Document	**P2822R0**
Date	**2024-02-15**
Reply-To	**Lewis Baker <lewissbaker@gmail.com>**
Audience	**EWG**

Providing user control of associated entities of class types

Abstract

This paper proposes an opt-in mechanism to allow explicitly specifying the set associated entities of classes and class templates.

This allows a fine control over argument-dependent lookup (ADL), overriding the default rules. ADL can then either be disabled or customized on a class-by class basis.

Default ADL lookup rules for types not using this feature remain unchanged.

Libraries that heavily use both class templates and argument-dependent-lookup can use this to better control the overload-set sizes to reduce compile times, without having to resort to elaborate library workarounds to define types without associated entities. It also allows adding associated entities to class types that would not normally have that type as associated.

Syntax

```
template<typename Source, typename Func>
class then_sender namespace(); // empty set of associated entities

template<typename T, typename Allocator = std::allocator<T>>
class array_map namespace(T); // only some of the template arguments are
                              // associated entities.

// Explicitly specifying an associated entity for a non-template.
class string_like namespace(std::string_view) {
public:
  // Implicit conversion to the associated type.
  operator std::string_view() const noexcept;
```

```
};

// Explicitly specifying the type of a non-type-template-parameter
// to be an associated entity. Example from P2781R3.
template<auto X>
struct std::constexpr_v namespace(decltype(X)) {
  constexpr operator decltype(X) const { return X; }
};

// TODO: Add example of how to emulate existing ADL behavior with new syntax

// Namespace containing various operator-overloads for foo_concept types
namespace foo_operators
{
  bool operator==(foo_concept const auto& a, foo_concept const auto& b);
}

// Brings in foo_operators as an associated namespace so that operators declared
// in foo_ops are considered when applying operators to this type.
struct some_foo namespace(foo_ops)
{
  // satisfies foo_concept
};
```

The proposed syntax is to add a new class-specifier `namespace(T1, T2, ...)` to class declarations immediately after the class identifier, before any `final` specifier (on class definitions).

Background

Current Behavior

The current rules of argument-dependent-lookup for unqualified calls requires the compiler to look in the associated entities and their enclosing namespaces of the types of arguments.

This behavior is intended for use in defining customisation points - originally for finding user-defined operator overloads. It allows user-defined types defined in other namespaces to be able to define overloads that are automatically considered whenever a call to that customisation-point passes an argument whose type is associated with the user-defined type without every overload for every type having to be considered for every call to that operator.

The associated entities of a class type, `T`, is the union of:

- the class, `T`, itself
- any direct or indirect base classes of `T` (but not their associated entities)

- if `T` is a nested class definition, then the class of which `T` is a member
  (Note, this is only the immediately enclosing class. It does not include any outer classes enclosing the immediately enclosing class, and does not include the enclosing class' associated entities)
- if `T` is a specialization of a class template, then;
  - for each type template argument, `A`, the associated entities of `A`
  - the templates used as template template arguments (but not their associated entities)

Then for each associated entity, the compiler searches for overloads of the named function declared as friends of that entity and also searches for overloads in the innermost enclosing, non-inline namespace as well as in every namespace in the set of inline namespaces [namespace.def] of that inner-most enclosing namespace.

Note that I suspect (although cannot find conclusive evidence for this) that the rationale for including template type arguments as associated entities was to allow the use of std::reference_wrapper<T> being passed as an argument to still find overloads declared in the namespace of T and, assuming that the arguments are

Previous Work

There have been a number of papers that previously tried to address issues with ADL.
This section attempts to summarize the prior art in this space.

P0934R0 - A Modest Proposal: Fixing ADL (Sutter, 2004/2018)

P0934R0 is a resubmission of a previous paper, N1792, with minor updates to the wording.

The paper lays out a number of footguns that result from the current rules of ADL, mainly focusing on accidental invocation of ADL and how this can lead to seemingly innocent code that is unspecified whether or not it is valid. It then proposes some changes to the rules to either eliminate or reduce the footguns.

It proposed making two changes to the rules around ADL:

1. Do not consider the types of template arguments as associated entities.
2. When determining viability of an overload found by ADL, requires that at least one of the parameter types of functions found by ADL is a parameter of the same type as the argument.

Discussion of the paper in Jacksonville was supportive, but raised some concerns about this breaking existing code.

It was also later pointed out that the change to remove template arguments as associated entities would break existing code that relied on passing a `std::reference_wrapper<T>`

argument to an unqualified call finding overloads in the namespace of `T` and then relying on the implicit conversion to `T&` for those overloads to be viable, assuming the argument corresponded to a parameter of type, `T&`.

The other concern with point 2 is that determining viability of overloads is done after template argument deduction, and so unconstrained overloads that deduce their parameters to the same type as the arguments passed to the ADL call would still be selected.

Implementation experience and usage experience was requested but the paper has not since been revised.

N1691 - Explicit Namespaces (Abrahams, 2004)

This paper proposed introducing a new `explicit namespace` syntax that would change the rules around unqualified function calls within any function declared/defined within that namespace.

Names that are intended to use ADL for unqualified lookup must be opted-in to with a using-declaration.

