Document Number: P1996R0
Date: 2019-11-08
Reply-to: Dmitry Sokolov
dimanne@gmail.com
Audience: Evolution

# Propagated template parameters

Abstract

Motivation

What we want to attain

Design alternatives

## Abstract

This proposal suggests making (as an opt-in option) class template parameters (type as well as non-type ones) visible outside the scope of a class with the same name as they appear in the list of template parameters (without having to manually redeclare them inside the class via `using`/`typedef` for types or `static constexpr auto` for variables).

## Motivation

Currently, class template parameters are not visible outside the scope of a class and it is often desirable to expose / propagate them to external users, which means that one has to manually redeclare them inside a class via `using`/`typedef` (or `static constexpr auto`, in case of non-type parameters). In the example below those names are `T` and `value_type`:

```
template <class T, ...>
class vector {
    using value_type = T;
};
```

The current state of affairs has a number of negative consequences.

### Contamination

it inevitably causes contamination of the scope of a class with names, indeed, there are two versions of (semantically) the same name (`T` and `value_type` in the example above). The current practice is to use an obfuscated/uglified version as a template parameter, and have a more clear and user-friendly one exposed via `using`/`typedef`.

### Ambiguity

Since there are two different names for the same entity, which one should be used inside the class?

### Wasting time

Last but not least, those repetitive declaration must be typed, which takes precious time.

### Learning curve

This [question from StackOverflow]() is an exemplary demonstration that the behaviour suggested in the proposal is expected by novices:

> *I am learning c++. I would like to use template parameter name as it is outside the class. I could not find the best solution but now I use "using" declaration. However it cannot use same name. Are there any better solution? Or Are there any good habit or good naming to re-declare template parameter by "using"?*

Note, how people learning C++ discover the harsh rules of C++:

- First, try to use the same names outside - error.
- Second, try to declare a typedef inside a class with the same name - error.
- Finally, realise that an additional "fake" name should be invented.

# What we want to attain

So far we have discussed what we don't want to do - we don't want to manually redeclare template arguments (with different names, obviously) in situations when we want to propagate them to external clients.
So, let's first discuss in greater detail what we want to attain. And then, in the sections below different approaches.
The best way of reasoning about the proposal is imagining what we can do in the scope of a template class with its template parameters, and try to make these actions available from outside the scope.

### Type template parameters

<table>
<tr><th>Given the code:</th><th>This will be allowed outside class:</th></tr>
<tr><td>

```
template <class value_type>
struct MyVector {
};

using VectorOfInts = MyVector<int>;
```

</td><td>

```
VectorOfInts::value_type a = {};

// the same as
// int a = {};
```

</td></tr>
</table>

### Non-type template parameters

| Given the code: | This will be allowed outside class: |
|---|---|
| `template <size_t Size>`<br>`struct MyArray {`<br>`};`<br><br>`using Array = MyArray<6>;` | `size_t a = Array::Size; // a == 6` |

### Template template parameters

| Given the code: | This will be allowed outside class: |
|---|---|
| `template <template <class> class Cont>`<br>`struct MyCont {`<br>`};`<br><br>`using Array = MyCont<std::array>;` | `Array::Cont<int, 3> array;`<br><br>`// the same as`<br>`// std::array<int, 3> array;` |

### Variadic template parameters

| Given the code: | This will be allowed outside class: |
|---|---|
| `template <class... Ts>`<br>`struct MyCont {`<br>`};`<br><br>`using Cont = MyCont<char, int, double>;` | `std::tuple<Cont::Ts...> tuple;`<br><br>`// the same as`<br>`// std::tuple<char, int, double> tuple;` |

### Unnamed template parameters

Little can be done with unnamed template parameters (`template <class = void> class MyClass {...};`), so nothing should be possible to do with unnamed template parameters. Moreover, using the feature with such parameters should cause a compiler error.

### Public, protected, private

Of course, it is possible to declare `using`/`typedef` in public/protected/private sections, so propagated template parameters also should allow it.

# Design alternatives

## Propagated by default => FAIL

The first approach is to change name lookup rules in the following way:

- when looking for a nested name, do normal lookup;
- if that didn't find anything, look into the template parameters.

Alternatively, this approach can be understood as implicitly generated and public `using`/`typedef` for type parameters (or `static constexpr auto` for non-type parameters).

| Before | After |
|---|---|
| `template <class _Tp,`<br>`          class _Allocator>`<br>`struct vector {`<br>`   typedef _Tp value_type;`<br>`   typedef _Allocator allocator_type;`<br>`};`<br><br>`vector<int>::value_type a = 0;` | `template <class value_type,`<br>`          class allocator_type>`<br>`struct vector {`<br>`};`<br><br>`vector<int>::value_type a = 0;` |

### Cons

The main disadvantage of the approach, however, is that parameter names become part of the API of a class. Even though it has been always possible to query n-th template parameter, given an instantiation of a class:

```
#include <vector>

template <class T>
struct TVectorTraits;

template <class T, class A>
struct TVectorTraits<std::vector<T, A>> {
   using value_type = T;
   using allocator = A;
};

template <class T>
using GetValueType = typename TVectorTraits<T>::value_type;

int main() {
   struct TMyStruct {};
   using ValueType = GetValueType<std::vector<TMyStruct>>;
   static_assert(std::is_same_v<ValueType, TMyStruct>, "");
   return 0;
}
```

the main difference with this approach, is that a user defines its own name to represent the n-the parameter (`TVectorTraits<>::`**`allocator`** in the example above). And, if/when the name of the n-th parameter changes, it will not break user code (`TVectorTraits<>`).
Another disadvantage, is that this will break (silently change the result of) existing type inspection classes.

