

# THE VARIADIC CRTP

- **Variadic CRTP: An Opt-in for Class Features, at Compile Time**
- **How to Reduce the Code Bloat of a Variadic CRTP**
- **Variadic CRTP Packs: From Opt-in Skills to Opt-in Skillsets**
- **Removing Duplicates in C++ CRTP Base Classes**

Fluent{C++}

<b><u>VARIADIC CRTP: AN OPT-IN FOR CLASS FEATURES, AT COMPILE TIME</u></b>	<b>4</b>
THE CRTP	4
A VARIADIC CRTP	5
<b><u>HOW TO REDUCE THE CODE BLOAT OF A VARIADIC CRTP</u></b>	<b>11</b>
AN EVER-GROWING TEMPLATE NAME	11
ONE SKILLSET INSTEAD OF A PACK OF MULTIPLE SKILLS	13
A ONE LINE INSTANTIATION	15
A TRADE-OFF	17
<b><u>VARIADIC CRTP PACKS: FROM OPT-IN SKILLS TO OPT-IN SKILLSETS</u></b>	<b>19</b>
VARIADIC CRTP PACKS	20
<b><u>REMOVING DUPLICATES IN C++ CRTP BASE CLASSES</u></b>	<b>24</b>
AN ADDITIONAL LEVEL OF INDIRECTION	25
CHECKING IF A FEATURE IS IN A PACK	25
MERGING TWO PACKS TOGETHER	26
MERGING ANY NUMBER OF PACKS	28
REDUCING THE AMOUNT OF COMPARISONS	29
IMPROVEMENTS	31
<b><u>HOW TO EMULATE THE SPACESHIP OPERATOR BEFORE C++20 WITH CRTP ERROR! BOOKMARK NOT DEFINED.</u></b>	

EMULATING THE SPACESHIP OPERATOR

**ERROR! BOOKMARK NOT DEFINED.**

COMPARISONS USING CRTP

**ERROR! BOOKMARK NOT DEFINED.**

DOING A NON LEXICAL COMPARISON

**ERROR! BOOKMARK NOT DEFINED.**

PERFORMANCE IMPACT

**ERROR! BOOKMARK NOT DEFINED.**

ANOTHER IMPLEMENTATION WITH FEWER RESTRICTIONS

**ERROR! BOOKMARK NOT DEFINED.**

# Variadic CRTP: An Opt-in for Class Features, at Compile Time

The CRTP is a technique that allows to add extra features to a class. Variadic templates, brought by C++11, make this technique more powerful by adding a new aspect to it: composition.

Combining the CRTP with variadic templates generates customizable classes, by opting in for a various set of features, and with a expressive syntax.

It is used extensively in the customizable skills of the [NamedType](#) library.

Before introducing variadic templates in the CRTP, here is a brief recap about the CRTP itself.

## The CRTP

On its most basic description, the technical definition of the CRTP is a class that inherits from a template base class, passing itself as template parameter:

```
template<typename Derived>
class Base
{
};

class X : public Base<X>
{
};
```

If you're not familiar with the CRTP, take a moment to wrap your head around the above code.

Now beyond the technical definition, what is [the point of the CRTP](#)? In a word, the CRTP allows to plug in extra features to your class, that use its public interface:

```

template<typename Derived>
class ExtraFeature
{
public:
    void extraMethod()
    {
        auto derived = static_cast<Derived&>(*this);
        derived.basicMethod();
        derived.basicMethod();
        derived.basicMethod();
    }
};

class X : public ExtraFeature<X>
{
public:
    void basicMethod() {}
};

```

By inheriting from `ExtraFeature`, the class `x` has indeed gained a new feature:

`extraMethod`. Indeed, it is now part of the public interface of `x`, and we can write this call:

```

X x;
x.extraMethod();

```

The code of this `extraMethod` uses the public interface of `x`. Once again, if this is the first CRTP you see, take some time to go through the definition of `x` line by line. If you'd like to see more details, check out this [detailed post about the CRTP](#).

The point of having `ExtraFeature` decoupled from `x` is that it can be reused with any other class `y`, as long as it also exposes the public interface that `ExtraFeature` uses (here, `basicMethod`).

## A variadic CRTP

This was about adding **one** extra feature to our class `x`. Now how can we add **several** extra features?

One way would be to add other methods to the base class of the CRTP, `ExtraFeature`. It makes sense if those new methods relate to `ExtraFeature` and to whatever `extraMethod` does.

But if you'd like to add an unrelated feature, it would make more sense to package it into another CRTP base class:

```
template<typename Derived>
class ExtraFeature2
{
public:
    void extraMethod2()
    {
        auto derived = static_cast<Derived*>(*this);
        // does something else with derived.basicMethod() ...
    }
};

class X : public ExtraFeature<X>, public ExtraFeature2<X>
{
public:
    void basicMethod() {}
};
```

Now `x` has been augmented with both `extraMethod` and `extraMethod2`.

In some cases, such a design is enough. But some cases have way more than two extra features, and you want to choose which ones to define on `x` depending on the context (for example, this is the case in the `NamedType` library, where you can choose amongst a various sets of operators (`operator+`, `operator*`, `operator<<`, `operator int`, ...) which one to tack on a given strong type).

One way to go about this is to make `x` a template, with a variadic pack of template arguments. And the elements of the pack are the extra features to add to `x`.

But how should we write this pack? Let's write it with a set of `typename` parameters:

```
template<typename... Skills>
class X : public Skills...
{
public:
    void basicMethod() {}
};
```

The `Skills` are supposed to be the set of extra features to tack on to `x`.

## Aside: which name for the template parameters?

Before attempting to compile this code, a little note on the name, “Skills”.

In general customizing the aspects of a class with template parameters is called using “policies”, not “skills”. A policy is one particular aspect of a class, that can have several behaviours. That class is a template, and you choose a behaviour when you instantiate it (one of them can be a default).