It also proposes allowing adding function overloads to other namespaces remotely by fully-qualifying the function name to make it easier to add overloads of customisable functions - presumably as an alternative to using ADL.

This paper is tackling the problem of accidentally invoking ADL when you didn't intend to, but does not address the problem of limiting the number of namespaces that ADL has to look in when you do (intentionally) invoke ADL.

N1912 - Namespace operator (Vandevoorde, 2005)

This paper points out the limitation in N1893 (a later revision of N1792) with regards to removing type template arguments from the set of associated entities.

This paper provides a sketch of a namespace operator as a potential fix for these issues. This would allow code that wants to call functions found in the namespace of some particular type to explicitly scope such a call by qualifying the name with `namespace(expr1, expr2, ...)::`.

The `namespace(x)::foo` syntax indicates that foo should be looked up in the associated namespaces of the type of the variable 'x - using the existing rules for computing associated entities/namespaces.

Like N1691, this paper tries to tackle the problem of accidentally making an ADL-call and proposes an explicit opt-in for looking up a name in the scope of some arguments. It does

improve upon N1691, however, in that it provides a bit more control over which arguments should be considered for ADL rather than being limited to using all of them.

This approach does not provide the ability to limit the set of associated namespaces that are searched for a given type, however. Applying the namespace operator to a variable of type `std::vector<liba::A<libb::B>>` still causes it to include `std::`, `liba::` and `libb::` namespaces in the set of namespaces to search.

Problems

Compile times

Libraries that make heavy use of a combination of class templates in conjunction with argument-dependent-lookup (ADL) based customisation points often have issues with compilation time.

Expression template libraries that have types with large template argument lists or that have deeply nested template arguments can often end up with a large number of associated entities, each of which need to be searched by the compiler whenever an argument of this type is passed to an ADL-call.

The P2300R7 `std::execution` design is such an expression-template library that is used with `tag_invoke` ADL calls. It explicitly calls out in the proposed wording the addition of class-templates annotated with "`// arguments are not associated entities`" for which the implementation is required to ensure that the template arguments for those types are not associated entities. This is done in order to reduce the number of entities that need to be searched during an invocation of one of the `tag_invoke`-based customisation points.

It is worth noting here, however, that there is a proposal in paper P2855R0 to move away from ADL-based customization points for `std::execution`, in favor of member-function-based concepts, partly because of the difficulties with controlling ADL name lookup.

The issue will still remain for expression-template libraries that want to make use of various mathematical operator overloads, however. For example, matrix expression templates, such as Eigen3, could potentially have compile-time benefits from a facility that allowed reducing the set of associated entities that the compiler needed to search whenever users use operator overloads.

Other libraries, such as `boost::hana`, have encountered situations where the number of associated entities were problematic and have had to develop workarounds to limit the set of associated types when passing them as arguments to ADL-calls.

Compiler diagnostics

Related to the compile-time issue of the compiler having to look at many more associated entity namespaces is the issue of compiler diagnostics.

Every time the compiler looks into some associated namespace for a name to be found by ADL and it finds a name, but the overload is not viable, the compiler adds that overload to the set of overloads considered so that it can print out the set of overloads that did not match.

If the set of associated entities is very large then the set of potential overloads considered can be very large, particularly for things like overloaded operators that are defined by many types.

Reducing the set of associated entities can therefore lead to better diagnostics by only showing the developer a smaller set of considered overloads.

The following example shows an example where we "accidentally" try to compare a `type_list` instance with an integer. The straight-forward implementation finds a large number of overloads and prints them all out in the diagnostic which the developer needs to trawl through to find the right info. The `type_list_noadl` version type limits the set of associated types and so finds no overloads, giving a much shorter diagnostic.

```
#include <string>
#include <vector>
#include <list>
#include <variant>

template<typename... Ts>
struct type_list {};

template<typename... Ts>
struct _type_list_noadl {
  struct type {};
};
template<typename... Ts>
using type_list_noadl = typename _type_list_noadl<Ts...>::type;

bool example() {
  type_list<std::string, std::vector<int>, std::list<bool>, float> x;
  return x == 42; // error: clang prints out 19 operator== candidates considered
}

bool example_noadl() {
  type_list_noadl<std::string, std::vector<int>, std::list<bool>, float> x;
  return x == 42; // error: clang prints out no operator== candidates found
}
```

See https://godbolt.org/z/vnY4aoTKM

Undesired template instantiation

Another potentially unexpected result of the current rules of ADL is that computing the set of associated entities of arguments of class type includes looking at the base-classes of that type, which in turn requires that the type is a complete type.

When combined with types that are specializations of a template, this can force instantiation of templates, which can lead to some unexpected compile-errors for users when incomplete types are involved.

For example: Given the following utility which allows users to test for membership in a type_list

```
#include <concepts>

template<typename... Ts> struct type_list {};

namespace detail
{
  template<typename T>
  constexpr bool contains(type_list<>) { return false; }

  template<typename T, typename... Rest>
  constexpr bool contains(type_list<T, Rest...>) { return true; }

  template<typename T, typename U, typename... Rest>
  requires (!std::same_as<T, U>)
  constexpr bool contains(type_list<U, Rest...>) {
    return contains<T>(type_list<Rest...>{});        // #1
  }
}

template<typename T, typename TL>
inline constexpr bool contains_v = detail::contains<T>(TL{});
```

The code here is passing empty structs (`type_list` specializations) as arguments to functions and the definition of these specializations does not require the types used as template arguments to be complete - a forward declaration will suffice.

