the Curiously Recurring Template Pattern can bring to your code**
- **An Implementation Helper For The Curiously Recurring Template**
- **Mixin Classes: The Yang of the CRTP**
- **How to Turn a Hierarchy of Virtual Methods into a CRTP**

Fluent {C++}

# The Curiously Recurring Template Pattern (CRTP)

The Curiously Recurring Template Pattern (CRTP) is a C++ idiom whose name was coined by James Coplien in 1995, in early C++ template code.

The "C" in CRTP made it travel the years in the C++ community by being this: a Curiosity. We often find definitions of what CRTP is, and it is indeed an intriguing construct.

But what's even more interesting is what CRTP *means* in code, as in what you can express and achieve by using it, and this is the point of this series.

If you have used CRTP in your own code then you certainly know what it is useful for. In this case you may know most of what's written in this series of posts (though you may want to give a quick glance to episode #2, just to make sure we're in line), and you can just skip over to episode #3 where I propose a generic helper for CRTP, that I found helpful when using it in code.

## What CRTP is

The CRTP consists in:

- inheriting from a template class,
- use the derived class itself as a template parameter of the base class.

This is what it looks like in code:

```
template <typename T>
class Base
{
    ...
};

class Derived : public Base<Derived>
{
    ...
};
```

The purpose of doing this is using the derived class in the base class. From the perspective of the base object, the derived object is itself, but downcasted. Therefore the base class can access the derived class by static_casting itself into the derived class.

```
template <typename T>
class Base
{
public:
    void doSomething()
    {
        T& derived = static_cast<T&>(*this);
        use derived...
    }
};
```

Note that contrary to typical casts to derived class, we don't use `dynamic_cast` here. A `dynamic_cast` is used when you want to make sure at run-time that the derived class you are casting into is the correct one. But here we don't need this guarantee: the Base class is *designed* to be inherited from by its template parameter, and by nothing else. Therefore it takes this as an assumption, and a `static_cast` is enough.

# What could go wrong

If two classes happen to derive from the same CRTP base class we likely get to undefined behaviour when the CRTP will try to use the wrong class:

```
class Derived1 : public Base<Derived1>
{
    ...
};

class Derived2 : public Base<Derived1> // bug in this line of code
{
    ...
};
```

There is a solution to prevent this, that has been proposed by **Marek Kurdej** in the comments section. It consists in adding a private constructor in the base class, and making the base class friend with the templated class:

```
template <typename T>
class Base
```

```
{
public:
    // ...
private:
    Base(){};
    friend T;
};
```

Indeed, the constructors of the derived class have to call the constructor of the base class (even if you don't write it explicitly in the code, the compiler will do his best to do it for you). Since the constructor in the base class is private, no one can access it except the friend classes. And the only friend class is... the template class! So if the derived class is different from the template class, the code doesn't compile. Neat, right?

Another risk with CRTP is that methods in the derived class will **hide** methods from the base class with the same name. As explained in Effective C++ Item 33, the reason for that is that these methods are not virtual. Therefore you want to be careful not to have identical names in the base and derived classes:

```
class Derived : public Base<Derived>
{
public:
    void doSomething(); // oops this hides the doSomething methods
from the base class !
}
```

The first time I was shown CRTP my initial reaction was: "wait, I didn't get it". Then I saw it a couple of other times and I got it. So if you didn't get how it works, just re-read section 1 a couple of times, and that should do it (if it doesn't just get in touch and I'll be happy to talk with you about it).

To tell you the truth, I started by writing a huge blog post about CRTP, which would have been daunting to read completely I think. For that reason I decided to split it up into several logical parts, which constitute the episodes of this series. This post was relatively short, but was necessary to put the basics in place.

Next up: how the CRTP can be useful to your code.

# What the Curiously Recurring Template Pattern can bring to your code

After having defined the basics on the CRTP in episode #1 of the series, let's now consider how the CRTP can be helpful in day-to-day code.
The episodes in this series are:

- The CRTP, episode One: Definition
- The CRTP, episode Two: What the CRTP can bring to your code
- The CRTP, episode Three: An implementation helper for the CRTP

I don't know about you, but the first few times I figured how the CRTP worked I ended up forgetting soon after, and in the end could never remember what the CRTP exactly was. This happened because a lot of definitions of CRTP stop there, and don't show you *what value* the CRTP can bring to your code.

But there are several ways the CRTP can be useful. Here I am presenting the one that I see most in code, **Adding Functionality**, and another one that is interesting but that I don't encounter as often: creating **Static Interfaces**.

