Document number: P1772R1
Revises: P1772R0
Date: 2019-10-07
Project: ISO JTC1/SC22/WG21: Programming Language C++
Audience: EWGI, SG7
Reply to: Vincent Reverdy
University of Illinois at Urbana-Champaign
vince.rev@gmail.com

Variadic overload sets and overload sequences

Abstract

This document constitutes a short remark to P1170: Overload sets as function parameters and P0051: C++ generic overload function. From a library design standpoint we argue that the functionality provided by these papers should be considered together, and not independently. In particular, we feel that the current form of P1170 will not solve the problem of variadic overload sets built from non-inheritable classes that P0051 is facing. Making `std::overload_set` a variadic template class would allow to solve this problem.

Contents

1 History

- **P1772R0/D1772R0**
  - Reviewed by LEWGI in Cologne
  - Positive feedback for the general suggested direction
  - Poll: "We want a variadic `std::overload_set` library API (as described by P1772) instead of 2 orthogonal but composable library APIs (as in both `std::overload` and `std::overload_set`, with necessary fixes to make them composable)."

Strongly For	For	Neutral	Against	Strongly Against
4	6	4	0	1

  - Forwarded to EWGI

2 Background

2.1 Overload sets as function parameters

To summarize the problem that P1170 is proposing to solve, we start with the following example, that currently works without problem:

Invoke a single function

```cpp
double f(double x) {return x;}

auto result = std::invoke(f, 1.);
```

However, if new overloads are introduced, possibly elsewhere in the program, the call is not working anymore as in the following example:

Cannot invoke an overload set

```cpp
double f(double x) {return x;}
double f(double x, double y) {return x * y;}
double f(double x, double y, double z) {return x * y * z;}

// The following is not working today
auto result = std::invoke(f, 1.);
```

To make it work today, the function overload set needs to be wrapped in a lambda:

Workaround

```cpp
double f(double x) {return x;}
double f(double x, double y) {return x * y;}
double f(double x, double y, double z) {return x * y * z;}

auto result = std::invoke([](auto... args){return f(args...);}, 1.);
```

In fact, to achieve perfect forwarding, the lambda needs to be defined in the following much more complicated way:

The exact workaround

```cpp
[&](auto&& ...args)
    noexcept(noexcept(f(std::forward<decltype(args)>(args)...)))
        -> decltype(f(std::forward<decltype(args)>(args)...)) {
    return f(std::forward<decltype(args)>(args)...);
}
```

P1170 proposes to introduce a magic class to wrap overload sets:

Hypothetical overload set wrapper proposed by P1170

```cpp
template <typename T>
class overload_set
{
    // Data members
    private:
    T f;

    // Lifecycle
    public:
    overload_set(/* unspecified */);
    overload_set(overload_set const&) = default;
    overload_set(overload_set&&) = default;
    overload_set& operator=(overload_set const&) = default;
    overload_set& operator=(overload_set&&) = default;
    ~overload_set() = default;

    // Invocation
    public:
    template <typename... Us>
    invoke_result_t<F&, Us...> operator()(Us&&... us) &
        noexcept(is_nothrow_invocable_v<F&, Us...>);
    template <typename... Us>
    invoke_result_t<F const&, Us...> operator()(Us&&... us) const&
        noexcept(is_nothrow_invocable_v<F const&, Us...>);
    template <typename... Us>
    invoke_result_t<F, Us...> operator()(Us&&... us) &&
        noexcept(is_nothrow_invocable_v<F, Us...>);
    template <typename... Us>
    invoke_result_t<F const, Us...> operator()(Us&&... us) const&&
        noexcept(is_nothrow_invocable_v<F const, Us...>);
};
```

`std::overload_set` would work on overload sets, but also with any callable (lambdas, function objects, function pointers, member function pointers...). In that sense, it would constitute a sort of universal wrapper for callables.

2.2 Generic overload function

P0051 proposes to introduce a `std::overload` function to make several callable members of the same overload set:

Use case of generic overload function

```cpp
auto f = std::overload(
    [](int x){return x;},                                    // 1
    [](double x){return x;},                                 // 2
    [](int x, int y){return x * y;},                         // 3
    [](double x, double y){return x * y;},                   // 4
    [](int x, int y, int z){return x * y * z;},              // 5
    [](double x, double y, double z){return x * y * z;}      // 6
);

f(1., 2., 3.); // Will call the 6th version
```

It also proposes a `std::first_overload` that will pick the first function that can be called on the provided parameters as in:

Use case of generic first overload function

```cpp
auto f = std::first_overload(
    [](int x){return x;},                                   // 1
    [](double x){return x;},                                // 2
    [](int x, int y){return x * y;},                        // 3
    [](double x, double y){return x * y;},                  // 4
    [](int x, int y, int z){return x * y * z;},             // 5
    [](double x, double y, double z){return x * y * z;}     // 6
);

f(1., 2., 3.); // Will call the 5th version
```

In terms of implementation, the way to make it work, is by inheriting from the callables as in:

Hypothetical overload set wrapper to make P0051 work

```cpp
template <class... F>
class overload_set: F...
{
    public:
    using F::operator()...;

    /* ... */
};
```

This would only work however for inheritable classes. Functions, function pointers, member function pointers, and other non-inheritable callables can be wrapped in an inheritable class. However, there is currently no workaround for non-inheritable classes as in:

The problem of non-inheritable classes

```cpp
union union_int
{
    int operator()(int x){return x;}
};

union union_double
{
    double operator()(double x){return x;}
};

union_int ui;
union_double ud;

auto f = std::overload(ui, ud); // Will not work
```

Two possible solutions are either to make `std::overload` rely on some compiler magic, or to find hacky tricks based on future reflection facilities.