For example, the following usages work fine (on current compilers*):

```
static_assert(contains_v<int, type_list<char, int, bool>>);

struct incomplete;
static_assert(contains_v<int, type_list<char, int, incomplete>>);

static_assert(contains_v<int, type_list<char, int, std::vector<incomplete>>>);
```

* - The current rules don't actually specify what the compiler should do for incomplete types. Issue CWG2857 tracks fixing this omission in the wording. Implementations currently treat an incomplete class type as if it has no base-classes, rather than treating this case as ill-formed, so we assume here that this is the intended design that will eventually be encoded in wording.

However, if we try to extend this to test a type-list with `std::array<incomplete, 5>`...

```
static_assert(contains_v<int, type_list<char, int, std::array<incomplete, 5>>>);
```

…we get the following: **error: field has incomplete type 'incomplete_type'**

The reason for this potentially surprising error is as follows:

- The call on line #1 is unqualified and so invokes rules for ADL
- One of the arguments is a `type_list<int, std::array<incomplete, 5>>`, which means that `std::array<incomplete, 5>` and its associated entities are added to the set of associated entities.
- Computing the set of associated entities of `std::array<incomplete, 5>` requires inspecting the base-classes, so the compiler tries to instantiate that specialization.
- This instantiation fails because the `std::array` class has a data-member whose type is an array of `incomplete` and data-member types are required to be complete at the point of instantiation.

But if we change the line marked #1 to qualify the `contains<T>()` function-name by prefixing with `detail::` then this no longer invokes ADL, the compiler no longer tries to instantiate the template and thus the compiler is now perfectly happy to accept the code.

Alternatively, if we happened to just have a forward-declaration of the `std::array` class template then the compiler would instantiate the template and this would result in an incomplete type. The compiler then treats the incomplete type as if it had no base-classes (see * note above) and the compiler can continue computing the set of associated entities.

While, in this example, the error is arguably the author's mistake for accidentally invoking ADL here, it does serve to show that ADL can cause implicit instantiation of templates in the name of computing the set of associated entities because the compiler needs to look at the base-classes, even if the template is not otherwise required by the call.

This can either affect compile-times due to additional, potentially undesired template instantiations, or can result in obscure compile errors such as the one described above.

The undesired template instantiation is an issue that the `boost::hana::type` has worked around by incorporating ADL isolation techniques to allow it to be used with function calls that compute information on types (in this case, `operator==` comparisons) without forcing those types to be instantiated. For example, see https://github.com/boostorg/hana/issues/5#issuecomment-51208723.

Sometimes we *want* another type to be an associated entity

If I have a type, `T`, that is implicitly convertible to some type, `U`, then users of `T` would often like to be able to use that type as a stand-in for `U` when calling functions.

However, if `T` does not have `U` as an associated entity then trying to call unqualified functions with an argument of type `T` it can fail to find the same overloads that passing the equivalent `U` type would have found.

For example, if we try to use the `std::constexpr_v` type from P2781, which just takes a non-type-template-parameter, and pass this to a function that expects that value, then if the `std::constexpr_v` doesn't have the type of the value as an associated entity then it won't find the appropriate overloads by ADL.

The paper P2781 gives the following example:

```
namespace std
{
  template<auto X>
  struct constexpr_v {
    using value_type = decltype(X);
    using type = value_type;
    constexpr operator value_type() const { return X; }
    // ...
  };

  template<auto X>
  inline constexpr constexpr_v<X> c_{};
}

namespace other
{
  // Example class from P2781.
  template<size_t N>
  struct strlit
  {
    constexpr strlit(char const (&str)[N]) { std::copy_n(str, N, value); }

    template<size_t M>
    constexpr bool operator==(strlit<M> rhs) const
    {
      return std::ranges::equal(bytes_, rhs.bytes_);
    }

    friend std::ostream & operator<<(std::ostream & os, strlit l)
    {
      assert(!l.value[N - 1] && "value must be null-terminated");
      return os.write(l.value, N - 1);
```

```
    }

    char value[N];
  };
}

std::cout << other::strlit("foo") << std::endl; // OK

auto f = std::c_<other::strlit("foo")>;
std::cout << f << std::endl;                    // error: Can't find operator<<
```

To work around this, the paper proposes adding an additional `typename T` template parameter to the `std::constexpr_v` class template, purely to add `T` as an associated entity so that the type has the desired ADL behavior.

i.e. the class declaration is

```
namespace std {
  template<auto X, typename T = decltype(X)>
  struct constexpr_v {
    // ...
  };
}
```

Such a template parameter should not need to be part of the public API, and would ideally be hidden from the user, but currently needs to be present due to the requirements to have the type of the non-type template parameter be an associated entity.

With this paper, it could instead be written as:

```
namespace std {
  template<auto X>
  struct constexpr_v namespace(decltype(X)) {
    // ...
  };
}
```

Current Approaches to taming ADL overload-set sizes

Hidden friends

Defining overloads as hidden friends restricts those overloads from being considered unless the unqualified call to a function includes a type that has this class as an associated entity.

