TEMPLATES

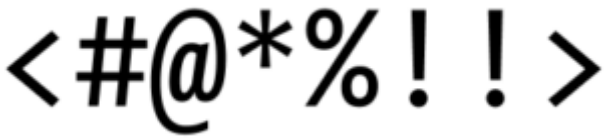
- **Expressive C++ Template Metaprogramming**
- **Template Partial Specialization In C++**
- **Function Templates Partial Specialization in C++**
- **The “Extract Interface” refactoring, at compile time**

Fluent{C++}

Expressive C++ Template Metaprogramming

There is a part of C++ developers that appreciate template metaprogramming.

And there are all the other C++ developers.



While I consider myself falling rather in the camp of the aficionados, I've met a lot more people that don't have a strong interest for it, or that even find it **downright disgusting**, than TMP enthusiasts. Which camp do you fall into?

One of the reasons why TMP is off putting for many people in my opinion is that it is often **obscure**. To the point that sometimes it looks like dark magic, reserved for a very peculiar sub-species of developers that can understand its dialect. Of course, we sometimes come across the occasional understandable piece of TMP, but on average, I find it harder to understand than regular code.

And the point I want to make is that **TMP doesn't have to be that way**.

I'm going to show you how to make TMP code much more expressive. And it's not rocket science.

TMP is often described as a language within the C++ language. So to make TMP more expressive, we just need to apply the same rules as in regular code. To illustrate, we're going to take a piece of code that only the bravest of us can understand, and apply on it the following two guidelines for expressiveness:

- choosing good names,
- and separating out levels of abstractions.

I told you, it's not rocket science.

Just before we start, I want to thank my colleague Jeremy for helping me with his impressive agility with TMP, and Vincent who's always so great for resonating ideas with. You guys rock.

The purpose of the code

We will write an API that checks whether an expression is valid for a given type.

For example given a type T, we would like to know whether T is incrementable, that is to say that, for an object t of type T, whether or not the expression:

```
++t
```

is valid. If T is `int`, then the expression is valid, and if T is `std::string` then the expression is not valid.

Here is a typical piece of TMP that implements it:

```
template< typename, typename = void >
struct is_incrementable : std::false_type { };

template< typename T >
struct is_incrementable<T,
          std::void_t<decltype( ++std::declval<T&>() )>
       > : std::true_type { };
```

I don't know how much time you need to parse this code, but it took me a significant amount of time to work it all out. Let's see how to rework this code to make it more quickly understandable.

In all fairness, I must say that to understand TMP there are constructs that you need to know. A bit like one needs to know "if", "for" and function overloading to understand C++, TMP has some prerequisites like "std::true_type" and SFINAE. But don't worry if you don't know them, I'll explain everything all along.

The basics

If you're already familiar with TMP you can skip over to the next section.

Our goal is to be able to query a type this way:

```
is_incrementable<T>::value
```

`is_incrementable<T>` is a type, that has one public boolean member, `value`, which is either true if T is incrementable (e.g. T is `int`) or false if it isn't (e.g. T is `std::string`).

We will use `std::true_type`. It is a type that only has a public boolean member `value` equal to true. We will make `is_incrementable<T>` inherit from it in the case that T can be incremented. And, as you'd have guessed, inherit from `std::false_type` if T can't be incremented.

To allow for having two possible definitions we use **template specialization**. One specialization inherits from `std::true_type` and the other from `std::false_type`. So our solution will look roughly like this:

```
template<typename T>
struct is_incrementable : std::false_type{};

template<typename T>
struct is_incrementable<something that says that T is incrementable>
: std::true_type{};
```

The specialization will be based on **SFINAE**. Put simply, we're going to write some code that tries to increment T in the specialization. If T is indeed incrementable, this code will be valid and the specialization will be instantiated (because it always has priority over the primary template). This is the one inheriting from `std::true_type`.

On the other hand if T isn't incrementable, then the specialization won't be valid. In this case SFINAE says that an invalid instantiation doesn't halt compilation. It is just completely discarded, which leaves as the only remaining option the primary template, the one inheriting from `std::false_type`.

Choosing good names

The code at the top of the post used `std::void_t`. This construct appears in the standard in C++17, but can be instantly replicated in C++11:

```
template<typename...>
using void_t = void;
```

EDIT: as u/Drainedsoul pointed out on Reddit, this implementation is guaranteed to work in C++14 but not in C++11, where unused template parameters of an alias declaration don't necessarily trigger SFINAE. The C++11 implementation uses an intermediate type an is available on cppreference.com.