In order to make the code examples shorter, I have omitted the private-constructor-and-template-friend trick seen in episode One. But in practice you would find it useful to prevent the wrong class from being passed to the CRTP template.

# Adding functionality

Some classes provide generic functionality, that can be re-used by many other classes.

To illustrate this, let's take the example of a class representing a sensitivity. A sensitivity is a measure that quantifies how much a given output would be impacted if a given input to compute it were to vary by a certain amount. This notion is related to derivatives. Anyway if

you're not (or no longer) familiar with maths, fear not: the following doesn't depend on mathematical aspects, the only thing that matters for the example is that a sensitivity has a **value**.

```
class Sensitivity
{
public:
    double getValue() const;
    void setValue(double value);
    // rest of the sensitivity's rich interface...
};
```

Now we want to add helper operations for this sensitivity, like scaling it (multiplying it by a constant value), and say squaring it or setting it to the opposite value (unary minus). We can add the corresponding methods in the interface. I realize that in this case it would be good practice to implement these functionalities as non-member non-friend functions, but bear with me just a moment and let's implement them as methods, in order to illustrate the point that is coming afterwards. We will come back to this.

```
class Sensitivity
{
public:
    double getValue() const;
    void setValue(double value);

    void scale(double multiplicator)
    {
        setValue(getValue() * multiplicator);
    }
    void square()
    {
        setValue(getValue() * getValue());
    }
    void setToOpposite()
    {
        scale(-1);
    };

    // rest of the sensitivity's rich interface...
};
```

So far so good. But imagine now that we have another class, that also has a value, and that also needs the 3 numerical capabilities above. Should we copy and paste the 3 implementations over to the new class?

By now I can almost hear some of you scream to use template non-member functions, that would accept any class and be done with it. Please bear with me just another moment, we will get there I promise.

This is where the CRTP comes into play. Here we can factor out the 3 numerical functions into a separate class:

```
template <typename T>
struct NumericalFunctions
{
    void scale(double multiplicator);
    void square();
    void setToOpposite();
};
```

and use the CRTP to allow `Sensitivity` to use it:

```
class Sensitivity : public NumericalFunctions<Sensitivity>
{
public:
    double getValue() const;
    void setValue(double value);
    // rest of the sensitivity's rich interface...
};
```

For this to work, the implementation of the 3 numerical methods need to access the `getValue` and `setValue` methods from the `Sensitivity` class:

```
template <typename T>
struct NumericalFunctions
{
    void scale(double multiplicator)
    {
        T& underlying = static_cast<T&>(*this);
        underlying.setValue(underlying.getValue() * multiplicator);
    }
    void square()
    {
        T& underlying = static_cast<T&>(*this);
        underlying.setValue(underlying.getValue() *
underlying.getValue());
    }
    void setToOpposite()
    {
        scale(-1);
    };
};
```

This way we effectively added functionality to the initial `Sensitivity` class by using the CRTP. And this class can be inherited from by other classes, by using the same technique.

## Why not non-member template functions?

Ah, there we are.

Why not use template non-member functions that could operate on any class, including `Sensitivity` and other candidates for numerical operations? They could look like the following:

```
template <typename T>
void scale(T& object, double multiplicator)
{
    object.setValue(object.getValue() * multiplicator);
}

template <typename T>
void square(T& object)
{
    object.setValue(object.getValue() * object.getValue());
}

template <typename T>
void setToOpposite(T& object)
{
    object.scale(object, -1);
}
```

What's all the fuss with the CRTP?

There is at least one argument for using the CRTP over non-member template functions: **the CRTP shows in the interface**.

With the CRTP, you can see that `Sensitivity` offers the interface of `NumericalFunctions`:

```
class Sensitivity : public NumericalFunctions<Sensitivity>
{
public:
    double getValue() const;
    void setValue(double value);
    // rest of the sensitivity's rich interface...
};
```

And with the template non-member functions you don't. They would be hidden behind a `#include` somewhere.

And even if you knew the existence of these 3 non-member functions, you wouldn't have the guarantee that they would be compatible with a particular class (maybe they call `get()` or `getData()` instead of `getValue()`?). Whereas with the CRTP the code binding `Sensitivity` has already been compiled, so you know they have a compatible interface.

### Who's Your Interface Now?

A interesting point to note is that, although the CRTP uses inheritance, its usage of it does not have the same meaning as other cases of inheritance.

