Document number:	P0051R3=yy-nnnn
Date:	2018-02-12
Project:	ISO/IEC JTC1 SC22 WG21 Programming Language C++
Audience:	Library Evolution Working Group
Reply-to:	Vicente J. Botet Escribá <vicente.botet@nokia.com>

C++ generic overload function (Revision 3)

Experimental overload function for C++. This paper proposes one function that allow to overload lambdas or function objects, but also member and non-member functions.

There will be another proposal to take care of grouping lambdas or function objects, member and non-member functions so that the first viable match is selected when a call is done.

The overloaded functions are copied and there is no way to access to the stored functions. There will be another proposal to take care state full function objects and a mean to access them.

Table of Contents

History

Revision 3

- Signal a limitation on the design. We have some unexpected behavior when using final function

objects, `reference_wrapper` or any perfect forwarder wrapper if the overloads overlap.
- Remove the wording waiting for a decision on how to manage with this limitation.

Revision 2

The 2nd revision of P0051R1 fixes some typos and takes in account the feedback from Oulu meeting. Next follows the direction of the committee:

- Add `constexpr` and conditional `noexcept`.
- Confirmed the use universal references as parameters of the `overload` function.
- Ensure that forward cv-qualifiers and reference-qualifiers are forwarded correctly.
- Note that the use case for a final *Callable* is accepted.
- Check the wording with an expert from the LGW before sending a new revision to LWG (**Not done yet**).

Revision 1

The paper has been split into 3 separated proposals as a follow up of the Kona meeting feedback for P0051R0:

- `overload` selects the best overload using C++ overload resolution (this paper)
- `first_overload` selects the first overload using C++ overload resolution (to be written).
- Providing access to the stored function objects when they are state-full (to be written).

Introduction

Experimental overload function for C++. This paper proposes one function that allow to overload lambdas or function objects, but also member and non-member functions.

There will be another proposal to take care of grouping lambdas or function objects, member and non-member functions so that the first viable match is selected when a call is done.

The overloaded functions are copied and there is no what to access to the stored functions. There will be another proposal to take care state full function objects and a mean to access them.

Motivation

As lambdas functions, function objects, can't be overloaded in the usual implicit way, but they can be "explicitly overloaded" using the proposed `overload` function:

This function would be especially useful for creating visitors, e.g. for variant.

```
    auto visitor = overload(
       [](int i, int j )         {          ...        },
       [](int i, string const &j )        {          ...        },
       [](auto const &i, auto const &j )       {          ...        }
    );

    visitor( 1, std::string{"2"} ); // ok - calls (int,std::string) "overload"
```

The `overload` function when there are only two parameters could be defined as follows (this is valid only for lambdas and non-final function objects)

```
  template<class F1, class F2> struct overload : F1, F2
  {
    overloaded(F1 x1, F2 x2) : F1(x1), F2(x2) {}
    using F1::operator();
    using F2::operator();
  };
```

Why do we need an overload function?

Instead of the previous example

```
auto visitor = overload(
  [](int i, int j )                 { … },
  [](int i, string const &j )        { … },
  [](auto const &i, auto const &j )  { … }
);
```

the user can define a function object

```
struct
{
  auto operator()(int i, int j ) { … }
  auto operator()(int i, string const &j ) { … }
  template <class T1, class T2>
  auto operator()(T1 const &i, T2 const &j ) { … }
) visitor;
```

So, what are the advantages and liabilities of the overload function. First the advantages:

1. With overload the user can use existing functions that it can combine, using the function object would need to define an overload and forward to the existing function.

2. The user can group the overloaded functions as close as possible where they are used and don't need to define a class elsewhere. This is in line with the philosophy of lambda functions.
3. Each overload can have its own captured data, either using lambdas or other existing function objects.
4. Any additional feature of lambda functions, automatic friendship, access to this, and so forth.

Next the liabilities:

1. The overload function generates a function object that is a little bit more complex and so would take more time to compile.
2. The result type of overload function is unspecified and so storing it in an structure is more difficult (as it is the case for `std::bind`).
3. With the function object the user is able to share the same data for all the overloads. Note that that the last point could be seen as an advantage and a liability depending on the user needs.