`void_t` is just instantiating the template types it is passed, and never uses them. It's like a surrogate mother for templates, if you would.

And to make the code work, we write the specialization this way:

```
template<typename T>
struct is_incrementable<T, void_t<decltype(++std::declval<T&>())>> :
std::true_type{};
```

Ok, to understand TMP you also need to understand `decltype` and `declval`: `decltype` returns the type of its argument, and `declval<T>()` does as if an object of type T was instantiated in the `decltype` expression (it's useful because we don't necessarily know what the constructors of T look like). So `decltype(++std::declval<T&>())` is the return type of `operator++` called on T.

And as said above `void_t` is just a helper to instantiate this return type. It doesn't carry any data or behaviour, it's just a sort of launchpad to instantiate the type returned by `decltype`.

If the increment expression is not valid then this intantiation made by `void_t` fails, SFINAE kicks in and `is_incrementable` resolves to the primary template inheriting from `std::false_type`.

It's a great mechanism, but I'm cross with the name. In my opinion it's absolutely at the wrong level of abstraction: it's **implemented** as void, but what it means to do is **trying to instantiate a type**. By working this piece of information into the code, the TMP expression immediately clears up:

```
template<typename...>
using try_to_instantiate = void;

template<typename T>
struct is_incrementable<T,
try_to_instantiate<decltype(++std::declval<T&>())>> :
std::true_type{};
```

Given that the specialization that uses two template parameters, the primary template has to have two parameters too. And to avoid the user to pass it, we provide a default type, say `void`. The question now is how to name this technical parameter?

One way to go about it is to not name it at all, (the code at the top took this option):

```
template<typename T, typename = void>
struct is_incrementable : std::false_type{};
```

It's a way of saying "don't look at this, it's irrelevant and it's there only for technical reasons" which I find reasonable. Another option is to give it a name that says what it means. The second parameter is the **attempt** to instantiate the expression in the specialization, so we could work this piece of information into the name, which gives for the complete solution so far:

```
template<typename...>
using try_to_instantiate = void;

template<typename T, typename Attempt = void>
struct is_incrementable : std::false_type{};

template<typename T>
```

```
struct is_incrementable<T,
try_to_instantiate<decltype(++std::declval<T&>())>> :
std::true_type{};
```

Separating out levels of abstraction

We could stop here. But the code in `is_incrementable` is still arguably too technical, and could be pushed down to a lower layer of abstraction. Besides, it is conceivable that we'll need the same technique for checking other expressions at some point, and it would be nice to factor out the checking mechanism in order to avoid code duplication.

We will end up with something resembling the `is_detected` experimental feature.

The part that can vary most in the above code is clearly the `decltype` expression. So let's take it in input, as a template parameter. But again, let's pick the name carefully: this parameter represents the type of an **expression**.

This expression itself depends on a template parameter. For this reason we don't simply use a `typename` as a parameter, but rather a template (hence the `template<typename> class`):

```
template<typename T, template<typename> class Expression, typename
Attempt = void>
struct is_detected : std::false_type{};

template<typename T, template<typename> class Expression>
struct is_detected<T, Expression, try_to_instantiate<Expression<T>>>
: std::true_type{};
```

`is_incrementable` then becomes:

```
template<typename T>
using increment_expression = decltype(++std::declval<T&>());

template<typename T>
using is_incrementable = is_detected<T, increment_expression>;
```

Allowing for several types in the expression

So far we've used an expression involving only one type, but it would be nice to be able to pass several types to expressions. Like for testing if two types are **assignable** to one another, for example.

To achieve this, we need to use **variadic templates** to represent the types coming into the expression. We'd like to throw in some dots like in the following code, but it's not going to work:

```
template<typename... Ts, template<typename...> class Expression,
typename Attempt = void>
struct is_detected : std::false_type{};

template<typename... Ts, template<typename...> class Expression>
struct is_detected<Ts..., Expression,
try_to_instantiate<Expression<Ts...>>> : std::true_type{};
```

It's not going to work because the variadic pack `typename... Ts` is going to eat up all the template parameters, so it needs to be put at the end (if you want to better understand variadic templates I suggest you watch this part of Arthur O'Dwyer's excellent talk Template Normal Programming). But the default template parameter `Attempt` also needs to be at the end. So we have a problem.

