Document number: D0051=15-nnnn
Date: 2015-09-15
Project: ISO/IEC JTC1 SC22 WG21
                    Programming Language C++,
                    Library Evolution Working Group
Reply-to: Vicente J. Botet Escriba <vicente.botet@wanadoo.fr>

C++ generic overload functions

Experimental overload functions for C++17. This paper proposes two functions that allow to overload lambdas or function objects, but also member and non-member functions.

Contents

Introduction

This paper proposes two functions that allow to overload lambdas or function objects, but also member and non-member functions:

- `overload` selects the best overload using C++ overload resolution and

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

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); }
```

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 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`.

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, the proposal includes two functions: `overload` and `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 means `first_overload`.

Result type of resulting function objects

The proposed `overload` and `first_overload` functions don'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.

Result type of **overload/first_overload**

The result type of these functions is unspecified as it is the result type of `std::bind`.

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?**
- **A better name for the proposed functions?**
- **Do we want to expose the result type of these functions?**

Technical Specification

Note: The following wording need of course much more details to describe the behavior of the unspecified result type.

Header <experimental/functional> Synopsis

Add the following declaration in experimental/functional.

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

Function Template `overload`

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

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

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

Function Template fisrt_overload

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

Returns: A function object that will forward its parameters to the first function that is callable with these parameters.

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 Fit and FTL authors from where the idea of the `first_overload` function comes from.

References

- Fit : https://github.com/pfultz2/Fit

- FTL: https://github.com/beark/ftl