To quote an example from *Modern C++ Design*, there are several ways to manage the life cycle of a `Singleton` object. Indeed, if a singleton depends on another singleton, their order of destruction matters. So the `Singleton` class defines a lifetime “policy” as a template parameter, that allows to choose between four options:

- the singleton is destroyed when the program exists,
- the singleton can be re-initialized after its destruction if is needed then (“phoenix singleton”),
- the singleton has a longevity that allows to customize the order of destructions of singletons,
- the singleton is never destroyed.

To implement this choice, the `Singleton` class has a `LifetimePolicy` template parameter:

```
template<LifetimePolicy, /* other points of customisation */>
class Singleton
{
    // ...
};
```

And this policy can be set to either one of `DefaultLifetime`, `PhoenixSingleton`, `SingletonWithLongevity`, `NoDestroy`. The beauty of this design is that there are other points of customization in the `Singleton` class, such as how it is created and how it handles multithreading. Those are two other policies, independent from the previous one but also implemented as template parameters.

For more about policy-based design, check out the book [Modern C++ Design](#).

In our case with the variadic CRTP, I don't think that the extra feature we're adding to `x` are policies, because they are not points of customization by themselves. Their *presence* is the customization. And since `x` can do new thing with them, I call them "skills".

Happy to discuss this naming point further in the comments section below.

## Template template parameters

Let's move on with the name `Skills` for the moment. So here was our attempt of fitting them into `x`:

```
template<typename... Skills>
class X : public Skills...
{
public:
    void basicMethod() {}
};
```

With, for example, this instantiation:

```
using X2 = X<ExtraFeature2>;
X2 x;
x.extraMethod2();
```

But this leads us right into a compilation error:

```
error: type/value mismatch at argument 1 in template parameter list for
template<class ...Skills> class X
    using X2 = X<ExtraFeature2>;
               ^
expected a type, got ExtraFeature2
```

Look at the last line of this error message. It says that `x` expected a type. Indeed, its template parameter is `typename`. But `ExtraFeature` is a template, not a type. A type would be `ExtraFeature<X2>`. But if we try this:

```
using X2 = X<ExtraFeature2<X2>>;
```

We now get:

```
error: 'X2' was not declared in this scope
```



The solution here is not to pass a type, but rather a template. So `x` should not expect `typename`s, but templates. Here is the correct code for `x`, and we review it in details just afterwards:

```
template<template<typename> typename... Skills>
class X : public Skills<X<Skills...>>...
{
public:
    void basicMethod();
};
```

So, step by step:

```
template<template<typename> typename... Skills>
```

The `template<typename> typename` indicates that we are expecting a template that takes one `typename`. Note that this is equivalent to `template<typename T> typename`: we expect a template that takes a `typename T`. But since we don't use `T` here, we can omit it.

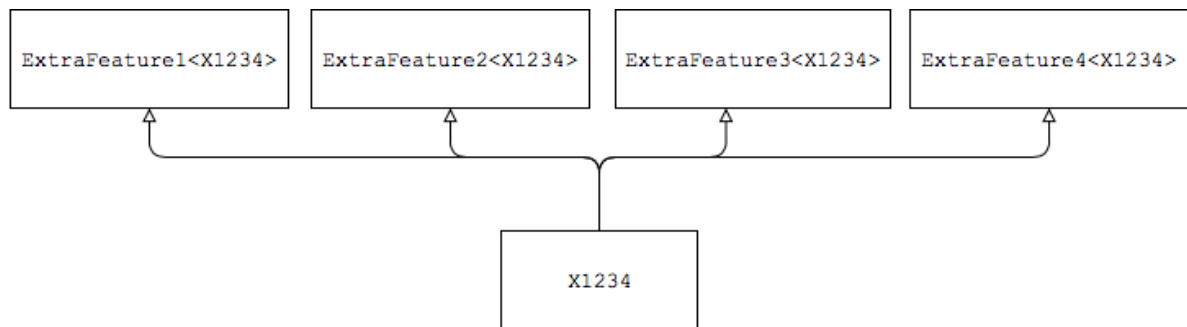
Note that before C++17, for some reason we couldn't use this exact expression. We had to use `class` instead of `typename` for the second `typename`: `template<typename> class`.

Finally, the variadic pack `...` allows to pass several such templates.

The second tricky bit is what `x` inherits from: `Skills<X<Skills...>>...`. Let's examine it from the inside out.

`Skills...` is the list of template parameters. `X<Skills...>` is the current class `x` that is being instantiated, because we merely repeat the template parameters of its instantiation. Then `Skills<X<Skills...>>...` is the pack of CRTP base class. It is equivalent to `ExtraFeature2<X<ExtraFeature2>>` in our example.

With more extra features, the relations between classes look like this:



Here is an example of calling code with one extra feature:

```
using X2 = X<ExtraFeature2>;
X2 x;
x.extraMethod2();
```

We can add to it other extra CRTP features, just by mentioning them in the template parameters list:

```
using X12 = X<ExtraFeature1, ExtraFeature2>;
X12 x;
x.extraMethod1();
x.extraMethod2();
```

Note the concise syntax.

So this is a variadic CRTP. It allows to add as many extra features as you want to a class, that enrich its interface by using its public method (including those of the other CRTP classes!).

One aspect to pay attention to is when there are many extra features, or if the extra features have complex types. Indeed, this can cause the type name of the instantiation of `x` to grow, sometimes too much. In a later post, we will see how to keep control of this and avoid the name to bloat.

# How to Reduce the Code Bloat of a Variadic CRTP

In the previous post we've seen how to introduce variadic templates into the [CRTP pattern](#), and how it allowed to create classes with various sets of opt-in features.

