Document number: D0050
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 match function

Vicente J. Botet Escriba

Experimental match function for C++17.

Contents

Introduction.... 1
Motivation and Scope.... 2
Tutorial.... 2
- Customizing the match function.... 2
- Using the match function to inspect one sum type.... 2
- Using the match function to visit several sum types.... 4

Design rationale.... 5
- Result type of match 5
- Multiple cases or overload.... 6
- Order of parameters and function name.... 6
- Variadics.... 6
- const aware matchers.... 7
- Non-const aware matchers.... 7
- Customization point.... 8
- Grouping with overload 8
- ss.... 8

Open Points.... 9
Technical Specification.... 10
- Header <experimental/meta> Synopsis.... 10
- Header <experimental/functional> Synopsis.... 10
- Header <experimental/optional> Synopsis.... 12
- Header <experimental/variant> Synopsis.... 12

Implementation.... 13
Acknowledgements.... 13
References.... 13

Introduction

This paper presents a proposal for generic `match` functions that allow to visit sum types individually or by groups (product of sum types).

Motivation and Scope

Getting the value stored in sum types as `variant<Ts...>` or `optional<T>` needs to know which type is stored in. Using visitation is a common technique that makes the access safer.

While the last variant proposal [N4542] includes visitation; it takes into account only visitation of homogeneous variant types. The accepted optional class doesn't provides visitation, but it can be added easily. Other classes, as the proposed expected class, could also have a visitation functionality. The question is if we want to be able to visit at once several sum types, as `variant`, `optional`, `expected`, and why not smart pointers.

`std::experimental::apply()` [N3915] can be seen as a particular case of visitation of multiple types if we consider that any type can be seen as a sum type with a single type.

Instead of a `visit` function, this proposal uses instead the `match` function that is used to inspect some sum types.

Tutorial

Customizing the `match` function

The proposed `match` function works for sum types `ST` that have customized the following overloaded function

```
template <class R, class F>
R match(ST const&, F&& );
```

For example, we could customize `boost::variant` as follows:

```
namespace boost {
  template <class R, class F, class ...Ts >
  R match(variant<Ts...> const& v, F&& f)
  { return apply_visitor(std::forward<F>(f), v); }
}
```

In addition we need to know the sum type alternatives if we want to use `match` with several sum types

```
temaplate <class ...Ts >
  struct sum_type_alternatives<variant<Ts...>>
{
  using type = types<Ts...>;
}
```

Using the `match` function to inspect one sum type

Given the `boost::variant` sum type, we could just visit it using the proposed `overload` function (See [D0051]).

```
boost::variant<int, X> a = 2;
boost::apply_visitor(overload(
  [](int i)
```

```
    {},
  [](X const& i)
    {
      assert(false);
    },
  [](...)
    {
      assert(false);
    }
  ), v);
```

The same applies to the proposed `std::experimental::variant`

```
std::experimental::variant<int, X> a = 2;
std::experimental::visit(overload(
  [](int i)
    {},
  [](X const& i)
    {
      assert(false);
    },
  [](...)
    {
      assert(false);
    }
  ), a);
```

We can use in both cases the variadic `match` function

```
boost::variant<int, X> a = 2;
std::experimental::match(a,
  [](int i)
    {},
  [](X const& i)
    {
      assert(false);
    },
  [](...)
    {
      assert(false);
    }
  );

std::experimental::::variant<int, X> a = 2;
std::experimental::match(a,
  [](int i)
    {},
  [](X const& i)
    {
      assert(false);
    },
  [](...)
    {
      assert(false);
    }
  );
```

We can also use a single matcher

```
boost::variant<int, X> a = 2;
std::experimental::match(a,
  std::experimental::overload([](int i)
    {},
  [](X const& i)
    {
      assert(false);
    },
  [](...)
    {
      assert(false);
    }
  ));
```