In general, a class deriving from another class expresses that the derived class somehow conceptually "is a" base class. The purpose is to use the base class in generic code, and to redirect calls to the base class over to code in the derived class.

With the CRTP the situation is radically different. The derived class does not express the fact it "is a" base class. Rather, it **expands its interface** by inherting from the base class, in order to add more functionality. In this case it makes sense to use the derived class directly, and to never use the base class (which is true for this usage of the CRTP, but not the one described below on static interfaces).

Therefore the base class is not the interface, and the derived class is not the implementation. Rather, it is the other way around: the base class uses the derived class methods (such as `getValue` and `setValue`). In this regard, **the derived class offers an interface to the base class**. This illustrates again the fact that inheritance in the context of the CRTP can express quite a different thing from classical inheritance.

## Static interfaces

The second usage of the CRTP is, as described in this answer on Stack Overflow, to create **static interfaces**. In this case, the base class does represent the interface and the

derived one does represent the implementation, as usual with polymorphism. But the difference with traditional polymorphism is that there is no `virtual` involved and all calls are resolved during compilation.

Here is how it works.

Let's take an CRTP base class modeling an amount, with one method, `getValue`:

```
template <typename T>
class Amount
{
public:
    double getValue() const
    {
        return static_cast<T const&>(*this).getValue();
    }
};
```

Say we have two implementations for this interface: one that always returns a constant, and one whose value can be set. These two implementations inherit from the CRTP `Amount` base class:

```
class Constant42 : public Amount<Constant42>
{
public:
    double getValue() const {return 42;}
};

class Variable : public Amount<Variable>
{
public:
    explicit Variable(int value) : value_(value) {}
    double getValue() const {return value_;}
private:
    int value_;
};
```

Finally, let's build a client for the interface, that takes an amount and that prints it to the console:

```
template<typename T>
void print(Amount<T> const& amount)
{
    std::cout << amount.getValue() << '\n';
}
```

The function can be called with either one of the two implementations:

```
Constant42 c42;
print(c42);
Variable v(43);
print(v);
```

and does the right thing:

```
42
43
```

The most important thing to note is that, although the `Amount` class is used polymorphically, there isn't any `virtual` in the code. This means that the polymorphic call has been resolved **at compile-time**, thus avoiding the run-time cost of virtual functions. For more about this impact on performance you can see the study Eli Bendersky made on his (great) website.

From a design point of view, we were able to avoid the virtual calls here because the information of which class to use was **available at compile-time**. And like we saw in the Extract Interface refactoring at compile-time, when you know the info, why wait until the last moment to use it?

EDIT: As u/quicknir pointed out on Reddit, this technique is not the best one for static interfaces, and nowhere as good as what concepts are expected to bring. Indeed, the CRTP forces to inherit from the interface, while concepts also specify requirements on types, but without coupling them with a specific interface. This allows independent libraries to work together.

Next up: how to make the implementation of the CRTP easy in practice.

# An Implementation Helper For The Curiously Recurring Template

In this final episode of the series on the Curiously Recuring Template Pattern, let's see an implementation that makes it easier to write CRTP classes.

In case you missed an episode in the series, here they are:

- The CRTP, episode One: Definition
- The CRTP, episode Two: What the CRTP can bring to your code
- The CRTP, episode Three: An implementation helper for the CRTP

## Getting rid of `static_cast`

Writing repeated `static_cast`s in CRTP base classes quickly becomes cumbersome, as it does not add much meaning to the code:

```
template <typename T>
struct NumericalFunctions
{
    void scale(double multiplicator)
    {
        T& underlying = static_cast<T&>(*this);
        underlying.setValue(underlying.getValue() * multiplicator);
    }
    ...
};
```

It would be nice to factor out these `static_cast`s. This can be achieved by forwarding the underlying type to a higher hierarchy level:

```
template <typename T>
struct crtp
{
    T& underlying() { return static_cast<T&>(*this); }
    T const& underlying() const { return static_cast<T
const&>(*this); }
};
```

Plus it deals with the case where the underlying object is const, which we hadn't mentioned yet.

This helper can be used the following way:

```
template <typename T>
struct NumericalFunctions : crtp<T>
{
    void scale(double multiplicator)
    {
        this->underlying().setValue(this->underlying().getValue() *
multiplicator);
    }
    ...
};
```

Note that the `static_cast` is gone and a `this->` appeared. Without it the code would not compile. Indeed, the compiler is not sure where `underlying` is declared. Even if it *is* declared in the template class `crtp`, in theory nothing guarantees that this template class won't be specialized and rewritten on a particular type, that would not expose an `underlying` method. For that reason, names in template base classes are ignored in C++.