For instance, the class `x` would have a basic interface but also augment them by inheriting from a set of CRTP base classes:

```
template<template<typename> typename... Skills>
class X : public Skills<X<Skills...>>...
{
public:
    void basicMethod() { /*...*/ }
};
```

After a quick recap on the variadic CRTP, we're going to have a look at the generated type names, and see how to make them shorter if necessary.

## An ever-growing template name

The variadic CRTP allows to add extra features that enrich the interface of `x`, by using its public interface. Let's take the example of 4 such extra features:

```
template<typename Derived>
class ExtraFeature1
{
public:
    void extraMethod1()
    {
        auto& derived = static_cast<Derived&>(*this);
        derived.basicMethod();
        derived.basicMethod();
        derived.basicMethod();
    }
};

template<typename Derived>
class ExtraFeature2
{
public:
    void extraMethod2()
```

```

        {
            auto& derived = static_cast<Derived&>(*this);
            // does something else with derived.basicMethod() ...
        }
    };

template<typename Derived>
class ExtraFeature3
{
public:
    void extraMethod3()
    {
        auto& derived = static_cast<Derived&>(*this);
        // does something else with derived.basicMethod() ...
    }
};

template<typename Derived>
class ExtraFeature4
{
public:
    void extraMethod4()
    {
        auto& derived = static_cast<Derived&>(*this);
        // does something else with derived.basicMethod() ...
    }
};

```

This design allows to tack extra features on `x`, with a fairly concise syntax. For example, to add `ExtraFeature1` and `ExtraFeature4` to the interface of `x`, we write:

```
using X14 = X<ExtraFeature1, ExtraFeature4>;
```

And we can then call:

```

X14 x;
x.extraMethod1();
x.extraMethod4();

```

To add all four extra features, we instantiate `x` this way:

```
using X1234 = X<ExtraFeature1, ExtraFeature2, ExtraFeature3,
ExtraFeature4>;
```

Which lets us write the following code:

```

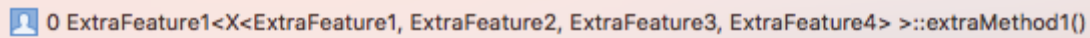
X1234 x;
x.extraMethod1();
x.extraMethod2();
x.extraMethod3();

```

```
x.extraMethod4();
```

x1234 is an alias. But what does its real name look like? Let's run the program in the debugger, and break the execution into the body of `extractMethod1` for example.

Making this experiment in XCode, the top line looks like this:

A screenshot of the Xcode call stack showing the top frame. The text is: 0 ExtraFeature1<X<ExtraFeature1, ExtraFeature2, ExtraFeature3, ExtraFeature4> >::extraMethod1(). The background is a light pink color.

0 ExtraFeature1<X<ExtraFeature1, ExtraFeature2, ExtraFeature3, ExtraFeature4> >::extraMethod1()

And if we put each extra feature into its own namespace, the top line of the call stack becomes:

A screenshot of the Xcode call stack showing the top frame with namespaces. The text is: 0 namespace1::ExtraFeature1<X<namespace1::ExtraFeature1, namespace2::ExtraFeature2, namespace3::ExtraFeature3, namespace4::ExtraFeature4> >::extraMethod1(). The background is a light grey color.

0 namespace1::ExtraFeature1<X<namespace1::ExtraFeature1, namespace2::ExtraFeature2, namespace3::ExtraFeature3, namespace4::ExtraFeature4> >::extraMethod1()

This could be a problem. Beyond the cumbersome symbol in the call stack, large template type names can have a detrimental effect on compilation time and binary size.

It could also be completely OK and unnoticeable. But for the cases where it's not, let's see how to keep this template name under control. The idea is to pack all the skills into one class, outside of `x`. I learnt about this idea from Nir Friedman on [Reddit](#), and I'm grateful to him for sharing that. Let's try to implement it.

## One skillset instead of a pack of multiple skills

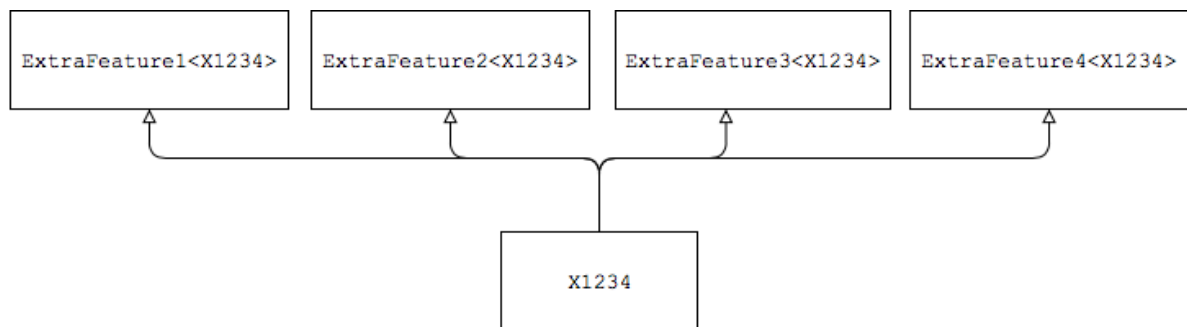
Here is our class `x` with the opt-in skills so far:

```
template<template<typename> typename... Skills>
class X : public Skills<X<Skills...>>...
{
public:
    void basicMethod() { /*...*/ }
};
```

An instantiation with all 4 extra features looks like this:

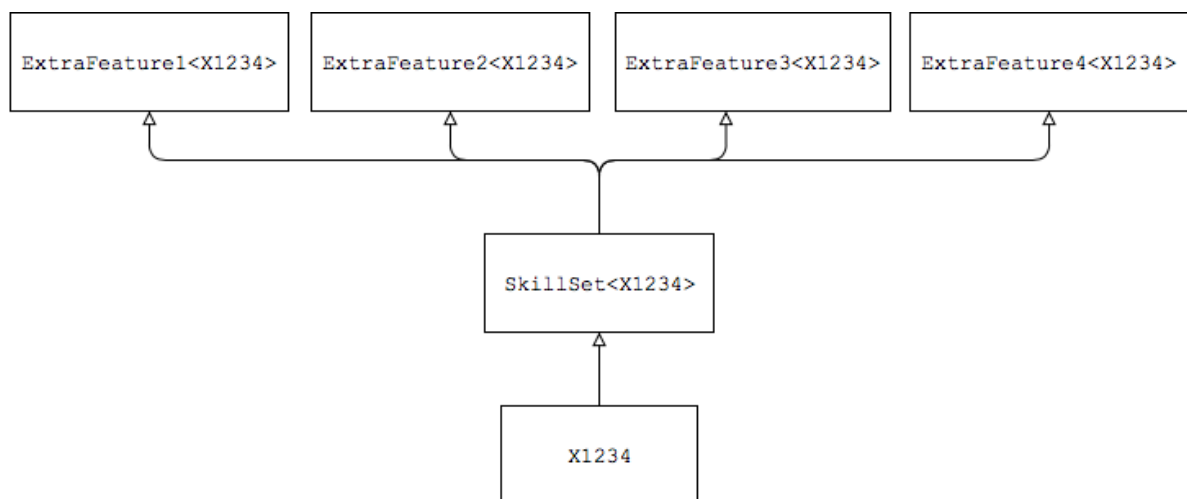
```
using X1234 = X<ExtraFeature1, ExtraFeature2, ExtraFeature3,
ExtraFeature4>;
```

Here are the inheritance relationships in a class diagram:



The types of the extra features are directly connected our class `x1234`, and this why they show in its type name.

What about adding an intermediary level, that would know the extra skills? It would be a sort of skill set. And `x1234` would only know of this one type, the skillset:



Let's modify the definition of `x` so that it only has one skill (the skillset, that groups them all):

```

template<template<typename> class SkillSet>
class X : public SkillSet<X<SkillSet>>
{
public:
    void basicMethod() { /*...*/ }
};
  
```

Then to define a CRTP skillset, we make it inherit from extra features. For example:

```

template<typename Derived>
  
```

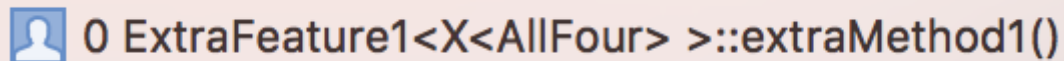
```
class AllFour : public ExtraFeature1<Derived>, public
ExtraFeature2<Derived>, public ExtraFeature3<Derived>, public
ExtraFeature4<Derived> {};
```

We use this skillset to instantiate `x`:

```
using X1234 = X<AllFour>;

X1234 x;
x.extraMethod1();
x.extraMethod2();
x.extraMethod3();
x.extraMethod4();
```

Let's now run this code in the debugger, and see what the type name looks like when we break into `extraMethod1`:



We now have the name of `x1234` under control! Its size no longer depends on the number or complexity of the extra features in the CRTP base classes.

Note how this is a different sort of skillsets than the one we saw in Variadic CRTP Packs: From Opt-in Skills to Opt-in Skillsets. There, the point of grouping skills relating together into skillsets was to make skills more discoverable for a user of `x`, and make the definition of `x` more concise.

This difference results in a different usage: there, `x` could inherit from several skillsets, along with other individual skills. Here, `x` inherit from one skillset that we design for it specifically, and that inherits from all the skills (and skillsets) we desire `x` to have.

## A one line instantiation

The type name is now under control, but the interface is less straightforward to use: we need to create a separate type and then use it:

```
template<typename Derived>
```

```
class AllFour : public ExtraFeature1<Derived>, public
ExtraFeature2<Derived>, public ExtraFeature3<Derived>, public
ExtraFeature4<Derived> {};

using X1234 = X<AllFour>;
```

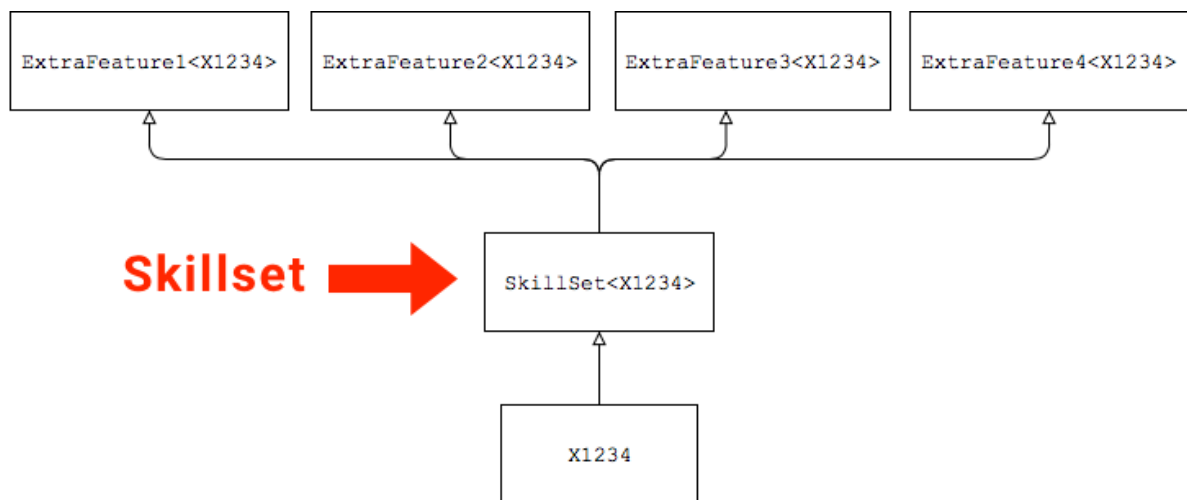
Compare this to the original syntax:

```
using X1234 = X<ExtraFeature1, ExtraFeature2, ExtraFeature3,
ExtraFeature4>;
```

It was more straightforward. But it doesn't compile any more, because x now expect only one template parameter, not four.