Using the `match` function to visit several sum types

The variant proposal provides visitation of multiple variants.

```
std::experimental::variant<int, X> a = 2;
std::experimental::variant<int> b = 2;
std::experimental::visit(overload(
  [](int const &i, int const &j )
    {
    },
  [](auto const &i, auto const &j )
    {
      assert(false);
    },
  [](...)
    {
      assert(false);
    }
  ), std::make_tuple(a, b));
```

The `match` function generalizes the visitation for several instances of heterogeneous sum types, e.g. we could visit `variant` and `optional` at once:

```
std::experimental::variant<int, X> a = 2;
std::experimental::optional<int> b = 2;
std::experimental::match(std::make_tuple(a, b),
  [](int const &i, int const &j )
    {
    },
  [](auto const &i, auto const &j )
    {
      assert(false);
    },
  [](...)
    {
      assert(false);
    }
  );
```

Alternatively we could use a `inspect` factory that would wrap the tuple and provide two functions `match` and `first_match`

```
std::experimental::inspect(a, b).first_match( // or match
```

```
[](int const &i, int const &j )
  {
  },
[](auto const &i, auto const &j )
  {
    assert(false);
  },
[](...)
  {
    assert(false);
  }
);
```

but this first draft doesn't include it.

Design rationale

Result type of `match`

We can consider several alternatives:

- same type: the result type is the type returned by all the overloads (must be the same),
- `common_type`: the result type is the `common_type` of the result of the overloads,
- explicit return type `R`: the result type is `R` and the result type of the overloads must be explicitly convertible to `R`,

Each one of these alternatives would need a specific interface:

- `same_type_match`
- `common_type_match`
- `explicit_type_match`

For a sake of simplicity (and because our implementation needed the result type) this proposal only contains a `match` version that uses the explicit return type. The others can be built on top of this version.

Let `Ri` be the return type of the overloaded functor for the alternative i of the ST.

- `same_type_match`: Check that all `Ri` are the same, let call it `R`, and only then call to the explicit `match<R>(...)`,
- `common_type_match`: Let be `R` the `common_type<Ri...>` and only if it exists call to the explicit `match<R>(...)`.

Multiple cases or overload

The matching functions accept several functions. In order to select the function to be applied we have two alternatives:

- use `overload` to resolve the function to be called

- use `first_overload` to do a sequential search for the first function that match.

The `match` function uses `overload` and the `first_match` uses `first_overload`.

An alternative is to restrict the interface to single `inspect` function and let the user use either `match` or `first_match`.

Order of parameters and function name

The proposed `visit` function has as first parameter the visitor and then the visited variant.

```
visit(visitor, visited);
```

The proposed `match` function reverse the order of the parameters. The reason is that we consider that it is better to have the object before the subject.

```
match(sumType, matcher);
```

Of course we can also reverse the roles of the object and subject and tell that the object is the visitor.

`std::experimental::apply` function follows the same pattern

```
apply(fct, tpl);
```

If uniform function syntax is adopted we would have

```
visitor.visit(visited);

sumType.match(matcher);

visitor.accept(visited);

fct.apply(tpl);
```

Variadics

The two parameters of the match function can be variadic. We can have several sum types and several functions overloading a possible match.

If we had a language type pattern matching feature (see [PM]) the authors guess it would be something like:

```
boost::variant<int, X> a = 2;
boost::optional<int> b = 2;
match (a, b) {
  case (int i, int j ) :
    //...
  case (int i, auto j ) :
    //...
  default:
    assert(false);
}
```

The sum types would be grouped in this case using the match (a,b) and the cases would be the variadic part. The cases would be matched sequentially.