Let's start by moving the pack to the end of the template parameters list, and also remove the default type for `Attempt`:

```
template<template<typename...> class Expression, typename Attempt,
typename... Ts>
struct is_detected : std::false_type{};

template<template<typename...> class Expression, typename... Ts>
struct is_detected<Expression,
try_to_instantiate<Expression<Ts...>>, Ts...> : std::true_type{};
```

But what type to pass to `Attempt`?

A first impulse could be to pass `void`, since the successful trial of `try_to_instantiate` resolves to `void` so we need to pass it to let the specialization be instantiated.

But I think that doing this would make the callers scratch their head: what does it mean to pass `void`? Contrary to the return type of a function, `void` doens't mean "nothing" in TMP, because `void` is a type.

So let's give it a name that better carries our intent. Some call this sort of thing "dummy", but I like to be even more explicit about it:

```
using disregard_this = void;
```

But I guess the exact name is a matter of personal taste.

And then the check for assignment can be written this way:

```
template<typename T, typename U>
using assign_expression = decltype(std::declval<T&>() =
std::declval<U&>());

template<typename T, typename U>
using are_assignable = is_detected<assign_expression,
disregard_this, T, U>
```

Of course, even if `disregard_this` reassures the reader by saying that we don't need to worry about it, it is still in the way.

One solution is to hide it behind a level of indirection: `is_detected_impl`. "impl_" often mean "level of indirection" in TMP (and in other places too). While I don't find this word natural, I can't think of a better name for it and it is useful to know it because a lot of TMP code uses it.

We'll also take advantage of this level of indirection to get the `::value` attribute, relieving all the elements further up from calling it each time they use it.

The final code is then:

```
template<typename...>
using try_to_instantiate = void;

using disregard_this = void;
```

```
template<template<typename...> class Expression, typename Attempt,
typename... Ts>
struct is_detected_impl : std::false_type{};

template<template<typename...> class Expression, typename... Ts>
struct is_detected_impl<Expression,
try_to_instantiate<Expression<Ts...>>, Ts...> : std::true_type{};

template<template<typename...> class Expression, typename... Ts>
constexpr bool is_detected = is_detected_impl<Expression,
disregard_this, Ts...>::value;
```

And here is how to use it:

```
template<typename T, typename U>
using assign_expression = decltype(std::declval<T&>() =
std::declval<U&>());

template<typename T, typename U>
constexpr bool is_assignable = is_detected<assign_expression, T, U>;
```

The generated values can be used at compile-time or at run-time. The following program:

```
// compile-time usage
static_assert(is_assignable<int, double>, "");
static_assert(!is_assignable<int, std::string>, "");

// run-time usage
std::cout << std::boolalpha;
std::cout << is_assignable<int, double> << '\n';
std::cout << is_assignable<int, std::string> << '\n';
```

compiles successfully, and outputs:

```
true
false
```

TMP doesn't have to be that complex

Sure, there are a few prerequisites to understand TMP, like SFINAE and such. But apart from those, there is no need to make the code using TMP look more complex than necessary.

Consider what is now a good practice for unit tests: it's not because it's not production code that we should lower our standards of quality. Well, it is even more true for TMP: it *is* production code. For this reason, let's treat it like the rest of the code and do our best to make it as expressive as possible. Chances are, more people would then be attracted to it. And the richer the community, the richer the ideas.

Template Partial Specialization In C++

Today I want to share something about the right way to emulate partial function template specialisation in C++. I learnt it by watching Arthur O'Dwyer's CppCon talk Template Normal Programming.

Actually, the technique for emulating function template partial specialization through class template specialization is well known, but the naming convention used by Arthur is the clearest I've seen. And he has kindly accepted that I share it with you on Fluent C++. Jump to the bottom of the post to get to it directly.

All this is a great opportunity for a general review of template partial specialization in C++.

Partial Template Specialization

C++ allows to partially specialize **class templates**:

```
template<typename T>
struct is_pointer : std::false_type {};

template<typename T>
struct is_pointer<T*> : std::true_type {};
```

In the above code, `is_pointer` has a primary template (the first struct) and a specialization (the second one). And since the specialization still has a template parameter, it is called a **partial** specialization.