Note that overloads declared as hidden friends of a type, `T`, will not be found when passing a type, `U`, that is implicitly convertible to `T`, unless `U` happens to have `T` as an associated entity (i.e. `T` is a base or an associated entity of a template argument).

Nested Non-Type class member of a class template

One technique, which allows defining types that need to be parameterised on some template argument without the template argument being an associated entity, is to define your type as a non-template nested type of an enclosing class template.

For example:

```
template<typename T>
struct _my_type {
  // A non-template member of a class template.
  struct type {
    int data;

    friend void swap(type& a, type& b) noexcept { /* impl here */ }
  };
};

template<typename T>
using my_type = typename _my_type<T>::type;
```

According to the rules in [basic.lookup.argdepl], `my_type` specializations have as associated entities, `_my_type<T>::type`, and `_my_type<T>` (as it is the immediately enclosing parent class) but do not have `T` as an associated entity as it does not transitively include the associated entities of parent classes.

Then you can pass instances of `my_type<T>` to an ADL call and it will only look for hidden friends in `_my_type<T>` and `_my_type<T>::type` and for functions declared in their enclosing namespace for overloads and will not consider `T` or its associated entities.

This approach has been employed widely in libunifex to reduce overload set sizes of calls to `tag_invoke` when passed sender/reciever objects, which tend to involve deeply nested templates.

This comes with a major downside, however, in that you can no longer use normal template argument deduction to deduce the template arguments to the `my_type<T>` type alias.

e.g. The following function declaration would be ill-formed:

```
template<typename T>
void example(const my_type<T>& obj) {
  // do something with 'obj' and 'T'
}
```

This is because we cannot deduce the template arguments for the outer class when passed an instance of the nested class.

This can be worked around by defining a separate concept that can be used to check whether or not a type is logically a specialization of the template-alias, and to extract the arguments. However, this requires a lot of boilerplate to implement.

For example: We can define the type instead as follows:

```
template<typename T>
struct _my_type {
  struct type {
    static constexpr bool _internal_is_my_type = true;
    using template_arg = T;

    int data;

    friend void swap(type& a, type& b) noexcept { /* impl here */ }
  };
};

template<typename T>
using my_type = typename _my_type<T>::type;

template<typename T>
constexpr bool is_my_type = false;

template<typename T>
  requires requires() {
    { T::_internal_is_my_type } -> std::same_as<bool>;
  }
constexpr bool is_my_type = T::_internal_is_my_type;
```

and then use it as follows:

```
template<typename MyType>
  requires is_my_type<MyType>
void example(const MyType& obj) {
  using T = typename MyType::template_arg;
  // do something with 'obj' and 'T'
}
```

This is a lot of extra boiler-plate compared to the version that hasn't had to limit the set of associated entities:

```
template<typename T>
struct my_type {
  int data;
  friend void swap(my_type& a, my_type& b) noexcept { /* impl here */ }
};

template<typename T>
void example(const my_type<T>& obj) {
  // do something with 'obj' and 'T'
}
```

Proposed Design

This paper proposes adding support for optionally annotating a class or class-template declaration with `namespace(`*`entity-list`*`)` specifier, known as an *associated-entities-specifier*, as a way of explicitly overriding the default rules for computing the associated entities of a class type.

The arguments to the associated-entities-specifier can either name types, class-templates or namespaces.

This can be used to either limit the set of associated entities/namespaces for class templates or extend the set of associated entities to include additional types not normally included in the set of associated entities (e.g. types that the object is implicitly convertible to) or namespaces not normally in the set of associated-namespaces.

If a type, `C`, has class type where the class declaration has an associated-entities-specifier or is a specialization of a class template which has an associated-entities-specifier then the set of associated entities of type `C` is the union of `C` with the sets of associated entities of each of the non-namespace entities listed in the associated-entities-specifier.

If the associated-entities-specifier's entity-list names a namespace then that namespace is added to the set of associated namespaces, allowing the type to bring in a namespace without having to name a type in that namespace as an associated entity.

Otherwise, the existing default rules for computing associated entities of a type will apply.

Examples

Defining a type with no associated entities:

```
// Template args are not associated entities
template<typename T>
class example namespace()
{ /* ... */ };
```

Defining a type with only a subset of template parameters as associated entities:

```
template<typename T, typename Alloc>
class some_container namespace(T)
{ /* ... */ };
```

Allows specifying multiple types as associated entities:

```
template<typename First, typename Second>
struct pair namespace(First, Second)
{ /* ... */ };
```

Allows specifying a pack of types as associated entities:

```
// Note that the base-class is not an associated entity
template<typename... Ts>
struct tuple namespace(Ts...) : detail::tuple_base<Ts...>
{ /* ... */ };
```

Allows specifying that types of non-type template parameters are associated entities (example from P2718)

```
template<auto X>
struct constexpr_v namespace(decltype(X))
{ /* ... */ };
```

All specifying another type as an associated entity for a non-template class that this class is implicitly convertible to:

```
struct string_like namespace(std::string_view) {
```

```
  operator std::string_view() const noexcept;
  // ...
};
```