Using `this->` is a way to include them back in the scope of functions considered to resolve the call. There are other ways to do it, although they are arguably not as adapted to this situation. In any case, you can read all about this topic in Effective C++ Item 43.

Anyway, the above code relieves you from writing the `static_cast`s, which become really cumbersome when they are several of them.

All this works if you class only add one functionality via CRTP, but it stops working if there are more.

# Adding several functionalities with CRTP

For the sake of the example let's split our CRTP classes into two: one that scales values and one that squares them:

```
template <typename T>
struct Scale : crtp<T>
{
    void scale(double multiplicator)
    {
        this->underlying().setValue(this->underlying().getValue() *
multiplicator);
    }
};

template <typename T>
struct Square : crtp<T>
{
    void square()
    {
        this->underlying().setValue(this->underlying().getValue() *
this->underlying().getValue());
    }
};
```

And add these two functionalities to the `Sensitivity` class:

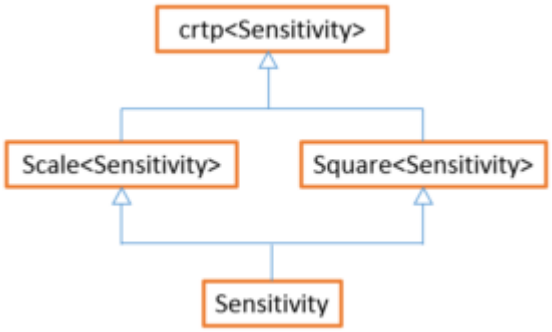
```
class Sensitivity : public Scale<Sensitivity>, public
Square<Sensitivity>
{
public:
    double getValue() const { return value_; }
    void setValue(double value) { value_ = value; }

private:
    double value_;
};
```

This looks ok at first glance but does not compile as soon as we call a method of either one of the base class!

```
error: 'crtp<Sensitivity>' is an ambiguous base of 'Sensitivity'
```

The reason is that we have a diamond inheritance here:




I tried to solve this with virtual inheritance at first, but quickly gave this up because I didn't find how to do it simply and without impacting the clients of the `crtp` class. If you have a suggestion, please, voice it!

Another approach is to steer away from the diamond inheritance (which sounds like a good idea), by having every functionality (scale, square) inherit from its own `crtp` class. And this can be achieved by… CRTP!

Indeed, we can add a template parameter to the `crtp` class, corresponding to the base class. Note the addition of the `crtpType` template parameter.

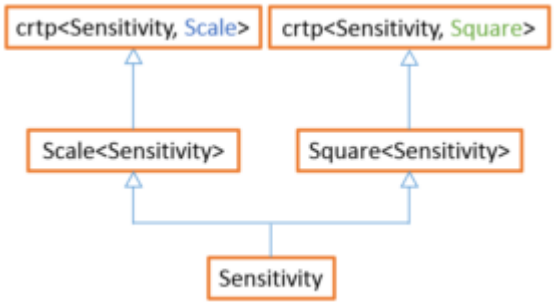
EDIT: as suggested by Daniel Houck in the comments section, the private-constructor-and-friend-with-derived technique should also be applied on this template template parameter here, because it forces `Scale` to inherit from the right crtp. Note that it doesn't force `Sensitivity` to inherit from the right CRTP though, so the friend and private constructor are still needed in `Scale` and `Square` (thanks to Amos Bird for pointing this out).

```
template <typename T, template<typename> class crtpType>
struct crtp
{
    T& underlying() { return static_cast<T&>(*this); }
    T const& underlying() const { return static_cast<T
const&>(*this); }
private:
    crtp(){}
    friend crtpType<T>;
};
```

Note that the template parameter is not just a `typename`, but rather a `template<typename> class`. This simply means that the parameter is not just a type, bu rather a template itself, templated over a type whose name is omitted. For example `crtpType` can be `Scale`.

This parameter is here to differentiate types only, and is not used in the implementation of `crtp` (except for the technical check in the friend declaration). Such an unused template parameter is called a “phantom type” (or to be more accurate here we could call it a “phantom template”).

The class hierarchy now looks like the following:




and we’re good to go.

A CRTP on a CRTP. Templates are *so* much fun.