Could we still define x1234 in one line, for the cases where the size of the generated template name doesn't matter? Or put another way, can we instantiate a skillset within the definition of x1234?

Let's put up the class diagram involving the skillset again:



The skillset is a class template with one parameter (the derived class x), and that inherits from the extra features. So we would need a function that takes the desired skills, and generate a class template expecting one parameter.

It wouldn't be a function, but rather a meta-function, as in a function that takes and returns types, not objects. Even more, it would **take templates and return templates**.



In template meta-programming, meta-functions are represented as template structs. Their inputs are their template parameters, and their outputs their nested types. Here we want the template skills as inputs, and the template skillset as outputs.

Let's call that function `make_skills`. A common convention for the output template is to name the corresponding nested template `templ`:

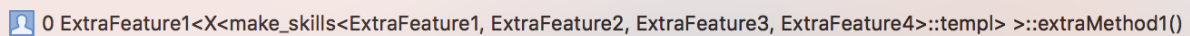
```
template<template<typename> class... Skills>
struct make_skills
{
    template<typename Derived>
    struct templ : Skills<Derived>...
    {

    };
};
```

We can then use it like so:

```
using X1234 = X<make_skills<ExtraFeature1, ExtraFeature2,
ExtraFeature3, ExtraFeature4>::templ>;
```

But here is what the generated type for X1234 then looks like in the debugger:

A screenshot of a debugger window showing the type of a variable named X1234. The type is displayed as '0 ExtraFeature1<X<make\_skills<ExtraFeature1, ExtraFeature2, ExtraFeature3, ExtraFeature4>::templ> >::extraMethod1()'. The text is in a light blue font on a dark background.

Indeed, now `X1234` knows again about the skills, because it passes them to the skillset class via `make_skills`.

## A trade-off

Has decoupling the skillset from `x` been an improvement to the design?

It has advantages and drawbacks. Its drawbacks are that `make_skills` make an even bigger typename for `x` than before we introduced a skillset, however we would use `make_skills` for the cases where the type name was not too long anyway. But its code is less direct to instantiate, with the ugly `::templ` sticking out.

But its advantages is that it leaves the flexibility to group all skills into a manually defined skillset, thus keeping the length of the type name under control. But the interface is less straightforward to use, with the separate type to define manually.

Have you used a variadic CRTP? Did you choose to separate the skillset? How did you go about it? Share your experiences, all feedback is welcome.

# Variadic CRTP Packs: From Opt-in Skills to Opt-in Skillsets

Last week we've seen the technique of the [variadic CRTP](#), that allowed to plug in generic extra features to a class.

For instance, we've seen the following class `x`:

```
template<template<typename> typename... Skills>
class X : public Skills<X<Skills...>>...
{
public:
    void basicMethod();
};
```

`x` can accept extra features that plug into its template parameters:

```
using X12 = X<ExtraFeature1, ExtraFeature2>;
```

To be compatible with `x`, each of those features follows the CRTP pattern:

```
template<typename Derived>
class ExtraFeature1
{
public:
    void extraMethod1()
    {
        auto derived = static_cast<Derived&>(*this);
        // uses derived.basicMethod()
    }
};

template<typename Derived>
class ExtraFeature2
{
public:
    void extraMethod2()
    {
        auto derived = static_cast<Derived&>(*this);
        // uses derived.basicMethod()
    }
};
```

Since each of these features is a member of the variadic pack of `x`'s template parameters, a natural name for this technique is variadic CRTP.

With it, `x` can be augmented with the methods coming from the extra features classes:

```
using X12 = X<ExtraFeature1, ExtraFeature2>;
X12 x;
x.extraMethod();
x.extraMethod2();
```

If you're not familiar with the variadic CRTP, you can read more about it in its [dedicated article](#). To extend the variadic CRTP, I'd like to focus today on a little addition to this pattern: grouping extra features into packs.

## Variadic CRTP packs

If you offer several extra features that can be plugged in your class `x`, like `NamedType` does with its skills for instance, it could make sense to bundle them into groups (which `NamedType` doesn't at the time of this writing, but it could make sense to refactor it in that way).

Indeed, bundling several related features into groups, or packs, has several advantages:

- it is less to learn for a user of `x`: they can only learn the groups, as opposed to learning every skill,
- it makes skills more discoverable: a user can explore groups, which is more logical than exploring the skills by alphabetical order or whatever,
- it makes the definition of `x` more readable: enumerating skillsets is shorter than enumerating skills.

So let's see how we could bundle several extra features into a pack, and pass that pack to `x` the same way that we'd pass individual skills.

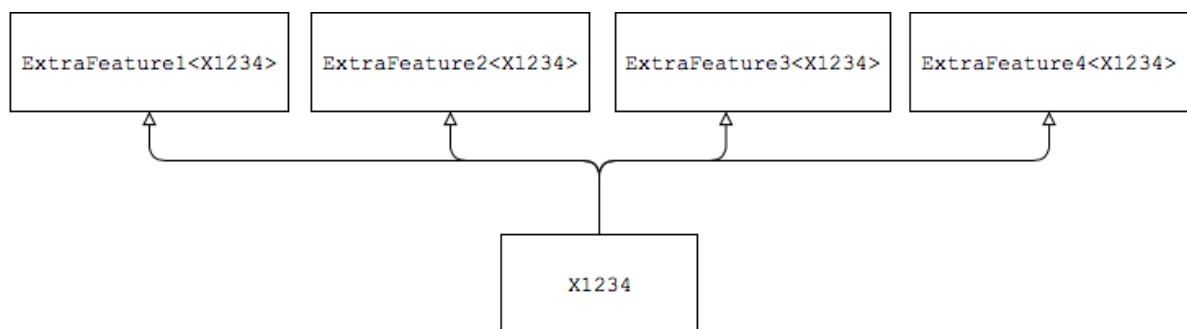
### Inheriting from packs

The trick is not a difficult one: it consists in using intermediary class in the inheritance hierarchy.