Design rationale

Which kind of functions would `overload` accept

The previous definition of `overload` is quite simple, however it doesn't accept member functions nor non-member function, as `std::bind` does, but only function objects and lambda captures.

As there is no major problem implementing it and that their inclusion doesn't degrade the run-time performances, we opt to allow them also. The alternative would be to force the user to use `std::bind` or wrap them with a lambda.

Binary or variadic interface

We could either provide a binary or a variadic `overload` function.

```
auto visitor =
overload([](int i, int j )          {          ...        },
overload([](int i, string const &j )         {          ...        },
   [](auto const &i, auto const &j )        {          ...        }
));
```

The binary function needs to repeat the overload word for each new overloaded function.

We think that the variadic version is not much more complex to implement and makes user code simpler.

Passing parameters by value or by forward reference

The function `overload` must store the passed parameters. If the interface is by value, the user will be forced to move movable but non-copyable function objects. Using forward references has not this inconvenient, and the implementation can optimize when the function object is copyable.

This has the inconvenient that the move is implicit. We follow here the same design than `when_all` and `when_any`.

`reference_wrapper<F>` to deduce `F&`

As with other functions that need to copy the parameters (as `std::bind`, `std::thread`, ...), the user can use `std::ref` to pass by reference.

The user could prefer to pass by reference if the function object is state-full or if the function object is expensive to move (copy if not movable) or even s/he would need it if the function object is not movable at all.

Final function objects

The basic design use inheritance from the function object. However when the function object is a final class, we cannot inherit from it. Nevertheless this final function object can be wrapped and the call be forwarded to the wrapped object. Note that the wrapper will need to provide all combinations of cv-qualifiers.

The same applies to classes with final virtual destructors.

Selecting the best or the first overload

Call the functions based on C++ overload resolution, which tries to find the best match, is a good generalization of overloading to lambdas and function objects.

However, when trying to do overloading involving something more generic, it can lead to ambiguities. So the need for a function that will pick the first function that is callable. This allows ordering the functions based on which one is more specific.

As both cases are useful, and even if this paper proposes only overload, there will be a separated proposal for `first_overload`.

- `overload` selects the best overload using C++ overload resolution and
- `first_overload` selects the first overload using C++ overload resolution.

Fit library name them `match` and `conditional` respectively. FTL uses `match` to mean `first_overload`. Boost.Hana names them `overload` and `overload_linearly` respectively.

Result type of resulting function objects

The proposed `overload` functions doesn't add any constraint on the result type of the overloaded functions. The result type when calling the resulting function object would be the one of the selected overloaded function.

However the user can force the result type and in this case the result type of all the overloads must be convertible to this type (contribution from Matt Calabrese).

This can be useful in order to improve the compiling time of a possible `match` / `visit` function that could take advantage when it knows the result type of all the overloads.

Result type of `overload`

The result type of this function is unspecified as it is the result type of `std::bind` or `std::mem_fn`.

However when the function objects have a state it will be useful that the user can inspect the state. The result type should provide an overload for `std::get<F>` / `std::get<I>` functions (contribution from Matt Calabrese).

These functions should take in account that the overload can be a `reference_wrapper<F>` in order to allow `get<F&>(ovl)`.

This paper doesn't include such access functions. Another paper will take care of this concern if there is interest.

`constexpr` and `noexcept`

There is no reason the result of the function object couldn't be `constexpr` if the parameters are literals.

In addition these functions shall be `noexcept` when the parameters are no throw move constructible.

forward `constexpr`

The overloaded functions should preserve `constexpr`. However, CWG-1581 prevents the use of `constexpr` functions in non-evaluated contexts.

There is some specific behavior for `std::overload`. The overloaded functions are in most of the cases not declared, they are introduced via a using declaration and so no `constexpr` is needed in these cases.

There are some cases where we need to declare a forwarding functions, e.g. for pointer to functions.