# Mixin Classes: The Yang of the CRTP

Now that we're clear on how the CRTP works, let me share with you another technique involving templates that is complementary to the CRTP: **Mixin classes**. I learnt about mixin classes by watching Arthur O'Dwyer's Template Normal Programming talk at CppCon (actually you can find them in the slides because they were skipped over during the presentation).

I find mixin classes interesting because they provide another approach to the CRTP to achieve something equivalent, and therefore provide a different trade-off.


## Plugging a generic functionality over your type

The main usage of the CRTP is to add a generic functionality to a particular class. Mixin classes do that too.

Mixin classes are **template classes** that define a generic behaviour, and are designed to **inherit** from the type you wish to plug their functionality onto.

Here is an example. Let's take a class representing the name of a person. It has a first name and a last name, and it can print out that name with a specific format:

```
class Name
{
public:
    Name(std::string firstName, std::string lastName)
      : firstName_(std::move(firstName))
      , lastName_(std::move(lastName)) {}

    void print() const
    {
        std::cout << lastName_ << ", " << firstName_ << '\n';
    }

private:
    std::string firstName_;
    std::string lastName_;
};
```

Here is a piece of code using it:

```
Name ned("Eddard", "Stark");
ned.print();
```

which outputs:

```
Stark, Eddard
```

Nothing spectacular so far, but here is a new requirement: we need to be able to print this name several times in a row.

We could add a `repeat` method to the `Name` class. But the concept of repeatedly call the `print` method is something that could apply to other classes, like a `PhoneNumber` class that could also have a `print()` method.

The idea of the mixin class is to isolate the generic functionality into its own class, template this class on the type we want to plug in onto, and derive from that type:

```
template<typename Printable>
```

```
struct RepeatPrint : Printable
{
    explicit RepeatPrint(Printable const& printable) :
Printable(printable) {}
    void repeat(unsigned int n) const
    {
        while (n-- > 0)
        {
            this->print();
        }
    }
};
```

In our example the `Name` class will play the role of `Printable`.

Note the `this->` in the implementation of the `repeat` method. Without it the code would not compile. Indeed, the compiler is not sure where `print` is declared: even if it *is* declared in the template class `Printable`, in theory nothing guarantees that this template class won't be specialized and rewritten on a particular type, that would not expose an `print` method. For that reason, names in template base classes are ignored in C++.

Using `this->` is a way to include them back in the scope of functions considered to resolve the call. There are other ways to do it, although they are arguably not as adapted to this situation. In any case, you can read all about this topic in Effective C++ Item 43.

To avoid specifying template arguments explicitly we use a function that deduces them:

```
template<typename Printable>
RepeatPrint<Printable> repeatPrint(Printable const& printable)
{
    return RepeatPrint<Printable>(printable);
}
```

And here is the client code:

```
Name ned("Eddard", "Stark");
repeatPrint(ned).repeat(10);
```

which outputs:

```
Stark, Eddard
Stark, Eddard
```

```
Stark, Eddard
Stark, Eddard
Stark, Eddard
Stark, Eddard
Stark, Eddard
Stark, Eddard
Stark, Eddard
Stark, Eddard
```

We can even change the names to get to code even more expressive:

```
Name ned("Eddard", "Stark");
repeatedlyPrint(ned).times(10);
```

(I'm changing the names only now in order to compare the previous code with the CRTP, for which these new names are not adapted.)

# The CRTP upside down

Mixin classes involve a mix of template and inheritance in order to plug a generic functionality onto an existing class. This kind of feels like the CRTP, doesn't it?

**Mixin classes are like the CRTP, but upside down**. Indeed our mixin class looks like this:

```
class Name
{
    ...
};
template<typename Printable>
struct RepeatPrint : Printable
{
    ...
};

repeatPrint(ned).repeat(10);
```

while the corresponding CRTP would rather look like this:

```
template<typename Printable>
```

```
struct RepeatPrint
{
   ...
};

class Name : public RepeatPrint<Name>
{
    ...
};

ned.repeat(10);
```

In fact, here is the whole implementation of the solution using the CRTP:

```
template<typename Printable>
struct RepeatPrint
{
    void repeat(unsigned int n) const
    {
        while (n-- > 0)
        {
            static_cast<Printable const&>(*this).print();
        }
    }
};

class Name : public RepeatPrint<Name>
{
public:
    Name(std::string firstName, std::string lastName)
      : firstName_(std::move(firstName))
      , lastName_(std::move(lastName)) {}

    void print() const
    {
        std::cout << lastName_ << ", " << firstName_ << '\n';
    }

private:
    std::string firstName_;
    std::string lastName_;
};

int main()
{
    Name ned("Eddard", "Stark");
    ned.repeat(10);
}
```

# So, CRTP or mixin class?

CRTP and mixin classes provide two approaches to the same problem: adding a generic functionality to an existing class, but with **different trade-offs**.

