Document number: D0051R1
Date: 27/10/31
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 first_overload function

Experimental `first_overload` function for C++17. This paper proposes one function that groups lambdas, function objects, member and non-member functions into a single callable object providing an overloaded conditional call operator. A call to this object will result in a call to the first function (in a generic way) that produce a valid call expression.

Contents

History

Revision 0

Extraction from P0032R0. As for the Kona meeting request, the `first_overload` has been, extracted from [P0032R0] as the motivation was less clear and some find even found more subject to errors.

Introduction

Experimental `first_overload` function for C++17. This paper proposes one function that groups lambdas, function objects, member and non-member functions into a single callable object providing an overloaded call operator that selects the first viable match.

P0051R1 proposes an overload function that is almost similar, selecting the best overload instead of the first one.

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

Motivation and Scope

In addition to the motivation for the overload function [P0051R1], the first_overload need appear as a consequence of overload ambiguity. When we want to overload functions based on type requirements, it can lead to ambiguities using the default overloading mechanism in C++.

There are some techniques that have been developed alo,g the time to resolve this overload ambiguity.

1. Disjoint requirements. This consists in negating the other requirements. Simple to implement but hard to maintain. Ensuring that all the overloads are disjoint could be a nightmare.
2. Tag dispatching [TagDispatching]. Tag dispatching works by creating tags that uses inheritance to order the functions. These tags follows a conceptual model, but in order to work correctly the conceptual model must be quite close to the function been overloaded. So as far as the algorithm/function that is been overloaded follows the conceptual tag every is correct.
3. Ranks [Rank]: This technique consists in adding a specific choice<N> tag to each overload. The choice<N> been more specific than the choice<N-1> allows to order the overloads, ensuring that there is no ambiguity as far as the overloads of the same rank have no ambiguities. While this tags can be seen as artificial, they have the advantage to been explicit and independent of the any conceptual model.
4. Linear [Conditional]: Grouping all the overloads in a single function object and select the first overload on the list that make the call well formed. This is a variation of the previous one and the overload grouping technique described in [P0051R1], but the user should instead of adding a rang, just puts the overloads in a given order. As it uses a function object to store the overloads it has also all the associated advantages and liabilities.

The first viable match has several advantages over the other approaches. First, additional overloads can be added easily. Secondly, it doesn't produce long and confusing compile errors. Thirdly, it doesn't require boilerplate every time.

The disadvantage of the first viable match over tag dispatching is the functions have to be ordered by their relationships for every function rather than being handle by the tag hierarchy once.

Up to the user to use whatever technique is more appropriated to their context.

How it works

The following code snippet shows the implementation restricted to 2 stateless function object overloads using C++ concepts. In C++14 we will use instead enable_if SFINAE.

```
template<class F1, class F2>
  struct first_overload_fn
{
  template<class... Ts>
    requires
          is_callable(F1(Ts&&...))
  auto operator()(Ts&&... xs)
  {
      return invoke(F1(), std::forward<Ts>(xs)...);
  }
  template<class... Ts>
    requires
          ! is_callable<F1(Ts&&...)>
          && is_callable<F2(Ts&&...)>
  auto operator()(Ts&&... xs)
  {
      return invoke(F2(), std::forward<Ts>(xs)...);
  }
};

template<class F1, class F2>
  first_overload_fn<decay_t<F1>, decay_t<F2>>
  first_overload(F1&& f1, F2&& f2)
{
  return first_overload_fn<F1, F2>(
      forward<F1>(f1), forward<F2>(f2));
}
```

Language type-switch pattern matching

The pattern matching presentation [PM] include a inspect/when/default statement that pattern matching linearly. The nth expression match is tried only when the previous one has failed. The cases are not disjoint, case order maters.

The library [Match7] follows the same strategy.

Sum type visitation

This `first_overload` function will be very useful while visit sum types [P0050] so that the visitation is done following the fist match strategy.

Design rationale

Selecting the first viable overload

See [P0051R1] if you are looking for the best viable overload.

Fit library name this function `conditional`. FTL uses `first_overload`, Boost.Hana uses `oveload_linearly`.

Which kind of functions would first_overload accept

This function follows the same rationale than the proposed `overload` function [P0051R1].

Passing parameters by value or by forward reference

This function follows the same rationale than the proposed `overload` function [P0051R1].

`reference_wrapper<F> to deduce F&`

This function follows the same rationale than the proposed `overload` function [P0051R1].

Result type of resulting function objects

This function follows the same rationale than the proposed `overload` function [P0051R1].

Result type of `first_overload`

This function follows the same rationale than the proposed `overload` function [P0051R1].

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 function?**

  first_overload, conditional, first_viable, overload_linearly.

Technical Specification

The wording is relative to [N4480].

Function objects

Header <experimental/functional> Synopsis

Add the following declaration in experimental/functional.

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

Function Template `first_overload`

```
template <class R, class ... Fs>
'see below' first_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 call to an instance of this type will select the first overload matching the invocation. If there is not such a overload the invocation expression will be ill-formed.

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

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

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

Requires: Fs are Callable and Movable.

Result type: A function object that behaves as if all the parameters were overloaded when calling it. The call to an instance of this type will select the first overload matching the invocation. If there is not such a overload the invocation expression will be ill-formed.

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

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

Implementation

Boost.Hana, Fit, FTL has each one its own implementation with each own specificities. The names are `overload_linearly`, `conditional` and `first_overload` respectively.

Acknowledgements

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

Thanks to Paul Fultz II and R. Martinho Fernandes for their excellent blogs.

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.

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
- [P0050] P0050 – C++ generic match function
  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0050r0.pdf
- [P0051R0] P0051 - C++ generic overload function
  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0051r0.pdf
- [P0051R1] P0051R1 - C++ generic overload function
  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0051r1.pdf
- [TagDispatching] Generic Programming Techniques - David Abrahams
  http://www.boost.org/community/generic_programming.html#tag_dispatching
- [Rank] Beating overload resolution into submission - R. Martinho Fernandes
  https://rmf.io/cxx11/overload-ranking/
- [Conditional] Overload of Type Requirements – Paul Fultz II
  http://pfultz2.com/blog/2014/08/22/overloading-requirements/
- [PM] Pattern Matching for C++ - Yuriy Solodkyy, Gabriel Dos Reis, Bjarne Stroustrup
- [Match7] Open Pattern Matching for C++ - Yuriy Solodkyy, Gabriel Dos Reis, Bjarne Stroustrup
  http://www.stroustrup.com/OpenPatternMatching.pdf