NOKIA

Nokia Networks

Traits: An Alternative Design for C++20 Customization Points

Vicente J. Botet Escribá

code:dive Nov 2016

What is the problem?

The standard customization points cannot be customized
for concepts (types satisfying some requirements) requiring recursion
… nor for non deduced types.

The standard customization points **swap, begin, end**, … behave like pseudo keywords

The standard customization approach **based on ADL** cannot be used for user libraries to customize STL types

ADL= Argument Dependent Looup

Outline

What is a customization point and why do we need them?

Goals of a good design

C++17 approach

N4381 - Range TS - customization point approach

N1691 Explicit namespace

Traits: Alternative customization point approach

Conclusion

Outline

What is a customization point and why do we need them?

Goals for a good design

C++17 approach

N4381 - Range TS - customization point approach

N1691 Explicit namespace

Traits: Alternative customization point approach

Conclusion

NOKIA

n1691 - Customization point: Definition

“A point of customization is a **procedure** (like swap) whose implementation might be **supplied or refined** by clients for specific types, and that is used by a library component (like sort)..”

N4381 - Customization point: Definition

[n4381] “A customization point, as will be discussed in this document, is a **function** used by the Standard Library that can be **overloaded** on user-defined types in the user’s namespace and that is found by **argument-dependent lookup**.”

What is Customization point

A customization point is **code** used by a library that can be **provided** for user-defined types or more generally for the types satisfying some specific requirements (concepts?)

We have two sides of the interface:

The client side

The customization side

It is not only about syntax, but about semantics.

Why do we need them?

```
class X {
public:
   iterator begin();
   iterator end();
   ...
};

template <class C)
void doSomething(C& c)
    for(auto it = c.begin(); it != c.end(); ++it) {…}
};

X x;
doSomething(x);
```

Doesn't work for C-arrays

NOKIA

Why do we need them?

```
namespace std {
  template< class C >
    constexpr auto begin( C& c ) -> decltype(c.begin());
  …

  template< class T, std::size_t N >
    constexpr T* begin( T (&array)[N] );
  ...
}

template <class C) void doSomething(C& c){
    for(auto it = std::begin(c); it != std::end(c); ++it) {…}
};
```

Overload, return type deduction and SFINAE

Substitution Failure Is Not An Error

NOKIA

Outline

- What is a customization point and why do we need them?
- **Goals of a good design**
- C++17 approach
- N4381 - Range TS - customization point approach
- N1691 Explicit namespace
- Traits: Alternative customization point approach
- Conclusion

Design Goals

A customization points must be associated to a concept

Requires clients to be explicit about which concept they mean to customize

Provides a clean way to supply customization points for UDT or types satisfying some requirements in a specific file

Provides a way to extend customizations for concepts

Provides a way to be explicit about where ADL shouldn't apply

Calls to the customization point should be **optimally efficient** by any reasonably modern compiler.

The solution should **not** introduce **excessive executable size bloat.**

Provides a path on how existing code can be migrated to use the new facility.

NOKIA

Goals of a (new) customization point design

A customization point shall always be associated to a concept

The customization shall respect the concept constraints

Easy to use correctly

- Easy to customize correctly (by the standard and the user)
- Easy to call correctly by the client

Hard to use incorrectly

- It is hard to customize by accident
- The client doesn't call it by accident

Efficient in time and space (this is C++)

Don't Repeat Yourself - Able to customize for concepts

As backward compatible as possible

NOKIA

Outline

What is a customization point and why do we need them?

Goals for a good design

C++17 approach

N4381 - Range TS - customization point approach

N1691 Explicit namespace

Traits: Alternative customization point approach

Conclusion

C++17 approach

```
namespace MyContainers {
    class X {
        auto Begin(); ...
    };
    auto begin(X const & x) { return x.Begin(); }
}
```

std::begin(x)

x.Begin()

NOKIA

C++17 approach

```
namespace MyContainers {
    class X {
      auto Begin(); ...
    };
    auto begin(X const & x) { return x.Begin(); }
}
```

Overload begin(X)

ADL : Argument Dependent Lookup

NOKIA

C++17 approach

```
namespace MyContainers {
    class X {
      auto Begin(); ...
    };
    auto begin(X const & x) { return x.Begin(); }
}

template <class C) void doSomething(C& c){
    for (auto it = std::begin(c); it != std::end(c); ++it)
    {...}
}
```

Qualification → No ADL

NOKIA

C++17 approach

```
namespace MyContainers {
    class X {
      auto Begin(); ...
    };
    auto begin(X const & x) { return x.Begin(); }
}


template <class C) void doSomething(C& c){
    for (auto it = begin(c); it != end(c); ++it)
    {…}
}
```

No Qualification →
No default std definitions

NOKIA

C++17 approach

```
namespace MyContainers {
    class X {
      auto Begin(); ...
    };
    auto begin(X const & x) { return x.Begin(); }
}

template <class C) void doSomething(C& c){
    using std::begin; using std::end;
    for (auto it = begin(c); it != end(c); ++it)
    {...}
}
template <class C) void doSomething(C& c){
    for (auto& v : c)
    {...}
}
```

