Document Number: P2600R0
Date: 2022-06-15
Reply-to: Matthias Kretz <m.kretz@gsi.de>
Audience: EWGI, EWG
Target: C++26

A minimal ADL restriction to avoid ill-formed template instantiation

ABSTRACT

I researched and tested a minimal change to ADL to avoid ill-formed instantiations of templates through ADL. If ADL ignores not-yet-instantiated-template types (just like ADL ignores incomplete types) little to no functionality is lost while losing a sharp edge of the language. This paper presents the idea, discusses potential code breakage, and presents potential alternatives/extensions of the idea.

CONTENTS

1 Introduction

Consider the innocent-looking code in Listing 1 (as posted to the core reflector by Jonathan Wakely[1]).

```
struct Incomplete;
template <typename T> struct Wrap { T t; };

template <typename Unused>
struct Testable
{
  explicit operator bool() const { return true; }
};

int main()
{
  Testable<Incomplete> l;
  if (l) // OK
    return 1;
  if (!(bool)l) // OK
    return 0;
  if (!l) // OK
    return 0;

  Testable<Wrap<Incomplete>> l2;
  if (l2) // OK
    return 1;
  if (!(bool)l2) // OK
    return 0;
  if (!l2) // ERROR
    return 0;
}
```

Listing 1: Ill-formed instantiation of `Wrap<Incomplete>` because of ADL

The expressions `!l` and `!l2` lead to name lookup of `operator!`. The associated entities of `l2` are `Testable<Wrap<Incomplete>>`, `Wrap<Incomplete>`, and `Incomplete`. In this example `Testable<Wrap<Incomplete>>` must already have been successfully instantiated, otherwise the declaration of `l2` would have been ill-formed. The type `Incomplete` is incomplete and the failure to look for `operator!` inside `Incomplete` is ignored. The type `Wrap<Incomplete>` is a template specialization which has not been instantiated at this point in the translation. But instead of treating the type like the incomplete `Incomplete` type, the compiler attempts to instantiate the specialization, which leads to an ill-formed definition of its data member since it has incomplete type.

1 `http://lists.isocpp.org/core/2021/06/11161.php` (the basic idea of this paper was already hinted at in the resulting discussion thread)

The expression `!l` is well-formed since ADL ignores incomplete types. Consequently, a later definition of `Incomplete` as

```
struct Incomplete {
  friend bool operator!(Testable<Incomplete>) { return true; }
}
```

changes the value of the expression `!l`.

Why does the standard allow incomplete types (i.e. not require completion) as associated entities even though this can lead to ODR violations? Isn't the same reasoning applicable to templates that have not been instantiated yet? And what are the use cases for defining a hidden friend in `Wrap<Incomplete>` which wins in overload resolution when the argument is a `Testable<Wrap<Incomplete>>`? (The last question is not only a rhetoric question, see Section 2.3 for a possible answer.)

1.1 Is this a real problem?

To determine whether Listing 1 poses a real problem, consider that `Testable<T>` is equivalent to `std::unique_ptr<T>` wrt. `operator!`. I.e. a few corner cases, but still legitimate uses of `unique_ptr` lead to puzzling errors[2].

Furthermore, for `std::unique_ptr<std::array<Incomplete, N>>`, instantiation on ADL can occur in many more places; for example on range-based for loops as shown in Listing 2. The reason for the error in Listing 2 isn't the lookup of `begin` and `end` (it would

```
1  class Incomplete;
2  using Data = std::array<Incomplete, 3>;
3  using Ptr = std::unique_ptr<Data>;
4
5  void assert_nonnull (std::span<Ptr> x)
6  {
7    for (const Ptr &ptr : x)  // ERROR: 'std::array<_Tp, _Nm>::_M_elems' has
8      assert (ptr);           //        incomplete type
9  }
```

Listing 2: Iteration over `std::span<std::unique_ptr<std::array<Incomplete, N>>>` is ill-formed

be if `x` had no `begin` or `end` members) but rather that operators applied to the iterator lead to ADL which instantiates `std::array<Incomplete, 3>`. Thus, to write an iterator that doesn't break on legitimate use of incomplete types a library developer has to use the ADL-proofing pattern from Listing 3 as explained by O'Dwyer [P2538R0]. We might

2 OTOH, `unique_ptr` is not *boolean-testable*, so maybe users should simply learn to cast `unique_ptr`, `optional`, etc to `bool`?

```cpp
template <typename T>
struct IteratorImpl {
  class type {
    // ...
  };
};

template <typename T>
using Iterator = typename IteratorImpl<T>::type;
```

Listing 3: ADL-proofed iterator type

expect standard library developers to learn and apply this pattern. But we should not expect from any other C++ library developer to work around ADL surprises.