Partial specialization also works for **variable templates**:

```
template<typename T>
bool is_pointer_v = false;

template<typename T>
bool is_pointer_v<T*> = true;
```

But C++ forbids partial specialization on anything else than classes (or structs) and variables.

That means that **alias template** partial specialization is forbidden. So the following code is also invalid:

```
template<typename T>
using myType = int;

template<typename T> // compilation error!
using myType<T*> = int*;
```

In fact, even total specialization of alias templates is forbidden.

And while **function templates** can be totally specialized, their partial specialization is illegal. So the following code:

```
template<typename T>
constexpr bool is_pointer(T const&)
{
    return false;
}

template<typename T> // compilation error!
constexpr bool is_pointer<T*>(T const&)
{
    return true;
}
```

leads to a compilation error.

Why can't we partially specialize everything?

To be honest, I don't know. But I wonder.

Herb Sutter touches upon the subject in More Exceptional C++ Item 10 and in Exceptional C++ Style Item 7, but it's more about total specialization than partial. I

suspect that the rationale for functions partial specialization is that it would allow mixing partial specializations with overloading, which would become too confusing.

Does anyone know the reason for restricting on function template specialization?

For aliases this answer on Stack Overflow gives some elements of information. In short, the using declaration is no more than an alias, that shouldn't fetaure more logic. This explains why even total specialization are not allowed for aliases.

Emulating Partial Template Specialization

To emulate partial specialization on aliases and on functions, the technique is to fall back on the specialization that works on structs.

Here is how to go about it for **alias templates**:

```
template<typename T>
struct MyTypeImpl { using type = int; };

template<typename T>
struct MyTypeImpl<T*> { using type = int*; };

template<typename T>
using myType = typename MyTypeImpl<T>::type;
```

As for **functions templates**, let me share the implementation that Arthur O'Dwyer's CppCon demonstrates in his talk. He also uses a fall back on structs, but his naming conventions is the clearest that I've seen:

```
template<typename T>
struct is_pointer_impl { static constexpr bool _() { return false; }
};

template<typename T>
struct is_pointer_impl<T*> { static constexpr bool _() { return
true; } };

template<typename T>
constexpr bool is_pointer(T const&)
```

```
{
    return is_pointer_impl<T>::_();
}
```

Did you notice the name of the static function inside the structs? It's just an **underscore**, which is a legal name for a function in C++. Since these functions are just a technical artifact, I think it's good to show that they carry no meaning by giving them (almost) no name.

With this you can emulate any missing template partial (or even total for aliases) specialization in C++.

Regardless of this, Template Normal Programming is a great talk, that shows you everything there is to know about templates, except in metaprogramming. It's definitely worth watching.

Function Templates Partial Specialization in C++

Why doesn't C++ allow partial specialization on function templates? Such was the question I asked to you, readers of Fluent C++, in the post covering Template Partial Specialization. Not because I wanted to test you, but simply because I couldn't find the answer.

And oh boy did I get an answer.

The post received comments, questions, suggestions and discussions, and even if the article covered template partial specialization in general, most of the reactions revolved around the subject of **function** template partial specialization. And I want to thank /u/sphere991, /u/quicknir, rroki and Simon Brand in particular for their contributions.

Each comment pointed to a particular facet of the subject. Now what I want to share with you is how, put together, they allowed me to take a step back and have a **wider vision on function template partial specialization**. I'm excited to share this big picture with you because it's exactly what I wished I could read when I was searching for more info on this, and that I couldn't find anywhere.

Function specialization? Just overload!

When you think about it, template specialization comes down to picking the right implementation for a given type. And the resolution occurs at compile time.

Now compare this with function overloading: it consists in picking the right function for a given type of argument. And the resolution also occurs at compile time.

In this perspective, these two features look very similar. Therefore, it is only normal that you can achieve something equivalent to function template (partial or total) specialization with function overloading. Let's illustrate with an example.

Consider the following template function `f`:

```
template <typename T>
void f(T const& x)
{
    // body of f
}
```

Say that we want a specific implementation when `T` is `std::string`.

We could either specialize `f`:

```
template <>
void f<std::string>(std::string const& x)
{
    // body of f for std::string
}
```

or we could simply overload:

```
void f(std::string const& x)
{
    // body of f for std::string
}
```

Either way, the execution will go through the specific implementation when you pass `f` a string.