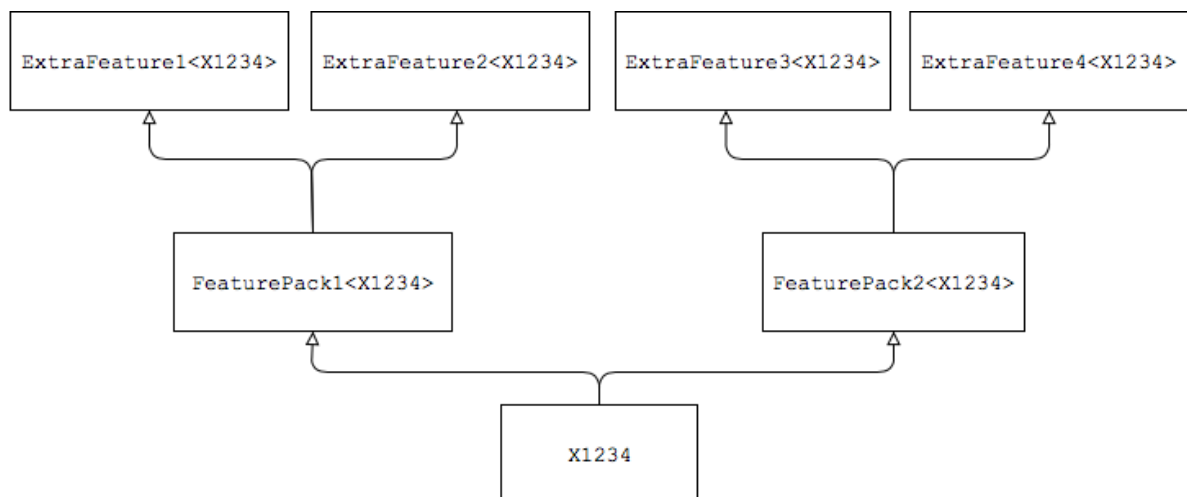
Indeed, the point of a feature pack is to behave as if  $x$  itself inherited from the features it contains. A natural way to do this is to make the pack inherits from the features, and  $x$  inherit from the pack.

To illustrate, let's consider 4 extra features, and say that the first two relate together, and the last two relate together too. So we'd like to have two packs: a first one with features 1 and 2, and a second one with features 3 and 4.

Without packs, the class hierarchy looks like this:



And by adding packs in:



Now let's see how to implement such feature packs.

## The implementation of a feature pack

We want packs to be CRTP classes (so that `x` inherits from them) and to inherit from the skill classes. So we have:

```
template<typename Derived>
struct FeaturePack1 : ExtraFeature1<Derived>,
ExtraFeature2<Derived> {};
```

And:

```
template<typename Derived>
struct FeaturePack1 : ExtraFeature1<Derived>, ExtraFeature2<Derived>
{};
```

`x` inherits from them through its template parameters:

```
using X1234 = X<FeaturePack1, FeaturePack2>;
```

Doing this augments `x` with the methods coming from all four extra features:

```
X1234 x;

x.extraMethod1();
x.extraMethod2();
x.extraMethod3();
x.extraMethod4();
```

## The composite design pattern

An interesting thing to note is that we haven't changed anything in `x` to allow for packs to plug in. This means that we can still add individual features to `x` along with the packs:

```
using X12345 = X<FeaturePack1, FeaturePack2, ExtraFeature5>;

X12345 x;

x.extraMethod1();
x.extraMethod2();
x.extraMethod3();
x.extraMethod4();
x.extraMethod5();
```

This looks like the **Composite** design pattern. Indeed, the Composite design pattern described in the classical GOF [book on Design Pattern](#) is about runtime polymorphism with `virtual` methods, but its spirit is the following: wholes and parts should look alike from the perspective of client code. And this is what the variadic CRTP is allowing here.

But the interest of skillsets does not stop here. One specific use of them allows to reduce the bloat of template symbols, and that's what we will see in a later post.

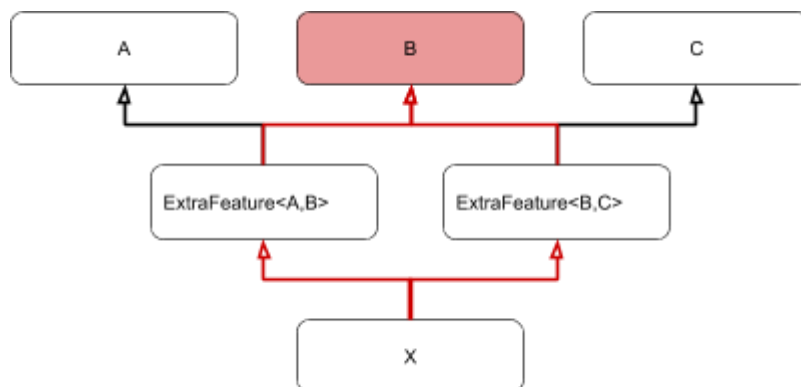
# Removing Duplicates in C++ CRTP Base Classes

*At the beginning of the summer, we talked on [Fluent C++](#) about [7 projects to get better at C++ during the summer](#). Reader Sergio Adán has taken up the challenge, and picked up Project #1 about how to avoid duplicates in a variadic CRTP. Today as summer is drawing to an end, Sergio shares with us his solution in a guest post!*