Here are the points where they differ:

The CRTP:

- impacts the definition of the existing class, because it has to inherit from the CRTP,
- client code uses the original class directly and benefits from its augmented functionalities.

The mixin class:

- leaves the original class unchanged,
- client code doesn't use the original class directly, it needs to wrap it into the mixin to use the augmented functionality,
- inherits from a the original class even if it doesn't have a virtual destructor. This is ok unless the mixin class is deleted polymorphically through a pointer to the original class.

Understanding these trade-off lets you choose the solution that fits best to a given situation.

There is more than that to the CRTP. If you want to know more about it, I've dedicated a whole series of posts to the CRTP, which has become quite popular.

Special thanks to Arthur for his talk, and also for taking the time to help me understand mixin classes.

# How to Turn a Hierarchy of Virtual Methods into a CRTP

After reading the series of posts on the CRTP, Fluent C++ reader Miguel Raggi reached out to me with the following email (reproduced with his permission):

*Dear Jonathan Boccara,*

*[...] After reading the posts on the curiously recurring template pattern, I'm wondering how to (expressively) implement this with 3 or more classes.*

*Say, you have 3 classes, A, B, C, and that C is derived from B which is derived from A, and, say, both B and A used to be pure virtual classes.*

*How would I convert this to CRTP? I have something similar to this that is currently suffering from some performance problems that go away if I copy and paste the code.*

```
struct A
{
    virtual ~A() = default;

    void bigAndSlow() const
    {
        // ...
        helperfunction1(); //in very inner loop, so performance
matters
        helperfunction2(); // same
        // ...
    }

    virtual void helperfunction1() const = 0;
    virtual void helperfunction2() const = 0;
};

struct B : public A
{
    void helperfunction1() const override;
};

struct C : public B
{
    void helperfunction2() const override;
};
```

```
int main()
{
    C c;
    c.bigAndSlow();
}
```

*I've done some tests with CRTP and it does speed things up considerably not having to do the virtual redirections, but I'm having trouble when you have 3 or more in a chain ☺*

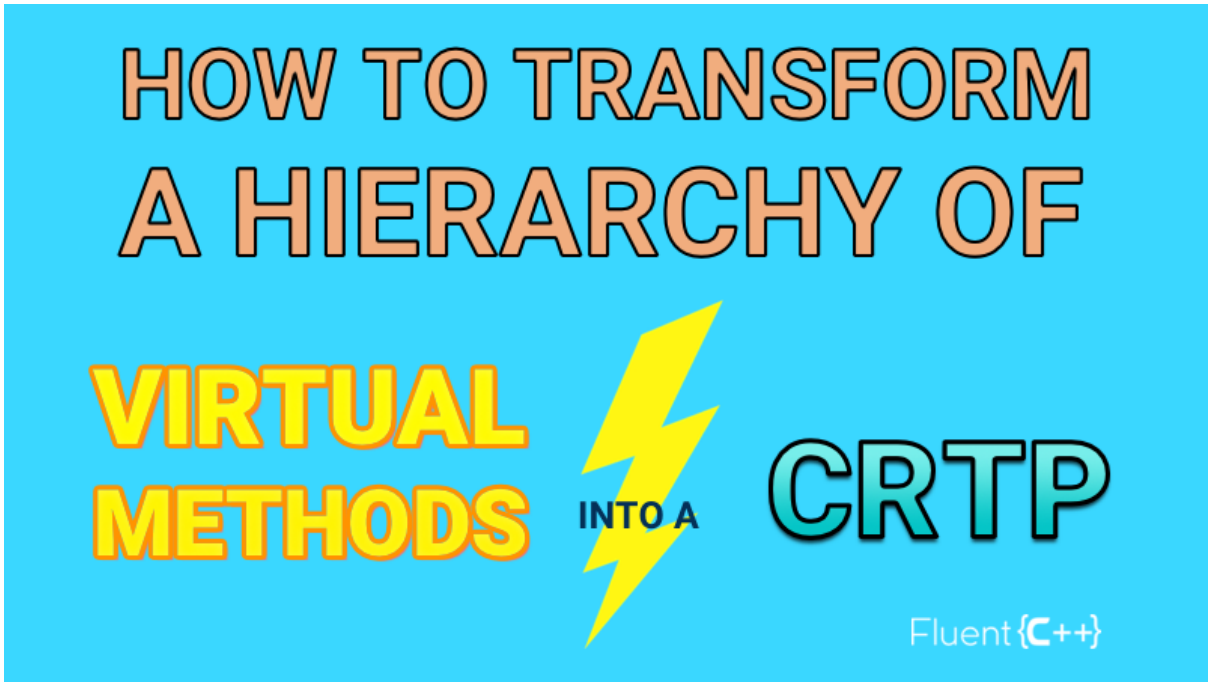
I want to thank Miguel for this great question.