Allow explicitly declaring another namespace containing generic operator definitions as an associated namespace:

```
template<typename T>
concept foo_concept = /* ... */;

// Namespace containing generic operators for types that satisfy some_concept
namespace foo_operators
{
  auto operator+(foo_concept const auto& t, foo_concept const auto& u) // #1
  { /* ... */ }

  // other operators ...
}

// Explicitly bring in operators from foo_operators namespace
struct my_foo namespace(foo_operators)
{ /* ... */ };
static_assert(foo_concept<my_foo>);

auto x = my_foo{} + my_foo{}; // calls #1
```

Allow specifying a template-template parameter as an associated entity:

```
template<template<typename T> class Container>
struct example namespace(Container)
{ /* ... */ };
```

Note that this will just bring in functions in the namespace containing the class template, and will not bring in hidden-friend functions declared within the class template, which will relate to a specific template specialization only.

Allow specifying a class template as an associated entity:

```
struct template_example namespace(std::vector)
{ /* ... */ };
```

Note that the above is equivalent to using the specifier, `namespace(std)`.

Design Discussion

Why this choice of syntax?

I initially explored the use of a context-sensitive keyword to use as part of the class declaration instead.
e.g. `struct foo associated(bar);`.

However, the problem with this is that this syntax is a valid forward declaration of a function named `associated` that returns a struct named `foo` and has a parameter of type `bar`.

If we instead use a keyword that is already reserved everywhere, like `namespace`, then this won't conflict with existing syntax.

Other keyword options to be considered:
- `struct foo using(bar);`
- `struct foo using namespace(bar);`

Forward declarations

One of the open design questions is whether or not the associated-entities-specifier is allowed to be placed on class forward declarations, or only on class definitions.

For example:

```
namespace bar { /*… */ }
namespace foo
{
  struct example namespace(bar);   // Valid?

  template<typename T>
  struct template_example namespace();  //  Valid?
}
```

An associated-entities specifier on a forward declaration allows restricting the set of namespaces considered for ADL for types that are still incomplete - in case they happen to be used in places that make them associated entities but that would not otherwise require the type to be complete.

It also allows usage of a forward-declaration of a type to have the same behavior with regards to ADL as if you had visibility of the class definition. This eliminates the possibility of different ADL behavior being a source of potential ODR violation for types that might be used when incomplete.

Note that the current rules don't actually specify what the compiler should do for incomplete types. Issue CWG2857 tracks fixing this omission in the wording. Implementations currently treat an incomplete class type as if it has no base-classes, rather than treating this case as ill-formed, so we assume here that this is the intended design that will eventually be encoded in wording.

With the default rules, a class template specialization implicitly has the associated entities of the template arguments, even if the class template itself is only a forward declaration. But once the type is defined, it also includes any base-classes in the set of associated entities. This can lead to inconsistent overload resolution if ADL calls are formed while the type is incomplete.

The cases where you need to be able to use an incomplete type often involve templates and meta-programming on types, where the actual type definition may not be needed. e.g. passing `type_list<...>` to an unqualified call / built-in operator expression.

In this case, however, ideally the type_list class have explicitly specified its set of associated entities to exclude the template arguments and so the passing `type_list<incomplete_type>` to an unqualified call would never be in a position where the associated entities needed to be computed. This weakens the need for declaring associated namespaces on forward declarations.

One use case that might benefit from supporting an associated-entities-specifier on a forward-declaration is where you have an opaque pointer to some struct and you want the user to pass pointers to this struct around and where you have the API defined in a separate namespace that you want to be found by ADL.

For example:

```
namespace foolib
{
  namespace detail::handle_ops
  {
    template<typename Handle>
    Handle clone_handle(Handle h);

    template<typename Handle>
    void close_handle(Handle h);

    // ... other ops
  }

  struct foo_t namespace(handle_ops);
  using foo_handle_t = foo_t*;

  foo_handle_t create_foo();
}
```

```
void usage() {
  foolib::foo_handle_t h = foolib::create_foo();

  // can call the functions using ADL without having to qualify them.
  foolib::foo_handle_t h2 = clone_handle(h);
  close_handle(h2);
}
```

The concrete use-case for this scenario seems weak, however.

This paper proposes adding the ability to include an associated-entities-specifier on declarations, mainly for the potential for reducing ODR violations that could be caused by inconsistent behavior of ADL around incomplete and complete types. This does complicate the design, however, as we need to now check that all forward declarations and definitions have the same associated-entities-specifier.
So I would also be fine with removing support for the specifier on declarations if there is not a lot of support for this.

Should it include base-classes by default?

The default ADL rule implicitly includes base-classes in the set of associated entities.

However, when explicitly specifying the set of associated entities, we may want to exclude some or all of the base-classes from participating in ADL - some base-classes may be privately inherited and used only as an implementation detail and do not want that type to contribute to the public interface of the class being defined.

To give more control, this paper proposes that types with an associated-entities-specifier should not include base-classes by default, requiring the user to instead explicitly list which base-classes should be considered associated entities.

