Language Support for Customizable Functions

Document #:	D2547R1
Date:	2022-03-12
Project:	Programming Language C++
Audience:	Evolution

Reply-to:	Lewis Baker (Woven-Planet) <lewissbaker@gmail.com> Corentin Jabot <corentinjabot@gmail.com> Gašper Ažman <gasper.azman@gmail.com>

Contents

1 Abstract

This paper proposes a language mechanism for defining customisable namespace-scope functions as a solution to the problems posed by [P2279R0] "We need a language mechanism for customization points".

2 Status of this proposal

This work is a preliminary initial design. We intend this proposal to replace the use of `tag_invoke` in [P2300R4] in the C++26 time frame. We need directional guidance from EWG and LEWG to refine the design over the next few months.

3 Short description of the proposed facilities

This proposal seeks to improve over existing customisation point mechanisms in a number of ways:

— Allows customisation-point names to be namespace-scoped rather than names being reserved globally.
— Allows generic forwarding of customisable functions through wrapper-types, including generic type-erasing wrappers (like `std::optional`) and adapters that customise some operations and pass through others.
— Concise syntax for defining customisable function-objects and adding customisations for particular types.

— Support for copy-elision of arguments passed by-value to customisation points.
— Far better compile times compared to the `tag_invoke` mechanism (avoids 3 layers of template instantiations and cuts down on SFINAE required to separate implementation functions)
— Far better error messages compared to the `tag_invoke` mechanism (`tag_invoke` does not distinguish between different customization point functions and results in sometimes hundreds of overloads)

This proposal improves on the semantics of the `tag_invoke` proposal [P1895R0], keeping namespace-scoped customisation points and generic customisation / forwarding, while providing a terser and cleaner syntax accessible to less experienced programmers.

The proposed syntax introduces use of:

— The `virtual` function-specifier for namespace-scope functions as a way of declaring that the function is a customisation-point (syntax idea borrowed from [P1292R0] "Customization Point Functions").
— The *virt-specifier* `override` for customisations of a customisable function-object.
— The *pure-specifier* `= 0` for declaring a customisable function without a default implementation
— The `default` keyword as an additional *virt-specifier* annotation for a customisable function's default implementation.
— A syntax for declaring customisations of a specific customisable function-object by using the fully-scoped name of the function-object as the function-name.
— The ability to deduce the customisable function object for the purposes of generic forwarding.

4 Terminology

This paper introduces the following terms:

— A function declaration for a *customisable function* is a **customisable function prototype**:

```
virtual void foo(auto&&) = 0;
```

— The set of *customisable function prototypes* naming the same entity represent a **customisable function**.

— A *customisable function* introduces a **Customizable Function Object (CFO)** in the scope it is declared.

— The set of functions or template functions with the `override` keyword are **customizations** of the corresponding customisable function. They form the **customization overload set**.

— The set of functions or template functions with the `default` keyword are **default implementations** of the corresponding customisable function. They form the **default overload set** for that customisable function object.

— A declaration of namespace-scope function of the form

```
template <auto func, typename T> auto func(T&& object) override;
```

declares a **generic customisation**.

5 Examples

5.1 Declaring a customisable function

The trailing `= 0` indicates this is a declaration without a default implementation.

```
namespace std::execution {
  template<sender S, receiver R>
  virtual operation_state auto connect(S s, R r) = 0;
}
```

5.2 Declaring a customisable function contains that has a default implementation

```
namespace std::ranges {
  template<input_range R, typename Value>
    requires equality_comparable_with<range_reference_t<R>, Value>
  virtual bool contains(R&& range, const Value& v) default {
    for (const auto& x : range) {
      if (x == v) return true;
    }
    return false;
  }
}
```

5.3 Defining a customisation as hidden friend

As an example we define a customization of the above customisable function contains (as a hidden friend) for `std::set` using the `override` keyword. Note that the name of the customised function is qualified.

```
namespace std {
  template<class Key, class Compare, class Allocator>
  class set {
  // ...
  private:
    template<typename V>
      requires requires(const set& s, const V& v) { s.contains(v); }
    friend bool ranges::contains(const set& s, const V& v) override {
      return s.contains(v);
    }
  };
}
```

5.4 Defining a customisation at namespace scope

Alternatively, we can define a customization at namespace scope using the `override` keyword. This can be useful when implicit conversions should be considered.

```
namespace std {
  template<class Key, class Hash, class KeyEq, class Allocator>
  class unordered_set { ... };

  // Defined outside class definition.
  template<class Key, class Hash, class Eq, class Allocator, class Value>
  requires(const unordered_set<Key,Hash,Eq, Allocator>& s, const Value& v) {
      s.contains(v);
  }
  bool ranges::contains(const unordered_set<K,H,Eq,A>& s,
                        const Value& v) override {
    return s.contains(v);
  }
}
```

6 References

[P1292R0] Matt Calabrese. 2018-10-08. Customization Point Functions.
https://wg21.link/p1292r0

[P1895R0] Lewis Baker, Eric Niebler, Kirk Shoop. 2019-10-08. tag_invoke: A general pattern for supporting customisable functions.
https://wg21.link/p1895r0

[P2279R0] Barry Revzin. 2021-01-15. We need a language mechanism for customization points.
https://wg21.link/p2279r0

[P2300R4] Michał Dominiak, Lewis Baker, Lee Howes, Kirk Shoop, Michael Garland, Eric Niebler, Bryce Adelstein Lelbach. 2022-01-19. std::execution.
https://wg21.link/p2300r4