It's a great question, as it aims at reducing the overload caused by something we don't need: here Miguel doesn't need the runtime polymorphism provided by virtual methods, and he doesn't want to pay for its cost.

This is part of the Programmer's Rights, protected by the Constitution of C++: no one shall pay for what they don't use.

So let's see how to implement static polymorphism in the above code. This question can be split into two parts:

- How to replace virtual methods by a CRTP,
- How to make a CRTP inherit from another CRTP




# From virtual methods to CRTP

Let's simplify the case of Miguel for the moment to keep only two levels in the hierarchy, `struct A` and `struct B` (we'll get back to the deeper hierarchy in a moment):

```
struct A
{
    virtual ~A() = default;

    void bigAndSlow() const
    {
        helperfunction1();
    }

    virtual void helperfunction1() const = 0;
};

struct B : public A
{
    void helperfunction1() const override{}
};
```

And the client code looks like this:

```
int main()
{
    B b;
    b.bigAndSlow();
```

```
}
```

The interface that the client code is invoking is the interface of `A`. And to be implemented, `A` needs some code behind the method `helperFunction1`, which is implemented in `B` here.

We can also have some polymorphic calling code, independent from `B`:

```
void f(A const& a)
{
    a.bigAndSlow(); // ends up in the code of B::helperFunction1
even though B doesn't appear here
}
```

The parallel with the CRTP goes like this: `B` has the functionality `helperFunction1`, and this functionality can **be extended**. This is what the CRTP is made for: adding functionality to a class.

The extension of functionality consists in a method that uses `helperFunction1`. In our starting example, that method was the one called `bigAndSlow`.

Now here is the resulting code using CRTP:

```
template<typename Derived>
struct A
{
    void bigAndSlow() const
    {
        return static_cast<Derived const&>(*this).helperfunction1();
    }
};

struct B : public A<B>
{
    void helperfunction1() const;
};
```

And to hide the ugly `static_cast` and to make the word “CRTP” appear in the interface, we can use the crtp helper:

```
template<typename Derived>
struct A : crtp<Derived, A>
{
    void bigAndSlow() const
    {
        return this->underlying().helperfunction1();
```

```
    }
};
```

Our calling code stays the same:

```
int main()
{
    B b;
    b.bigAndSlow();
}
```

And this code also ends up calling `helperFunction1` in `B`. But the virtual function mechanism, that incurs a certain cost (the size of a virtual pointer and the indirection of a virtual table) is gone.

We could also have some polymorphic code independent from `B`:

```
template<typename T>
void f(A<T> const& a)
{
    a.bigAndSlow(); // ends up in the code of B::helperFunction1
even though B doesn't appear here
}
```

And, just like with virtual functions, we can reuse `A` with other classes offering a `helperFunction1` methods, to augment their functionalities.

## Inheritance without a virtual destructor?

As you may have noticed, the virtual destructor is gone after this transformation. But is it OK? Is it safe to inherit from a class that does not have a virtual destructor?

Let's see. Writing this:

```
class A
{

};

class B : public A
{

};
```

is totally valid and legal C++.

The problems come when you delete a pointer to a base class that points to an object of a derived class:

```
B* b = new B;
A* pa = b;
delete pa; // undefinded behaviour
```

Indeed, the third line calls the destructor on `A`, which is not virtual so it does not redirect to the code of the destructor of `B`. The destructor of `B` never gets called. This is undefined behaviour.

Whereas with a virtual destructor, the call to the destructor on `A` is resolved by calling the destructor of `B` (just like when calling any other virtual method on `A` that is overriden in `B`). The destructor of `B` does its stuff and then calls the destructor of `A` (similarly to constructors of derived classes that call the constructor of their base class).