Correct standard way

Range-based for loop

NOKIA

C++17 approach

The previous way is risky

What we want is that the library do the ADL introduction for us

```
// explicit qualification
for (auto it = stdx::range::begin(c);
              it != stdx::range::end(c); ++it)
{...}
```

Make Interfaces
Easy to Use Correctly
and
Hard to Use Incorrectly

User interface != customization interface. Method Pattern

stdx stands for std2 or std::experimental

NOKIA

Techniques – Function Overload and ADL - begin/end

Overloading when using references or pointers has a specific feature.
What if we use the overloaded function with a derived class?

```
class Derived : vector<int> {… };
using stdx::range::begin;
using stdx::range::end;
Derived d;
 for (auto it = begin(d);
           it != end(d); ++it)
 {...}
```

We cannot make the difference between the customized type and its derived classes

This can be considered an advantage or a disadvantage of function overloads, depending on what you use overloaded types for.

NOKIA

Outline

What is a customization point and why do we need them?
Goals for a good design
C++17 approach
N4381 - Range TS - customization point approach
N1691 Explicit namespace
Traits: Alternative customization point approach
Conclusion

Range TS - Customization point: Definition

1 A customization point object is a **function object** (20.9) with a literal class type that interacts with user-defined types while enforcing **semantic requirements** on that interaction.

2 The type of a customization point object shall satisfy **Semiregular** (19.4.8).

3 **All instances** of a specific customization point object type **shall be equal**.

4 The type of a customization point object T shall satisfy **Function<const T, Args..>**() (19.5.2) when the types of Args... meet the requirements specified in that customization point object's definition. Otherwise, T shall not have a function call operator that participates in overload resolution.

5 Each customization point object type **constrains its return type** to satisfy a particular concept.

6 The library defines several named customization point objects. In every **translation unit** where such a name is defined, it shall refer to the **same instance** of the customization point object.

N4381 - Customization point - Goals

Code that calls `cust` either qualified as

```
stdx::cust(a);
```

or unqualified as

```
using stdx::cust;
cust(a);
```

should **behave identically**.

In particular, it should find any user-defined overloads in the argument's associated namespace(s).

Code that calls `cust` as

```
using stdx::cust;
cust(a);
```

should **not bypass any constraints** defined on `stdx::cust`.

Calls to the customization point should be **optimally efficient** by any reasonably modern compiler.

The solution should **not** introduce any potential **violations of the one-definition rule** or **excessive executable size bloat**.

NOKIA

N4381 - Customization point: - Approach

```
namespace std::experimental::ranges {
  namespace __detail {
    template <class T, size_t N>
    constexpr T* begin(T (&a)[N]) noexcept {return a;}
    template <class RangeLike>
    constexpr auto begin(RangeLike && rng) ->
      decltype(forward<RangeLike>(rng).begin()) { return forward<RangeLike>(rng).begin();}
    struct __begin_fn {
      template <class R> constexpr auto operator()(R && rng) const
        noexcept(noexcept(begin(forward<R>(rng)))) -> decltype(begin(forward<R>(rng))) {
        return begin(forward<R>(rng));
      }
    };
  }
  // To avoid ODR violations:
  template <class T>  constexpr T __static_const{};
  // std::begin is a global function object
  namespace {
    constexpr auto const & begin = __static_const<__detail::__begin_fn>;
  }
}
```

Note that ranges don't mean a concept

Add overload of the customization function `cust` in a specifc `_detail` namespace with the default definitions

N4381 - Customization point: - Approach

```
namespace std::experimental::ranges {
  namespace __detail {
    template <class T, size_t N>
    constexpr T* begin(T (&a)[N]) noexcept {return a;}
    template <class _RangeLike>
    constexpr auto begin(_RangeLike && rng) ->
      decltype(forward<_RangeLike>(rng).begin()) { return forward<_RangeLike>(rng).begin();}
    struct __begin_fn {
      template <class R> constexpr auto operator()(R && rng) const
        noexcept(noexcept(begin(forward<R>(rng)))) -> decltype(begin(forward<R>(rng))) {
        return begin(forward<R>(rng));
      }
    };
  }
  // To avoid ODR violations:
  template <class T>  constexpr T __static_const{};
  // std::begin is a global function object
  namespace {
    constexpr auto const & begin = __static_const<__detail::__begin_fn>;
  }
}
```

Define `_detail::cust_fn` as a function object

The function call operator of `cust_fn` makes an unqualified call to `cust`

N4381 - Customization point: - Approach

```
namespace std::experimental::ranges {
  namespace __detail {
    template <class T, size_t N>
    constexpr T* begin(T (&a)[N]) noexcept {return a;}
    template <class _RangeLike>
    constexpr auto begin(_RangeLike && rng) ->
      decltype(forward<_RangeLike>(rng).begin()) { return forward<_RangeLike>(rng).begin();}
    struct __begin_fn {
      template <class R> constexpr auto operator()(R && rng) const
        noexcept(noexcept(begin(forward<R>(rng)))) -> decltype(begin(forward<R>(rng))) {
        return begin(forward<R>(rng));
      }
    };
  }
  // To avoid ODR violations:
  template <class T>  constexpr T __static_const{};
  // std::begin is a global function object
  namespace {
    constexpr auto const & begin = __static_const<__detail::__begin_fn>;
  }
}
```

Define `cust` as
an instance of the function object
`_detail::cust_fn`

NOKIA

N4381 - Customization point: - Approach

```
using stde::cust;
cust(a);
```

`stde::cust` is always called as the first phase of lookup, the name `cust` will resolve to the global object `stde::cust`.