The same goes for **partial specialization**. Say we want a specific implementation of `f` for vectors. We can't write it with partial specialization since the following would be illegal:

```
// Imaginary C++
template <typename T>
void f<std::vector<T>>(std::vector<T> const& v)
{
    // body of f or vectors
}
```

But we can write it with overloading instead:

```
template <typename T>
void f(T const& x) // #1
{
    // body of f
}

template <typename T>
void f(std::vector<T> const& v) // #2
{
    // body of f for vectors
}

f(std::vector<int>{}); // this call goes to #2
```

and we get the desired effect just as well.

What if you can't overload

Is this to say that there is no case for partial specialization on template functions? No! There are cases where **overloading won't do**.

Overloading works for template **arguments**, which arguably represent a fair share of use cases for template functions. But what if the template is not in the argument? For instance, it could be in the return type of the function:

```
template <typename T>
T f(int i, std::string s)
{
    // ...
}
```

Or it could even be nowhere in the function prototype:

```
template <typename T>
void f()
{
    // ...
}
```

The common point between those cases is that you have to specify the template type explicitly at call site:

```
f<std::string>();
```

In such cases, overloading can't do anything for us, so for those cases I believe there is a case for partial specialization on function templates. Let's review our options for working around the fact that C++ does not support it natively.

Fall back on overloading

This is the technique that Simon Brand proposed. It consists in adding a parameter that carries the information about what type `T` is. This parameter, `type`, is something that just carries another type `T`:

```
template <typename T>
struct type{};
```

(we could also omit the name `T` here since it isn't used the body of the template.)

This allows to return to the case where we can use overloading instead of specialization.

Consider the following example to illustrate. We want to design a template function `create` that returns an object of type `T` initialized by default:

```
return T();
```

except when the type to be returned is a vector, in which case we want to allocate a capacity of 1000 to anticipate for repeated insertions:

```
std::vector<T> v;
v.reserve(1000);
return v;
```

So we want a default implementation, and one for all `vector<T>` for all `T`. In other terms, we need to partially specialize `f` with `vector<T>`.

Here is how to achieve this with this technique:

```
template <typename T>
struct type{};

template <typename T>
T create(type<T>)
{
    return T();
}

template <typename T>
std::vector<T> create(type<std::vector<T>>)
{
    std::vector<T> v;
    v.reserve(1000);
    return v;
}

template <typename T>
T create()
{
    return create(type<T>{});
}
```

Fall back on class template partial specialization

Even if we can't do partial specialization on function template, we can do it for **class templates**. And there is a way to achieve the former by reusing the latter. To see the cleanest way to do this, you can refer to the post Template Partial Specialization where I get into this (and more) in details.

Whatever you do, don't mix specialization and overloading

When you use a set of overloads for several implementations of a function, you should be able to understand what's going on.

When you use a set of (total) specializations for several implementations of a function, you still should be able to understand what's going on.

But when you mix both overloading and specializations for the same function, you're entering the realm where the magic, the voodoo and the Bermuda Triangle live, a world where things behave in an inexplicable way, a world where you're better off not knowing too many explanations about because they could suck you in it and your mind will end up in a plane cemetery and pierced with spikes carrying dark magic spells.

To illustrate, consider this insightful example given by /u/sphere991 that says it all:

```
template <typename T> void f(T ); // #1
template <typename T> void f(T* ); // #2
template <> void f<>(int* ); // #3

f((int*)0); // calls #3
```

but:

```
template <typename T> void f(T ); // #1
template <> void f<>(int* ); // #3
template <typename T> void f(T* ); // #2

f((int*)0); // calls #2 !!
```

The mere **order of declarations** of the overloads and the specializations determines the behaviour of the call site. Reading this piece of code sends a chill down my spine. Brrrr.

Whichever technique you choose, don't mix function overloading and function specialization for the same function name.

Will C++ support function template partial specialization?

We've seen **when** we need partial specialization for function templates and **how** to emulate it. But we haven't answered our original question: Why doesn't C++ allow partial specialization on function templates?

The only element of answer that I got was provided to me by rroki in this comment, referring to an old document written by Alistair Meredith. If I summarize what this document says, the feature has been considered a while ago, and left out because the concept_maps could do the job instead. But concept_maps are part of the version of concepts that was abandoned since then!