*Sergio Adán is a Spanish C++ programmer. He began programming when he was 5 years old and his parents offered him an Amstrad CPC. Sergio has been programming in C++ for six years and he really likes code looks clean and expressive.*

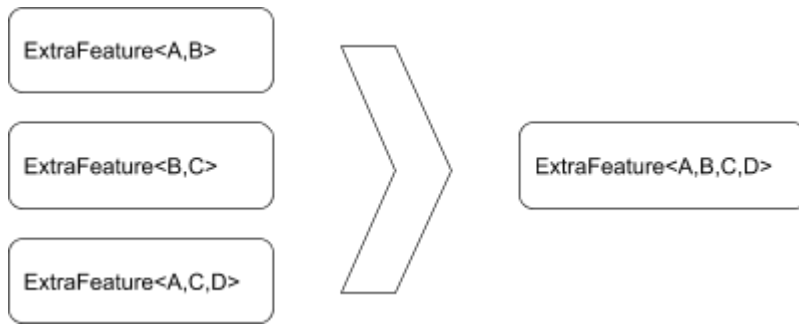
*Interested to write on [Fluent C++](#) too? Check out the [guest posting area](#).*

As we can see in the [original post](#), if some packs have the same feature, our class will inherit the same base class two or more times and then the direct call of the feature will fail:



We need to modify the inheritance to ensure each feature will be inherited only once. The solution I propose is to join, at compile time, all feature packs into a single pack removing all duplicates.





## An additional level of indirection

To perform some compile-time work on the set of skill packs so as to remove the duplicates among skills, we introduce an additional level of indirection: the `ExtraFeatures` class. This class takes the packs as template parameters and does some cutting work that we'll see in details just afterwards. Features packs such as `ExtraFeaturesA` use it to declare their set of skills.

```
template<typename Derived, template<typename> typename ... Features>
struct ExtraFeatures : Features<Derived>...
{ };
```

So once the declaration is in our project, feature packs must be declared as below:

```
template<typename Derived>
using ExtraFeaturesA =
ExtraFeatures<Derived, ExtraFeature1, ExtraFeature2>;

template<typename Derived>
using ExtraFeaturesB =
ExtraFeatures<Derived, ExtraFeature2, ExtraFeature3>;

template<typename Derived>
using ExtraFeaturesC =
ExtraFeatures<Derived, ExtraFeature1, ExtraFeature3>;
```

Let's now see how to remove duplicate skills across the packs.

## Checking if a feature is in a pack

As a first step we need a tool that checks if a given feature is already in a list. A first attempt could look like this:

```

template<typename Derived,
        template<typename> typename ToCheck,
        template<typename> typename Current,
        template<typename> typename ... Features>
constexpr bool HasFeature()
{
    if constexpr(
std::is_same<ToCheck<Derived>, Current<Derived>>::value )
        return true;
    else if constexpr( sizeof...(Features) == 0 )
        return false;
    else
        return HasFeature<Derived, ToCheck, Features...>();
}

```

The function `HasFeature` receives the type to be checked and a list of types. Then the function iterates over the list and check if the `ToCheck` template is in the list. The function works properly but it has a problem: it relies on recursion.

Compilers limit the maximum number of iterations done at compile time, and even if we remain with the authorised limits, recursion incurs more compilations time, so the common practice for operating on a list of types is to avoid recursion.

One solution is to use C++17's fold expressions:

```

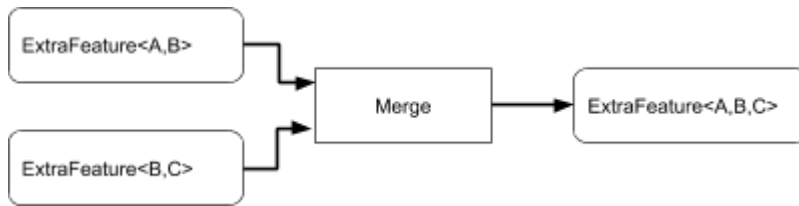
template<typename Derived,
        template<typename> typename ToCheck,
        template<typename> typename ... Features>
constexpr bool HasFeature()
{
    return (std::is_same<ToCheck<Derived>, Features<Derived>>::value ||
...);
}

```

The function now looks more simple and expressive, and it no longer uses recursion.

## Merging two packs together

Now we need an utility that merges two feature packs into a new one, ensuring that each feature exists only once in the new feature pack:



To implement this functionality we can start with a recursive approach again:

```

template<typename ...>
struct JoinTwoExtraFeatures;

template<typename Derived,
        template<typename> typename Feature,
        template<typename> typename ... Features1,
        template<typename> typename ... Features2>
struct JoinTwoExtraFeatures<
    ExtraFeatures<Derived, Features1...>,
    ExtraFeatures<Derived, Feature, Features2...>
>
{
    using type= typename
        std::conditional<
            HasFeature<Derived, Feature, Features1...>(),
            typename JoinTwoExtraFeatures<
                ExtraFeatures<Derived, Features1...>,
                ExtraFeatures<Derived, Features2...>
            >::type,
            typename JoinTwoExtraFeatures<
                ExtraFeatures<Derived, Features1..., Feature>,
                ExtraFeatures<Derived, Features2...>
            >::type
        >::type;
};

template<typename Derived,
        template<typename> typename ... Features1>
struct JoinTwoExtraFeatures<
    ExtraFeatures<Derived, Features1...>,
    ExtraFeatures<Derived>
>
{
    using type= ExtraFeatures<Derived, Features1...>;
};

```

But unlike `HasFeature` utility, I haven't been able to find a way to avoid the recursion. If you see how to refactor this code to remove the recursion, please let us know by leaving a comment below.