The second phase of lookup is not performed when lookup has found an object.

Compilers should have no difficulty removing the global reference to a function object, eliding the parameter, removing the indirection, and inlining the call.

N4381 uses a trick (valid) to ensure that.
C++17 inline variables would solve the issue at the language level.

```
constexpr inline auto const begin = __detail::__begin_fn{};
```


NOKIA

Range TS - Customization point: Definition

1 A customization point object is a **function object** (20.9) with a literal class type that interacts with user-defined types while enforcing **semantic requirements** on that interaction.

2 The type of a customization point object shall satisfy **Semiregular** (19.4.8).

3 **All instances** of a specific customization point object type **shall be equal**.

4 The type of a customization point object T shall satisfy **Function<const T, Args..>**() (19.5.2) when the types of Args... meet the requirements specified in that customization point object's definition. Otherwise, T shall not have a function call operator that participates in overload resolution.

5 Each customization point object type **constrains its return type** to satisfy a particular concept.

6 The library defines several named customization point objects. In every **translation unit** where such a name is defined, it shall refer to the **same instance** of the customization point object.

NOKIA

N4381 - Customization point: - drawbacks

As Range TS define function objects in namespace `std::experimental::ranges`, ADL in this namespace is not applicable.

For class customizable types define a friend function

```
namespace std::experimental::ranges {
  template <class Base, TagSpecifier... Tags>
  class tagged {
    // ...
    friend void swap(tagged & x, tagged & y) {
      // ...
    }
  };
}
```

More intrusive

Even if this works, it makes customization points more complex to explain than necessary

N4381 - Customization point: - drawbacks

For non-class customizable types

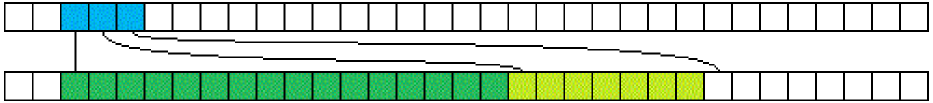
```
namespace std {
  namespace __hidden {
    enum memory_order {
      // ...
    };
    // If memory_order needed to hook cust...
    size_t cust(memory_order) {
      // ...
    }
  }
  using __hidden::memory_order;
}
```

User is unable to add an overload for memory_order

This is really more complex than we would expect.

NOKIA

N4381 - Customization point - mem_usage



```
std::vector<std::pair<X, Y>> vp;

framework::mem_usage(vp);
```

A customizable framework - Andrzej's C++ blog
https://akrzemi1.wordpress.com/2016/01/16/a-customizable-framework/

NOKIA

N4381 - Customization point - mem_usage

```
// framework/mem_usage.hpp
#include <type_traits> <product_type>
namespace std::experimental {
namespace __detail {
  template< class C >
    constexpr auto mem_usage( C& c ) -> decltype(c.mem_usage());
  template <typename T>
      constexpr enable_if_t<is_trivial_v<T>, size_t>
      mem_usage(const T& v) { return sizeof v; }
  // ...
  struct mem_usage_fn {
    template <typename T>
    constexpr auto operator()(const T& v) -> decltype(mem_usage(v))
    { return mem_usage(v); }
  };
}
   constexpr inline __detail::mem_usage_fn const mem_usage;
}
```

Forward to member

Trivial types

Forward into ADL namesapce

NOKIA

N4381 - Customization point - issues

```
// framework/mem_usage.hpp
#include <type_traits> <product_type>
namespace std::experimental {
namespace __detail {


    // ...
    template <typename T>
    auto mem_usage(T* v)-> decltype(*v);
    {
      return sizeof v + ( v ? mem_usage(*v) : 0 );
    }
    // ...
}
   constexpr inline __detail::mem_usage_fn const mem_usage;
}
```

Customization for pointers

Canot use stdx::mem_usage

NOKIA

N4381 - Customization point - issues

```
// framework/mem_usage.hpp
#include <type_traits> <product_type>
namespace std::experimental {
namespace __detail {
    // ...
    namespace pt {
      template <class PT, std::size_t... I>
      constexpr decltype(auto) mem_usage_impl( PT&& pt, index_sequence<I...> ) {
        return  sizeof(pt)
            + ( mem_usage(product_type::get<I>(forward<PT>(pt)) ) + ... )
            - ( sizeof(   product_type::get<I>(forward<PT>(pt)) ) + ... );
      }
    }

    template <typename PT>
      enable_if_t<is_product_type_v<remove_cv_reference_t<PT>>, size_t>
    mem_usage(PT&& pt) {
      return pt::mem_usage_impl(forward<PT>(pt),
                 product_type::element_sequence_for<PT>{});
    }
    // ...
}
```