This is a major motivation to place the sum type variables as the first parameter and let the matchers variadic since the second parameter.

const aware matchers

The proposed match function don't allows to change the sum type. That is the

```
const boost::variant<int, X> a = 2;
std::experimental::match(a,
  [](int& I)
    {
      ++i;
    },
  [](X const& i)
    {
      assert(false);
    },
  [](...)
    {
      assert(false);
    }
  ));
```

A matcher must match the exact type. The following will assert false

```
const boost::variant<int, X> a = 2;
std::experimental::inspect(a).overload(
  [](int& I) { ++i; },
  [](...)    { assert(false); }
  );
```

and the following will even not compile

```
const boost::variant<int, X> a = 2;
std::experimental::inspect(a).overload(
  [](int& I) { ++i; }
  [](X const& i)
    {
      assert(false);
    },
  );
```

Non-const aware matchers

The question is if we want the matcher to be able to change the sum types. As the matcher has access only to the current alternative for each match it could at best be able to change the value preserving the alternative type.

This proposal doesn't includes match overloads for non-const sum types, but there is no major problem to support them.

Customization point

The customization point

```
template <class ST, class F>
  match(ST const&, F&& );
```

Grouping with `overload`

We can also group all the functions with the `overload` function and let the variadic part for the sum types.

```
boost::variant<int, X> a = 2;
boost::optional<int> b = 2;
match<void>(overload(
  [](int i, int j )
    {
      //...
    },
  [](...)
    {
      assert(false);
    }
  ), a, b);
```

This has the advantage of having a single prototype resolving the problem with a single function overload. The liability is much a question of style. The author prefer to give the sum types first and the overloads later. If we had a language type pattern matching feature the author guess it would be much more like

```
boost::variant<int, X> a = 2;
boost::optional<int> b = 2;
match (a, b) {
  case (int i, int j ) :
    //...
  case (auto const&i, auto const&j ) :
    assert(false);
  default:
    assert(false);
}
```

SS

We could try to apply the language-like math with types that have as single alternative, themselves

```
int a = 2;
boost::optional<int> b = 2;
match (a, b) {
  case (int i, int const &j ) :
    //... make use of i and j
  case (int i, auto const &j ) :
    assert(false);
  default:
    assert(false);
}
```

This seems not natural as we are able to use directly the variable a inside each match case.

```
int a = 2;
boost::optional<int> b = 2;
match (b) {
  case (int j ) :
    //... make use of a and j
  case (auto const &j ) :
    assert(false);
  default:
    assert(false);
}
```

Using the library solution each case is represented by a function and the function would not have direct access to the variable a

```
int a = 2;
boost::optional<int> b = 2;
auto x = inspect(b).match(
  [a] (int j ) {
    return sum(a,j);
  },
  [a] auto const &j ) :
    assert(false);
  default:
    assert(false);
}
```

Providing

```
int a = 2;
boost::optional<int> b = 2;
inspect(a, b).match(
  [] (int i, int j) {
    return sum(i,j);
  },
  [] (auto const &j ) :
    assert(false);
  [](...) {
    assert(false);}
);
```

Open Points

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

- **`match` versus `visit`**

  N proposes a `visit` function to visit variants that takes the arguments in the reverse order.

  What do we prefer `visit` or `match`?

  Which order of arguments do we prefer?

  Do we want a variadic function of overloads or just an overloaded visitor functor?

- **Seen a type T as a sum type with a single type.**

  Do we want to support this case?

- **Matching several sum types**

  Do we want the inspect factory?

Technical Specification

Header <experimental/meta> Synopsis

```
namespace std {
namespace experimental {
inline namespace fundamental_v2 {
namespace meta {
  template <class ...Ts>
    struct types {};
}
}
}
}
```

Header <experimental/functional> Synopsis

```
namespace std {
namespace experimental {
inline namespace fundamental_v2 {

  template <class ST>
    struct sum_type_alternatives; // undefined
  template <class ST>
    struct sum_type_alternatives<const ST>;
  template <class ST>
    struct sum_type_alternatives<volatile ST>;
  template <class ST>
    struct sum_type_alternatives<const volatile ST>;
  template <class ST>
    using sum_type_alternatives_t = typename sum_type_alternatives<ST>::type;

  template <class T, class F>
    'see below' match(const T &that, F && f);

  template <class R, class ST, class... Fs>
    'see below' match(const ST &that, Fs &&... fcts);

  template <class R, class... STs, class... Fs>
    'see below' match(const std::tuple<STs...> &those, Fs &&... fcts);

}}}
```

Type trait `sum_type_alternatives`

The nested type `sum_type_alternatives<ST>::type` must define the tuple-like helper meta-functions `std::tuple_size` and `std::tuple_element`.