## Opt-in + keywords, implicitly generated members

So, it is obvious that we need a more explicit, opt-in approach. One of the ways forward would be using keywords, presumably inside `template <>` clause, since any new syntax inside class itself will not be short and convenient enough.
For the sake of completeness there are potentially suitable for this task keywords:

1. Prime candidates: `private`, `protected`, `public`.
2. `export` / `explicit`.
3. `default`, `extern`, `using`.

`private`, `protected`, `public` - look very easy and intuitive. There is a direct correspondence between those keywords and expectation about accessibility of the implicitly generated `using`/`typedef`:

**Before**

```
template <                  class _A,
                            int   _B,
          template <class> class _C
>
struct X {
public:
   using A = _A;

protected:
   static constexpr int B = _B;

private:
   template<class T>
   using C = _C<T>;
};
```

**After**

```
template<                   public    class A,
                            protected int   B,
          template <class> private   class C
>
struct X {
};
```

### What to do with class types in non-type template parameters?

Namely, do we want to have references (`static constexpr TClassType &X = _X;`), or values (`static constexpr TClassType X = _X;`)?
Unlike “normal” non-type template parameters, which are rvalues, and therefore their address cannot be taken, class types as non-type template parameters are const lvalues, and therefore their address can be taken. In order to make behaviour of implicitly generated constexpr values outside the scope of a class more similar to that of template parameter inside the scope of the class (in particular, their behaviour in regard to their addresses), the proposal suggests using references - `static constexpr TClassType &X = _X;`.

### Partial and full specialisations

We have to have an option to suppress propagated template parameters in (partial) specialisations. As usual, there are opt-in and opt-out approaches.
Opt-in approach is to explicitly use the `public` keyword (once more) in a specialisation:

**Before**

```
template <class T,
          class A>
class MyVector {
public:
   using value_type = T;
};

// Specialisation for bool:
template <class A>
class MyVector<bool, A> {
public:
   using value_type = bool;
};
```

**After**

```
template <public class T,
                 class A>
class MyVector {
};

// Specialisation for bool:
// without “public” there will be no
// “value_type” in “MyVector<bool, A>”
// specialisation
template <class A>
class MyVector<public bool, A> {
};
```

Opt-out approach can be achieved via the following options:

1. Use `delete` in the list of template parameters:

| Before | After |
|---|---|
| `template <class T>`<br>`class Foo {`<br>`public:`<br>`   using value_type = T;`<br>`};`<br><br>`// Specialisation for pointer:`<br>`template <class T>`<br>`class Foo<T*> {`<br>`public:`<br>`};` | `template <public class value_type>`<br>`class Foo {`<br>`};`<br><br><br><br>`// Specialisation for pointer:`<br>`template <class T>`<br>`class Foo<T* = delete> {`<br>`public:`<br>`};` |

2. If the goal is not to just suppress a propagated template parameter, but also to redeclare it with a different type, we can just allow it in this way:

| Before | After |
|---|---|
| `template <class T>`<br>`class Foo {`<br>`public:`<br>`   using value_type = T;`<br>`};`<br><br>`// Specialisation for pointer:`<br>`template <class T>`<br>`class Foo<T*> {`<br>`public:`<br>`   using value_type = T;`<br>`};` | `template <public class value_type>`<br>`class Foo {`<br>`};`<br><br><br><br>`// Specialisation for pointer:`<br>`template <class T>`<br>`class Foo<T*> {`<br>`public:`<br>`   using value_type = T;`<br>`};` |

All in all, it is yet undecided which approach is better. Probably, the first one (opt-in) looks reasonable.

### New compiler errors

Depending on the decision regarding suppressing propagated template parameters, we might want to  issue an error when the name of a propagated template parameter clashes with user-defined one. Similarly, when propagated template parameter is unnamed:

| Collision of names | Unnamed propagated parameter |
|---|---|
| `template <public class value_type>`<br>`struct MyVector {`<br>`   using value_type = int; // ERROR`<br>`};` | `template <public class = void> // ERROR`<br>`struct MyVector {`<br>`};` |

### Relation to concepts

Since concepts are just predicates for types, the proposal does not interfere with concepts.

### Multiple forward declaration

Drawing on the logic described in What we want to attain, if there are forward declarations of a class and class itself, and propagated template argument names differ:

```
template <public class T1, public class T2>
struct X;

template <public class RealNames, public class GoHere>
struct X {
};
```

only the names in the definition of the class X are used.

Similarly, if there is only declarations of a class, without definition, any attempts to refer to propagated template parameters are ill-formed and should cause an error.