P1665R0

Tag Based Customization Point Functions

Author: John Bandela

Audience: LEWGI

Project: ISO/IEC JTC1/SC22/WG21 14882: Programming Language — C++

Reply-To: John Bandela (jbandela@gmail.com), Bryce Adelstein-Lelbach (brycelelbach@gmail.com)

Abstract

This paper proposes a library change to improve how customization point functions are specified, overridden, and invoked. This is a library only proposal.

Example

```
namespace std {
namespace customization_points {
class swap;
template <typename T>
auto std__customization_point(swap*, T*) {
  return [](T& lhs, T& rhs) { /*...*/ };
}
}  // namespace customization_points
template <typename Tag>
struct __customization_point_caller {
  template <typename T, typename... Args>
  decltype(auto) operator()(T&& t, Args&&... args) const {
    return (std__customization_point(
        static_cast<Tag*>(nullptr),
        static_cast<add_pointer_t<remove_cvref_t<T>>>(nullptr))(
        std::forward<T>(t), std::forward<Args>(args)...));
  }
};

template <typename Tag>
inline constexpr __customization_point_caller<Tag> call_customization_point;
}  // namespace std

// User override
auto std__customization_point(std::customization_points::swap*, foo*) {
  return [](foo& a, foo& b) { /*...*/ };
}
```

```
// Calling swap
std::call_customization_point<std::customization_points::swap>(a, b);
```

Background

Using single function overloads is a common pattern throughout the C++ Standard Library and other C++ libraries. Examples include `begin`,`end`, `size`, and `swap`.

However, the problems with them are well known [1,2,3].

Some issues briefly are:

- Customization points are not differentiated from other functions
- While customization points are easy to override via overloading, invoking them can be subtle, involving the std two-step: `using std::swap; swap(a,b);`
- The dispatch part of a call and the call itself are done in a single call expression. For customization points with more than 1 parameter, this can lead to extra namespaces considered for ADL which can lead to surprising overloads being called [4].
- The names of customization points are de facto reserved across other namespaces. This can cause existing code to break. The introduction of `size` in C++17 broke at least one code base [5].

Tag Based Customization Point Functions

What the standard library provides for all customization points

The standard library provides variable template `customization_point_caller<Tag>` function object which passes a null pointer to `Tag`, along with a null pointer to the type of the first argument to `std__call_customization_point`. It then calls the returned function object forwarding all passed in parameters.

```
template <typename Tag>
struct __customization_point_caller {
  template <typename T, typename... Args>
  decltype(auto) operator()(T&& t, Args&&... args) const {
    return (std__customization_point(
        static_cast<Tag*>(nullptr),
        static_cast<add_pointer_t<remove_cvref_t<T>>>(nullptr))(
        std::forward<T>(t), std::forward<Args>(args)...));
  }
};
template <typename Tag>
inline constexpr __customization_point_caller<Tag> call_customization_point;
}
```

Specifying a customization point

To specify a customization point, the library declares a tag class in the desired namespace.

```
namespace std {
namespace customization_points {
class swap;
}
}  // namespace std
```

Overriding a customization point

To override a customization point, the user provides an overload of `std__customization_point` in the namespace of type for which the customization point is being overridden, which takes a pointer to `Tag` and a pointer to the type for which the the customization point is being specialized, and returns a function object which provides the required behavior.

```
auto std__customization_point(std::customization_points::swap*, foo*) {
  return [](foo& a, foo& b) { /*...*/ };
}
```

Calling a customization point

To call a customization point, the user calls function object `std::call_customization_point<Tag>` with any required parameters.

```
std::call_customization_point<std::customization_points::swap>(a,b);
```

Discussion

This addresses the issues that were brought up previously.

- Customization points are differentiated from other functions
- There is a canonical way to call customization points that is easy to teach and immediately recognizable as a call to customization point.
- The customization point dispatch is separated from the actual call. Because only the tag and the type is subject to ADL lookup and not additional parameters, this decreases the opportunity for surprising overloads due to ADL for additional argument types.
- Customization points can be added without the risk of name collisions in other namespaces.

In addition, this provides a framework for other libraries to use. Instead of coming up with their own customization point naming convention and hoping they don't clash with other names, libraries can just declare a tag in the appropriate namespace.

```
namespace awesome_library {
namespace customization_points {
class bar;
}
}  // namespace awesome_library
```

This can then be overridden:

```
auto std__customization_point(awesome_library::customization_points::bar*,
                              foo*) {
  return [](foo& f, parameter1& p1, parameter2& p2) { /**/ }
}
```

and called:

```
std::call_customization_point<awesome_library::customization_points::bar>(f,
p1,

p2);
```

As mentioned previously, this would also be a pure library change and not require any language changes.

Bikeshedding

- Shorten the names by abbreviating `customization_point` to `cp`
  - `std::call_cp<std::cp::swap>(a,b);`
- Naming conventions and namespaces of customization points
  - `std::begin_cp` vs `std::customization_points::begin` vs `std::cp::begin`
- Name of overloaded function
  - `std__customization_point` chosen with two underscores so that it is a reserved name already.
    - shorten to `std__cp`?
    - Change to some other name?

References

1. Matt Calabrese. "P1292R0 Customization Point Functions" http://wg21.link/p1292r0
2. Eric Niebler: "Customization Point Design in C++11 and Beyond" http://ericniebler.com/2014/10/21/customization-point-design-in-c11-and-beyond/
3. Arthur O'Dwyer. "C++ doesn't know how to do customization points that aren't operators" https://quuxplusone.github.io/blog/2018/08/23/customization-point-or-named-function-pick-one/
4. Michał Dominiak. Personal Communication
5. Arthur O'Dwyer. "Avoid ADL calls to functions with common names" https://quuxplusone.github.io/blog/2018/06/17/std-size/