The way I interpret this is that there is nothing wrong in allowing function template partial specialization in C++, but we don't know if it will be in the language one day.

The “Extract Interface” refactoring, at compile time

We haven’t talked too much about refactoring on Fluent C++ so far, but this is a topic related to code expressiveness. Indeed, most of the time we don’t start working on production code from scratch, but we rather work on an existing base. And to inject expressiveness into it, this can come through refactoring.

To make a long story short, refactoring goes with tests, and tests go with breaking dependencies.

Indeed, having unit tests covering the code being refactored allows being bold in refactoring while ensuring a certain level of safety. And to test a portion of code, **this code has to be relatively independent** from the rest of the application, particularly from the parts that really don’t play well with tests, such as UI and database for example.

The “Extract Interface” technique is a classical method to break dependencies that can be found in any good book about refactoring, such as Working Effectively with Legacy Code from Michael Feathers.

My purpose here is to propose a way to perform the Extract Interface technique, in a way that is idiomatic in C++. Indeed, even if they are C++ legal code, I find that typical implementations are directly translated from Java, and I think we can change them to make them fit much better in C++.

Extract Interface

Let’s start by a quick description of what Extract Interface is and what problem it aims at solving. If you are already familiar with it you can safely skip over to the next section.

One of the situations where Extract Interface comes in handy, is breaking a dependency related to an argument passed to a function or a method.

For example, here is a class we would like to get into a unit test:

```
// In a .h file

class ClassToBeTested
{
public:
    void f(Argument const& arg);
};
```

Here is what `Argument` can do:

```
class Argument
{
public:
    void whoIsThis() const
    {
        std::cout << "This is Argument\n";
    }
    // more methods...
};
```

and the above method `f` uses it in its body:

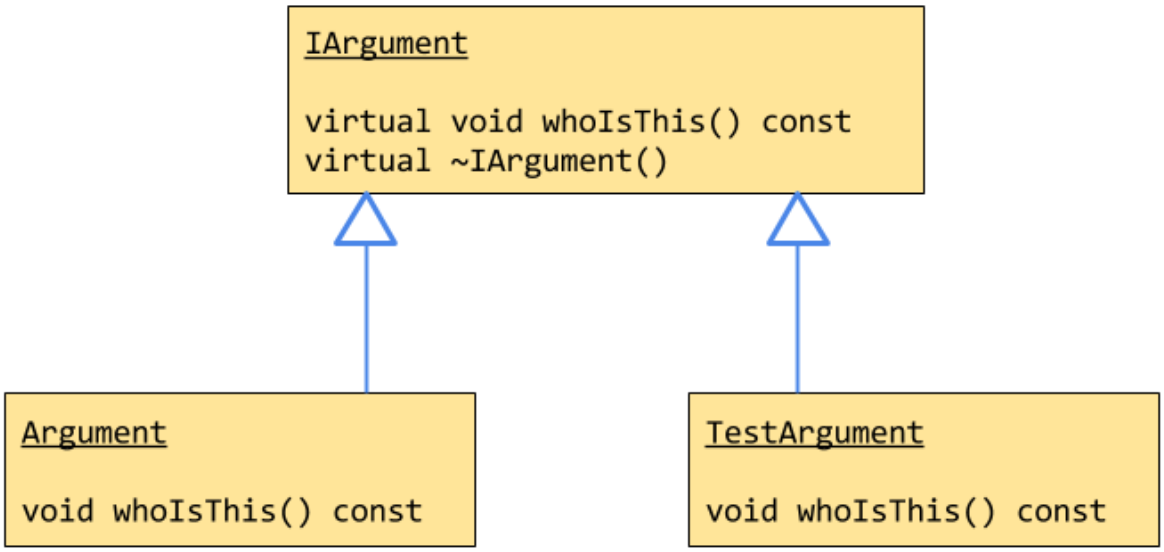
```
// In a .cpp file

void ClassToBeTested::f(Argument const& arg)
{
    arg.whoIsThis();
}
```

Let's imagine that, like some real classes, `ClassToBeTested` won't let itself into a test harness, because building an object of type `Argument` is, say, terribly complicated as it depends on so many other things.