1.2 Instantiation via ADL interferes with the evolution of C++

With the current behavior of ADL, adding non-member **`operator[]`** or overloadable **`operator?:`** [P0917R3] to the language[3] is a breaking change. Existing code that currently compiles just fine would suddenly instantiate templates through ADL and thus potentially lead to ill-formed instantiation.

Instantiation via ADL is a concern that needs to be considered for language and library evolution whenever ADL is required for a new feature. This proposal makes language and library evolution simpler (or even turn breaking changes into non-breaking changes).

1.3 Less need for ADL proofing

To complete the picture, note that the issue is not specific to operators (e.g. Listing 4). However, for operators there's no simple "always fully qualify your calls" rule to avoid ADL.

```cpp
template<class T> struct Holder { T t; };
struct Incomplete;
Holder<Incomplete> *p;
int f(Holder<Incomplete>*);
int x = f(p);    // error: Holder<Incomplete> is an associated entity
int y = ::f(p);  // ok: no ADL
```

Listing 4: Minimal example triggering ill-formed instantiation, as presented in [P2538R0]

O'Dwyer [P2538R0] goes on to show how the original definition of `std::projected` leads to ill-formed instantiation on ADL. His proposed solution requires ADL-proofing of `std::projected`, which requires a change in how the class template is defined.

3 Motivation is / will be given in their own papers.

This proposal makes ADL-proofing for avoiding ill-formed instantiation unnecessary[4]. The language would lose one sharp edge at a minor cost (more complex wording & new, unlikely, and easy to work around corner case for users).

1.4 History

To estimate how much code might be affected by limiting template instantiation on ADL to the argument type itself, it is helpful to know that template instantiation via ADL had not been implemented in compilers for a long time. Template instantiation via ADL works since GCC 4.5.0 (April 2010), Clang 3.1.0 (May 2012), and since an ICC release between ICC 14 and 16[5].

2 Proposed solution

Let us consider a simple solution to make ADL avoid the above situations:
Don't instantiate templates via ADL except for the argument type itself.

Rationale: If a given associated entity has not been instantiated at this point in the translation,

1. the user might have avoided instantiation intentionally since instantiation would be ill-formed; and
2. the type was not important enough up to this point that instantiation was necessary — the chances for it to make a semantic difference are small —.

If I interpret the current wording correctly, the reason compilers instantiate templates via ADL is [temp.inst] p2:

[temp.inst]

2 **Unless a class template specialization is a declared specialization, the class template specialization is implicitly instantiated when the specialization is referenced in a context that requires a completely-defined object type or when the completeness of the class type affects the semantics of the program. […]**

In basically all cases, the completeness of the class type does not affect the semantics of the program, though. But the compiler cannot know that instantiation is unnecessary until after it instantiated the template. The subsequent note in [temp.inst] clarifies that the knowledge about whether a name exists or not is considered affecting semantics. Therefore implementations have no choice of avoiding instantiation, even if they kept track of ADL-relevant names (i.e. hidden friends).

4 ADL-proofing is still a useful tool for avoiding ADL hijacking.
5 I have only been able to test with the compilers available on Compiler Explorer

2.1 Necessary instantiation

The one case where instantiation via ADL is still required is shown in Listing 5. The situation in Listing 5 is unlikely (but certainly not impossible[6]), since almost every other use of the lvalue `x` would lead to instantiation.

```
template <typename T>
struct A {
  friend void f(const A&);
};

void g(const A<int>& x) {
  f(x);
}
```

Listing 5: Requires instantiation or reasonable code could break.

Therefore, given an argument of reference to `T`, ADL should instantiate `T` and its bases but none of its template arguments.