Customization for product types

Canot use stdx::mem_usage

NOKIA

N4381 - Customization point - issues

```
// framework/mem_usage.hpp
#include <type_traits> <product_type>
namespace std::experimental {
namespace __detail {
    // ...
    namespace pt {
      template <class PT, std::size_t... I>
      constexpr decltype(auto) mem_usage_impl( PT&& pt, index_sequence<I...> ) {
        return  sizeof(pt)
            + ( mem_usage(product_type::
            - ( sizeof(   product_type::
      }
    }
    template <typename PT>
      enable_if_t<is_product_type_v<remo
    mem_usage(PT&& pt) {
      return pt::mem_usage_impl(forward<
               product_type::element_seq
    }
    // ...
}
```

```
#include <framework/mem_usage.hpp>
namespace stdx = std::experimental;
//...
std::pair<int,int> v ;
std::cout << stdx::mem_usage(v);
```

N4381 - Customization point - issues

```
// framework/mem_usage.hpp
#include <type_traits> <product_type>
namespace std::experimental {
namespace __detail {
    // ...
    template <typename T>
    auto mem_usage(T* v)
     -> decltype(*v);

    template <typename PT>
      enable_if_t<is_product_type_v<remo
    mem_usage(PT&& pt);
    // ...
}
```

```
#include <framework/mem_usage.hpp>
namespace stdx = std::experimental;
//...

std::pair<int,int>* v ;
std::cout << stdx::mem_usage(v);
```

COMPILE FAILS
Order maters

N4381 - Customization point - issues

```
// framework/mem_usage.hpp
#include <type_traits> <product_type>
namespace std::experimental {
namespace __detail {
    // ...
    namespace pt {
      template <class PT, std::size_t... I>
      constexpr decltype(auto) mem_usage
        return  sizeof(pt)
            + ( mem_usage(product_type::
            - ( sizeof(   product_type::
      }
    }
    template <typename PT>
      enable_if_t<is_product_type_v<remo
    mem_usage(PT&& pt) {
      return pt::mem_usage_impl(forward<
                product_type::element_seq
    }
    // ...
}
```

```
#include <framework/mem_usage.hpp>
namespace stdx = std::experimental;
//...

std::pair<Point2D, int>> v ;
std::cout << framework::mem_usage(v);
```

COMPILE FAILS
Nested product types

NOKIA

Serializable

```
struct player {
  std::optional<int> name;
  int hitpoints;
  int coins;
};

serializable::save(archive, opt);
 ...
auto x = serializable::load<optional<T>>(archive);
```

A *ProductType* of *Serializables* is *Serializable*

There is no parameter on which overload can dispatch

N4381 - Customization point - issues

I don't know how to apply N4381 customization approach to customize functions where the type to customize doesn't appears as function parameter

```
auto x = serializable::load<optional<T>>(archive);
```

We will need to pass the parameter by reference, which implies the type is default constructed

```
optional<T> x;
serializable::load(archive, x);
```

Or pass a tag that conveys the type to construct in the customization point

```
auto x load(type<optional<T>>{}, Archive&) {};
```

Others generic factories cases:

```
none<optional>();
make<optional>(x);
```


NOKIA

Outline

What is a customization point and why do we need them?
Goals for a good design
C++17 approach
N4381 - Range TS - customization point approach
N1691 Explicit namespace
Traits: Alternative customization point approach
Conclusion

n1691 - Customization point: Definition

“A point of customization is a procedure (like swap) whose implementation might be supplied or refined by clients for specific types, and that is used by a library component (like sort)..”

n1691 Explicit Namespaces Approach for Customizing points - Goals

Eliminates the problem of unintended "name reservation" when overloading is used for customization

Allows clients to be explicit about which algorithm they mean to customize via overloading

Provides a clean way to supply points of customization for user-defined types

Provides a way to be explicit about where ADL should apply

Eliminates unpredictable and unintended overloading due to ADL

Reduces the "syntactic weight" of client-supplied template specializations.

Describes how existing code can be migrated to use the new facility.

NOKIA

n1691 Explicit Namespaces - Approach for Customizing points - Problems

Unintended Overloading

« When calling functions in her own namespace, a conscientious programmer must disable ADL ...»

Name Reservation

« Users must take care to avoid defining these names in their own namespaces except where the intention is to customize a given library. ...»

Name Reservation - Interoperation

«...If two library implementors happen to choose the same function name as a point-of-customization with different semantics, the result is at best confusing: the user may need to create two overloads of the same name in his own namespace with different semantics.
Since the name is the same, it's quite likely that the semantics will be similar, but not identical. In the worst case, the functions have identical signatures, and the libraries simply refuse to interoperate in the user's application.
A corollary problem is that when providing a customization with a given name, there's no way for a client to indicate in code which library's meaning of that name she is implementing. »

NOKIA

C++17 - Customization point: Name reservation

The Standard Library already defines several customization points:

```
swap (C++98)
begin, cbegin, ... (C++11)
end, cend, ....
size, (C++17)
data, cdata (C++17)
```

We will need more in the future
Namespace std pollution
Which names are generic enough?
Behave almost as language keywords

Avoiding ADL

```
user_ns::swap(a,b);
(swap)(a,b);
swap_fn(a,b);
```


NOKIA

Outline

What is a customization point and why do we need them?