Declaring this case as `constexpr` will prevent to use a call to the result of `std::overload` (the overloaded set) in non-evaluated contexts CWG-1581 when there are pointer functions that are non `constexpr`.

Not declaring it `constexpr` will prevent to call the result of `std::overload` (the overloaded set) in a `constexpr` when there are pointer to functions even if the wrapped function is `constexpr`.

We believe that adding `constexpr` is the best approach even if it has some liabilities. These liabilities will be fixed when CWG-1581 will be resolved. This is in line with P0356R3.

forward `noexcept`

The overloaded functions can be conditionally `noexcept` depending on wether the stored functions are `noexcept`.

Not adding the conditional `noexcept` could make the call less efficient and suggest to the user to write directly a function object by hand. This is why this proposal request to preserve the `nonexcept` of the stored functions.

forward cv and ref qualifiers

The overloaded functions shall preserve cv and ref qualifiers.

Issue with perfect forwarding and implicit conversions

In the case of a final class we need to wrap the final class and perfect forward the call to the final class. However, this perfect forwarding implementation interacts with implicit conversions as shown in the following example

```
struct X;
struct Y {
//  Y(const X&){} // enable this to fail
};
struct X {
//  operator Y() const { return Y(); }  // or enable this to fail
};

struct Foo final { void operator()(X) const { printf("X\n"); } };
struct Bar final { void operator()(Y) const { printf("Y\n"); } };

int main()
{
    auto ol = std::experimental::overload(Foo(), Bar());
    ol(X());
    return 0;
}
```

The perfect forwarding overload cannot select the best matching when there are types that implicitly convert one to each other.

In order to fix the issue we will surely need some help from the compiler. Maybe `std::experimental::invocation_type` could help, but we don't have yet a compiler implementing those traits.

This issue happens also with `reference_wrapper` and any object function that has forwarding references. Consider the following example:

```
struct A { operator()(std::string const&) };
struct B { operator()(std::string_view) };
A a; B b;
std::string s;

auto f = std::overload(a, b);
f(s);  //-> calls a(s); b requires user defined conversion
```

Now suppose that B becomes so big that copying it is an issue and we decide to pass it by reference.

```
auto f = std::overload(a, std::ref(b));
f2(s); //-> calls b(s)
```

This function is used to combine multiple functors into one entity. Using the proposed best overload approach, makes the code that combines such functors very fragile to code change, as the subtle changes in the interface of the functor that is used (like `std::string const&` to `std::string_view`) may

lead to silent change in the result of invocation of the functor produced from `std::overload`.

The use of standard library features shouldn't be error prone while doing such kind of code refactoring, due above I would suggest direction, when the domain of combined functors should not overlap - this should still cover most of use cases of `std::overload`, i.e. visitation, as it make little sense to create an variant of object that can be convertible to each other (like `std::string`, `std::string_view` and `const char*`).

Open points

Do we want an `unique_overload` function with no surprises?

The surprises come when more than one overload matches and the wrapped perfect forwarder is preferred. We can avoid these surprises, by requesting that only one single overload could be possible. This can be implemented by wrapping with a perfect forwarder even for function objects that can be used as a base class. In this case all the overloads will use perfect forwarding and so no one would have preference.

This has the drawback, that basic examples wouldn't work, as e.g.

```
    std::experimental::unique_overload(
        [](int) {
            //...
        }
        ,
        [](float) {
            //...
        }
    )(1.0f);
```

No workaround other that using an additional `first_overload`.

Do we want a `simple_overload` function?

As using perfect forwarding wrappers are subject to surprises, the guideline would be to don't use them when there are several possible matches. Maybe we could simplify the overload function `simple_overload` and don't wrap implicitly. This means that the overload function will work only with (derivable) function objects and lambdas.

We can protect ourselves from the perfect forwarder wrappers by forbidding the standard ones as `reference_wrapper`, `not_fn`, `bind_front`, ... but we cannot forbid the user defined perfect

forwarders wrappers. So we could just have a guideline for the user to tell them to don't use perfect forwarder wrappers with `std::simple_overload`. This function at least will not introduce implicitly new ones.