2.2 Namespaces of base classes

Consider Listing 6 which defines a base class in a different namespace (`A`) than the derived class template. In order for ADL to consider `A` as associated namespace when an

```
namespace A {
  class B {};
  void f(B*);
}
void f(void*);

template <class T> class C : public A::B {};

void g(C<int>* p) {
  f(p);    // calls  ::f  because C<int> has not been instantiated
  A::B* other_ptr = p; // instantiates C<int> to find its base types
  f(p);    // calls A::f  because C<int> has been instantiated before ADL
}

// analogue situation with incomplete types:
class I;
void g0(I* p) { f(p); } // calls ::f
class I : public A::B {};
void g1(I* p) { f(p); } // calls A::f
```

Listing 6: ADL in namespace of base class

6 E.g. `https://gcc.gnu.org/PR34870`, which lead to GCC instantiating templates via ADL

argument of type `C<int>*` is used, the base types of the class template need to be know. However, if ADL were not to instantiate `C<int>` anymore, the behavior would depend on the preceding code: whether `C<int>` had already been instantiated or not. The analogue issue for an incomplete type `I` exhibits the same behavior change in ADL after defining `I`. However, for the incomplete type it is more obvious why the base types' namespaces are not considered ("You didn't say that `A::B` is a base!").

Note that overload resolution still has to instantiate class templates to find bases, as shown in Listing 7. It is an open question whether template instantiation via ADL on

```
class B0 {};
namespace A {
  class B : public B0 {};
  void f(B*);
}
void f(void*);
void f(B0*);

template <class T> class C : public A::B {};

void g(C<int>* p) {
  f(p); // calls ::f(B0*)
        // candidates: ::f(void*) and ::f(B0*), overload resolution instantiates
        // C<int> and picks ::f(B0*), i.e. overload resolution works unchanged
  f(p); // calls A::f(A::B*)
        // because ADL now finds the additional candidate A::f(A::B*)
}
```

Listing 7: Class template instantiation on overload resolution if ADL does not instantiate `C<int>`

arguments of pointer type is enough of a problem to warrant the surprising behavior of the (contrived) example in Listing 7 (vs. the surprising behavior of Listing 4). As a more conservative change to ADL, given an argument of pointer to `T`, let ADL instantiate `T` and its bases but none of its template arguments.

Beyond references and pointers — as far as I have seen — there is little use in ADL instantiating templates. I propose to modify [basic.lookup.argdep] and/or [temp.inst] to treat not-yet-instantiated-template types like incomplete types except if the not-yet-instantiated-template type is the type of the function argument[7].

2.3 `std::reference_wrapper` example

`std::reference_wrapper<X>`, where `X` implements hidden friends, is the canonical example where this proposal has the potential for breaking existing code. An idea for

7 *stops handwaving*; this is nowhere near wording …

avoiding the breaking change is presented in Section 4.1. Note that, since C++20, `std::reference_wrapper<T>` does not require `T` to be complete and we can therefore construct valid examples with the type `std::reference_wrapper<Wrap<Incomplete>>`. The relevant class of examples follows the pattern shown in Listing 8. This pattern

```
struct X {
  friend constexpr int f(const X&) { return 1; }
  friend constexpr bool operator!(const X&) { return false; }
};

int g(std::reference_wrapper<X> ref) {
  return !ref ? 0 : f(ref);  // returns 1
}
```

Listing 8: Hidden friends are transparent for `std::reference_wrapper`

relies on:

1. `std::reference_wrapper<X>` is convertible to `X`, and
2. ADL looks into `X` for hidden friends when `std::reference_wrapper<X>` is a function argument (operand).

Note that `X` doesn't need to know about (or even spell out) `std::reference_wrapper` and can therefore work with any type that has a conversion operator to `X` *and where `X` is an associated entity*. Notably, and inconsistently, function arguments of type `XRef`, as

```
struct XRef {
  X* ptr;
  constexpr operator X&() const noexcept { return *ptr; }
};
```

Listing 9: Hidden friends are not transparent for a non-template reference wrapper

defined in Listing 9, will not find hidden friends of `X` on name lookup. See Section 4.1 for an idea that would resolve the inconsistency.