Goals for a good design

C++17 approach

N4381 - Range TS - customization point approach

N1691 Explicit namespace

Traits: Alternative customization point approach

Conclusion

NOKIA

Traits+Explicit namespace - Design Goals

A customization points must be associated to a concept

Requires clients to be explicit about which concept they mean to customize

Provides a clean way to supply customization points for UDT or types satisfying some requirements in a specific file

Provides a way to extend customizations for concepts

Provides a way to be explicit about where ADL shouldn't apply

Calls to the customization point should be **optimally efficient** by any reasonably modern compiler.

The solution should **not** introduce **excessive executable size bloat.**

Provides a path on how existing code can be migrated to use the new facility.

NOKIA

Traits + Explicit Namespaces - Approach

Add a new explicit namespace C++ and associated a namespace to the concept

Define a traits class template associated to the concept

Locate all the customization points associated to a concept in the explicit namespace that forward to the traits.

Client specialize explicitly the `traits` class template associated to the concept

Code calling `cust` qualified as

```
concept::cust(a);
```

or unqualified as

```
using stde::concept::cust;
cust(a);
```

should **behave identically**.

NOKIA

Traits : Definition

« **17.3.25 [defns.traits] traits class**
a class that encapsulates a set of types and functions necessary for class templates and function templates to manipulate objects of types for which they are instantiated »

Traits + Explicit Namespaces – explicit namespace

Proposal: “An explicit namespace is a namespace that requires qualification as a struct or a class does. »

```
explicit namespace range {

}
```


NOKIA

Traits + Explicit Namespaces – explicit namespace

Requires explicit qualification

```
stdx::range::begin(c);
```

Explicit introduction disallows ADL

```
using stdx::range::begin;
begin(c);
```

Behaves as if `stdx::range:begin` was a function object and so ADL is not used.

```
auto begin = stdx::range::begin; // P0119 overload set
begin(c);
```

Explicit introduction of two customization points conflicts

```
using stdx::range::begin;
using _3pp::book::begin; // compile error?
begin(c);
```

Reduces the problem of unintended "name reservation"

Provides a way to be explicit about where ADL shouldn't apply

NOKIA

Traits + Explicit Namespaces – explicit namespace

Requires explicit qualification

```
stdx::range::begin(c);
```

Explicit introduction disallows ADL

```
using stdx::range::begin;
begin(c);
```

Behaves as if `stdx::range:begin` was a function object and so ADL is not used.

```
auto begin = stdx::range::begin; // P0119 overload set
begin(c);
```

```
auto begin = [](auto&&... args)
  -> decltype(stdx::range::begin(std::forward<decltype(args)>(args)...))
{
  return stdx::range::begin(std::forward<decltype(args)>(args)...);
};
```


NOKIA

Traits + Explicit Namespaces – traits - begin/end

```
namespace std::experimental {
explicit namespace range { // here range stands for a concept

    template <class R, class Enabler=void>
    struct traits: traits<R, when<true>> {};

    // Default failing specialization
    template <typename R, bool condition>
    struct traits<R, when<condition>>
    {
        template <class T>
            static constexpr auto begin(T&& x) = delete;
        template <class T>
            static constexpr auto end(T&& x) = delete;
    };
}
}
```

Request clients to be explicit about which concept they mean to customize

NOKIA

Traits + Explicit Namespaces – traits - begin/end

```
namespace std::experimental {
explicit namespace range { // here range stands for a concept

        template <class R, class Enabler=void>
        struct traits: traits<R, when<true>> {};

        // Default failing specialization
        template <typename R, bool condition>
        struct traits<R, when<condition>>
        {
            template <class T>
                static constexpr auto begin(T&& x) = delete;
            template <class T>
                static constexpr auto end(T&& x) = delete;
        };
}
}
```

Enabler

An alias for `void`

template <bool cnd>
struct when ;

NOKIA

Traits + Explicit Namespaces – begin/end

```
namespace std::experimental {
explicit namespace range { // here range stands for a concept
    template <class T>
    static constexpr auto begin(T&& x)noecept
      -> decltype( traits<remove_cv_reference_t<T>>::apply_begin(forward<T>(x)) )

      { return traits<remove_cv_reference_t<T>>::apply_begin(forward<T>(x)); }
    template <class T>
    static constexpr auto end(T&& x)      noecept
      -> decltype( traits<remove_cv_reference_t<T>>::apply_end(forward<T>(x)) )
      { return traits<remove_cv_reference_t<T>>>::apply_end(forward<T>(x)); }
}
template <class R, class Enabler=void> struct is_range : false_type {};
template <class T> constexpr bool is_range_v = is_range<T>::value;
template <class T> struct is_range<T, void_t<
            decltype( range::begin(declval<T&>()) )
          , decltype( range::end(declval<T&>()) )
          >>: true_type {};
```

