Document number: P0051R1
Date: 2015-11-05
Project: ISO/IEC JTC1 SC22 WG21
Programming Language C++
Audience: Library Evolution Working Group
Reply-to: Vicente J. Botet Escriba <vicente.botet@wanadoo.fr>

C++ generic `overload` function (Revision 1)

Experimental overload function for C++17. 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.

Contents

History

Revision 1

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

- `overload` selects the best overload using C++ overload resolution (this paper)
- `first_overload` selects the first overload using C++ overload resolution [D0050B].
- Providing access to the stored function objects when they are state-full [D0050C].

Introduction

Experimental overload function for C++17. 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 and Scope

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 follows 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.

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`, you could find `first_overload` in [D0050B]

- `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 means `first_overload`.

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 compile time of a possible match/visit function that could take advantage when the 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`.

However when the function objects have an 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 [D0050C] will take care of this concern.

Open points

The authors would like to have an answer to the following points if there is at all an interest in this proposal:

- **Should the callable be passed by value, forcing the use of std::move?**

Technical Specification

Header <experimental/functional> Synopsis

Add the following declaration in experimental/functional.

```
namespace std {
namespace experimental {
inline namespace fundamental_v2 {
    template <class ... Fs>
    'see below' overload(Fs &&... fcts);
    template <class R, class ... Fs>
    'see below' overload(Fs &&... fcts);
}
}
}
```

Function Template `overload`

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

Requires: Fs are Callable and Movable and the result type of each parameter must be convertible to `R`.

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 effect of call 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`.

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

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

Requires: Fs are Callable and Movable and the result type of each parameter must be convertible to `R`.

Result type: A function object that behaves as if all the parameters were overloaded when calling it.

The effect of call 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`.

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

Implementation

There is an implementation of `overload` at https://github.com/viboes/tags.

Acknowledgements

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

Thanks to Paul Fultz II and Bjorn 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.

References

- [Boost.Hana] - Louis Dionne
  http://boostorg.github.io/hana/
- [Fit] - Paul Fultz II
  https://github.com/pfultz2/Fit
- [FTL]
  https://github.com/beark/ftl
- [P0051R0] P0051 - C++ generic overload function
  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0051r0.pdf