With the following ingredients we can build an example (see Listing 10) that would break with the proposed language change:

- A class template specialization `X<Y>` that has not been instantiated yet.
- A hidden friend in `X<Y>`.
- A function call (or operator) which wants to find said hidden friend in name lookup.
- Wrap everything as `std::reference_wrapper<X<Y>>`.

```cpp
template <typename T>
struct X
{
  T data;
  friend auto operator<=>(const X&, const X&) = default;
};

// static_assert(std::totally_ordered<X<int>>);
// the following fails unless X<int> was instantiated already (e.g. with the
//   line above)
static_assert(std::totally_ordered<std::reference_wrapper<X<int>>>);
```

Listing 10: `std::reference_wrapper` example which would fail unless line 8 is uncommented

As far as I can tell there is no class template in the standard library that fits this description. However, the `std::experimental::simd<T, Abi>` class template of the Parallelism TS 2 specifies its operators as hidden friends. Consequently, the example in Listing 11 would not instantiate `simd<float>` anymore via ADL and thus fail to compile.

```cpp
#include <experimental/simd>
#include <functional>

auto f(std::reference_wrapper<std::experimental::simd<float>> x) {
  return x == x; // would not instantiate simd<float> anymore
}
```

Listing 11: Regression when combining `std::reference_wrapper` with `std::experimental::simd`.

An example similar to Listing 11 is still highly unlikely to occur outside of code snippets specifically written for demonstrating the issue. In general, an expression instantiating `simd<float>` (i.e. the wrapped type) would precede the use of the reference wrapper with high probability. More importantly, the explicit use of `std::reference_wrapper` or similar reference types is not common. At least `reference_wrapper` is unwrapped transparently for `std::bind` and typically unwraps automatically when a function with a reference to the wrapped type is called.

While the chance of breakage after restricting instantiation via ADL may be close to zero, we have no way of making a more substantial statement. It is impossible to prove that no breakage will occur, because the absence of certain code patterns cannot be proven.

3 IMPLEMENTATION EXPERIENCE

I implemented the presented idea for GCC. In this implementation the type of a pointer argument is not instantiated (cf. Section 2.2). The necessary change to the ADL code was straightforward: In principle the change involved only making template instantiation conditional on whether the type is the argument type itself or an associated entity. A modified GCC 12.1 can be obtained at `https://github.com/mattkretz/gcc/tree/7266f1c5f75fc7a970de`.

One test of GCC's testsuite (shown in Listing 12) broke with the change. The test case

```
template <typename> struct g { static const int h = 0; };
template <typename i> void declval() { static_assert(!g<i>::h,""); }
template <typename> struct a {
  template <typename d, typename c>
  friend auto f(d &&, c &&)
    noexcept(declval<c>) -> decltype(declval<c>); // { dg-error "different exception" }
};
template <typename d, typename c> auto f(d &&, c &&) -> decltype(declval<c>);
struct e {};
static_assert((e{}, declval<a<int>>),""); // { dg-error "no context to resolve type" }
```

Listing 12: GCC test that broke after implementation of less eager ADL (`declval<a<int>>` without parenthesis is no error — with parenthesis `a<int>` isn't an associated entity)

started failing because the expression `(e, declval<a<int>>)` requires name lookup of `operator,` and the type `a<int>` is an associated entity and therefore was instantiated (i.e. GCC was looking for a hidden friend comma operator in `a<int>`) which lead to the actual failure this test was looking for. To me Listing 12 is no motivation for holding back on this proposal, rather the opposite. For what it's worth, Clang doesn't even instantiate `a<int>` for this test case.

I found no further code breakage.

4 ALTERNATIVE SOLUTIONS / ADDITIONAL ADL IDEAS

I believe the above suggestion would be a strict improvement of the C++ language and better than the alternatives listed in the following. However, in order to reduce the potential for breaking existing code and to resolve an inconsistency of ADL with regard to reference wrappers, I believe the idea in Section 4.1 should be part of the final solution.

4.1 Add conversion operator return types to associated entities

Observe the inconsistency in Listing 13. Each of the three types `ref<foo>`, `int_ref`, and

```
struct foo {
  friend bool operator!(const foo&) noexcept;
};

template <class T>
struct ref {
  T* data;
  constexpr operator T&() const noexcept { return *data; }
};

struct int_ref {
  int* data;
  constexpr operator int&() const noexcept { return *data; }
};

struct foo_ref {
  foo* data;
  constexpr operator foo&() const noexcept { return *data; }
};

void f(const ref<foo>& r0, const int_ref& r1, const foo_ref& r2) {
  !r0; // finds foo::operator! because foo is an associated entity
  !r1; // OK
  !r2; // ERROR
}
```

Listing 13: Different name lookup for three reference types that should be equivalent (`https://godbolt.org/z/83fqT76vn`)

`foo_ref` (partially) model a reference (conversion operator defined as for `std::reference_wrapper`). But they behave considerably different because name lookup is different.