remove_cv-reference_t is a shorthand

Decay is not convenient here

NOKIA

Traits + Explicit Namespaces – begin/end

```
namespace std::experimental:range {
    template <typename R>
    struct traits<R, when<has_members_begin_end_v<R>>>
    {
        template <class T>
            static constexpr auto apply_begin(T&& x) { return x.begin(); }
        template <class T>
            static constexpr auto apply_end(T&& x) { return x.end(); }
    };
    template <typename R>
    struct traits<R, when<not has_members_begin_end_v<R>
                          and has_adl_begin_end_v<R>>>
    {
        template <class T>
            static constexpr auto apply_begin(T& x) { return begin(x); }
        template <class T>
            static constexpr auto apply_end(T& x) { return end(x); }
    };
}
```

Provides a clean way to supply customization points for user defined types or types satisfying some requirements

Provides a path on how existing code can be migrated to use the new facility.

NOKIA

Traits + Explicit Namespaces – Type class refinement

In the same way we inherit from a class we can define custom interfaces that refines others, e.g.

```
explicit namespace functor {
    …
}
explicit namespace applicative {
    using explicit namespace functor;
    …
}
explicit namespace monad {
    using explicit namespace applicative;
    …
}
```

Provides a way to extend customizations for concepts

We can as well use multiple refinement

Techniques – Class Template Specialization - mem_usage

```
// stdx/mem_usage.hpp
#include <type_traits>
namespace std::experimental::mem_usage_able {
  template <typename T, typename Enabler = void>
  struct trait : trait<T, when<true>>  {  };
  template <typename T, bool B>
  struct trait<T, when<B>>
  {
    static size_t apply(const T& v) = delete;
  };
  template <typename T>
  struct trait<T, when<is_trivial_v<T>>>
  {
    static size_t apply(const T& v) { return sizeof v; }
  };
  ...
}
```


NOKIA

Techniques – Class Template Specialization

```
// stdx/optional.hpp
#include <optional>
#include <stdx/mem_usage.hpp>
namespace std::experimental::mem_usage_able {
  template <typename U>
  struct traits<std::optional<U>>
  {
    template <typename T>
    static size_t apply(const T& v)
        -> decltype( mem_usage_able::mem_usage(*v) )     {
      size_t ans = sizeof(v);
      if (v) ans += mem_usage_able::mem_usage(*v) - sizeof(*v);
      return ans;
    }
  };
}
```

The customization can be associated to a specific file

SFINAE friendly

The customization uses the user interface

NOKIA

Techniques – Class Template Specialization

```
// optional
#include <stdx/mem_usage.hpp>

namespace std::experimental::mem_usage_able {
  template <typename U>
  struct traits<std::optional<U>>
  {
    template <typename T>
    static size_t apply(const T& v) -> mem_usage_able::mem_usage(*v)  {
      size_t ans = sizeof(v);
      if (v) ans += mem_usage_able::mem_usage(*v) - sizeof(*v);
      return ans;
    }
  };
}
```


NOKIA

Techniques – Class Template Specialization - mem_usage

```
// stdx/product_type.hpp
#include <stdx/mem_usage.hpp>
//...
namespace std::experimental::mem_usage_able {
  template <typename U>
  struct traits<U, when<is_product_type_v<T>>>
  {
    template <typename T>
    static size_t apply(const T& v) { return ...; }
  };
  ...
}
```

Generic customization can be associated to the concept file

Possible conflict between generic customizations

NOKIA

Techniques – Class Template Specialization - mem_usage

```
// stdx/product_type.hpp
#include <stdx/mem_usage.hpp>
//...
namespace std::experimental::mem_usage_able {
  template <typename U>
  struct traits<U, when<
        not is_trivial_v<T>
        and is_product_type_v<T>
    >>
  {
    template <typename T>
    static size_t apply(const T& v) { return ...; }
  };
  ...
}
```

How to solve a conflict ?

Manage priority between concepts

NOKIA

Techniques – Class Template Specialization - mem_usage

```
// stdx/product_type.hpp
#include <stdx/mem_usage>
namespace std::experimental {
namespace product_type {
  template <typename PT> auto mem_usage(PT&& v);
  struct as_mem_usage_able  {
    template <typename PT> static auto apply(PT&& v)
        -> product_type::mem_usage(forward<PT>(v));
  };
}
namespace mem_usage_able {
  template <typename U>
  struct traits<U, when<is_product_type_v<T>>>
  : product_type::as_mem_usage_able {};
  ...
}}
```

Define a specific mem_usage function for product types

Define a specific and reusable traits

Use it by default for product types

NOKIA

Techniques – Class Template Specialization - mem_usage

```
// Conflicting.hpp
#include <stdx/product_type.hpp>

class Conflicting {};

// Conflicting is a product type and trivial

namespace framework::mem_usage_able {
  struct traits<Conflicting>
  : product_type::as_mem_usage_able {};
}
```

