Injected class name in the base specifier list

Document #:	P3009R0
Date:	2023-10-07
Project:	Programming Language C++
Audience:	EWG

Reply-to:	Joe Jevnik <joejev@gmail.com>

1 Abstract

Inside of a class template body, the *class-name* of the class template refers to the current template instantiation; this is referred to as the *injected-class-name*. The *injected-class-name* is not present until the class body begins, which is after the opening brace of the class definition. The consequence of this is that the *injected-class-name* is not available in the *base-specifier-list*, which means it is not accessible when using the curiously recurring template pattern (CRTP). While [P0847R7] has removed the need for using CRTP in many places, there still exist cases that must be expressed using this pattern. This paper proposes allowing the use of the *injected-class-name* in the *base-specifier-list* to simplify code using CRTP. This paper also aims to reduce complexity in the language by simplifying the interpretation of the *class-name* and the rules around the *injected-class-name*.

2 Introduction

Using the *injected-class-name* when implementing class templates is not only easier to read, it is also less bug prone. Many complicated container types use multiple template arguments, often with rarely changed default arguments. For example, a mapping type have may some `Key` type and a `Value` type in addition to `Hash`, `KeyEqual`, and `Allocator`. The last three arguments are often defaulted, as it is for the standard library's `std::unordered_map`, which means people can easily forget that they exist. Using the *injected-class-name* to refer to the current instantiation makes it impossible to forget to include all of these extra template arguments. When the *injected-class-name* is not used, it is possible to write code that works for common cases, but produces difficult to understand compiler errors for some template instantiations. For example, consider the following template class:

```
template <typename Derived>
struct CRTPBase {
    // ...
};

template <typename Key,
          typename Value,
          typename KeyEqual = std::equal<Key>,
          typename Hash = std::hash<Key>
          typename Allocator = std::allocator<std::pair<Key const, Value>>>
class Container : CRTPBase<Container<Key, Value KeyEqual, Hash>> {
    // ...
};
```

In this case, we likely wanted to pass the current instantiation of `Container` as the template argument to `CRTPBase`; however, we accidentally forgot the `Allocator` argument. `Allocator` will likely often use the default type, which means that most of the time this code will actually behave as intended. The bug will only be revealed when a non-default allocator is used, and it will likely yield a complicated error message.

3 Proposal

This paper proposes introducing the *injected-class-name* in the *base-specifier-list*.

3.1 Behavior Changes

Unfortunately, while the proposal itself is concise, it can alter the behavior of existing code.

3.2 Unambiguous to ambiguous

The following code is currently unambiguous according to the rules of C++: the base type will always be `WasTemplate`:

```
#include <type_traits>

struct WasType {};
struct WasTemplate {};

template <typename Type>
auto foo() -> WasType;  // overload 1

template <template <typename...> class Template>
auto foo() -> WasTemplate;  // overload 2

template <typename Type>
struct CurrentlyUnambiguousBase : decltype(foo<CurrentlyUnambiguousBase>()) {};

static_assert(std::is_base_of_v<WasTemplate, CurrentlyUnambiguousBase<void>>);
```

In this example, we have a function template `foo` which is overloaded once for a type template argument, and once for a template template argument. In the *base-specifier-list* the *class-name* `CurrentlyUnambiguousBase` will always refer to the class template and never refer to an instantiation of the class template. Because the name unambiguously refers to the class template, the compiler chooses overload 2 for `foo`.

Inside the class body, the *class-name* can refer to the current template instantiation being defined, which means that `CurrentlyUnambiguousBase` may be the type `CurrentlyUnambiguousBase<Type>` or the class template `CurrentlyUnambiguousBase`. This means that moving the exact same expression from the *base-specifier-list* inside the class body results in an ambiguous overload.

```
template <typename Type>
struct CurrentlyUnambiguousBase : decltype(foo<CurrentlyUnambiguousBase>()) {
    // Error: call to overloaded `foo` is ambiguous between overloads 1 and 2
    using InsideBody = decltype(foo<CurrentlyUnambiguousBase>());
};
```

If the *injected-class-name* was introduced in the *base-specifier-list*, then this previously unambiguous code would become an error and force the user to disambiguate.

3.3 Same code with different behavior.

Changing well formed code into ill-formed code is not ideal, but users will mostly be able to mechanically change the few occurrences and move on. Changing the meaning of code in such a way that both the old and the new are well formed with a different result is much more serious.

Consider the following code:

```
#include <type_traits>
```

```
struct WasType {};
struct WasTemplate {};

template <typename Type>
auto bar(int) -> WasType;  // overload 1

template <template <typename...> class>
auto bar(long) -> WasTemplate;  // overload 2

template <typename Type>
struct DifferentBehavior : decltype(bar<DifferentBehavior>(0)) {
    using InsideBody = decltype(bar<DifferentBehavior>(0));
};

static_assert(std::is_base_of_v<WasTemplate, DifferentBehavior<void>>);
static_assert(std::is_same_v<WasType, DifferentBehavior<void>::InsideBody>);
```

This code uses an overloaded function template: `bar`. `bar` has two overloads: one which has a type template parameter with an `int` parameter and a second which takes a template template parameter and a `long` parameter. In the *base-specifier-list*, where `DifferentBehavior` unambiguously refers to the class template, `bar<DifferentBehavior>(0)` only considers overload 2 and allows an implicit conversion from `0` (which is of type `int`) to `long`. Inside the class body where `DifferentBehavior` can refer to either the class template or the current instantiation depending on the context, both overloads 1 and 2 are considered. Overload 1 takes exactly an `int`, which makes the function an exact match. If the *injected-class-name* was introduced in the *base-specifier-list*, then this code would begin to fail the first `static_assert`, and instead the base class would become `WasType`.

3.4 Proposed Wording

In 6.4.2 [basic.scope.pdecl]/8:

The locus of an *injected-class-name* declaration (11.1 [class.pre]) is immediately following the ~~opening brace~~*class-head-name* of the class definition.

4 Prior Work

[CWG432] was reported in 2003 to ask about this exact feature. At the time the author believed that not introducing the *injected-class-name* in the *base-specifier-list* was accidental, and should be corrected with a defect report. This defect report did not go anywhere as far as I can tell. Given that this feature would change the meaning of existing code, I believe it is more appropriate to treat this as a feature with its own paper.

5 Conclusion

Allowing the *injected-class-name* in the *base-specifier-list* will make existing template metaprogramming more concise, easier to read, and less bug-prone. While some code may change behavior, it is the author's opinion that the code that will change behavior is already ambiguous to readers, even if it is not ambiguous in the standard. The code that changes from working to not-working will only require the author to clarify their intent. The code that changes meaning is, in the author's opinion, contrived and unclear in intent. The fact that the code already does something different if written one line later inside the body of the class means that most readers of the code will be unaware that the result is different. Picking one consistent interpretation for this expression will make it easier for people to understand and build a mental model for the rules of the language.

6 Acknowledgments

At C++Now 2023 at the first language feature in a week session someone shouted this idea out as a good proposal. Unfortunately I did not see who it was but thank you to the anonymous C++Now attendee who proposed working on this idea. Thank you to Barry Revzin and Steven Watanabe who helped find code that would change behavior given this change.

7 References

[CWG432] Daveed Vandevoorde. 2003-08-29. Is injected class name visible in base class specifier list? https://wg21.link/cwg432

[P0847R7] Barry Revzin, Gašper Ažman, Sy Brand, Ben Deane. 2021-07-14. Deducing this. https://wg21.link/p0847r7