We can then create a new type, `TestArgument`. It offers the same interface as `Argument`, so that our `ClassToBeTested` can use it, but it has a simplified implementation, containing just enough for the purpose of conducting the test.

To materialize this interface we can create an `IArgument` class, from which both `Argument` and `TestArgument` would derive:



The interface of `ClassToBeTested` becomes:

```
// In a .h file

class ClassToBeTested
{
public:
    void f(IArgument const& arg);
};
```

And f can be passed an `Argument` coming from production code, or a `TestArgument` coming from the test harness. This is the result of **Extract Interface**.

Pay only for what you need

The above implementation of Extract Interface works very well in languages such as Java and C#, because inheriting from interfaces with runtime polymorphism is so ubiquitous that these languages do an excellent job optimizing these constructs.

But this is not the case in C++, where this is much less idiomatic.

First off, there is a technical consideration: the above implementation adds runtime polymorphism, which has a cost:

- it adds an indirection at each call to the interface, to redirect the execution to the code of the correct derived class,
- it makes the objects bigger, typically by adding a virtual pointer to each one, to help in this indirection.

But even if this can be problematic in performance sensitive parts of the code, this cost may be negligible in many situations.

The real problem here is about design: **we don't need runtime polymorphism here**. We know when we're in production code or in test code when invoking the class to be tested, and we know this **at the moment of writing code**. So why wait until the last moment at runtime to do this check and redirect to the right argument?

We do need polymorphism though, because we want two possible classes to be used in the same context. But this is **compile-time polymorphism that we need**. And this can be achieved with templates.

Extract "compile-time" Interface

Templates offer a polymorphism of sorts: template code can use **any type** in a given context, provided that the generated code compiles. This is defining an interface, although not as explicitly stated as in runtime polymorphism with inheritance and virtual functions (although concepts will make template interfaces more explicit, when they make it into the language).

Here is how Extract Interface can be implemented with templates:

```
// In a .h file

class ClassToBeTested
{
public:
    template<typename TArgument>
    void f(TArgument const& arg)
    {
        arg.whoIsThis();
    }
```

```
};
```

Then you can pass either an `Argument` or a `TestArgument` to the method `f`, and they no longer need to inherit from `IArgument`. No more runtime polymorphism and virtual pointers and indirections.

However, the template code has to be visible from the point it is instantiated. So it is generally put **into the header file**, mixing the declaration and the implementation of the method.

"We don't want that!", I hear you say, undignified. "We don't want to show the internals of the method to everyone, thus breaking encapsulation and really increasing compilation dependencies!"

But template code forces us to do this… or does it?

Explicit instantiation

C++ holds a discrete feature related to templates: **explicit instantiation**. It consists in declaring an instantiation of a template on a particular type, which can be done in a .cpp file.

```
// In the .cpp file

template void ClassToBeTested::f(Argument);
template void ClassToBeTested::f(TestArgument);
```

When the compiler sees this, it instantiates the template with the type, generating all the corresponding code, **in the .cpp file** (if you have heard of the "export" keyword, it has nothing to do with it. If you haven't… then good for you ☺) Then the implementation of the method no longer needs to be in the header file, because only the explicit instantiation needs to see it.

At this point we may wonder why all template classes don't use this formidable feature. The answer is because we would need an explicit instantiation to specify each of the types

the template can be instantiated with, if we really want to keep implementation in the .cpp file. So for `std::vector` for example, this feature has no use.

But in our case, **we know each of the possible instantiations**, and they are just the two of them: `Argument` and `TestArgument`. This was actually the whole purpose of the operation!

To sum up where we are know, here is what the header and the implementation files look like:

In the .h file:

```
class ClassToBeTested
{
public:
    template <typename TArgument>
    void f(TArgument const& arg);
};
```

In the .cpp file:

```
#include "ClassToBeTested.h"
#include "Argument.h"
#include "TestArgument.h"

template<typename TArgument>
void ClassToBeTested::f(TArgument const& arg)
{
    arg.whoIsThis();
}

template void ClassToBeTested::f(Argument);
template void ClassToBeTested::f(TestArgument);
```

Now we can still construct a `TestArgument` in the test harness without paying for runtime polymorphism, nor displaying the implementation of the method in the header.

There is one more problem left to tackle: the above example `#include`s the `"Argument.h"` header. And this header may itself contain dependencies to complicated things that the test harness will have a hard time linking against. It would be nice to somehow avoid `#include`ing `"Argument.h"` in the context of the test harness.