For example:

```
namespace foo
{
  struct base {
    friend void adlcall(const base&);
  };
}

namespace bar
{
  struct other_base {
  };

  template<typename T>
```

```
  void adlcall(const T& x);
}


struct my_type namespace(foo::base)
  : foo::base, bar::other_base
{
  // ...
};

void example() {
  my_type t;
  adlcall(t); // Calls foo::adlcall(const foo::base&)
              // Not ambiguous because bar::other_base is not
              // an associated entity and so bar::adlcall() is
              // not considered.
}
```

Should it include the class itself?

Another question is whether the associated entities of a class type with an associated-entities-specifier should implicitly include itself in the set of associated entities when performing argument-dependent-lookup, or whether the type declaration must list itself in its associated-entities-specifier to have the type considered its own associated entity.

This could potentially be useful for types defined in the same namespace as a lot of other namespace-scope functions which the user doesn't want to be found accidentally by ADL when passing an argument of that class type.

However, having to explicitly list the type itself is potentially tedious to type, particularly for class templates where you would have to repeat the type name with all of the template arguments as the alias for the current specialization is not in-scope yet.

Example: We'd need to repeat the full type-name in the namespace() list if a type wasn't implicitly its own associated entity

```
namespace foo
{
  template<typename Key, typename Value,
           typename Allocator = std::allocator<std::pair<const Key,Value>>>
  struct my_map namespace(my_map<Key, Value, Allocator>) {
    // ...

    friend bool operator==(const my_map& a, const my_map& b) { /* … */ }
  };
}
```

Further, if you wanted to limit the scope to not include any template arguments as associated entities, but forget to list the type itself, then you can end up with some potentially confusing error messages:

```
namespace foo
{
  template<typename Key, typename Value,
           typename Allocator = std::allocator<std::pair<const Key,Value>>>
  struct my_map namespace() {
    // ...

    friend bool operator==(const my_map& a, const my_map& b) { /* … */ }
  };
}

void example(const foo::my_map& a, const foo::my_map& b) {
  if (a == b)     // error: No viable overloads of operator== found (huh?)
    std::print("equal");
  else
    std::print("not equal");
}
```

This could be both confusing and also, in some cases, silently mask bugs.

For example: Here, if we didn't implicitly include the type itself, we would silently fail to call the user-provided overload of swap()

```
namespace foo
{
  template<typename T>
  struct my_container namespace() {
    // ...

    void swap(my_container& other) noexcept;

    friend void swap(my_container& a, my_container& b) { a.swap(b); }
  };
}

void example() {
  my_container<int> a = ...;
  my_container<int> b = ...;

  using std::swap;
  swap(a, b); // whoops! calls std::swap() instead of user-provided


  if (a == b)     // error: No viable overloads of operator== found (huh?)
    std::print("equal");
  else
```

```
    std::print(“not equal”);
}
```

On balance, I think it makes sense to always include the class itself in the set of associated entities being searched as the loss of flexibility of being able to define types that have zero associated entities is outweighed by the potential for confusion and bugs that can occur if the user forgets to list their own type in the list.

This paper proposes that a type should implicitly be in its own set of associated entities.

Should it include the immediately enclosing namespace of the class?

The current rules for argument-dependent lookup compute the set of associated entities and then look in the immediately enclosing non-inline namespace for overloads, in addition to also looking for hidden-friend functions declared in the scope of a class that is an associated entity.

As the associated-entities-specifier allows including namespaces in the list of entities, we could potentially give a finer grain of control and have the behavior of listing an empty set of associated entities only search for hidden friends of the current type. And if the author of the type wants overloads to be found in the parent namespace, then the author needs to explicitly list the parent namespace in the list of associated entities.

For example: If explicitly listing the enclosing namespace was required

```
namespace foo
{
  struct X namespace() {
    friend bool operator==(X, X) { return true; }
  };
  X operator+(X, X);

  struct Y namespace(foo) {
    friend bool operator==(Y, Y) { return true; }
  };
  Y operator+(Y, Y);
}

X usage(X a, X b) {
  if (a == b) {    // ok: found as a hidden friend
    return a + b;  // error: 'foo' is not an associated namespace.
                   // can't find foo::operator+(X,X)
  }
  return a;
}

Y usage(Y a, Y b) {
  if (a == b) {    // ok: found as a hidden friend
    return a + b;  // ok: 'foo' is an associated namespace.
                   // calls foo::operator+(Y,Y)
  }
  return a;
}
```

This might help authors of types declared in the same namespace as a number of other namespace-scope functions (e.g. types in std::) to limit the set of functions that can be found when invoking unqualified calls with an argument of that type.

However, a relatively straight-forward workaround for this is to declare each type its own private sub-namespace and then import the type into the parent namespace.

For example: Give each type its own namespace to prevent functions in parent namespace from being found by ADL

```
namespace foo
{
  // lots of namespace-scope functions here...

  namespace _some_type {
    template<typename T>
    struct some_type namespace() {
    };
  }
  using namespace _some_type;
```

```
}
```

Giving the user a way to do something similar using just the associated-entities-specifier might be more convenient and give shorter symbol names, but it wouldn't give any functionality that wasn't already available today.

This paper currently proposes that the existing behavior is retained - i.e. that both hidden friends and the immediately enclosing namespace are considered for an associated entity of class type, even if that class type has an associated-entities-specifier.