Template function `match`

```
template <class R, class ST, class... Fs>
'see below' match(const ST &that, Fs &&... fcts);
```

Requires: Let R`i` be `decltype(overload(forward<Fs>(fcts)...)(declval<sum_type_alternative<ST,i>()))` `for i in 1...tuple_size<sum_type_alternatives<ST>>.` `Ri` must be explicitly convertible to `R`.

Returns: the result of calling the overloaded functions `fcts` depending on the type stored on the sum type using the customization point match_custom as if

```
return match_custom(id<R>, that, overload(forward<Fs>(fcts)...));
```

Remarks: This function will not participate in overload resolution if ST is a tuple type.

Throws: Any exception thrown during the construction any internal object or thrown by the call of the selected overloaded function.

Template function `match`

```
template <class R, class... STs, class... Fs>
'see below' match(const std::tuple<STs...> &those, Fs &&... fcts);
```

Requires: Let `{i,j, …}` one element of the cartesian product `1...tuple_size<sum_type_alternatives<STs>>....`

`decltype(overload(forward<Fs>(fcts)...)(declval<sum_type_alternative<STs,i>>(), declval<sum_type_alternative<STs,j>>(), ...)` must be explicitly convertible to `R`.

Returns: the result of calling the overloaded functions `fcts` depending on the type stored on the sum types `STs....`

Throws: Any exception thrown during the construction any internal object or thrown by the call of the selected overloaded function.

Customization point `match`

Given a types ST and the visitors V and CV and the variables of S s, const S cs and V v and CV cv the following expression must be well formed

```
R r = match(s, v);

R r = match(cs, cv);
```

Header <experimental/optional> Synopsis

```
namespace std {
```

```
namespace experimental {
inline namespace fundamental_v2 {
      template <class T >
        struct sum_type_alternatives<optional<T>>
      {
        using type = types<nullopt_t, T>;
      }

      template <class R, class F, class ...Ts >
      R match_custom(id<R>, optional<T> const& v, F&& f)
      {
        if (v)
          return f(v.get());
        else
          return f(nullopt);
      }
      template <class R, class F, class ...Ts >
      R match_custom(id<R>, optional<T>& v, F&& f)
      {
        if (v)
          return f(v.get());
        else
          return f(nullopt);
      }
}}}
```

Header <experimental/variant> Synopsis

```
namespace std {
namespace experimental {
inline namespace fundamental_v2 {
      template <class ...Ts >
        struct sum_type_alternatives<variant<Ts...>>
      {
        using type = types<Ts...>;
      }

      template <class R, class F, class ...Ts >
      R match_custom(id<R>, variant<Ts...> const& v, F&& f)
      { return visit(std::forward<F>(f), v); }
      template <class R, class F, class ...Ts >
      R match_custom(id<R>, variant<Ts...>& v, F&& f)
      { return visit(std::forward<F>(f), v); }
}}}
```

Implementation

There is an implementation at https://github.com/viboes/tags including customization for `boost::variant` and `std::experimental::optional`.

Acknowledgements

Many thanks to Mat Calabrese who suggested the `common_type` approach.

References

- [N4542] N4542 - Variant: a type-safe union (v4) http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4542.pdf
- [N3915] N3915 - apply() call a function with arguments from a tuple (V3)

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3915.pdf
- [D0051] D0051 - C++ generic overload function

  https://github.com/viboes/tags/tree/master/doc/proposals/overload/D0051.pdf
- [PM] Open Pattern Matching for C++

  http://www.stroustrup.com/OpenPatternMatching.pdf