To make Listing 13 consistent, we could add another rule to ADL and add the return types of all conversion operators as associated entities. Since these types are much more likely to make a semantic difference on name lookup, template instantiation should be performed. However, only the conversion operators of the function argument type itself must be considered; conversion operators of return types of conversion operators are irrelevant, as are conversion operators of template arguments. As a consequence the examples in Listing 10 and Listing 11 would work again.

Obviously, more ADL has the potential to add more problems. But if this is introduced together with not instantiating templates anymore, the net effect should be positive. Counter-examples are welcome to improve the discussion of the idea.

I have implemented this idea in GCC and found no regressions in the GCC and libstdc++ test suites. More testers are welcome. A modified GCC 12.1 can be obtained at `https://github.com/mattkretz/gcc/tree/7895934f858fbb2e6039`.

4.2 opt-out/in ADL

If the committee cannot agree on changing the way ADL and template instantiation work, we are basically only left with new opt-in/out syntax, if we want to solve this problem. I am not ready to explore this idea. I mention it here to give a complete picture of the solution space.

As a straw-man example consider Listing 14. Note that such an opt-in/out solution

```
// opts out of considering implicit associated entities; instead `bar` and its
// associated entities are the associated entities of foo<T>:
template <typename T>
struct foo : adl bar { /* ... */ };

// no associated entities:
template <typename T>
struct meow : adl void { /* ... */ };

// fix the `unique_ptr` problem from Listing 1
namespace std {
  template<class T, class D = default_delete<T>>
  class unique_ptr : adl void {
    // ...
  };
}
```

Listing 14: Straw-man opt-in/out syntax for ADL

would not open a path for the evolution of non-member **operator**`[]` and overloadable **operator**`?:`.

4.3 tentative instantiation

It seems like an obvious solution to require IFINAE (instantiation failure is not an error) on ADL, i.e. to require tentative template instantiation. However, this would place a huge burden on implementations: The instantiation depth might be very deep, before the condition is found. IFINAE requires a rollback of all unfinished instantiations that lead to the issue. It seems like a huge but solvable task for compilers, but without implementation experience it is not a realistic path forward.

4.4 Inhibit instantiation if a template parameter is incomplete

A minimal solution for solving the problem in Listing 1 inhibits instantiation on ADL if a template parameter is incomplete. This would cover some of the cases where ADL appears too eager. However, only minor variations of such examples would instantiate the class templates again, leading to the same errors we were trying to fix. The solution would thus appear to be rather fragile and potentially more confusing than helpful.

4.5 Instantiation if and only if there are hidden friends with a matching name

The current wording already makes instantiation conditional on whether "the completeness of the class type affects the semantics of the program". Can we take this a step further, and require instantiation only if the compiler determines name lookup will find something it wouldn't find otherwise? Such a condition would come close to tentative instantiation but isn't necessarily the same thing. Name lookup requires less information of the complete type. Only when determining viability and performing overload resolution is the completely instantiated type unavoidable.

An implementation would have to

1. keep a list of names of all hidden friends, and
2. be able to determine *additional* associated namespaces from base classes.

Instantiation is only necessary if a hidden friend was found via name lookup. For the other case instantiation will likely be required for overload resolution, though.

5 Wording

TBD.

6 Suggested straw polls

None at this point.

7 Acknowledgements

This paper and my implementation for GCC are a reaction to a discussion on the CWG list with input from a Jonathan Wakely, Richard Smith, Peter Dimov, Barry Revzin, Arthur O'Dwyer, Ville Voutilainen, and Daveed Vandevoorde. Arthur O'Dwyer suggested to consider `std::reference_wrapper` examples and strengthen the motivation.

A BIBLIOGRAPHY

[P0917R3] Matthias Kretz. *P0917R3: Making operator?: overloadable*. ISO/IEC C++ Standards Committee Paper. 2019. URL: https://wg21.link/p0917r3.

[P2538R0] Arthur O'Dwyer. *P2538R0: ADL-proof std::projected*. ISO/IEC C++ Standards Committee Paper. 2022. URL: https://wg21.link/p2538r0.