Recursively computing associated entities

The current rules for computing the set of associated entities are complicated and non-uniform. For example, the set of associated entities includes the base classes but not the associated entities of those base classes, the immediately enclosing class if it is a class member, but not its associated entities, while for template arguments it does include the associated entities of the template arguments.

This can lead to some odd behavior:

```
namespace foo {
  struct X {
     friend void adl_func(const X& x) {}
  };
}

template<typename  T>
struct convertible {
  operator T&() const noexcept;
};

struct derived : convertible<X> {};

void example() {
  derived d;
  convertible<X>& c = d;

  adl_func(c); // ok
  adl_func(d); // error: no such function found adl_func()
}
```

In this case, when we inherited publicly from a template base class, we didn't get all of the public interface of that base-class because with the current rules of ADL we don't consider the associated entities of base classes, and so `foo::X` was not considered an associated entity.

This paper makes the observation that the main reason for including another type as an associated entity is because functions in a type's namespace (or hidden friends) form part of the public interface of the current type being defined.

If we then consider this from a third type's perspective - if the first type's namespace forms part of the public interface of a second type, and the second type's public interface forms part of the third type's public interface, then this means that the first type's public interface also forms part of the third type's public interface - i.e. the relationship here should be transitive.

It is for this reason that when computing the set of associated entities for an ADL call, we include the associated entities of each non-namespace entity in the explicitly specified set of associated entities.

Class template instantiation

If a class template declaration includes an associated-entities-specifier then there should be no need to instantiate the template in order to compute the set of associated entities as we don't need to inspect the base classes.

We want to avoid unnecessarily instantiating templates in case instantiation of those templates would be ill-formed.

This situation can be common when template-metaprogramming to compute types.
For example, see https://github.com/boostorg/hana/issues/5#issuecomment-51208723

Class template specializations

Do we want to allow class template specializations to provide different associated namespace patterns?

For example: Do we want the following behavior

```
namespace obj_ops {
  void f(auto);
}
namespace ptr_ops {
  void f(auto);
}
namespace baz {
  template<typename T>
  struct X namespace(obj_ops) {};

  template<typename T>
  struct X<T*> namespace(ptr_ops) {};
}
```

```
void test() {
  baz::X<int> x1;
  f(x1); // calls obj_ops::f()

  baz::X<int*> x2;
  f(x2); // calls ptr_ops::f()
}
```

I think the answer here is “yes”.

Associated entities of template template arguments

Example 1: Should a class template with an associated-entities-specifier include the associated entities from the primary class declaration?

```
namespace foo {
  void f(auto);
}
namespace bar {
  template<typename T>
  struct X namespace(foo) {};
}
namespace baz {
  template<template<typename> class X>
  struct Y namespace(X) {};
}
void test(){
  baz::Y<bar::X> y;
  f(y); // finds foo::f() ?
}
```

Example 2: Should a class template with an associated-entities-specifier that includes some dependent types include the non-dependent entities from the primary class declaration?

```
namespace foo {
  void f(auto);
}
namespace bar {
  template<typename T>
  struct X namespace(foo, T) {};
}
namespace baz {
  template<template<typename> class X>
  struct Y namespace(X) {};
}
void test(){
  baz::Y<bar::X> y;
  f(y); // finds foo::f() ?
}
```

This paper currently does not attempt to compute any additional associated entities for class templates with associated-entities-specifiers. The associated-entities-specifiers are only used to compute the associated entities of class specializations of that class template.

Implementation Experience

Not yet implemented.

An initial survey of the Clang code-base did not indicate any expected implementation difficulties.

Proposed wording changes

Modify “Classes [class]” subsection “Preamble [class.pre]”

Edit paragraph 1 as follows:

A class is a type. Its name becomes a *class-name* ([class.name]) within its scope.

class-name:
identifier
simple-template-id

A *class-specifier* or an *elaborated-type-specifier* ([dcl.type.elab]) is used to make a *class-name*. An object of a class consists of a (possibly empty) sequence of members and base class objects.

class-specifier:
class-head `{` *member-specification*$_{opt}$ `}`

class-head:
class-key attribute-specifier-seq$_{opt}$ *class-head-name* *associated-entities-specifier*$_{opt}$ *class-virt-specifier*$_{opt}$ *base-clause*$_{opt}$
class-key attribute-specifier-seq$_{opt}$ *associated-entities-specifier*$_{opt}$ *base-clause*$_{opt}$

class-head-name:
nested-name-specifier$_{opt}$ *class-name*

class-virt-specifier:
`final`

class-key:
`class`
`struct`
`union`

Modify “Elaborated type specifiers [dcl.type.elab]”

Modify the grammar production as follows:

elaborated-type-specifier:
class-key attribute-specifier-seq$_{opt}$ *nested-name-specifier*$_{opt}$ *identifier* *associated-entities-specifier*$_{opt}$
class-key simple-template-id *associated-entities-specifier*$_{opt}$
class-key nested-name-specifier `template`$_{opt}$ *simple-template-id* *associated-entities-specifier*$_{opt}$
`enum` *nested-name-specifier*$_{opt}$ *identifier*

Modify paragraph 2 as follows:

If an *elaborated-type-specifier* is the sole constituent of a declaration, the declaration is ill-formed unless it is an explicit specialization ([temp.expl.spec]), an explicit instantiation ([temp.explicit]) or it has one of the following forms:
class-key attribute-specifier-seq$_{opt}$ *identifier* *associated-entities-specifier*$_{opt}$;
class-key attribute-specifier-seq$_{opt}$ *simple-template-id* *associated-entities-specifier*$_{opt}$;

Insert the following paragraph at the end of [dcl.type.elab]

If an *elaborated-type-specifier* that declares a class type contains an *associated-entities-specifier* then the class type has an *explicitly specified set of associated entities* that is the set of entities named in the *associated-entity-seq*.