3 The problem

However the problem goes beyond non-inheritable classes. The current designs proposed by P1170 and P0051 seem to be incompatible with each other. The problem arises when combining the two as in the following example:

Problem when combining overload set and overload function

```cpp
// Overload set with floating-points
double f(double x) {return x;}
double f(double x, double y) {return x * y;}
double f(double x, double y, double z) {return x * y * z;}

// Overload set with integers
int g(int x) {return x;}
int g(int x, int y) {return x * y;}
int g(int x, int y, int z) {return x * y * z;}

// Combining std::overload_set and std::overload
auto function = std::overload(
    std::overload_set(f),
    std::overload_set(g)
);

function(1., 2., 3.); // Does not work: ambiguous!
```

It would be natural for users to expect the following:

Expectation

```cpp
struct function
{
    int operator()(int x) {return x;}
    double operator()(double x) {return x;}
    int operator()(int x, int y) {return x * y;}
    double operator()(double x, double y) {return x * y;}
    int operator()(int x, int y, i z) {return x * y * z;}
    double operator()(double x, double y, double z) {return x * y * z;}
};
```

but in reality it produces the following:

Reality

```cpp
struct function
{
    // Coming from f
    template <typename... Us>
    invoke_result_t<F&, Us...> operator()(Us&&... us) &
        noexcept(is_nothrow_invocable_v<F&, Us...>);
    template <typename... Us>
    invoke_result_t<F const&, Us...> operator()(Us&&... us) const&
        noexcept(is_nothrow_invocable_v<F const&, Us...>);
    template <typename... Us>
    invoke_result_t<F, Us...> operator()(Us&&... us) &&
        noexcept(is_nothrow_invocable_v<F, Us...>);
    template <typename... Us>
    invoke_result_t<F const, Us...> operator()(Us&&... us) const&&
        noexcept(is_nothrow_invocable_v<F const, Us...>);

    // Coming from g
    template <typename... Us>
    invoke_result_t<F&, Us...> operator()(Us&&... us) &
        noexcept(is_nothrow_invocable_v<F&, Us...>);
    template <typename... Us>
    invoke_result_t<F const&, Us...> operator()(Us&&... us) const&
        noexcept(is_nothrow_invocable_v<F const&, Us...>);
    template <typename... Us>
    invoke_result_t<F, Us...> operator()(Us&&... us) &&
        noexcept(is_nothrow_invocable_v<F, Us...>);
    template <typename... Us>
    invoke_result_t<F const, Us...> operator()(Us&&... us) const&&
        noexcept(is_nothrow_invocable_v<F const, Us...>);
};
```

which leads to the ambiguity at the call site.

4 Suggested resolution

Given these incompatibilities, we suggest that the handling of overload sets should be unified instead of separated in two proposals:

Suggested hypothetical resolution

```
template <class... F>
class overload_set: overload_set<F>...
{
    public:
    using overload_set<F>::operator()...;

    /* ... */

};

template <class F>
class overload_set<F>
{
    // Data members
    private:
    T f;

    // Lifecycle
    public:
    overload_set(/* unspecified */);
    overload_set(overload_set const&) = default;
    overload_set(overload_set&&) = default;
    overload_set& operator=(overload_set const&) = default;
    overload_set& operator=(overload_set&&) = default;
    ~overload_set() = default;

    // Invocation
    public:
    /* ... compiler magic to import the overload set as is ... */
    /* ... replacing the name of the functions by operator() ... */
};
```

so that the following works as expected:

Example of use of the resolution

```
// Overload set with floating-points
double f(double x) {return x;}
double f(double x, double y) {return x * y;}
double f(double x, double y, double z) {return x * y * z;}

// Overload set with integers
int g(int x) {return x;}
int g(int x, int y) {return x * y;}
int g(int x, int y, int z) {return x * y * z;}

// Variadic overload set
auto function = std::overload_set(f, g);
```

```cpp

// Calls "double f(double x, double y, double z) {return x * y * z;}"
function(1., 2., 3.);
```

Based on the same principle, a whole family of utilities can be created:

Overload set utilities

```cpp
// Picks the best overload according to overload resolution
template <class... F>
class overload_set;

// Picks the first overload that works
template <class... F>
class overload_sequence;

// Overload set with specific result (like is_invocable_r)
template <class R, class... F>
class overload_set_r;

// Overload sequence with specific result (like is_invocable_r)
template <class R, class... F>
class overload_sequence_r;
```

At this point, this proposal suggests a direction of investigation, but does not provide wording. The main goal was to raise the issue about the incompatibility of approaches in existing proposals P1170: Overload sets as function parameters and P0051: C++ generic overload function and start a discussion on how to properly solve the problem. The authors think that the suggested approach would work at the price of some compiler magic (that P1170 requires anyway) but other suggestions are very welcomed. What is certain, however, is that P1170 and P0051 propose partial solutions of a more general problem, and these solutions are currently not compatible with each other.

5 References

- P1170: Overload sets as function parameters, Barry Revzin and Andrew Sutton, *ISO / IEC – JTC1 / SC22 / WG21* (October 2018)
- P0051: C++ generic overload function, Vicente J. Botet Escriba, *ISO / IEC – JTC1 / SC22 / WG21* (November 2015)
- Custom Overload Sets and Inline SFINAE for Truly Generic Interfaces, Vincent Reverdy, *CppCon 2018* (September 2018)