In order to make work the following example using `std::ref`

```
	function_with_state foo;
	function_with_state_2 foo2;

	// Ambiguous. reference_wrapper provide perfect forwarding calls :(
	std::experimental::overload(std::ref(foo), std::ref(foo2))(1.0f);
```

we can use the workaround using lambdas

```
	function_with_state foo;
	function_with_state_2 foo2;

	std::experimental::overload(
			[&foo](int i) { return foo(i); },
			[&foo2](float f) { return foo2(f); }
			)(1.0f);
```

The following will call the `nonMemberInt` function even if the lambda float function seems a better match

```
int nonMember( int );
{
	std::experimental::simple_overload(
			nonMemberInt,
			[](float f) { return foo2(f); }
			)(1.0f);

}
```

The fix consists again in using a lambda that forwards the call to `nonMemberInt`, but being explicit enough, no perfect forwarding should be used.

```
int nonMember( int );
{
      std::experimental::simple_overload(
                        [](int i) { return nonMemberInt(i); },
                        [](float f) { return foo2(f); }
                        )(1.0f);

}
```

Do we want a `first_overload` function that selects the first match?

We have some difficulties with the best match. A function that selects the first match will avoid any confusion.

Do we want a `overload` function that selects the best possible match?

While the proposed `std::overload` function has some issues, it is quite practical in a lot of cases.

For the author,

- `unique_overload` while avoiding surprises is quite restrictive when the overloads overlap.
- `simple_overload` makes the work for the user more complex in most of the cases, but don't prevent the user of using user defined perfect forwarding wrappers and introduce surprises.
- `overload` makes the work for the user more simpler in most of the cases, but don't prevent the user of using the perfect forwarding wrappers either explicitly or implicitly introducing possible surprises.
- `first_overload` should avoid any surprising issues, as it imposes an order.

If we decide to get rid of surprised and select `unique_overload`, `first_overload` would be necessary. At the end this look like a good combination.

Proposed wording

No wording provided until the open point is closed.

Implementation

There is an implementation of the best match version of overload at https://github.com/viboes/std-make/blob/master/include/experimental/fundamental/v3/functional/overload.hpp.

Acknowledgements

Thanks to Daniel Krügler who helped me to improve the wording and that pointe out to me the use case for a final *Callable*.

Thanks to Scott Pager who suggested to add overloads for non-member and member functions.

Thanks to Paul Fultz II and Bjørn Ali authors of Fit and FTL from where the idea of the `first_overload` function comes from.

Thanks to Matt Calabrese for its useful improvement suggestions on the library usability.

Thanks to Tony Van Eerd for championing the original proposal at Kona and for insightful comments.

Thanks to Stephan TL for pointing CWG-1581 "When are constexpr member functions defined?".

Thanks to Peter Remmers that reported the issue Issue16.

Thanks to Tomasz Kaminski helping me to refine the implementation for final function object and to the private discussion about the possibility to have the combination of `unique_overload` and `first_overload` as a much safer solution.

Special thanks and recognition goes to Technical Center of Nokia - Lannion for supporting in part the production of this proposal.

References

- Boost.Hana - Louis Dionne

  http://boostorg.github.io/hana/

- Fit - Paul Fultz II

  https://github.com/pfultz2/Fit

- FTL - Bjorn Ali

  https://github.com/beark/ftl

- N4564 N4564 - Working Draft, C++ Extensions for Library Fundamentals, Version 2 PDTS

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4564.pdf

- P0051R0 C++ generic overload function

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0051r0.pdf

- P0051R1 C++ generic overload function (Revision 1)

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0051r1.pdf

- P0051R2 C++ generic overload function (Revision 2)

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0051r2.pdf

- P0356R3 Simplified partial function application

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0356r3.html

- CWG-1581. When are constexpr member functions defined?

  http://www.open-std.org/jtc1/sc22/wg21/docs/cwg_active.html#1581

- Issue16 overload: ambiguous for compatible types

  https://github.com/viboes/std-make/issues/16