# Merging any number of packs

Now we are able to merge two feature packs into a new one. Our next step is to build an utility that merges *any number* of feature packs into a new one:

```
template<typename ...>
struct JoinExtraFeatures;

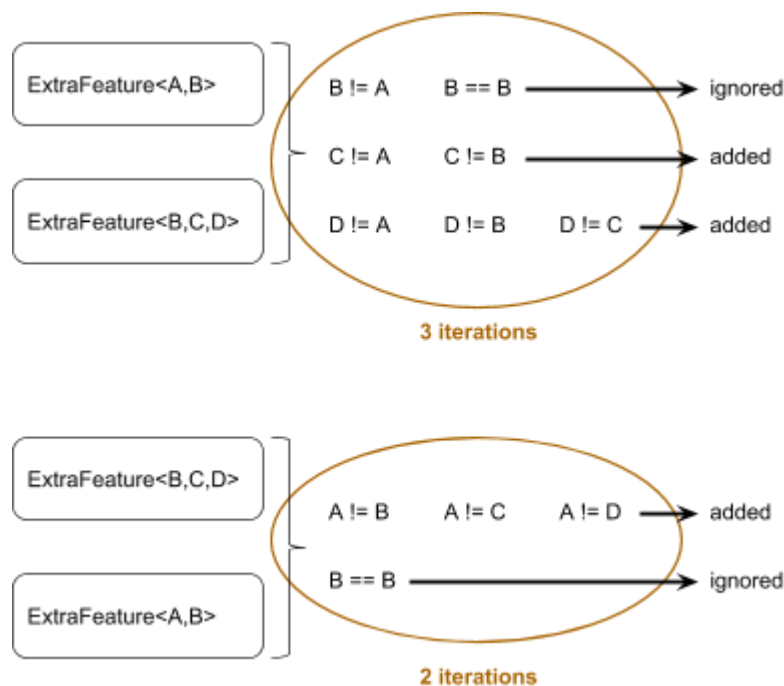
template<typename Derived,
        typename ... Packs,
        template<typename> typename ... Features1,
        template<typename> typename ... Features2>
struct JoinExtraFeatures<
    ExtraFeatures<Derived, Features1...>,
    ExtraFeatures<Derived, Features2...>,
    Packs...
>
{
    using type= typename
        JoinExtraFeatures<
            typename JoinExtraFeatures<
                ExtraFeatures<Derived, Features1...>,
                ExtraFeatures<Derived, Features2...>
            >::type,
            Packs...
        >::type;
};

template<typename Derived,
        template<typename> typename ... Features1,
        template<typename> typename ... Features2>
struct JoinExtraFeatures<
    ExtraFeatures<Derived, Features1...>,
    ExtraFeatures<Derived, Features2...>
>
{
    using type= typename
        JoinTwoExtraFeatures<
            ExtraFeatures<Derived, Features1...>,
            ExtraFeatures<Derived, Features2...>
        >::type;
};
```

The library now has all its components, and you can find all the code put together [here](#).

# Reducing the amount of comparisons

The library so far does the job, but we can add an additional optimization. As you can see `JoinExtraFeatures` adds the unique features from the second feature pack to the first one. What happens if the second feature pack is larger than the first one? Then we are forcing the compiler to perform more iterations, for nothing:



Indeed, the algorithm here is to check if a feature from pack 2 is already in pack 1, and to add it if it's not. So pack 1 is growing with some the features of pack 2. So to consider a feature of pack 2, we need to compare it with all the initial features of pack 1, plus the features of pack 2 added so far. So the smaller pack 2, the less comparisons.

Another way to put it is that the algorithm ends up comparing the features coming from pack 2 with each other, which it doesn't do for pack 1. And this comparison is not necessary since we can assume that features are unique within a single pack.

Note that this solution ensures that pack 2 is the smallest of the two, but doesn't remove the comparisons of the elements of pack 2 together. If you see how to get rid of those too, I'll be happy to read your ideas in the comments.

To reduce comparisons, we can count the number of features in each feature pack and place in the first position the larger one.

With this improvement smaller pack will be merged into the larger one so the number of needed iterations can be slightly reduced:

```
template<typename Derived,
        template<typename> typename ... Features1,
        template<typename> typename ... Features2>
struct JoinExtraFeatures<
    ExtraFeatures<Derived, Features1...>,
    ExtraFeatures<Derived, Features2...>
>
{
    using type = typename
        std::conditional<
            sizeof...(Features1) >= sizeof...(Features2),
            typename JoinTwoExtraFeatures<
                ExtraFeatures<Derived, Features1...>,
                ExtraFeatures<Derived, Features2...>
            >::type,
            typename JoinTwoExtraFeatures<
                ExtraFeatures<Derived, Features2...>,
                ExtraFeatures<Derived, Features1...>
            >::type
        >::type;
};
```

Finally we just need to update the declaration of the `x` class. As explained at the beginning, `x` can no longer inherit from the feature packs directly. Rather it now inherits from the merged one:

```
template<template<typename> typename... Skills>
class X : public JoinExtraFeatures<Skills<X<Skills...>>...>::type
{
public:
    void basicMethod(){};
};
```

The code can be tested easily without modifying the original `x` class posted by Jonathan in the [original pre-summer post](#):

```
int main()
{
    using XAB = X<ExtraFeaturesA, ExtraFeaturesB, ExtraFeaturesC>;

    XAB x;
    x.extraMethod1();
    x.extraMethod2();
    x.extraMethod3();
}
```

## Improvements

As I told before `JoinTwoExtraFeatures` structure can be improved if we can remove recursion to ease the load on the compiler. Also, the merge of two packs still makes some comparisons that could be avoided.

I'm thinking about those two possible improvements it but I couldn'tcannot find a nice solution. If you discover a way to avoid the recursion and the superfluous comparisons, please share it with us by leaving a comment below.