Re-Use it for
a specific product type

NOKIA

Techniques – Class Template Specialization - mem_usage

```
// stdx/product_type.hpp
#include <stdx/mem_usage>
namespace std::experimental {
struct product_type_tag{};
namespace product_type {
  template <typename PT> auto mem_usage(PT&& v);
}
namespace mem_usage_able {
  struct traits<product_type_tag>  {
    template <typename PT> static auto apply(PT&& v)
        -> product_type::mem_usage(forward<PT>(v));
  };
}
namespace mem_usage_able {
  template <typename U>
  struct traits<U, when<is_product_type_v<T>>>
  : traits<product_type_tag> {};
}
}
```

Define a specific mem_usage function for product types

Define a specific and reusable traits

Use it by default for product types

NOKIA

Techniques – Function Overload and ADL - begin/end

When we specialize a class template trait, the specialization only works for the type X and nothing more.

If the user wants the traits to work for derived classes either it can reuse the X specialization, for a specific class D or specialize the trait for all derived classes

```
namespace lib::concept {
  template <typename T>
  struct traits<T, when<
    not std::is_same_v<X,T>
    and std::is_base_of_v<X,T>>>
  : struct traits<X> {};
}
```


NOKIA

Techniques – Standard library Comparison

N4381 approach	Traits
- All the concepts specialization go in the file where the function is defined	+ Specialization possible in a independent file
- NOT SFINAE friendly	+ SFINAE friendly
- Cannot be used when the customization for concepts is recursive	+ works for any type
+ people are used to ADL	- user customization less friendly

NOKIA

Techniques – Standard library Comparison

N4381 approach	Traits
- The user cannot know the namespace of some enums types	
+ Cannot be used to customize a user defined concept	- Users can add customizations for unexpected concepts (ODR?)
- Cannot be used to customize a user defined concept	+ works for any type
- Customizations can customize derived classes by default and unexpectedly	+ Specialization possible for a specific class, for the classes derived from a class and for a class and the derived class

Traits + Explicit Namespaces – Open Points -

One trait by function or a trait for all the concept functions

A trait specialized by type or a trait specialized by type class

Operators

NOKIA

Traits + Explicit Namespaces – Open Points - Individual/group traits

One traits by function versus one trait by concept

- Boost.Hana moved to one trait by function before the Boost review
- Each approach has surely its advantages and liabilities
- I need to do more research is this field

Traits + Explicit Namespaces – Open Points - Individual/group traits

```
explicit namespace range {
  template <class T>
  static constexpr auto begin(T& x)noecept
     -> decltype(    traits<T>::apply_begin(x)    );
  template <class T>
  static constexpr auto end(T& x) noecept
     -> decltype(     traits<T>::apply_end(x)    );
}
```

```
explicit namespace range {
  template <class T>
  static constexpr auto begin(T& x)noecept
     -> decltype(    trait_begin<T>::apply(x)    );
  template <class T>
  static constexpr auto end(T& x) noecept
     -> decltype(     trait_end<T>::apply(x)    );
}
```


NOKIA

Traits + Explicit Namespaces – Open Points - type class

A trait by type or a trait by type class.

- Boost.Hana uses a `tag_of<T>` to share the customization of similar types.
- Facebook Thrift library uses `type_class<T>`

```
template <class T>
constexpr auto begin(T& x) noexcept
    -> decltype( traits<tag_of<T>>::begin(x) );
```

This might reduce code bloat if a lot of types share the same tag.
Once you specialize `tag_of<T>`, any customization of T must be done using `tag_of<T>`.
Alternatively the user can reuse explicitly traits specializations for each concept.

```
template <class R>
struct traits<optional<R>>: traits<nullable_tag>> {};
```

Traits + Explicit Namespaces – Open Points - type class

```
// stdx/product_type.hpp
#include <stdx/mem_usage>
namespace std::experimental {
struct product_type_tag{};
namespace product_type {
  template <typename PT> auto mem_usage(PT&& v);
}
namespace mem_usage_able {
  struct traits<product_type_tag>  {
    template <typename PT> static auto apply(PT&& v)
        -> product_type::mem_usage(forward<PT>(v));
  };
}
namespace mem_usage_able {
  template <typename U>
  struct traits<U, when<is_product_type_v<T>>>
  : traits<product_type_tag> {};
}
}
```

Define a specific mem_usage function for product types

Define a specific and reusable traits

Use it by default for product types

NOKIA

Traits + Explicit Namespaces – Open Points - Operators

Having an infix syntax is sometimes more readable.
Operators are usually found by ADL and cannot be friendly qualified

```
auto x = monad::chain(monad::chain(m, f1), f2);
auto x  = m >> f1 >> f2;
```

UCS (Uniform call syntax) **couldn't** help here as far as the function call must be qualified

```
auto x = m.chain(f1).chain(f2);
```

Some functional libraries provide some kind of **infix** syntax for binary functions

```
auto x = m <chain> f1 <chain> f2; // Fit
auto x = m ^chain^ f1 ^chain^ f2; // Hana
```

Traits + Explicit Namespaces – Operators

Locate the operators associated to a concept in a specific operators namespace

```
namespace monad::operators {
    template <class Monad, class Callable>
        auto operator>>(Monad&& m, Callable&& f)
            -> monad::chain(m,f);
}
```