Otherwise, if an elaborated-type-specifier that is part of a template-declaration contains an associated-entities-specifier then the class template declaration has a template associated-entities-specifier. A specialization of the declared class template computes that type’s explicitly specified set of associated entities by performing template argument substitution into any type-ids in the associated-entities-specifier. [Note: this does not require instantiation of the class template. — End note]

The *declaration* in an *explicit-instantiation* shall not contain an *associated-entities-specifier*.

Add a new section “Explicitly specified associated entities [class.assoc]”

Insert section after the section [class.name].

Explicitly specified associated entities [class.assoc]

associated-entities-specifier:
`namespace (` *associated-entity-seq*$_{opt}$ `)`

associated-entity-seq:
associated-entity-item
associated-entity-seq `,` *associated-entity-item*

associated-entity-item:
type-id `...`$_{opt}$
qualified-namespace-specifier

A class type whose declaration contains an *associated-entities-specifier* has an *explicitly specified set of associated entities* consisting of the entities named by any *associated-entity-item* terms in the *associated-entity-seq*.

An *associated-entity-item* shall either;
- name a namespace; or
- denote a type; or
- denote an expanded pack of types

The explicitly specified set of associated entities, if present on a class declaration or class definition, is used to compute the set of associated namespaces for argument-dependent name lookup ([basic.lookup.argdep]).

Modify “Argument-dependent name lookup [basic.lookup.argdep]”

Modify paragraph 3 of [basic.lookup.argdep] as follows:

For each argument type T in the function call, there is a set of zero or more *associated entities* to be considered. The set of entities is determined entirely by the types of the function arguments (and any template template arguments). Any *typedef-names* and *using-declarations* used to specify the types do not contribute to this set. The set of entities is determined in the following way:

- If T is a fundamental type, its associated set of entities is empty.
- If T is a class type (including unions)
  - If T has an explicitly specified set of associated entities ([class.assoc]), then;
    - its associated entities are;
      - the class itself; and
      - the associated entities of the non-namespace members of its explicitly specified set of associated entities; and
      - the namespace members of its explicitly specified set of associated entities;
    - [Note:
      A class type that is a specialization of a class template that has an explicitly specified set of associated entities in its declaration or definition is not required to be instantiated to compute its set of associated entities. Only template argument substitution into the

associated-entities-seq is required.
— *end note*]

- otherwise, its associated entities are: the class itself; the class of which it is a member, if any; and its direct and indirect base classes.

  Furthermore, if T is a class template specialization, its associated entities also include: the entities associated with the types of the template arguments provided for template type parameters; the templates used as template template arguments; and the classes of which any member templates used as template template arguments are members.

  [*Note*:
  Non-type template arguments do not contribute to the set of associated entities.
  — *end note*]

- If T is an enumeration type, its associated entities are T and, if it is a class member, the member's class.
- If T is a pointer to U or an array of U, its associated entities are those associated with U.
- If T is a reference to U or a cv U, its associated set of entities are those associated with U.
- If T is a function type, its associated entities are those associated with the function parameter types and those associated with the return type.
- If T is a pointer to a member function of a class X, its associated entities are those associated with the function parameter types and return type, together with those associated with X.
- If T is a pointer to a data member of class X, its associated entities are those associated with the member type together with those associated with X.

In addition, if the argument is an overload set or the address of such a set, its associated entities are the union of those associated with each of the members of the set, i.e., the entities associated with its parameter types and return type.
Additionally, if the aforementioned overload set is named with a *template-id*, its associated entities also include its template *template-arguments* and those associated with its type *template-arguments*.

Modify paragraph 4 of [basic.lookup.argdep] as follows:

The *associated namespaces* for a call are the namespace associated entities and the innermost enclosing non-inline namespaces for its non-namespace associated entities as well as every element of the inline namespace set ([namespace.def]) of those namespaces.
Argument-dependent lookup finds all declarations of functions and function templates that

- are found by a search of any associated namespace, or
- are declared as a friend ([class.friend]) of any class with a reachable definition in the set of associated entities, or
- are exported, are attached to a named module M ([module.interface]), do not appear in the translation unit containing the point of the lookup, and have the same innermost enclosing non-inline namespace scope as a declaration of an associated entity attached to M ([basic.link]).

If the lookup is for a dependent name ([temp.dep], [temp.dep.candidate]), the above lookup is also performed from each point in the instantiation context ([module.context]) of the lookup, additionally ignoring any declaration that appears in another translation unit, is attached to the global module, and is either discarded ([module.global.frag]) or has internal linkage.

Acknowledgements

Thanks to Corentin Jabot, Walter Brown for feedback and input to the design.