The Chinese wall between explicit instantiations

This solution has been found by my colleague Romain Seguin.

When you think about it, the only thing we need to include Argument for is the template instantiation. The idea then is to take the explicit instantiations and the include directives out into separate files.

In the production binary:

```
// file ClassToBeTested.templ.cpp

#include "Argument.h"
#include "ClassToBeTested.cpp"

template void ClassToBeTested::f(Argument);
```

And in the test binary:

```
// file ClassToBeTestedTest.templ.cpp

#include "TestArgument.h"
#include "ClassToBeTested.cpp"

template void ClassToBeTested::f(TestArgument);
```

And the initial implementation file is reduced to:

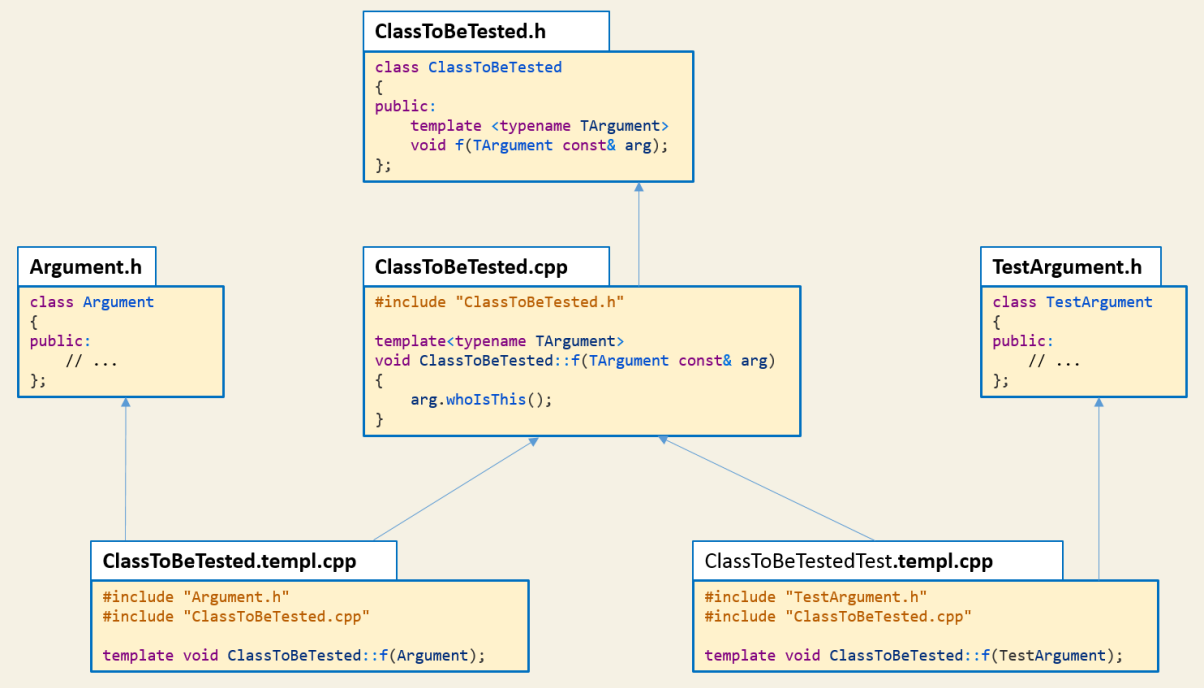
```
// file ClassToBeTested.cpp

#include "ClassToBeTested.h"

template<typename TArgument>
void ClassToBeTested::f(TArgument const& arg)
{
    arg.whoIsThis();
}
```

This way, the test binary does not have to link against anything coming from the header of the `Argument` production class.

Here is a schema showing all file inclusions:



(Note that the proposed extension for these files (templ.cpp) is subject to debate. Maybe we should use ".cpp" for *them*, and rather ".templ.hpp" for the implementation of the template method, which was Romain's opinion.)

Now over to you

What do you think about this proposed way of performing an Extract Interface in C++? I haven't found it described anywhere, so it might be either innovative or so wrong that no one cared to talk about it before.

In any case, your impression on this would be very welcome. It's crazy how questions and thoughts can improve the quality of an idea as a group, so please lads (and ladies!), knock yourselves out.