Document number:	P0051R2=yy-nnnn
Date:	2016-10-13
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 2)

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 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 function objects)

```
 template<class F1, class F2> struct overloaded : F1, F2
 {
   overloaded(F1 x1, F2 x2) : F1(x1), F2(x2) {}
   using F1::operator();
   using F2::operator();
 };
 template<class F1, class F2>
 overloaded<F1, F2> overload(F1 f1, F2 f2)
 { return overloaded<F1, F2>(f1, f2); }
```

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 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.

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 this function 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 is at least one case where we need to declare a forwarding functions, for pointer to non member 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 to non member functions that is 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 non member functions even if the wrapped function is `constexpr`.

For pointer to member functions, the current implementation uses `std::mem_fn` so it inherits the `std::mem_fn` behavior. Nevertheless, a flat implementation declaring the forwarding functions could choose whether the overloaded functions are `constexpr` or not.

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.

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.

For pointer to member functions, the current implementation uses `std::mem_fn` so it inherits the `std::mem_fn` behavior. Nevertheless, a flat implementation declaring the forwarding functions could make the overloaded functions conditionally `noexcept`.

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.

Proposed wording

The proposed changes are expressed as edits to N4564 the Working Draft - C++ Extensions for Library Fundamentals V2.

Header Synopsis

Add the following declaration in experimental/functional.

```
namespace std
{
namespace experimental
{
inline namespace fundamental_v3
{
    template <class R, class ... Fs>
    constexpr 'unspecified' overload(Fs &&... fcts) noexcept('see below');
    template <class ... Fs>
    constexpr 'unspecified' overload(Fs &&... fcts) noexcept('see below');
}
}
}
```

Function Template `overload`

```
    template <class R, class ... Fs>
    constexpr 'see below' overload(Fs &&... fcts) noexcept('see below');
```

"The expression inside `noxcept` is equivalent to: `is_nothrow_move_constructible<Fs>::value && ...`

Requires: Each `Fs_i` in `Fs` shall satisfy the requirements of and *MoveConstructible* (Table 20 [moveconstructible]) and of a callable type ([func.def]). [Note: The result type of each one of the possible calls to the overloaded functions shall be convertible to `R`, otherwise the program is ill formed -end Note]

Result type: A function object that behaves as if all the parameters were overloaded when calling it. The result type will contain the nested `result_type` type alias `R`. The overloads shall preserve `constexpr`, `noexcept`, cv-qualifiers and reference qualifiers.

The effect of calling to an instance of this type with parameters `ti` will select the best overload. If there is not such a best overload, either because there is no candidate or that there are ambiguous candidates, the invocation expression will be ill-formed.

If there is a best overload, lets say that is `f`, `INVOKE(DECAY_COPY(f), forward<T1>(t1) ..., forward<T1>(tN), R)`, where `t1`, `t2`, ..., `tN` are values of the corresponding types in `Ts...`, shall be a valid expression. Invoking a decay copy of `f` shall behave the same as invoking `f`.

Returns: An instance of the result type, that contains a decay copy of each one of the arguments `fcts`.

Thows: Any exception thrown during the construction of the resulting function object.

Remarks: This function as well as the overloaded `operator()` for each `fcts` on the resulting type shall be a `constexpr` functions. The overloaded `operator()` for each `f` in `fcts...` on the resulting type shall be `noexcept(noexcept(INVOKE(DECAY_COPY(f), forward<T1>(t1) ..., forward<T1>(tN))))`.

The return type is a callable type that meets the *MoveConstructible* requirements. Let `FV_i` be `remove_reference_t< Fs_i>`. If all `FV_i` in `Fs` satisfy the *CopyConstructible* requirements, then the return type shall meet the *CopyConstructible* requirements.

```
    template <class ... Fs>
    constexpr 'see below' overload(Fs &&... fcts) noexcept('see below');
```

"The expression inside `noxcept` is equivalent to: `is_nothrow_move_constructible<Fs>::value && ...`

Requires: Each `Fs_i` in `Fs` shall satisfy the requirements of and *MoveConstructible* ([moveconstructible]) and of a callable type ([func.wrap.func]).

Result type: A function object that behaves as if all the parameters were overloaded when calling it. The overloads shall preserve `constexpr`, `noexcept`, cv-qualifiers and reference qualifiers.

The effect of calling to an instance of this type with parameters `ti` will select the best overload. If there is not such a best overload, either because there is no candidate or that there are ambiguous candidates, the invocation expression will be ill-formed.

If there is a best overload, lets say that is `f`, `INVOKE(DECAY_COPY(f), forward<T1>(t1) ..., forward<T1>(tN))`, where `t1`, `t2`, ..., `tN` are values of the corresponding types in `Ts...`, shall be a valid expression. Invoking a decay copy of `f` shall behave the same as invoking `f`.

Returns: An instance of the result type, that contains a decay copy of each one of the arguments `fcts`.

Thows: Any exception thrown during the construction of the resulting function object.

Remarks: This function as well as the overloaded `operator()` for each `fcts` on the resulting type shall be a `constexpr` functions. The overloaded `operator()` `f` in `fcts...` on the resulting type shall be `noexcept(noexcept(INVOKE(DECAY_COPY(f), forward<T1>(t1) ..., forward<T1>(tN))))`.

The return type is a callable type that meets the *MoveConstructible* requirements. Let `FV_i` be `remove_reference_t< Fs_i>`. If all `FV_i` in `Fs` satisfy the *CopyConstructible* requirements, then the return type shall meet the *CopyConstructible* requirements.

Implementation

There is an implementation of the explicit return type version 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 Tomasz Kaminski helping me to refine the implementation for final function object .

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
- CWG-1581. When are constexpr member functions defined? http://www.open-std.org/jtc1/sc22/wg21/docs/cwg_active.html#1581