In our case, the class is not designed to be used with dynamic polymorphism (see below) and pointers to base class. So I haven't left the virtual destructor.

You could add it though, the price will only be an increased size of the object (so that the compiler can fit in a virtual pointer to redirect calls to the destructor), and arguably it would be less clear that this class is not meant to be used with dynamic polymorphism.

## Why pay for virtual functions at all?

It seems that the code using CRTP does just the same thing as the code using virtual methods, but it does not incur the cost of virtual methods. Is this to say that virtual methods are useless?

In this case, yes.

But in general, no.

Virtual methods are just more powerful than the CRTP, and therefore they cost more.

They're more powerful in the sense that, unlike the CRTP, they are able to discover the implementation of an interface at **each runtime call**. This is **dynamic polymorphism**.

For example, if you hold a pointer to an interface `A` that has virtual methods:

```
std::unique_ptr<A> pa;
```

You can use the polymorphic function `f`:

```
void f(A const& a)
{
    a.bigAndSlow();
}
```

on `pa`, even if the implementation of the interface changes at runtime.

To illustrate, let's assume that we have another class `B2` that inherits from `A`:

```
struct B2 : public A
{
    void helperfunction1() const override;
};
```

With dynamic polymorphism we can write the following code:

```
std::unique_ptr<A> pa = std::make_unique<B>(); // pa is a B
f(*pa); // calls B::helperFunction1

pa = std::make_unique<B2>(); // pa is now a B2
f(*pa); // calls B2::helperFunction1
```

The first call to `f` ends up calling the code of the class `B`, and the second one calls the code of the class `B2`.

This is an incredible flexibility. But it comes at a cost.

But if you don't need it, you don't have to pay for it. If you don't need the power of this dynamic polymorphism with virtual methods you can use **static polymorphism** with templates and (for example) CRTP.

# A deeper hierarchy of CRTPs

Now that we have our CRTP with one layer of inheritance, we can tackle Miguel's case and replace by a CRTP the following virtual methods:

```
struct A
{
    virtual ~A() = default;

    void bigAndSlow() const
    {
        helperfunction1();
        helperfunction2();
    }

    virtual void helperfunction1() const = 0;
    virtual void helperfunction2() const = 0;
};

struct B : public A
{
    void helperfunction1() const override;
};

struct C : public B
{
    void helperfunction2() const override;
};
```

Note that `B` overrides only one virtual method, `helperFunction1`, and leaves `helperFunction2` to be implemented by another class deeper down the hierarchy. Here, that class is `C`.

So to implement the CRTP in this hierarchy, we also need `B` to be a CRTP base class:

```
template<typename Derived>
struct A
{
    void bigAndSlow() const
    {
        static_cast<Derived const&>(*this).helperfunction1();
        static_cast<Derived const&>(*this).helperfunction2();
    }
};

template<typename Derived>
struct B : public A<B<Derived>>
{
    void helperfunction1() const;

    void helperfunction2() const
```

```
    {
        return static_cast<Derived const&>(*this).helperfunction2();
    };
};

struct C : public B<C>
{
    void helperfunction2() const;
};
```

(Note that we could use the crtp helper in only one of `A` or `B`. Indeed, if both inherit from `crtp` that defines the method `underlying` then this method becomes ambiguous for `B`)

EDIT: As pointed out by Simon Nivault in the comments sections, we can simplify this code. Indeed, no need for `B` to inherit from `A<B<Derived>>`: inheriting from `A<Derived>` is enough, because it makes `A` manipulate `C`, which also exposes the methods of `B` since it is is base class. This has the advantage of not needing any implementation of `helperFunction2` in `B`:

```
template<typename Derived>
struct A
{
    void bigAndSlow() const
    {
        static_cast<Derived const&>(*this).helperfunction1();
        static_cast<Derived const&>(*this).helperfunction2();
    }
};

template<typename Derived>
struct B : public A<Derived>
{
    void helperfunction1() const;
};

struct C : public B<C>
{
    void helperfunction2() const;
};
```

So this is a hierarchy of virtual methods turned into a hierarchy of CRTP!

# Let me know how I can help

If, like Miguel, you have a question about a topic we tackled on Fluent C++, or if you have a question related to expressive code in C++, you can write me at jonathan@fluentcpp.com. I'm always happy to hear from you.

I don't promise to have the answers, but I'll do my best to answer your question, and that could be by writing an article out of it!