Explicit usage: The user needs to introduce this operator namespace

```
using namespace monad::operators;
auto x  = m >> f1 >> f2;
```

Usable inside Generic code

NOKIA

Outline

- What is a customization point and why do we need them?
- Goals for a good design
- C++17 approach
- N4381 - Range TS - customization point approach
- N1691 Explicit namespace
- Traits: Alternative customization point approach
- **Conclusion**

NOKIA

Conclusions

The standard customization points cannot be customized in general for
concepts
… nor for non deduced types

The standard customization ADL approach cannot be used for user libraries
Teaching people the standard customization approach doesn't help them build their frameworks

Method Pattern: split the interface into client and customization interface.
Generally being explicit is safer than implicit.
The presented approach solves some of the major ADL issues
The presented approach is more cumbersome that it should.

Questions ?

References

NOKIA

Standard proposals

[N1691] Explicit Namespaces
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1691.html

[N4381] Suggested Design for Customization Points
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4381.html

[N4569] Working Draft, C++ Extensions for Ranges
www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/n4569.pdf

[P0119R2] Overload sets as function arguments
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0119r2.pdf

[P0370r1] Ranges TS Design Updates Omnibus
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0370r1.html

[P0382R0] Comments on P0119: Overload sets as function arguments
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0382r0.html

Libraries

[Boost.Hana] Boost.Hana library
http://boostorg.github.io/hana/index.html

Blogs and Presentations

True Story: I Will Always Find You - Tales of C++ K-ballo - Agustín "K-ballo" Bergé
http://talesofcpp.fusionfenix.com/post-8/true-story-i-will-always-find-you

A customizable framework - Andrzej's C++ blog - Andrzej Krzemieński
https://akrzemi1.wordpress.com/2016/01/16/a-customizable-framework/

Customization Point Design in C++11 and Beyond - Eric Niebler
http://ericniebler.com/2014/10/21/customization-point-design-in-c11-and-beyond/

BoostCon 2014 - C++11 Library Design
https://www.youtube.com/watch?v=zgOF4NrQllo

CppCon 2016: “The Power of Reflection with Facebook's Thrift" - Marcelo Juchem
https://www.youtube.com/watch?v=tq0YfWFIVZA&index=27&list=PLHTh1InhhwT7J5jl4vAhO1WvGHUUFgUQH

NOKIA

Backup

nullable::none<TC> / make<TC>(x)

```
using stdx;
auto no  = nullable::none<std::optional>();
auto nup = nullable::none<std::unique_ptr>();
auto nsp = nullable::none<std::shared_ptr>();
auto na  = nullable::none<std::any>();
```

There is no parameter on which overload could dispatch

```
auto no  = factory::make<std::optional>(x);
auto nup = factory::make<expected<_,E>>(x);
auto nsp = factory::make<std::future>(x);
```

functor::transform

```
std::optional<Person> opt_p;
std::array<Person> arr_p;
std::variant<Person, Company> var_p_c;
auto age = [](auto const& x)-> unsigned {return x.age();};
using stdx;
auto res1 = functor::transform(age, opt_p);
auto res2 = functor::transform(age, arr_p);
auto res3 = functor::transform(age, var_p_c);

std::variant<unsigned, unsigned> <=> unsigned
```

Given `F(T) → U`

```
transform(F, optional<T>) → optional<U>
transform(F, vector<T>) → vector<U>
```

Traits as Customization Points in C++17

The Standard Library already defines several customization points used by the algorithms that are not functions:

```
numeric_traits<T>
char_traits<T>
iterator_traits<T>
allocators_traits<T>
pointer_traits<T>

common_type<T,U>
regex_type<T,U>
use_allocator

treat_as_floating_point<Rep>
duration_values<Rep>

tuple_size<T>
tuple_element<I,T>

default_order<T>

Executors (Concurrency and Parallelism TS)
Coroutines TS
```


NOKIA

Traits as Customization Points in proposals or TS

The Standard Library already defines several customization points used by the algorithms:

```
executors

coroutines
```

Techniques – User defined library Comparison

N4381 approach	Class Template Specialization
- depends possibly on the whole standard library	+ Specialization possible in a independent file
- NOT SFINAE friendly	+ SFINAE friendly
- canot recurse for std and concepts	+ works for any type
	- less usual

NOKIA

Traits + Explicit Namespaces – Operators

Locate all the operators in a operators namespace with an global adl tag

```
// operators_adl.hpp
namespace operators {    struct adl {};   }
// monad/operators.hpp
#include <operators_adl.hpp>
template <class T> struct monad_operators;
namespace operators {
    template <class Monad, class Callable>
        requires monad_operators_v<Monad>
        auto operator>>(Monad&& m, Callable&& f) -> monad::chain(m,f);
}
```

when the class inherits from this ADL tag.

```
class M : operators::adl {...};
template <> struct monad_operators<M> : true_type;
```

Usage

```
M m; auto x  = m >> f1 >> f2;
```


NOKIA